EPICS Application Developer's Guide

EPICS Base Release 3.16.0

10 November 2016

Martin R. Kraimer, Janet B. Anderson (Retired)

Andrew N. Johnson (Argonne National Laboratory)

W. Eric Norum (Lawrence Berkeley National Laboratory)

Jeffrey O. Hill (Los Alamos National Laboratory)

Ralph Lange (ITER Organization)

Benjamin Franksen (Helmholtz-Zentrum Berlin)

Peter Denison (Diamond Light Source)

Michael Davidsaver (Osprey DCS)

Contents

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 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
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) interrfaces 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" accessable 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 be aware the following additional documentation:

- *EPICS Record Reference Manual*, Philip Stanley, Janet Anderson and Marty Kraimer
- *EPICS R3.14 Channel Access Reference Manual*, Jeffrey O. Hill
- *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 Zieman (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 the functionality of the `dbLoadRecords` and `dbLoadTemplate` commands.

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.dctsdr`) were replaced by all ASCII files. The new method provides architecture independence and a more flexible environment for configuring the record types, device and 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 HPUX.

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 familarity with IOC concepts such as record types, device and 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:

Unix/Linux

```
echo $EPICS_HOST_ARCH
```

or

Windows

```
set EPICS_HOST_ARCH
```

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 must be invoked with the command `perl <scriptname>` on Windows. Perl script names are case sensitive. For example to create an application on Windows:

```
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.stt` can be compiled, the sequencer module must have been 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 this line:

  ```
  #seq sncExample, "user=<user>"
  ```

The Makefile contains commands for building the sncExample code both as a component of the example IOC application and as a standalone program called `sncProgram`, an executable that connects through Channel Access to a separate IOC 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 Solaris

```
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 the 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 program expects a pvname argument, connects and reads the current value for the pv, 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 program expects a filename argument 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 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 4 "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` and `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 contains references to the appropriate library modules. The following database definitions keywords are used for this purpose:

```
recordtype
device
driver
function
variable
registrar
```

The method also requires that IOC components contain an appropriate epicsExport statement. All components must contain the statement:

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

Any component that defines any exported functions must also contain:

```
#include <registryFunction.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,drvXxx);
```

Functions are registered using an `epicsRegisterFunction` macro in the C source file containing the function, along with a `function` statement in the application database description file. The makeBaseApp example thus contains the following statements to register a pair of functions for use with a subroutine record:

```
epicsRegisterFunction(mySubInit);
epicsRegisterFunction(mySubProcess);
```

The database definition keyword `variable` forces a reference to an integer or double variable, 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 named registrar function that has the prototype:

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

This function normally registers things, as described in Chapter 21, "Registry" on page 335. The makeBaseApp example provides a sample iocsh command which is registered with the following registrar function:

```
static void helloRegister(void) {
    iocshRegister(&helloFuncDef, helloCallFunc);
}
epicsExportRegistrar(helloRegister);
```

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` variable requests that a library 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

# Add support from base/src/vxWorks if needed
myexample_OBJS_vxWorks += $(EPICS_BASE_BIN)/vxComLibrary

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

`PROD_IOC` sets the name of the ioc application, here called `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 that definition, the created `myexampleInclude.dbd` will contain an include statement for that filename. In this case 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. Names of the form `<name>_registerRecordDeviceDriver.cpp`, are special; when they are seen the perl script `registerRecordDeviceDriver.pl` is executed and given `<name>.dbd` as input. This script generates the `<name>_registerRecordDeviceDriver.cpp` file automatically.

2.6 makeBaseApp.pl

`makeBaseApp.pl` 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.pl` 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`
: 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 its value is used.
  - If a `<top>/configure/RELEASE` file exists and defines a value for `EPICS_BASE` it is used.
  - It is obtained from the invocation of the makeBaseApp program. 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 its value is used.
  - If a `<top>/configure/RELEASE` file exists and defines a value for `TEMPLATE_TOP` it is used.
  - If `EPICS_MBA_TEMPLATE_TOP` is defined its value 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 its value 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`, this file may:
  - Replace one or both of the above subroutines with its own versions.
  - Provide a subroutine `ReplaceFilenameHook($file)` which will be called at the end of the subroutine `ReplaceFilename` described above.
  - Provide a subroutine `ReplaceLineHook($line)` which is called at the end of `ReplaceLine`.
  - Include other code which is run after the command line options have been interpreted.

2.6.5 makeBaseApp templetes 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 can connect the console to a terminal window on your workstation. On Linux the 'screen' program lets you communicate through a local serial port; run `screen /dev/ttyS0` if the IOC is connected to `ttyS0`.

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 applicaion 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

Motrola 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?

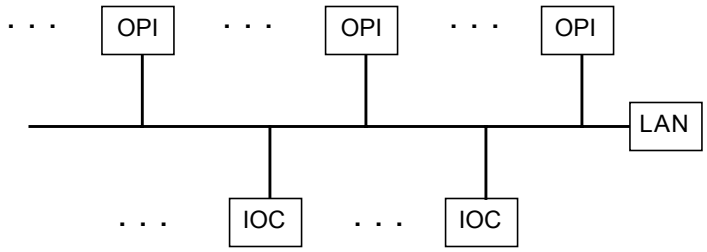
The Experimental Physics and Industrial Control System (EPICS) consists of a set of software components and tools that Application Developers can use to create control systems. The basic components are:

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.

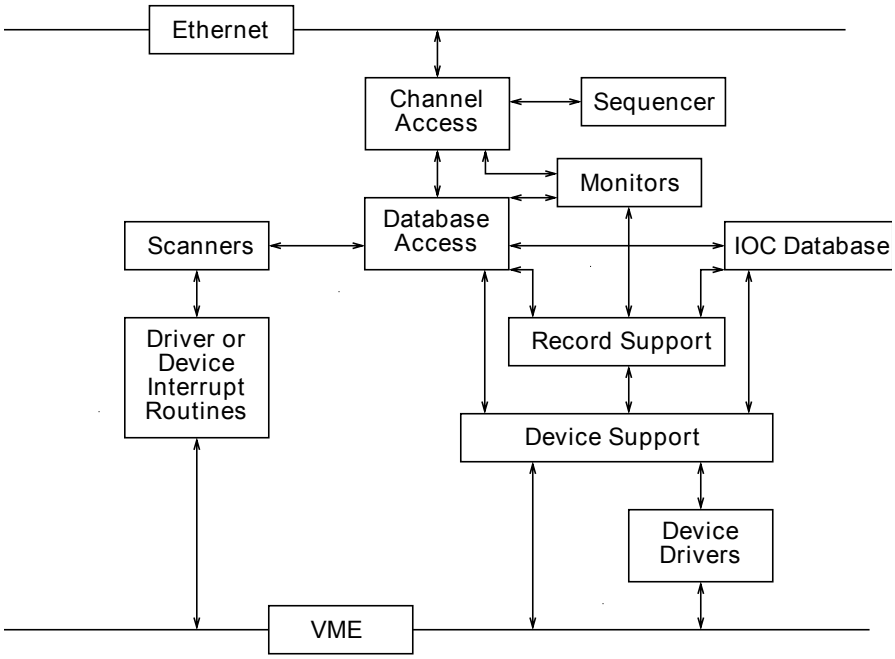
3.2 Basic Attributes

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 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.

3.3.1 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.

3.3.2 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 for details.

3.3.3 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.
- **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.

3.3.4 Record Support, Device Support and Device Drivers

Database access needs no record-type specific knowledge, each record-type provides a set of record support routines that implement all record-specific behavior. 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 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.

3.3.5 Channel Access

Channel Access is discussed in the next section.

3.3.6 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.

3.4 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.

3.4.1 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.

3.4.2 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 and "I have it" message.

3.4.3 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`.

3.4.4 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.

3.5 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.

3.5.1 Examples of Channel Access Tools

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

- **CSS**: Control System Studio, an Eclipse RCP application with many available plug-ins.
- **EDM**: Extensible Display Manager.
- **MEDM**: Motif Editor and Display Manager.
- **StripTool**: A general-purpose stripchart program.
- **ALH**: Alarm Handler. General purpose alarm handler driven by an alarm configuration file.
- **Sequencer**: Runs in an IOC and emulates a finite state machine.
- **Probe**: Allows the user to monitor and/or change a single process variable specified at run time.

3.5.2 Examples of other Tools

- **VDCT**: A Java based database configuration tool which is quickly becoming the recommended database configuration tool.
- **SNC**: State Notation Compiler. It generates a C program that represents the states for the IOC Sequencer tool.

3.6 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, most applications will need equivalent functionality to 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

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`, where `dir1`, `dir2`, ... are user created subdirectory trees with `Makefiles` and source files to be built. Because the build rules allow make commands like "`make install.vxWorks-68040`", subdirectory names within a `<top>` directory structure may not contain a period "`.`" character.

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, extensions, and IOC applications, the default value can be changed in the `configure/CONFIG_SITE` file. The installation directory for the EPICS components is controlled by the definition of `INSTALL_LOCATION`

The following subdirectories may 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` does not equal `TOP`).
- `cfg` – The directory into which user created configure files are installed

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
- User created configuration files in build created `$(INSTALL_LOCATION)/cfg` directories.

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 `cygwin-x86`.

4.2.2 Software Prerequisites

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

- Perl version 5.8 or greater
- GNU make, version 3.81 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 or vxWorks 6.x 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. You can override the default settings by defining PERL, CC and CCC, GCC and G++, GNU_DIR ... in the appropriate file (usually `configure/os/CONFIG_SITE.$EPICS_HOST_ARCH.Common`)

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)/` to your path so the shared libraries, dlls, can be found during the build.. 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 Directory names

Because the build rules allow make commands like "`make <dir>.<action>,<arch>`", subdirectory names within a `<top>` directory structure may not contain a period"." character.

4.2.5 EPICS_HOST_ARCH environment variable

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.You must also modify the following tow macros in the base/configure/CONFIG_SITE file:

COMPAT_TOOLS_313 - Set to YES to build R3.13 extensions with this base.

COMPAT_313 - Set to YES to build R3.13 ioc applications and extensions with this base.

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 in the `configure/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)/javalib

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 application source files. Build definitions such as
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>, and may also include other files to take those definitions from elsewhere. The macros defined in the RELEASE file (or its includes) may reference other defined macros, but cannot rely on environment variables to provide definitions.

When make is executed, macro definitions for include, bin, and library directories are automatically generated for each external <top> definition given in the RELEASE file. Also generated are include statements for any existing RULES_BUILD files, cfg/RULES* files, and cfg/CONFIG* files from each external <top> listed in the RELEASE file.

For example, if configure/RELEASE contains the definition

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

then the generated macros will be:

```
CAMAC_HOST_BIN = /home/epics/modules/bus/camac/bin/$(EPICS_HOST_ARCH)
CAMAC_HOST_LIB = /home/epics/modules/bus/camac/lib/$(EPICS_HOST_ARCH
CAMAC_BIN = /home/epics/modules/bus/camac/bin/$(T_A)
CAMAC_LIB = /home/epics/modules/bus/camac/lib/$(T_A)
RELEASE_INCLUDES += -I/home/epics/modules/bus/camac/include/os
RELEASE_INCLUDES += -I/home/epics/modules/bus/camac/include
RELEASE_DBDFLAGS += -I /home/epics/modules/bus/camac/dbd
RELEASE_DBFLAGS += -I/home/epics/modules/bus/camac/db
RELEASE_PERL_MODULE_DIRS += /home/epics/modules/bus/camac/lib/perl
```

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 host and target architectures by providing the appropriate file or files containing overriding definitions.

```
configure/RELEASE.<epics_host_arch>.Common
configure/RELEASE.Common.<targetarch>
configure/RELEASE.<epics_host_arch>.<targetarch>
```

For <top> directory structures created by makeBaseApp.pl, an EPICS base perl script, convertRelease.pl can perform consistency checks for the external <top> definitions in the RELEASE file and its includes as part of the <top> level build. Consistancy checks are controlled by value of CHECK_RELEASE which is defined in <top>/configure/CONFIG_SITE. CHECK_RELEASE can be set to YES, NO or WARN, and if YES (the default value), consistency checks will be performed. If CHECK_RELEASE is set to WARN the build will continue even if conflicts are found.

4.3.5 Modifying configure/RELEASE* files

You should always do a gnumake clean uninstall in the <top> 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>/configure/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>/configure/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 support <top> 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/R3.14.9
EPICS_BASE = $(SUPPORT)/base/3-14-9-asd1
```

4.3.6 OS Class 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 architectures for which a particular definition is used. Most Makefile definition names can be specified with an appended underscore "_" followed by an osclass name. 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 definition exists but should not apply to a particular system OS Class, 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 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 vwWorks-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.

4.3.9 User specific override definitions

User specific override definitions are allowed in user created files in the user's `<home>/configure` subdirectory. These override definitions will be used for builds in all `<top>` directory structures. The files must have the following names.

```
<home>/configure/CONFIG_USER
<home>/configure/CONFIG_USER.<epics_host_arch>
<home>/configure/CONFIG_USER.Common.<targetarch>
<home>/configure/CONFIG_USER.<epics_host_arch>.<targetarch>
```

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 help This command can be executed from the `<top>` directory only. This command prints a page describing the most frequently used make commands.

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

gnumake all This is the same as "gnumake install".

gnumake buildInstall This is the same as "gnumake install".

gnumake <arch> This rebuilds and installs everything that is not up to date first for the host arch and then (if different) for the 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 archclean This command will remove the current build's `O.<arch>` directories but not `O.Common` directory.

gnumake realclean This command 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 cvsclean This command can be executed from the <top> directory only. It removes cvs .#* files in the make 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 an architecture such as solaris-sparc, vxWorks-68040, win32-x86, etc.

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

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

NOTE: realclean cannot be specified inside an O.<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.

The build system ensures the host architecture is up to date before building a cross-compiled target, thus Makefiles must be explicit in defining which architectures a component should be built for.

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.

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.<arch>` 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. Source code files may have the suffix .c, .cc, .cpp, or .C.

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 using the EPICS base utility program makeBpt 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 using the EPICS base utility program dbToRecordTypeH. 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 using the EPICS base utility program dbToMenuH 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 EPICS base 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 or can be created using other make rules.

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

An example of registering the variable mySubDebug and the routines mySubInit and mySubProcess is `<name>.dbd` containg 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 are made dependencies of the created <name>.db by the makeDbDepends.pl perl tool.

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 CREATESUBSTITUTIONS variable (e.g. CREATESUBSTITUTIONS=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 "`-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_<arch> += <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_<arch> += <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_<arch> += <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>`

c++ 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 `$(INSTALL_LOCATION)/lib/<arch>`. Two library files are created when SHARED_LIBRARIES=YES, `<name>.lib`, an import library for DLLs, which is installed into `$(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. NOTE: the `<name>.lib` files are different for shared and nonshared builds.

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_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.

`LIB_OBJLIBS += <name>`

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

`LIB_OBJLIBS_<osclass> += <name>`

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

`LIB_OBJLIBS_DEFAULT += <name>`

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

`<libname>_OBJLIBS += <name>`

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

`<libname>_OBJLIBS_<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>_LDOBJS_<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 3.14 Base releases 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 configure/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 osclasswhen 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.

item `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/<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/<osc
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 yac

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. Lex converts a <name>.l Lex code file to a lex.yy.c file which the build rules renames to <name>.c. Yacc converts a <name>.y yacc code file to a y.tab.c file, which the build rules renames to <name>.c. Optionally yacc can create a y.tab.h file which the build rules renames to <name>.h.

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>_LDOBJS += <name>`

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

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

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

`<prodname>_LDOBJS_DEFAULT += <name>`

Object files will be used in builds of the product for archs without a `<prodname>_LDOBJS_<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 osclasswhen 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 Generate version header

A header can be generated which defines a single string macro with an automatically generated identifier. The default is the ISO 8601 formatted time of the build. A revision id is used if a supported version control system is present. This will typically be used to make an automatically updated source version number visible at runtime (eg. with a stringin record).

To enable this the variable `GENVERSION` must be set with the desired name of the generated header. By default this variable is empty and no header will be generated. If specified, this variable must be set before `configure/RULES` is included.

It is also necessary to add an explicit dependency for each source file which includes the generated header.

An Makefile which generates a version header named "myversion.h" included by "devVersionString.c" would have the following.

```
TOP=../..
include $(TOP)/configure/CONFIG
# ... define PROD or LIBRARY names sometarget
sometarget_SRCS = devVersionString.c
GENVERSION = myversion.h
include $(TOP)/configure/RULES
# for each source file
devVersionString$(DEP): $(GENVERSION)
```

The optional variables `GENVERSIONMACRO` and `GENVERSIONDEFAULT` give the name of the C macro which will be defined in the generated header, and its default value if no version control system is being used. To avoid conflicts, the macro name must be changed from its default `MODULEVERSION` if the version header is to be installed.

4.6.23.8 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.Static builds may not be possible on all systems. For static builds, all nonsystem libraries must have an archive version, and this may not be true form all libraries.

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 definition script files for all products and libraries.
`RCS_<osclass> += <name>`

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

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

`<name>_RCS += <name>` Resource definition script 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 resource definition script 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 depreciation 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.

`JAR_PACKAGES += <name>`

Names of java packages to be installed and added to 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
```

4.6.34 User Created CONFIG* and RULES* files

Module developers can now create new CONFIG and RULES* files ia a `<top>` application source directory. These new CONFIG* or RULES* files will be installed into the directory $(INSTALL_LOCATION)/cfg by including lines like the following Makefile line:

```
CFG += CONFIG_MY1 RULES_MY1
```

The build will install the new files `CONFIG_MY1` and `RULES_MY1` into the $(INSTALL_LOCATION)/cfg directory.

Files in a $(INSTALL_LOCATION)/cfg directory are now included during a build by so that the definitions and rules in them are available for use by later src directory Makefiles in the same module or by other modules with a RELEASE line pointing to the TOP of this module.

4.6.35 User Created File Types

Module developers can now define a new type of file, e.g. ABC, so that files of type ABC will be installed into a directory defined by INSTALL_ABC. This is done by creating a new `CONFIG_<name>` file, e.g. CONFIG_ABC, with the following lines:

FILE_TYPE += ABC

INSTALL_ABC = $(INSTALL_LOCATION)/abc

The INSTALL_ABC directory should be a subdirectory of $(INSTALL_LOCATION). The file type ABC should be target architecture independent (alh files, medm files, edm files.

Optional rules necessary for files of type ABC should be put in a RULES_ABC file.

The module developer installs new CONFIG_ABC and RULES_ABC files for the new file type into the directory $(INSTALL_LOCATION)/cfg by including the following Makefile line:

CFG += CONFIG_ABC RULES_ABC

Files of type ABC are installed into INSTALL_ABC directory by adding a line like the following to a Makefile.

```
ABC += <filename1> <filename2> <filename3>
```

Since the files in $(INSTALL_LOCATION)/cfg directory are now included by the base config files, the ABC += definition lines are available for use by later src directory Makefiles in the same module or by other modules with a RELEASE line pointing to the TOP of this module.

4.6.36 Assemblies

A single output file is generated from assembling specified snippet files. Snippet file names start with numbers and are sorted when the snippets are concatenated: first by the number, then alphabetical by the remaining part of the name. (This mechanism is conceptually similar to the Linux convention of collecting configuration file snippets in *.d directories.)

Snippets with file names not starting with a number or ending in '~' are ignored. The specified snippets are processed in the order they appear on the command line. Multiple snippets with the same number are concatenated. "Commands" (tags in the snippet name) can be used to control the treatment of snippets with the same number:

D - Default
: Snippet is treated as a default, which is replaced (overwritten) by any other snippet with the same number.

R - Replace
: Snippet is replacing (overwriting) already processed snippets with the same number.

Specification of the target file is different for architecture dependent or independent files.

```
COMMON_ASSEMBLIES += st.cmd

ASSEMBLIES += mytool.rc
```

Snippet files are configured specifically (relative or absolute path) or as patterns (searched relative to all source directories).

```
mytool.rc_SNIPPETS += ../rc.d/10_head ../rc.d/20_init

st.cmd_PATTERN += st.cmd.d/*
```

4.6.36.1 Macros

The following macros can be used in snippets, and will be replaced by the current value when assembling is done.

`_DATETIME_` Date and time of the build

`_USERNAME_` Name of the user running the build

`_HOST_` Name of the host on which the build is run

`_OUTPUTFILE_` Name of the generated file

`_SNIPPETFILE_` Name of the current snippet

4.6.36.2 Example

This mechanism can be used to create an IOC startup file from snippets in a global and an application specific directory, allowing applications to add commands to different phases of the IOC startup by dropping appropriately numbered snippets into the directory.

Given the following directories and snippets:

```
/global/st.cmd.d:   (G=GLOBAL)
    D10_init
    20_environment
    30_drivers
    D40_settings
    70_start-ioc

../st.cmd.d:        (L=LOCAL)
    D10_init
    40_settings
    40_settings~
    30_another-driver
    R70_start-my-ioc
```

And the following Makefile declaration:

```
SCRIPTS += $(COMMON_DIR)/st.cmd
COMMON_ASSEMBLIES += st.cmd
st.cmd_SNIPPETS += $(wildcard /global/st.cmd.d/*)
st.cmd_PATTERN += st.cmd.d/*
```

The build will create and install a `st.cmd` script using the following snippets:

Source	Snippet	Comment
L	`10_init`	L default resets the G default
G	`20_environment`	
L	`30_another-driver`	implicit addition, alphabetical sorting
G	`30_drivers`	
L	`40_settings`	replacing a default, ignoring backup file
L	`70_start-my-ioc`	explicit replace

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
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
Test scripts to be built	
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 or Ca.dll on WIN32
LIBRARY_<osclass>	os specific libraries to build and install

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 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 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
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 $(AB_BIN)/lb)
LIB_OBJS_<osclass>	os-specific library object files, specify without suffix,
LIB_OBJS_DEFAULT	library object files, specified without suffix, for archs with no LIB_OBJS_<osclass> specified
<name>_OBJS	object files, specified without suffix, to build a specific PROD or LIBRARY
<name>_OBJS_<osclass>	os specific object files, specified without suffix, to build a specific PROD or LI|BRARY
<name>_OBJS_DEFAULT	object files, without suffix, needed to build a specific PROD or LIBRARY for archs with no <prod>_OBJS_<osclass> specified
Product and library R3.13 combined object files	
USR_OBJLIBS	combined object files with filenames that do not have a suffix, needed for building all PRODs and LIBRARYs (e.g. USR_OBJLIBS+=$(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 LI|BRARY
<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)
<name>_LDOBJS_<osclass>	os specific combined object files with filenames that do not have a suffix, to build a specific PROD or LI|BRARY (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)System libraries do not need a <name>_dir definition.
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
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_CFLAGS=-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$(EPICS_EXTENSIONS_INCLUDE))
USR_INCLUDES_<osclass>	directories, with -I prefix, to search for include files for a specific os class
USR_INCLUDES_DEFAULT	directories, with -I prefix, to search for include files for systems with no <name>_INCLUDES_<osclass> specified
<name>_INCLUDES	directories, with -I prefix, to search for include files when building a specific object file (e.g. -I$(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)
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	
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
USR_DBFLAGS	optional flags for msi (EPICS Macro Substitution Tool)
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
USR_ARFLAGS	ar options
Facilities for building Java programs	
JAVA	names of Java source files to be built and installed
TESTJAVA	names of Java source 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
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
Assemblies	
ASSEMBLIES	names of files to be assembled from snippets
COMMON_ASSEMBLIES	names of arch-independent files to be assembled from snippets
<name>_SNIPPETS	snippet files needed to build a specific assembly
<name>_PATTERN	patterns for snippet files (searched from all source directories) needed to build a specific assembly
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))
GENVERSION	If set, the name of a generated header file with the module version string.
GENVERSIONMACRO	The CPP macro name written into the generated version header (default MODULEVERSION).
GENVERSIONDEFAULT	The default version string written into the generated header if no VCS system is in use. Leave unset to use build time.

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_APP_INCLUDE`

Definitions to generate include, bin, lib, perl module, db, and dbd directory definitions for RELEASE `<top>`s.

`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_FILE_TYPE`

Definitions to allow user created file types.

`CONFIG_SITE`

File in which you add to or modify make variables in EPICS base. A definition commonly 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.ioc`

Rules which allow building in the `iocBoot/<iocname>` directory of a makeBaseApp created ioc application.

`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_EXPAND`

Definitions and rules to use expandVars.pl to expand @VAR@ variables in a file.

`RULES_FILE_TYPE`

Definitions and rules to allow user created CONFIG* and RULES* files and rules to allow user created file types.

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

`RULES_TARGET`

Makefile code to create target specific dependency lines for libraries and product targets.

`RULES_TOP`

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

`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 targets

`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 definitions for all hosts

`CONFIG_SITE.<host>.Common`

Site specific host definitions for all targets

4.8.4 Base src/tools File Descriptions

The src/tools directory contains Perl script tools used for the build. The are installed by the build into `$(INSTALL_LOCATION)/bin/$(T_A)` for Host type target archs. The tools currently in this directory are:

convertRelease.pl This Perl script does consistency checks for the external `<top>` definitions in the RELEASE file. This script also creates envPaths, cdCommands, and dllPath.bat files for vxWorks and other IOCs.

cvsclean.pl This perl script finds and deletes cvs .#* files in all directories of the directory tree.

dos2unix.pl This perl script converts text file in DOS CR/LF format to unix ISO format.

expandVars.pl This perl tool expands `@VAR@` variables while copying a file.

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.

makeDbDepends.pl This perl script searches .substitutions and .template files for entries to create a DEPENDS file.

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.

makeTestfile.pl This perl script generates a file $target.t which executes a real test program in the same directory.

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.

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.

useManifestTool.pl This tools uses MS Visual C++ compiler version number to determine if we want to use the Manifest Tool (status=1) or not (status=0).

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.MS_WINDOWS Microsoft WIN32 specific instructions

README.niCpu030 NI cpu030 specific instructions

README.hpux HPUX 11 (hpux-parisc) specific instructions

README.cris Cris architecture specific instructions

README.tru64unix Tru64Unix/Alpha 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.

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

cygwin.bat WIN32 bat file to set path and environment variables for building with cygwin gcc/g++ compilers

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 (mutual exclusion) 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 (`CONSTANT`).

    Not discussed in this chapter

  - database link (`DB_LINK`).

    A link to another record in the same IOC.

  - channel access link (`CA_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 Link Operations

The basic operations which can be preformed on a link (excluding hardware links) are as follows.

- 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 points to a (normally passive) record that should be processed after the record that contains the link.

For input and output links, two additional attributes can be specified by the application developer: process passive, and maximize severity.

5.3.1 Process Passive

The Process Passive attribute takes the value `NPP` (Non-Process Passive) or `PP` (Process Passive). 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 only if link's Process Passive attribute is `PP` and the target record's `SCAN` field is Passive.

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

The Maximize Severity attribute is one of `NMS` (Non-Maximize Severity), `MS` (Maximize Severity), `MSS` (Maximize Status and Severity) or `MSI` (Maximize Severity if Invalid). It determines whether alarm severity is propagated across links. If the attribute is `MSI` only a severity of `INVALID_ALARM` is propagated; settings of `MS` or `MSS` propagate all alarms that are more severe than the record's current severity. 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. If the severity is changed the associated alarm status is set to `LINK_ALARM`, except if the attribute is `MSS` when the alarm status will be copied along with the severity.

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 alarm status reflects the first one detected.

5.4 Database Locking

Locking is required to prevent corruption of record fields due to concurrent access by different threads. Record locking can either be done for a single record with dbScanLock, or for a list of records with dbScanLockMany.

Before any record field is accessed, the record must be locked by calling either dbScanLock or dbScanLockMany.

Further details on the algorithms used to implement locking operations can be found in section 5.13.

5.4.1 Single Record Locking

```
dbScanLock(struct dbCommon *precord);
dbScanUnlock(struct dbCommon *precord);
```

A single record may be locked for access with a call to dbScanLock and unlocked later with a call to dbScanUnlock.

A thread must only lock one record at a time with dbScanLock, except as discussed in section 5.4.3.

5.4.2 Multi-Record Locking

```
typedef struct dbLocker dbLocker;
dbLocker *dbLockerAlloc(struct dbCommon **precs,
                        size_t nrecs,
                        unsigned int flags);

void dbLockerFree(dbLocker *);
void dbScanLockMany(dbLocker*);
void dbScanUnlockMany(dbLocker*);
```

It is possible to lock multiple records safely using dbScanLockMany. First a dbLocker* must be created from an array of record pointers. This object can be used to lock and unlock that particular group of records as many times as necessary with dbScanLockMany.

dbScanLockMany may not be called recursively. After calling dbScanLockMany a thread must call dbScanUnlockMany with the same dbLocker* before calling dbScanLockMany again.

dbScanLock may be called recursively as described in section 5.4.3.

The first argument to dbScanLockMany is an array of dbCommon* (i.e pointers to record instances), and the second is the number of elements in this array. The array may contain duplicate elements. Elements may be NULL.

The third argument to dbScanLockMany (flag) must be zero since no flags are defined at present.

5.4.3 Recursive Locking

Recursive locking is an attempt by a thread to lock a record which it has already locked. As for example:

```
/* This is valid recursive locking */
dbCommon *prec = ...;
dbScanLock(prec);
dbScanLock(prec);
dbScanUnlock(prec);
dbScanUnlock(prec);
```

But not:

```
/* This is NOT valid */
dbCommon *prec1 = ..., *prec2 = ...;
assert(prec1!=prec2);
dbScanLock(prec1);
dbScanLock(prec2); /* potential deadlock here! */
dbScanUnlock(prec2);
dbScanUnlock(prec1);
```

The rules for recursive locking with `dbScanLock` and `dbScanLockMany` are as follows:

- `dbScanLockMany` does not support recursion. A single thread can only hold one group lock (`dbLocker*`) at a time.
- `dbScanLock` may be used to recursively lock a record.
- `dbScanLock` may be used on a record which is already locked with `dbScanLockMany`.

Therefore the following is valid.

```
/* This is valid multi-locking */
dbCommon *precs[2] = {prec1, prec2};
dbLocker *L = dbLockerAlloc(precs, 2, 0);
dbScanLockMany(L);
dbScanLock(precs[0]);
dbScanUnlock(precs[0]);
dbScanLock(precs[1]);
dbScanUnlock(precs[1]);
dbScanUnlockMany(L);
dbLockerFree(L);
```

5.4.4 When to lock

A record is always locked while it is being processed by the IOC. So Device and Record Support code must never call `dbScanLock` nor `dbScanLockMany` from within any support callback function.

However, asynchronous device support may explicitly call `dbScanLock` when the asynchronous operation completes from a user thread or `CALLBACK`.

The functions `dbPutField` and `dbGetField` implicitly call `dbScanLock` or `dbScanLockMany`. The functions `dbPut` and `dbGet` do not.

All records connected by any kind of database link are placed in the same lock set. Versions of EPICS Base prior to R3.14 allowed an NPP NMS input link to span two different lock sets, but this was not safe when the read and write operations on the field value were not atomic in nature. This feature is no longer available to break a lockset.

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 the record processing routines `dbGetLink`, `dbPutLink`, and `dbPutField` call the `dbScanPassive` routine. Record processing routines call it for each forward link in the record.
- dbPutField: This routine sets the target field value and then, if the field was marked `pp(TRUE)` it calls `dbScanPassive`. Each field of each record type has an attribute `pp` declared as either `TRUE` or `FALSE` in the record definition file. The attribute is a global property which is set by the record type. This use of `pp` only affects calls to the `dbPutField` routine. 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 once the current processing completes.
- dbGetLink: If the link includes the process passive flag `PP` this routine first calls `dbScanPassive` to process the target record. Whether or not `dbScanPassive` was called, it then obtains the value from the target field.
- dbPutLink: This routine sets the target field. Then, if the link includes the process passive flag `PP` it calls `dbScanPassive` to process the target record. `dbPutLink` is only called from record processing routines. 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 later, once the current processing completes.

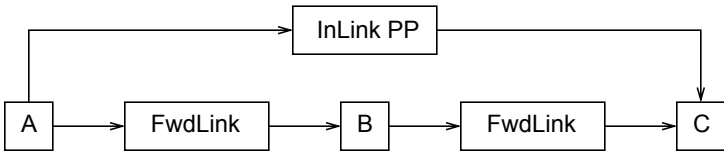
All non-record processing tasks (Channel Access, Sequence Programs, etc.) call `dbGetField` to obtain database values. `dbGetField` just reads values without asking that a record be processed.

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 the Chapter "Record Support".

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.

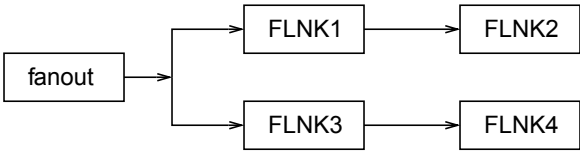
This brief example demonstrates that database links need more discussion.

5.7.1 Rules Relating to Database Links

5.7.1.1 Processing Order

The processing order follows 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 (such as the calculation or select records) the input values are nornally fetched in the natural order. For example for link fields named `INPA`, `INPB`, ..., `INPL`, the links would be read in the order A, B, C etc. Thus if obtaining an input results in a record being processed, the processing order is guaranteed. Some record types may not follow this rule however.
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` or `dbScanLockMany` 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.

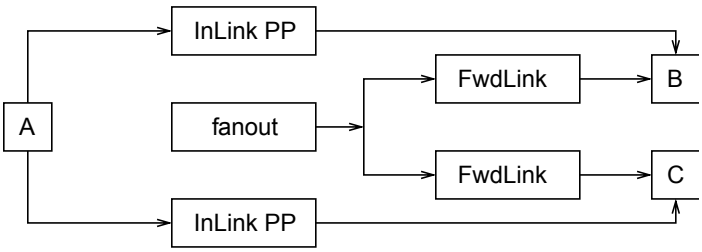
5.7.1.3 PACT - Process Active

Every 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. To prevent infinite processing loops, whenever a record gets processed through a forward link, or a database link with the `PP` link option, the linking record's `PACT` field is saved and set to

TRUE, then restored again afterwards. 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

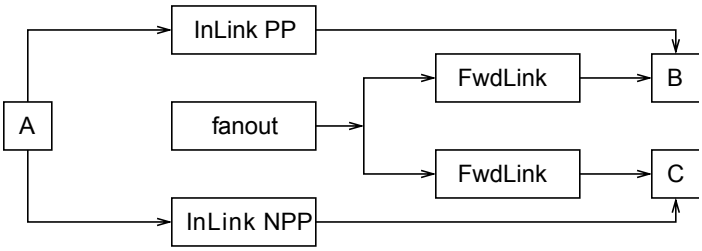
Input and output links have an option called process passive. For each such link the application developer can specify process passive TRUE (PP) or process passive FALSE (NPP). Consider the following example:



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 was processed twice. This is unnecessary. If the input link to C were declared No Process Passive then A would only be processed once. Thus a better solution would be:



5.7.1.5 Process Passive: Field attribute

All record type field definitions have an attribute called process_passive which is specified in the record definition file. It cannot be changed by an IOC application developer. This attribute is used only by dbPutField. It determines if a passive record will be processed after dbPutField sets 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 the option as `MS` (Maximize Severity), `NMS` (Non-Maximize Severity), `MSS` (Maximize Status and Severity) or `MSI` (Maximize Severity if Invalid).

When database input or output links are defined, the application developer can use this option to specify whether and how alarm severities should be propagated across links with the data. The alarm severity is transferred only if the new severity will be greater than the current severity of the destination record. If the severity is propagated the alarm status is set equal to `LINK_ALARM` (unless the link option is `MSS` when the alarm status will also be copied from the source record).

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 `PHAS` 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` attribute of each record field determines if a passive record will be 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 using this mechanism.
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 cover asynchronous device support. An example might be 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. Let's 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

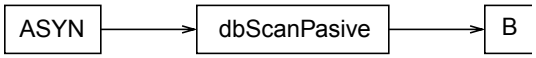
The asynchronous completion routine performs the following algorithm:

1. Record processing continues
2. Record specific alarm conditions are checked
3. Monitors are raised
4. Forward links are processed
5. `PACT` is set `FALSE`.

A few attributes of the above rules are:

1. Asynchronous record processing does not delay the scanners.
2. Between the time that 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`.
3. Forward and output links are triggered only when the asynchronous completion routine completes record processing.

With these rules the following works just fine:

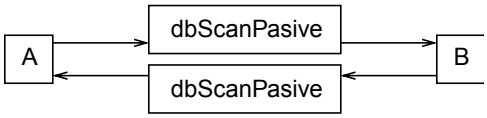


When `dbProcess` is called for record ASYN, processing will be started but `dbScanPassive` will not be called. Until the asynchronous completion routine executes any additional attempts to process ASYN are ignored. When the asynchronous callback is invoked the `dbScanPassive` is performed.

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.

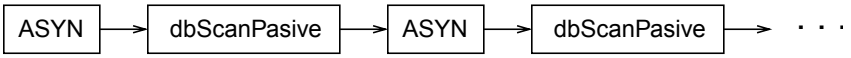
5.9.2 Obtain Old Data

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

If A is a passive asynchronous record then record B's `dbGetLink` request forces `dbProcess` to be called for record A. `dbProcess` starts the processing but returns immediately, before the operation has finished. `dbGetLink` then reads the field 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.

5.10 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 processNotify

`dbProcessNotify` is used when a Channel Access client calls `ca_put_callback` and makes a request to notify the caller when all records processed as a result of this put are complete. Because of asynchronous records and conditional use of database links between records this can be complicated and the set of records that are processed because of a put cannot be determined in advance. The processNotify system is described in section 15.4.3.3 on page 230. The result of a `dbProcessNotify` with type `putProcessRequest` is the same as a `dbPutField` except for the following:

- `dbProcessNotify` requests are queued rather than cached. Thus when additional requests are directed to a record that already has an active `dbProcessNotify`, they are queued. As each one finishes it releases the next one in the queue.
- If a `dbProcessNotify` links to a record that is not active but has a `dbProcessNotify` attached to it, 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, using the channel's native field type and element count. It also monitors the alarm status. Whenever the monitor callback gets invoked the new data is stored in a buffer belonging to dbCa. When iocCore or the record support module asks for data from the link, the contents of the buffer are converted to the requested type.
- For each output link, a buffer is allocated the first time iocCore/record support issues a put after the channel access connection has been made. This buffer is allocated large enough to store the channel's native field type and 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 use 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 support the passing of a process passive flag, 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:

- Input links always act like NPP.
- CA - Forces the link to be a channel access link.
- CP - Forces the link to be a channel access link and also requests that the record containing the link be processed whenever a monitor occurs.
- CPP - Force the link to be a channel access link and also requests 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:

- It is not possible to honor PP or NPP options; the put operation completes immediately but whether the destination record will process depends on the process passive attribute of the destination field.
- 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 performed each time a forward link request is issued. Because of this implementation, the requirement that a forward link can only point to a passive record does not apply to channel access forward links; the target record will be processed irrespective of the value of its `SSCAN` field.

The available options are:

- CA - Force the link to be a channel access link.

Maximize Severity is not honored.

5.13 Record Locking Algorithms

This section describes details of the implementation of `dbScanLock` and `dbScanLockMany`. Any discussion of links and linking in this section refers only to database links (`DB_LINK`). Other link types do not require record locking.

A lockset guards one or more records with an `epicMutexId`. Each lockset maintains a list of its member records.

The relationship between a record and a lockset forms the basis of the locking algorithms. Every record is always a member of some lockset throughout its lifetime. However, a record may move between locksets. The relationship between record and lockset is established in the `lockRecord*` private structure which is the `LSET` field of each record. Each `lockRecord` structure includes an `epicsSpin*` to maintain its consistency.

Records are associated with each other through links with the `DBF_INLINK`, `DBF_OUTLINK`, and `DBF_FWDLINK` field types. These links are directional, from the record with the link field, to the field of the record it is targeted at. This is a directed graph of records (nodes) and links (edges).

The existence of a database link between two records places them in the same lockset. This allows database processing chains involving multiple records to maintain consistency. Records which are not currently connected by any database link (directly or indirectly) are placed in different locksets. This enables parallel scanning of unrelated processing chains.

When a database link is created between two records in two different locksets, all the records in the locksets are moved into one lockset. The other (now empty) lockset is free'd. This is referred to as a merge operation.

Each time a database link between two records is broken it is possible that the lockset (graph) has become partitioned (split in two). When this occurs, a new lockset is created and populated with one set of connected records. This is referred to as a split operation.

Access and modification of the association between record and lockset is governed by the following rules:

- When changing the association, both the lockset mutex and the `lockRecord` spinlock must be locked.
- When reading the association, either the lockset mutex or the `lockRecord` spinlock must be locked.

A basic property of a spin lock is that it must not be held during any blocking operation, including locking a mutex. This defines the order of locking. The mutex (lockset) must be locked first, then the spinlock (`lockRecord`).

This complicates things because locking operations begin with record pointer(s) (`dbCommon*`). The spinlock must be locked first in order to find a record's current lockset. However, the spinlock must be unlocked before the lockset can be locked. Care must be taken as the association may change when neither is locked. Furthermore, when two locksets are merged, one of them will be free'd.

To handle this safely, each lockset contains a reference counter. The lockset will only be free'd when this counter falls to zero. This counter has one "count" for each active reference. Each `lockRecord` is an active reference. Further, a `dbLocker` may also hold active references.

The process of locking a lockset is as follows:

- Lock the `lockRecord` (spinlock)
- Increment the reference counter of the lockset
- Unlock the `lockRecord`
- Lock the lockset (mutex)
- Again lock the `lockRecord`
- Check that the record's lockset hasn't changed
- Unlock the `lockRecord`
- Decrement the reference counter of the lockset

There remains the possibility that the association between record and lockset may change during the moment between unlocking the spinlock and locking the mutex. This can be detected after the mutex has been locked. When it occurs, the whole operation must be re-tried with the new lockset.

We assume that database link modification is a relatively rare operation.

Locking multiple locksets is necessary when a database link is created. The underlying `epicsMutex` API only supports locking a single mutex in one call. Care must be taken to avoid a deadlock when locking the second, and beyond.

Two common strategies for avoiding deadlocks are to use a try-lock operation with ownership tracking, or to establish a global ordering. At present the second strategy is used. All lockset mutexes are placed into a global order by comparing their memory (pointer) address. Locking is done in order of increasing address.

Merging two locksets when a link is created is accomplished by locking both locksets, then concatenating their record lists into one. This leaves one empty lockset.

Splitting one lockset into two when a link is broken requires finding if the lockset has become partitioned. It is helpful to recognize that the act of removing one link between two records (say 'A' and 'B') can result in at most two locksets.

To determine if a lockset has been partitioned it is sufficient to start with one of the two records ('A'), then recursively traverse the remaining links to or from record 'A'. If record 'B' is encountered during this traversal, then the lockset has not been partitioned. If all records connected to 'A' can be traversed without finding 'B', then the lockset has been partitioned. All the records connected with 'A' become one lockset, while the remaining records (including 'B') become the second.

During IOC startup, the complete list of records is iterated (by `dbLockInitRecords`) and the required locksets are created and populated based on the links defined at the time.

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 statements.

6.2 Summary of Database Syntax

The following summarizes the Database Definition syntax:

```
path "path"
addpath "path"
include "filename"
#comment
menu(name) {
    include "filename"
```

```
        choice(choice_name, "choice_value")
        ...
    }

    recordtype(record_type) {}

    recordtype(record_type) {
        include "filename"
        field(field_name, field_type) {
            asl(asl_level)
            initial("init_value")
            promptgroup("group_name")
            prompt("prompt_value")
            special(special_value)
            pp(pp_value)
            interest(interest_level)
            base(base_type)
            size(size_value)
            extra("extra_info")
            menu(name)
            prop(yesno)
        }
        %C_declaration
        ...
    }

    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")
        alias(alias_name)
        info(info_name, "value")
        ...
    }
    alias(record_name,alias_name)
```

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
function
variable
breaktable
record
grecord
info
alias
```

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 need to be quoted unless it contains one of the above keywords:

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

These are all legal characters for process variable names, although `.` is not allowed in a record name since it separates the record from the field name in a PV name. Thus in many cases quotes are not needed around record or field names in database files. Any string containing a macro does need to be quoted though.

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 a back-slash (`\`) as an escape character. For example `"\""` is a quoted string containing a single double-quote character.

6.3.4 Macro Substitution

Macro substitutions are permitted inside quoted strings. Macro instances take the form:

```
$(name)
```

or

```
${name}
```

There is no distinction between the use of parentheses or braces for delimiters, although the opening and closing characters must match for each macro instance. A macro name can be constructed using other macros, for example:

```
$(name_$(sel))
```

A macro instance can also provide a default value that is used when no macro with the given name has been defined. The default value can itself be defined in terms of other macros if desired, but may not contain any unescaped comma characters. The syntax for specifying a default value is as follows:

```
$(name=default)
```

Finally macro instances can also set the values of other macros which may (temporarily) override any existing values for those macros, but the new values are in scope only for the duration of the expansion of this particular macro instance. These definitions consist of `name=value` sequences separated by commas, for example:

```
$(abcd=$(a)$(b)$(c)$(d),a=A,b=B,c=C,d=D)
```

6.3.5 Escape Sequences

The database routines translate standard C escape sequences inside database field value strings only. The standard C escape sequences supported are:

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

`\ooo` represents an octal number with 1, 2, or 3 digits. `\xhh` represents a hexadecimal number which may have any number of hex digits, although only the last 2 will be represented in the character generated.

6.3.6 Comments

The comment symbol is "#". Whenever the comment symbol appears outside of a quoted string, it and all subsequent characters through the end of the line will be ignored.

6.3.7 Define before referencing

In general items cannot be referenced until they have been defined. For example a `device` definition cannot appear until the `recordtype` that it references has been defined or at least declared. Another example is that a record instance cannot appear until its associated record type has been defined.

One notable exception to this rule is that within a `recordtype` definition a menu field may reference a menu that has not been included directly by the record's `.dbd` file.

6.3.8 Multiple Definitions

If a menu, device, driver, or breakpoint table is defined more than once, then only the first instance will be used. Subsequent definitions may be compared to the first one and an error reported if they are different (the `dbdExpand.pl` program does this, the IOC currently does not). Record type definitions may only be loaded once; duplicates will cause an error even if the later definitions are identical to the first. However a record type declaration may be used in place of the record type definition in `.dbd` files that define device support for that type.

Record instance definitions are (normally) cumulative, so multiple instances of the same record may be loaded and each time a field value is encountered it replaces the previous value.

6.3.9 Filename Extensions

By convention:

- Record instances files have the extension "`.db`" or "`.vdb`" if the file also contains visual layout information
- Database definition files have the extension "`.dbd`"

6.4 Database Definition Statements

6.4.1 `path addpath` – Path Definition

6.4.1.1 Format

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

The path string follows the standard convention for the operating system, i.e. directory names are separated by a colon "`:`" on Unix and a semicolon "`;`" on Windows.

The `path` statement specifies the current search path for use when loading database and database definition files. The `addpath` statement appends directories to the current path. The path is used to locate the initial database file and included files. An empty path component at the beginning, middle, or end of a non-empty path string means search 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 that initial path.

The initial path is determined as follows:

1. If path is provided with the command, it is used. Else:
2. If the environment variable `EPICS_DB_INCLUDE_PATH` is defined, it is used. Else:
3. the path is "`.`", i.e. the current directory.

The search path is not used at all if the filename being searched for contains a `/` or `\` character. The first instance of the specified filename is used.

6.4.2 `include` – Include Statement

6.4.2.1 Format

```
include "filename"
```

An include statement can appear at any place shown in the summary. It uses the search path as described above to locate the named file.

6.4.3 `menu` – Menu Definition

6.4.3.1 Format

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

6.4.3.2 Definitions

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 used in the `enum` generated by `dbdToMenuH.pl` or `dbdToRecordtypeH.pl`. This must be a legal C/C++ identifier.

choice_string The text string associated with this particular choice.

6.4.3.3 Example

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

6.4.4 `recordtype` – Record Type Definition

6.4.4.1 Format

```
recordtype(record_type) {}

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

A record type statement that provides no field descriptions is a declaration, analagous to a function declaration (prototype) or forward definition in C. It allows the given record type name to be used in circumstances where the full record type definition is not needed.

6.4.4.2 Field Descriptor Rules

asl Sets the Access Security Level for the field. Access Security is discussed in chapter 8.

initial Provides an initial (default) value for the field.

promptgroup The group to which the field belongs, for database configuration tools.

prompt A prompt string for database configuration tools. Optional if `promptgroup` is not defined.

special If specified, special processing is required for this field at run time.

pp Whether a passive record should be processed when Channel Access writes to this field.

interest Interest level for the field.

base For integer fields, the number base to use when converting the field value to a string.

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.

prop Must be `YES` or `NO` (default). Indicates that the field holds Channel Access meta-data.

6.4.4.3 Definitions

record_type The unique name of the record type. Duplicate definitions are not allowed and will be rejected.

field_name The field name, which must be a valid C and C++ identifier. When include files are generated, the field name is converted to lower case for use as the record structure member name. If the lower-case version of the field name is a C or C++ keyword, the original name will be used for the structure member name instead. Previous versions of EPICS required the field name be a maximum of four all upper-case characters, but these restrictions no longer apply.

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`

as_level This must be one of the following values:

- `ASL0`
- `ASL1` (default value)

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.

init_value A legal value for data type.

prompt_value A prompt value for database configuration tools.

group_name A string used by database configuration tools (DCTs) to group related fields together.

A `promptgroup` should only be set for fields that can sensibly be configured in a record instance file.

The set of group names is no longer fixed. In earlier versions of Base the predefined set of choices beginning `GUI_` were the only group names permitted. Now the group name strings found in the database definition file

are collected and stored in a global list. The strings given for group names must match exactly for fields to be grouped together.

To support sorting and handling of groups, the names used in Base have the following conventions:

- Names start with a two-digit number followed by a space-dash-space sequence.
- Names are designed to be presented in ascending numerical order.
- The group name (or possibly just the part following the dash) may be displayed by the tool as a title for the group.
- In many-of-the-same-kind cases (e.g. 21 similar inputs) fields are distributed over multiple groups. Once-only fields appear in groups numbered in multiples of 5 or 10. The groups with the multiple instances follow in +1 increments. This allows more sophisticated treatment, e.g. showing the first group open and the other groups collapsed.

Record types may define their own group names. However, to improve consistency, records should use the following names from Base where possible. (This set also demonstrates that the group names used in different record types may share the same number.)

10 - Common	General fields that are common to all or many record types
20 - Scan	Scanning mechanism, priority and related properties
30 - Action	Record type specific behavior and processing action
40 - Link	Links and related properties
40 - Input	Input links and properties
50 - Output	Output links and properties
60 - Convert	Conversion between raw and engineering values
70 - Alarm	Alarm related properties, severities and thresholds
80 - Display	Client related configuration, strings, deadbands
90 - Simulate	Simulation mode related properties

NOTE: Older versions of Base contained a header file `guigroup.h` defining a fixed set of group names and their matching index numbers. That header file has been removed. The static database access library now provides functions to convert between group index keys and the associated group name strings. See 14.7.6 for details.

special_value Must be one of the following:

- `SPC_MOD` – Notify record support when modified. The record support `special` routine will be called whenever the field is modified by the database access routines.
- `SPC_NOMOD` – No external modifications allowed. This value disables external writes to the field, so it can only be set by the record or device support module.
- `SPC_DBADDR` – Use this if the record support's `cvt_dbaddr` routine should be called to adjust the field description when code outside of the record or device support makes a connection to the field.

  The following values are for database common fields. 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 values are deprecated, use `SPC_MOD` instead:

- An integer value greater than 103.
- 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:

- FALSE (default)
- TRUE

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.

%C_declaration A percent sign % inside the record body introduces a line of code that is to be included in the generated C header file.

6.4.4.4 Example

The following is the definition of the event record type:

```
recordtype(event) {
    include "dbCommon.dbd"
    field(VAL,DBF_STRING) {
        prompt("Event Name To Post")
        promptgroup("40 - Input")
        special(SPC_MOD)
        asl(ASL0)
        size(40)
    }
    field(EPVT, DBF_NOACCESS) {
        prompt("Event private")
        special(SPC_NOMOD)
        interest(4)
        extra("EVENTPVT epvt")
    }
    field(INP,DBF_INLINK) {
        prompt("Input Specification")
        promptgroup("40 - Input")
        interest(1)
    }
    field(SIOL,DBF_INLINK) {
        prompt("Sim Input Specifctn")
        promptgroup("90 - Simulate")
        interest(1)
    }
    field(SVAL,DBF_STRING) {
```

```
            prompt("Simulation Value")
            size(40)
        }
        field(SIML,DBF_INLINK) {
            prompt("Sim Mode Location")
            promptgroup("90 - Simulate")
            interest(1)
        }
        field(SIMM,DBF_MENU) {
            prompt("Simulation Mode")
            interest(1)
            menu(menuYesNo)
        }
        field(SIMS,DBF_MENU) {
            prompt("Sim mode Alarm Svrty")
            promptgroup("90 - Simulate")
            interest(2)
            menu(menuAlarmSevr)
        }
    }
```

6.4.5 `device` – Device Support Declaration

6.4.5.1 Format

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

6.4.5.2 Definitions

record_type Record type. The combination of `record_type` and `choice_string` must be unique. If the same combination appears more than once, only 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 name of the device support entry table for this device support.

choice_string The `DTYP` choice string for this device support. A `choice_string` value may be reused for different record types, but must be unique for each specific record type.

6.4.5.3 Examples

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

6.4.6 `driver` – Driver Declaration

6.4.6.1 Format

```
driver(drvet_name)
```

6.4.6.2 Definitions

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

6.4.6.3 Examples

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

6.4.7 `registrar` – Registrar Declaration

6.4.7.1 Format

```
registrar(function_name)
```

6.4.7.2 Definitions

function_name The name of an C function that accepts no arguments, returns `void` and has been marked in 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 Section 2.2, "Example IOC Application" on page 13 gives an example of how to register functions for subroutine records.

6.4.7.3 Example

```
registrar(myRegistrar)
```

6.4.8 `variable` – Variable Declaration

6.4.8.1 Format

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

6.4.8.2 Definitions

variable_name The name of a C variable which has been marked in its source file with an `epicsExportAddress` declaration.

type The C variable's type. If not present, `int` is assumed. Currently only `int` and `double` variables are supported.

This registers a diagnostic/configuration variable for device or driver support or a subroutine record subroutine so that the variable can be read and set with the iocsh `var` command (see Section 18.2.5 on page 261). The example application described in Section 2.2 on page 13 provides an example of how to register a debug variable for a subroutine record.

6.4.8.3 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)
```

6.4.9 `function` – Function Declaration

6.4.9.1 Format

```
function(function_name)
```

6.4.9.2 Definitions

function_name The name of a C function which is exported from its source file in an `epicsRegisterFunction` declaration.

This registers a function so that it can be found in the function registry for use by record types such as sub or aSub which refer to the function by name. The example application described in Section 2.2 on page 13 provides an example of how to register functions for a subroutine record.

6.4.9.3 Example

In an application C source file:

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

static long myFunction(void *argp) {
    /* my code ... */
}
epicsRegisterFunction(myFunction);
```

In an application database definition file:

```
function(myFunction)
```

6.4.10 `breaktable` – Breakpoint Table

6.4.10.1 Format

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

6.4.10.2 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.

6.4.10.3 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
}
```

6.4.11 `record` – Record Instance

6.4.11.1 Format

```
record(record_type, record_name) {
    alias(alias_name)
    field(field_name, "field_value")
    info(info_name, "info_value")
    ...
}
alias(record_name, alias_name)
```

6.4.11.2 Definitions

record_type The record type, or `"*"` (see discussion under record_name below).

record_name The record name. This must be composed out of only the following characters:

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

NOTE: If macro substitutions are used the name must be quoted.

Duplicate definitions are normally allowed for a record as long as the record type is the same. The last value given for each field is the value used. If the duplicate definitions are being used and the record has already been loaded, subsequent definitions may use `"*"` in place of the record type in the record instance.

The variable `dbRecordsOnceOnly` can be set to any non-zero value using the iocsh `var` command to make loading duplicate record definitions into the IOC illegal.

alias_name An alternate name for the record, following the same rules as the record name.

field_name A field name.

field_value A value for the named field, appropriate for its particular field type. When given inside double quotes the field value string may contain escaped characters which will be translated appropriately when loading the database. See section 6.3.5 for the list of escaped characters supported. Permitted values for the various field types are as follows:

- `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. Infinities or NaN are also allowed.
- `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`, `DBF_FWDLINK`
  NOTES:
  - If the field name is `INP` or `OUT` then this field is associated with `DTYP`, and the permitted values are determined by the link type of the device support selected by the current `DTYP` choice string. Other `DBF_INLINK` and `DBF_OUTLINK` fields must be either `CONSTANT` or `PV_LINK`s.
  - A device support that specifies a link type of `CONSTANT` can be given either a constant or a `PV_LINK`.

  The allowed values for the field depend on the device support's link type as follows:

  - `CONSTANT`
    A numeric literal, valid for the field type it is to be read into.
  - `PV_LINK`
    A value of the form:

    ```
    record.field process maximize
    ```

    `record` is the name of a record that exists in this or another IOC.

    The `.field`, `process`, and `maximize` parts are all 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 `DBF_INLINK` fields.

  `DBF_FWDLINK` fields can use `PP` or `CA`. If a `DBF_FWDLINK` is a channel access link it must reference the target record's `PROC` field.

`maximize` can have one of the following values:

- `NMS` – No Maximize Severity (Default)
- `MS` – Maximize Severity
- `MSS` – Maximize Severity and Status
- `MSI` – Maximize Severity if Invalid

- `VME_IO`
  `#Ccard Ssignal @parm`

  `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 specific (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` – optional device-specific character string (27 char max)

- `GPIB_IO`
  `#Llink Aaddr @parm`

  `link` – gpib link, i.e. interface
  `addr` – GPIB address
  `parm` – device-specific 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

- `BBGPIB_IO`
  `#Llink Bbbaddr Ggpibaddr @parm`

  `link` – link, i.e. vme bitbus interface
  `bbadddr` – bitbus address
  `gpibaddr` – gpib address
  `parm` – optional device-specific 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)

info_name The name of an Information Item related to this record. See section 6.5 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.

6.4.11.3 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")
    }
```

6.5 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.

6.6 Record Attributes

Each record type can have any number of record attributes. Each attribute is a psuedo field that can be accessed via database and channel access. Each attribute has a name that acts like a field name but returns the same value for all instances 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 *rtype, char *name, char *value);
```

The arguments are:

`rtype` – The name of recordtype.

`name` – The attribute name, i.e. the psuedo field name.

`value` – The value assigned to the attribute.

6.7 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 values 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 calls `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:

- Copy the `menuConvert.dbd` file from EPICS `base/src/ioc/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`:

```
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`.

6.8 Menu and Record Type Include File Generation.

6.8.1 Introduction

Given a file containing menu definitions, the program `dbdToMenuH.pl` generates a C/C++ header file for use by code which needs those menus. Given a file containing any combination of menu definitions and record type definitions, the program `dbdToRecordtypeH.pl` generates a C/C++ header file for use by any code which needs those menus and record type.

EPICS Base uses the following conventions for managing menu and recordtype definitions. Users generating local record types are encouraged to follow these.

- Each menu that is used by fields in database common (for example `menuScan`) or is of global use (for example `menuYesNo`) should be defined in its own 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 is defined in its own file. This file should also contain any menu definitions that are used only by that record type. Menus that are specific to one particular record type should use that record type name as a prefix to the menu name. The name of the file is the same as the record type, followed by `Record.dbd`. The name of the generated include file is the same as the `.dbd` file but with an extension of `.h`. Thus the record type `ao` 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 will 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`).

Note that developers don't normally execute the `dbdToMenuH.pl` or `dbdToRecordtypeH.pl` programs manually. If the proper naming conventions are used, it is only necessary to add definitions to the appropriate `Makefile`. Consult the chapter on the EPICS Build Facility for details.

6.8.2 dbdToMenuH.pl

This tool is executed as follows:

```
dbdToMenuH.pl [-D] [-I dir] [-o menu.h] menu.dbd [menu.h]
```

It reads in the input file `menu.dbd` and generates a C/C++ header file containing enumerated type definitions for the menus found in the input file.

Multiple `-I` options can be provided to specify directories that must be searched when looking for included files. If no output filename is specified with the `-o menu.h` option or as a final command-line parameter, then the output filename will be constructed from the input filename, replacing `.dbd` with `.h`.

The `-D` option causes the program to output Makefile dependency information for the output file to standard output, instead of actually performing the functions describe above.

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` that is generated contains:

```
/* menuPriority.h generated from menuPriority.dbd */

#ifndef INC_menuPriority_H
#define INC_menuPriority_H

typedef enum {
    menuPriorityLOW                 /* LOW */,
    menuPriorityMEDIUM              /* MEDIUM */,
    menuPriorityHIGH                /* HIGH */,
    menuPriority_NUM_CHOICES
} menuPriority;

#endif /* INC_menuPriority_H */
```

Any code that needs the priority menu values should include this file and make use of these definitions.

6.8.3 dbdToRecordtypeH.pl

This tool is executed as follows:

```
dbdTorecordtypeH.pl [-D] [-I dir] [-o xRecord.h] xRecord.dbd [xRecord.h]
```

It reads in the input file `xRecord.dhd` and generates a C/C++ header file which defines the in-memory structure of the given record type and provides other associated information for the compiler. If the input file contains any menu definitions, they will also be converted into enumerated type definitions in the output file.

Multiple `-I` options can be provided to specify directories that must be searched when looking for included files. If no output filename is specified with the `-o xRecord.h` option or as a final command-line parameter then the output filename will be constructed from the input filename, replacing `.dbd` with `.h`.

The `-D` option causes the program to output Makefile dependency information for the output file to standard output, instead of actually performing the functions describe above.

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")
        promptgroup("50 - Output")
        asl(ASL0)
        pp(TRUE)
    }
    field(OVAL,DBF_DOUBLE) {
        prompt("Output Value")
```

```
    }
    ... many more field definitions
}
```

The include file `aoRecord.h` that is generated contains:

```
/* aoRecord.h generated from aoRecord.dbd */

#ifndef INC_aoRecord_H
#define INC_aoRecord_H

#include "epicsTypes.h"
#include "link.h"
#include "epicsMutex.h"
#include "ellLib.h"
#include "epicsTime.h"

typedef enum {
    aoOIF_Full                      /* Full */,
    aoOIF_Incremental               /* Incremental */,
    aoOIF_NUM_CHOICES
} aoOIF;

typedef struct aoRecord {
    char                name[61];   /* Record Name */
    ... define remaining fields from database common
    epicsFloat64        val;        /* Desired Output */
    epicsFloat64        oval;       /* Output Value */
    ... define remaining record specific fields
} aoRecord;

typedef enum {
    aoRecordNAME = 0,
    aoRecordDESC = 1,
    ... indices for remaining fields in database common
    aoRecordVAL = 43,
    aoRecordOVAL = 44,
    ... indices for remaining record specific fields
} aoFieldIndex;

#ifdef GEN_SIZE_OFFSET

#ifdef __cplusplus
extern "C" {
#endif
#include <epicsExport.h>
static int aoRecordSizeOffset(dbRecordType *prt)
{
    aoRecord *prec = 0;
    prt->papFldDes[aoRecordNAME]->size = sizeof(prec->name);
    ... code to compute size for remaining fields
    prt->papFldDes[aoRecordNAME]->offset = (char *)&prec->name - (char *)prec;
    ... code to compute offset for remaining fields
    prt->rec_size = sizeof(*prec);
    return 0;
```

```
}
epicsExportRegistrar(aoRecordSizeOffset);

#ifdef __cplusplus
}
#endif
#endif /* GEN_SIZE_OFFSET */

#endif /* INC_aoRecord_H */
```

The analog output record support module and all associated device support modules should include this 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 provide values for the field associated with that menu.
- The `typedef struct` defining the record are used by record support and device support to access the fields in an analog output record.
- The next `enum` defines an index number 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 generated routine `aoRecordSizeOffset` is executed when the record type gets registered with an IOC. The routine is compiled with the record type code, and is marked static so it will not be visible outside of that file. The associate record support source code MUST include the generated header file only after defining the `GEN_SIZE_OFFSET` macro like this:

```
#define GEN_SIZE_OFFSET
#include "aoRecord.h"
#undef GEN_SIZE_OFFSET
```

This convention ensures that the routine is defined exactly once. The `epicsExportRegistrar` statement ensures that the record registration code can find and call the routine.

6.9 dbdExpand.pl

```
dbdExpand.pl [-D] [-I dir] [-S mac=sub] [-o out.dbd] in.dbd ...
```

This program reads and combines the database definition from all the input files, then writes a single output file containing all information from the input files. The output content differs from the input in that comment lines are removed, and all defined macros and include files are expanded. Unlike the previous `dbExpand` program, this program does not understand database instances and cannot be used with `.db` or `.vdb` files.

Multiple `-I` options can be provided to specify directories that must be searched when looking for included files. Multiple `-S` options are allowed for macro substitution, or multiple macros can be specified within a single option. If no output filename is specified with the `-o out.dbd` option then the output will go to stdout.

The `-D` option causes the program to output Makefile dependency information for the output file to standard output, instead of actually performing the functions describe above.

6.10 dbLoadDatabase

```
dbLoadDatabase(char *dbdfile, char *path, char *substitutions)
```

This IOC command loads a database file which may contain any of the Database Definitions described in this chapter. The `dbdfile` string may contain environment variable macros of the form `${MOTOR}` which will be expanded before the file is opened. Both the `path` and `substitutions` parameters can be null or empty, and are usually ommitted. Note that `dbLoadDatabase` should only used to load Database Definition (`.dbd`) files, although it is currently possible to use it for loading Record Instance (`.db`) files as well.

As each line of the file is read, the substitutions specified in `substitutions` are performed. Substitutions are specified as follows:

```
"var1=sub1,var2=sub3,..."
```

Variables are used in the file with the syntax `$(var)` or `${var}`. If the substitution string

```
"a=1,b=2,c=\"this is a test\""
```

were used, any variables `$(a)`, `$(b)`, `$(c)` in the database file would have the appropriate values substituted during parsing.

6.11 dbLoadRecords

```
dbLoadRecords(char* dbfile, char* substitutions)
```

This IOC command loads a file containing record instances, record aliases and/or breakpoint tables. The `dbfile` string may contain environment variable macros of the form `${MOTOR}` which will be expanded before the file is opened. The `substitutions` parameter can be null or empty, and is often ommitted. Note that `dbLoadRecords` should only used to load Record Instance (`.db`) files, although it is currently possible to use it for loading Database Definition (`.dbd`) files as well.

6.11.1 Example

For example, let the file `test.db` contain:

```
record(ai, "$(pre)testrec1")
record(ai, "$(pre)testrec2")
record(stringout, "$(pre)testrec3") {
    field(VAL, "$(STR)")
    field(SCAN, "$(SCAN)")
}
```

Then issuing the command:

```
dbLoadRecords("test.db", "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")
}
```

6.12 dbLoadTemplate

```
dbLoadTemplate(char *subfile, char *substitutions)
```

This IOC command reads a template substitutions file which provides instructions for loading database instance files and gives values for the `$(xxx)` macros they may contain. This command performs those substitutions while loading the database instances requested.

The `subfile` parameter gives the name of the template substitution file to be used. The optional `substitutions` parameter may contain additional global macro values, which can be overridden by values given within the substitution file.

The MSI program can be used to expand templates at build-time instead of using this command at run-time; both understand the same substitution file syntax.

6.12.1 Template File Syntax

The template substitution file syntax is described in the following Extended Backus-Naur Form grammar:

```
substitution-file ::= ( global-defs | template-subs )+

global-defs ::= 'global' '{' variable-defs? '}'

template-subs ::= template-filename '{' subs? '}'
template-filename ::= 'file' file-name
subs ::= pattern-subs | variable-subs

pattern-subs ::= 'pattern' '{' pattern-names? '}' pattern-defs?
pattern-names ::= ( variable-name ','? )+
pattern-defs ::= ( global-defs | ( '{' pattern-values? '}' ) )+
pattern-values ::= ( value ','? )+

variable-subs ::= ( global-defs | ( '{' variable-defs? '}' ) )+
variable-defs ::= ( variable-def ','? )+
variable-def ::= variable-name '=' value

variable-name ::= variable-name-start variable-name-char*
file-name ::= file-name-char+ | double-quoted-str | single-quoted-str
value ::= value-char+ | double-quoted-str | single-quoted-str

double-quoted-str ::= '"' (double-quoted-char | escaped-char)* '"'
single-quoted-str ::= "'" (single-quoted-char | escaped-char)* "'"
double-quoted-char ::= [^"\]
single-quoted-char ::= [^'\]
escaped-char ::= '\' .
```

```
value-char ::= [a-zA-Z0-9_+:;./\<>[] | '-' | ']'
variable-name-start ::= [a-zA-Z_]
variable-name-char ::= [a-zA-Z0-9_]
file-name-char ::= [a-zA-Z0-9_+:;./\] | '-'
```

Note that the current implementation may accept a wider range of characters for the last three definitions than those listed here, but future releases may restrict the characters to those given above.

Any record instance file names must appear inside quotation marks if the name contains any environment variable macros of the form `${ENV_VAR_NAME}`, which will be expanded before the named file is opened.

6.12.2 Template File Formats

Two different template formats are supported by the syntax rules given above. 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 `${VAR_NAME}` which must be expanded to generate the file name:

```
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.

6.12.3 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")
}
```

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 which are executed by `iocsh`. An example main program is:

```
int main(int argc,char *argv[])
{
    if (argc >= 2) {
        iocsh(argv[1]);
        epicsThreadSleep(.2);
    }
    iocsh(NULL);
    epicsExit(0)
    return 0;
}
```

The first call to `iocsh` executes commands from the startup script filename which must be passed as an argument to the program. The second call to `iocsh` with a `NULL` argument puts `iocsh` into interactive mode. This allows the user to issue the commands described in the chapter on "IOC Test Facilities" as well as some additional commands like `help`.

The command file passed is usually called the startup script, and contains statements like these:

```
< envPaths
cd ${TOP}
dbLoadDatabase "dbd/appname.dbd"
appname_registerRecordDeviceDriver pdbbase
dbLoadRecords "db/file.db", "macro=value"
cd ${TOP}/iocBoot/${IOC}
iocInit
```

The envPaths file is automatically generated in the IOC's boot directory and defines several environment variables that are useful later in the startup script. The definitions shown below are always provided; additional entries will be created for each support module referenced in the application's `configure/RELEASE` file:

```
epicsEnvSet("ARCH","linux-x86")
epicsEnvSet("IOC","iocname")
epicsEnvSet("TOP","/path/to/application")
epicsEnvSet("EPICS_BASE","/path/to/base")
```

7.2 Overview - vxWorks

After vxWorks is loaded at IOC boot time, commands like the following, normally placed in the vxWorks startup script, are issued to load and initialize the application code:

```
# Many vxWorks board support packages need the following:
#cd <full path to IOC boot directory>
< cdCommands
cd topbin
ld 0,0, "appname.munch"

cd top
dbLoadDatabase "dbd/appname.dbd"
appname_registerRecordDeviceDriver pdbbase
dbLoadRecords "db/file.db", "macro=value"

cd startup
iocInit
```

The `cdCommands` script is automatically generated in the IOC boot directory and defines several vxWorks global variables that allow `cd` commands to various locations, and also sets several environment variables. The definitions shown below are always provided; additional entries will be created for each support module referenced in the application's `configure/RELEASE` file:

```
startup = "/path/to/application/iocBoot/iocname"
putenv "ARCH=vxWorks-68040"
putenv "IOC=iocname"
top = "/path/to/application"
putenv "TOP=/path/to/application"
topbin = "/path/to/application/bin/vxWorks-68040"
epics_base = "/path/to/base"
putenv "EPICS_BASE=/path/to/base"
epics_basebin = "/path/to/base/bin/vxWorks-68040"
```

The `ld` command in the startup script loads EPICS core, the record, device and driver support the IOC needs, and any application specific modules that have been linked into it.

`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 and starts the IOC running.

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 is often similar or identical to the one used with vxWorks. The RTEMS startup code calls

```
epicsRtemsInitPreSetBootConfigFromNVRAM(struct rtems_bsdnet_config *);
```

just before setting the initialization parameters from non-volatile memory, and

```
epicsRtemsInitPostSetBootConfigFromNVRAM(struct rtems_bsdnet_config *);
```

just after setting the initialization parameters. An application may provide either or both of these routines to perform any custom initialization required. These function prototypes and some useful external variable declarations can be found in the header file `epicsRtemsInitHooks.h`

7.4 IOC Initialization

An IOC is normally started with the `iocInit` command as shown in the startup scripts above, which is actually implemented in two distinct parts. The first part can be run separately as the `iocBuild` command, which puts the IOC into a quiescent state without allowing the various internal threads it starts to actually run. From this state the second command `iocRun` can be used to bring it online very quickly. A running IOC can be quiesced using the `iocPause` command, which freezes all internal operations; at this point the `iocRun` command can restart it from where it left off, or the IOC can be shut down (exit the program, or reboot on vxWorks/RTEMS). Most device support and drivers have not yet been written with the possibility of pausing an IOC in mind though, so this feature may not be safe to use on an IOC which talks to external devices or software.

IOC initialization using the `iocBuild` and `iocRun` commands then consists of the following steps:

7.4.1 Configure Main Thread

Providing the IOC has not already been initialized, `initHookAtIocBuild` is announced first.

The main thread's `epicsThreadIsOkToBlock` flag is set, the message "`Starting iocInit`" is logged and `epicsSignalInstallSigHupIgnore` called, which on Unix architectures prevents the process from shutting down if it later receives a HUP signal.

At this point, `initHookAtBeginning` is announced.

7.4.2 General Purpose Modules

Calls `coreRelese` which prints a message showing which version of iocCore is being run.

Calls `taskwdInit` to start the task watchdog. This 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, and can also invoke callback routines registered by the task itself or by other software that is interested in the state of the IOC. See "Task Watchdog" on page 247 for details.

Starts the general purpose callback tasks by calling `callbackInit`. Three tasks are started at different scheduling priorities.

`initHookAfterCallbackInit` is announced.

7.4.3 Channel Access Links

Calls `dbCaLinkInit`. The initializes the module that handles database channel access links, but does not allow its task to run yet.

`initHookAfterCaLinkInit` is announced.

7.4.4 Driver Support

`initDrvSup` locates each device driver entry table and calls the `init` routine of each driver.

`initHookAfterInitDrvSup` is announced.

7.4.5 Record Support

`initRecSup` locates each record support entry table and calls the `init` routine for each record type.

`initHookAfterInitRecSup` is announced.

7.4.6 Device Support

`initDevSup` locates each device support entry table and calls its `init` routine specifying that this is the initial call.

`initHookAfterInitDevSup` is announced.

7.4.7 Database Records

`initDatabase` is called which makes three passes over the database performing the following functions:

1. Initializes the fields `RSET`, `RDES`, `MLOK`, `MLIS`, `PACT` and `DSET` for each record.

   Calls record support's `init_record` (first pass).
2. Convert each `PV_LINK` into a `DB_LINK` or `CA_LINK`

   Calls any extended device support's `add_record` routine.
3. Calls record support's `init_record` (second pass).

Finally it registers an `epicsAtExit` routine to shut down the database when the IOC application exits.

Next `dbLockInitRecords` is called to create the lock sets.

Then `dbBkptInit` is run to initialize the database debugging module.

`initHookAfterInitDatabase` is announced.

7.4.8 Device Support again

`initDevSup` locates each device support entry table and calls its `init` routine specifying that this is the final call.

`initHookAfterFinishDevSup` is announced.

7.4.9 Scanning and Access Security

The periodic, event, and I/O event scanners are initialized by calling `scanInit`, but the scan threads created are not allowed to process any records yet.

A call to `asInit` initailizes access security. If this reports failure, the IOC initialization is aborted.

`dbProcessNotifyInit` initializes support for process notification.

After a short delay to allow settling, `initHookAfterScanInit` is announced.

7.4.10 Initial Processing

`initialProcess` processes all records that have PINI set to YES.

`initHookAfterInitialProcess` is announced.

7.4.11 Channel Access Server

The Channel Access server is started by calling `rsrv_init`, but its tasks are not allowed to run so it does not announce its presence to the network yet.

`initHookAfterCaServerInit` is announced.

At this point, the IOC has been fully initialized but is still quiescent. `initHookAfterIocBuilt` is announced. If started using `iocBuild` this command completes here.

7.4.12 Enable Record Processing

If the `iocRun` command is used to bring the IOC out of its initial quiescent state, it starts here.

`initHookAtIocRun` is announced.

The routines `scanRun` and `dbCaRun` are called in turn to enable their associated tasks and set the global variable `interruptAccept` to `TRUE` (this now happens inside `scanRun`). Until this is set all I/O interrupts should have been ignored.

`initHookAfterDatabaseRunning` is announced. If the `iocRun` command (or `iocInit`) is being executed for the first time, `initHookAfterInterruptAccept` is announced.

7.4.13 Enable CA Server

The Channel Access server tasks are allowed to run by calling `rsrv_run`.

`initHookAfterCaServerRunning` is announced. If the IOC is starting for the first time, `initHookAtEnd` is announced.

A command completion message is logged, and `initHookAfterIocRunning` is announced.

7.5 Pausing an IOC

The command `iocPause` brings a running IOC to a quiescent state with all record processing frozen (other than possibly the completion of asynchronous I/O operations). A paused IOC may be able to be restarted using the `iocRun` command, but whether it will fully recover or not can depend on how long it has been quiescent and the status of any device drivers which have been running. The operations which make up the pause operation are as follows:

1. `initHookAtIocPause` is announced.
2. The Channel Access Server tasks are paused by calling `rsrv_pause`
3. `initHookAfterCaServerPaused` is announced.
4. The routines `dbCaPause` and `scanPause` are called to pause their associated tasks and set the global variable `interruptAccept` to `FALSE`.
5. `initHookAfterDatabasePaused` is announced.

6. After logging a pause message, `initHookAfterIocPaused` is announced.

7.6 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 `dbLoadDatabase` commands.

```
callbackSetQueueSize(size)
dbPvdTableSize(size)
scanOnceSetQueueSize(size)
errlogInit(buffersize)
errlogInit2(buffersize, maxMessageSize)
```

7.6.1 callbackSetQueueSize

Requests for the general putpose 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.

7.6.2 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 records) then a larger hash table size speeds up searches for records.

7.6.3 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. It should rarely be necessary to override this default. Normally the ring buffer overflow messages appear when the scanOnce task fails.

7.6.4 errlogInit or errlogInit2

These commands can increase (but not decrease) the default buffer and maximum message sizes for the errlog message queue. The default buffer size is 1280 bytes, the maximum message size defaults to 256 bytes.

7.7 initHooks

The inithooks facility allows application functions to be called at various states during ioc initialization. The states are defined in `initHooks.h`, which contains the following definitions:

```
typedef enum {
    initHookAtIocBuild = 0,         /* Start of iocBuild/iocInit commands */
    initHookAtBeginning,
    initHookAfterCallbackInit,
    initHookAfterCaLinkInit,
    initHookAfterInitDrvSup,
```

```
    initHookAfterInitRecSup,
    initHookAfterInitDevSup,
    initHookAfterInitDatabase,
    initHookAfterFinishDevSup,
    initHookAfterScanInit,
    initHookAfterInitialProcess,
    initHookAfterCaServerInit,
    initHookAfterIocBuilt,          /* End of iocBuild command */

    initHookAtIocRun,               /* Start of iocRun command */
    initHookAfterDatabaseRunning,
    initHookAfterCaServerRunning,
    initHookAfterIocRunning,        /* End of iocRun/iocInit commands */

    initHookAtIocPause,             /* Start of iocPause command */
    initHookAfterCaServerPaused,
    initHookAfterDatabasePaused,
    initHookAfterIocPaused,         /* End of iocPause command */

/* Deprecated states, provided for backwards compatibility.
 * These states are announced at the same point they were before,
 * but will not be repeated if the IOC gets paused and restarted.
 */
    initHookAfterInterruptAccept,   /* After initHookAfterDatabaseRunning */
    initHookAtEnd,                  /* Before initHookAfterIocRunning */
}initHookState;

typedef void (*initHookFunction)(initHookState state);
int initHookRegister(initHookFunction func);
const char *initHookName(int state);
```

Any functions that are registered before `iocInit` reaches the desired state will be called when it reaches that state. The `initHookName` function returns a static string representation of the state passed into it which is intended for printing. The following skeleton code shows how to use this 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

Various environment variables are used by iocCore:

```
EPICS_CA_ADDR_LIST
EPICS_CA_AUTO_ADDR_LIST
EPICS_CA_CONN_TMO
EPICS_CAS_BEACON_PERIOD
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_...` and `EPICS_CAS_...` variables see the EPICS Channel Access Reference Manual. For an explaination of the `EPICS_IOC_LOG_...` variables see "iocLogClient" on page 178 of this manual. `EPICS_TS_NTP_INET` is used only on vxWorks and RTEMS, where it sets the address 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 set through iocsh via the `epicsEnvSet` command, or on vxWorks using `putenv`. 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 just the environment variables used by iocCore.

`epicsEnvShow` – This shows all environment variables on your system.

7.9 Initialize Logging

Initialize the logging system. See the chapter on "IOC Error Logging" for details. The following can be used to direct the log client to use a specific host log server.

```
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:

Overview This section

Quick start A summary of the steps necessary to start access security.

User's Guide This explains what access security is and how to use it.

Design Summary Functional Requirements and Design Overview.

Application Programmer's Interface

Database Access Security Access Security features for EPICS IOC databases.

Channel Access Security Access Security features in Channel Access

Trapping Channel Access Writes This allows trapping of all writes from external channel access clients.

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 startup script must contain the following command before iocInit.

```
asSetFilename("/full/path/to/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, either `ASL0` or `ASL1`. The security level is defined in the 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 that network access is protected via normal networking and physical security methods.

No attempt has been made to protect against the sophisticated saboteur. Network and physical 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.

`#` introduces a comment line

The elements `<name>`, `<user>`, `<host>`, `<pvname>` and `<calculation>` can be given as 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. A user name may appear in more than one UAG. To match, a user name must be identical to the user name read by the CA client library running on the client machine. For vxWorks clients, the user name is usually 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. To match, a host name must match the host name read by the CA client library running on the client machine; both names are converted to lower case before comparison however. For vxWorks clients, the host name is usually taken from the target name of the boot parameters.
- 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 permissions. The TRAPWRITE setting used for a client is determined by the first WRITE rule that passes the rule checks.

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("/full/path/to/access/security/file.acf")
```

If this command is not used then access security will not be started by `iocInit`. If an error occurs when iocInit calls `asInit` than all access to the ioc is disabled, i.e. no channel access client will be able to access the ioc. Note that this command does not read the file itself, it just saves the argument string for use later on, nor does it save the current working directory, which is why the use of an absolute path-name for the file is recommended (a path name could be specified relative to the current directory at the time when `iocInit` is run, but this is not recommended if the IOC also loads the subroutine record support as a later reload of the file might happen after the current directory had been changed).

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 permissable 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:

1. When the value is found to be 1, `asInit` is called and the value set back to 0.
2. 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:lev1permit` 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:lev1permit)
    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:lev1permit)
    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 secuity 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 secururity 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(ASINPUTFUNPTR 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 initilization 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 `ASG`s.

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 *ppvt, const char *asgName);
```

This routine adds a new member to ASG `asgName`. The calling routine must provide storage for `ASMEMBERPVT`. Upon successful return `*ppvt` 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 *ppvt);
```

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 the `ASCLIENTPVT` pointer. `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` if the client has read access rights.

8.5.5.8 check Put

```
long asCheckPut(ASCLIENTPVT pvt);
```

This routine, actually a macro, returns `TRUE` if the client has write 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, to permit any registered listeners to be notified. The value returned by the call to `asTrapWriteBefore` is the `trapPvt` value that must subsequently be passed to the `asTrapWriteAfter` routine. 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 user 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. struct dbFldDes in dbBase.h contains a field as_level. In addition definitions are provided 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

struct 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 Database Access Library

Two files asDbLib.c and asCa.c implement the interface between IOC databases and access control. They contain the following routines:

8.6.3.1 Initialization

```
int asSetFilename(char *acf);
```

Calling this routine sets the filename of an access configuration file, but does not save the current working directory, so the use of an absolute pathname is strongly recommended. The next call to asInit uses this filename. asSetFilename must be called before iocInit otherwise access configuration is disabled. Is access security is disabled during iocInit it will never be turned on.

```
int asSetSubstitutions(char *substitutions);
```

This routine specifies macro substitutions for use while reading the configuration file.

```
int asInit();
int asInitAsyn(ASDBCALLBACK *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. After initialization all records in the database are made members of the appropriate access control group.

asInit is called by iocInit, and can also be called after iocInit 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.

If the caller provides an ASDBCALLBACK then when either initialization completes or taskwd detects a failure the user's 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.

8.6.3.2 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.

8.6.3.3 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.

8.6.3.4 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 asynchronously. When asInitTask calls the completion routine supplied by asSubProcess, the completion status is used to determine whether to place the record in alarm or not.

8.6.3.5 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 accidently 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 an exception message will be 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 registers a callback with `asRegisterClientCallback()` in order to receives asynchronous notification of access rights changes. 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, the server reenables 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, the 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 whenever the client library updates its knowledge of the access rights state. The application's callback function is installed using `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, 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, 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 monitor write requests and pass them to any software that registers a listener function. In order to use this facility three things are necessary:

1. The server using this library must call `asTrapWriteBefore()` and `asTrapWriteAfter()`. These routines are defined in asLib.h. The RSRV channel access server running 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 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 a `dbChannel *` describing the PV.

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.

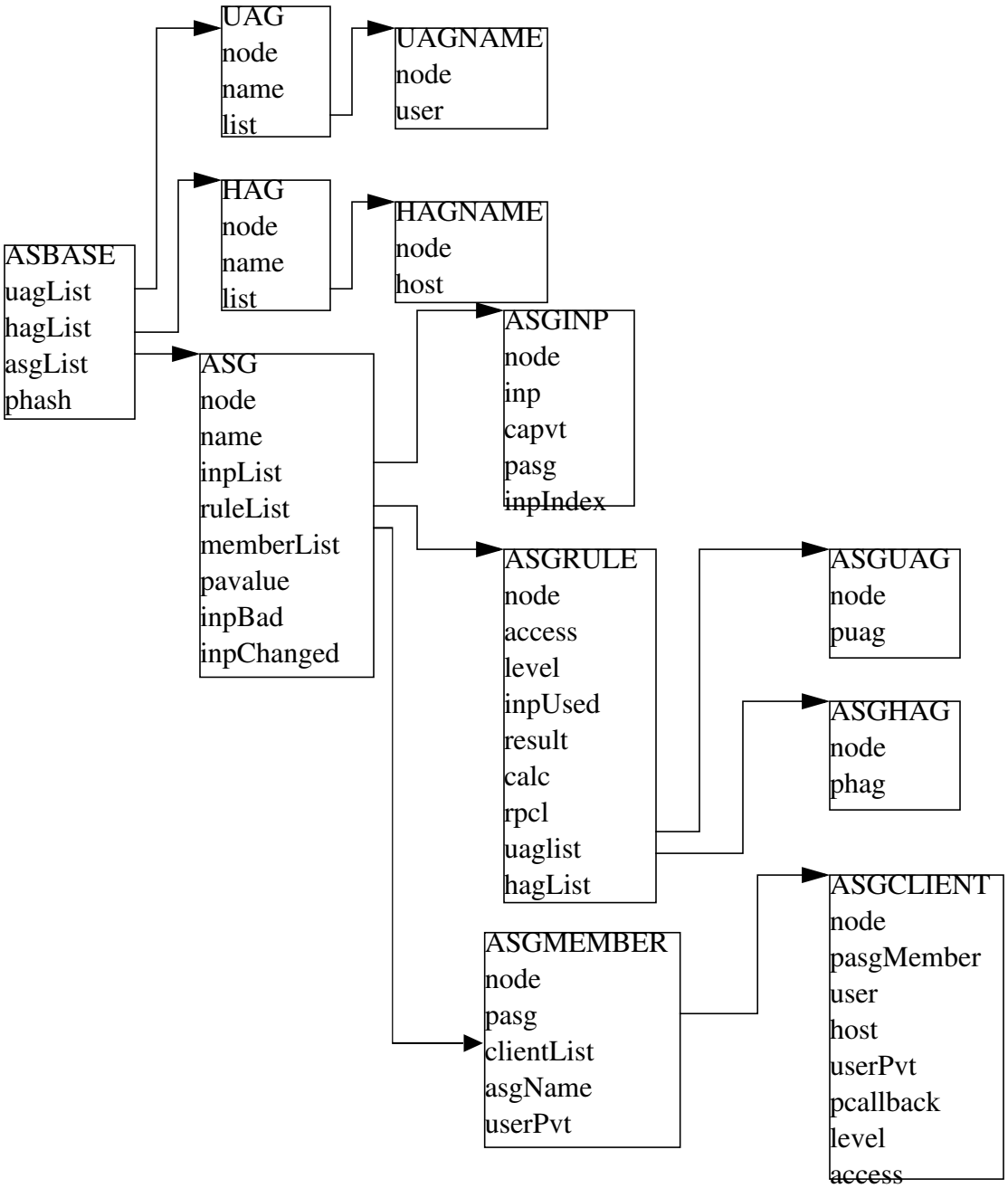
ascheck.c The Unix program which performs a syntax check on a configuration file.

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 from either iocsh or the vxWorks shell. In both shells the parentheses around arguments are optional. On vxWorks all character string arguments must be enclosed in double quote characters `""` and all arguments must be separated by commas. For iocsh single or double quotes must be used around string arguments that contain spaces or commas but are otherwise optional, and arguments may be separated by either commas or spaces. 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.

Both iosch and vxWorks shells allow output redirection, i.e. the standard output of any command can be redirected to a file. For example

```
dbl > dbl.lst
```

will send the output of the `dbl` command to the file `dbl.lst`

If iocsh is being used it provides help for all commands that have been registered. Just type

```
help
```

or

```
help pattern*
```

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 dbla

List Record Alias Names with optional pattern:

```
dbla
dbla("<pattern>")
```

Lists the names of all aliases (which match the pattern if given) and the records they refer to. Examples:

```
dbla
dbla "alia*"
```

9.2.4 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).

9.2.5 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.

9.2.6 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.

9.2.7 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.

9.2.8 dbtr

Test Record:

```
dbtr("<record_name>")
```

This calls dbNameToAddr, then dbProcess and finally dbpr (interest level 3). Its purpose is to test record processing.

9.2.9 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 Trace Processing

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.

9.5 Error Logging

9.5.1 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.

9.5.2 errlogInit, errlogInit2

Initialize error log client buffering

```
errlogInit(int bufSize)
errlogInit2(int bufSize, int maxMsgSize)
```

The error log client maintains a circular buffer of messages that are waiting to be sent to the log server. If not set using one or other of these routines the default value for bufSize is 1280 bytes and for maxMsgSize is 256 bytes.

9.5.3 errlog

Send a message to the log server

```
errlog("<message>")
```

This command is provided for use from the ioc shell only. It sends its string argument and a new-line to the log server, without displaying it on the IOC console. Note that the iocsh will have expanded any environment variable macros in the string (if it was double-quoted) before passing it to errlog.

9.6 Hardware Reports

9.6.1 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.

9.6.2 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.

9.7 Scan Reports

9.7.1 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.

9.7.2 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 nunber. If event_number is 0 all event scan lists are shown.

9.7.3 scanpiol

Print I/O Event Lists:

```
scanpiol
```

This routine prints a list of all records in the I/O event scan lists.

9.8 General Time

The built-in time providers depend on the IOC's target architecture, so some of the specific subsystem report commands listed below are only available on the architectures that use that particular provider.

9.8.1 generalTimeReport

Format:

```
generalTimeReport(int level)
```

This routine displays the time providers and their priority levels that have registered with the General Time subsystem for both current and event times. At level 1 it also shows the current time as obtained from each provider.

9.8.2 installLastResortEventProvider

Format:

```
installLastResortEventProvider
```

Installs the optional Last Resort event provider at priority 999, which returns the current time for every event number.

9.8.3 NTPTime_Report

Format:

```
NTPTime_Report(int level)
```

Only vxWorks and RTEMS targets use this time provider. The report displays the provider's synchronization state, and at interest level 1 it also gives the synchronization interval, when it last synchronized, the nominal and measured system tick rates, and on vxWorks the NTP server address.

9.8.4 NTPTime_Shutdown

Format:

```
NTPTime_Shutdown
```

On vxWorks and RTEMS this command shuts down the NTP time synchronization thread. With the thread shut down, the driver will no longer act as a current time provider.

9.8.5 ClockTime_Report

Format:

```
ClockTime_Report(int level)
```

This time provider is used on several target architectures, registered as the time provider of last resort. On vxWorks and RTEMS the report displays the synchronization state, when it last synchronized the system time with a higher priority provider, and the synchronization interval. On workstation operating systems the synchronization task is not started on the assumption that some other process is taking care of synchronzing the OS clock as appropriate, so the report is minimal.

9.8.6 ClockTime_Shutdown

Format:

```
ClockTime_Shutdown
```

Some sites may prefer to provide their own implementation of a system clock time provider to replace the built-in one. On vxWorks and RTEMS this command stops the OS Clock synchronization thread, allowing the OS clock to free-run. The time provider *will* continue to return the current system time after this command is used however.

9.9 Access Security Commands

9.9.1 asSetSubstitutions

Format:

```
asSetSubstitutions("substitutions")
```

Specifies macro substitutions used when access security is initialized.

9.9.2 asSetFilename

Format:

```
asSetFilename("<filename>")
```

This command defines a new access security file.

9.9.3 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.

9.9.4 asdbdump

Format:

```
asdbdump
```

This provides a complete dump of the access security database.

9.9.5 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.

9.9.6 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.

9.9.7 asprules

Format:

```
asprules("<access security group>")
```

Print the rules for the specified access security group or if no group is specified for all groups.

9.9.8 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.

9.10 Channel Access Reports

9.10.1 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.

9.10.2 dbel

Format:

```
dbel("<record_name>")
```

This routine prints the Channel Access event list for the specified record.

9.10.3 dbcar

Database to Channel Access Report - See “Record Link Reports”

9.10.4 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.

9.11 Interrupt Vectors

9.11.1 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

9.12 Miscellaneous

9.12.1 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`.

9.12.2 epicsEnvShow

Format:

```
epicsEnvShow("<name>")
```

Show Environment variables. On vxWorks it shows the variables created via calls to `putenv`.

9.12.3 coreRelease

Format:

```
coreRelease
```

Print release information for iocCore.

9.13 Database System Test Routines

These routines are normally only of interest to EPICS system developers NOT to Application Developers.

9.13.1 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.

9.13.2 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.

9.13.3 dbtpn

Test Process Notify:

```
dbtpn("<record_name.field_name>")
dbtpn("<record_name.field_name>","<value>")
```

Example:

```
dbtpn("aitest")
dbtpn("aitest","5.0")
```

This command performs a `dbProcessNotify` request. If a non-null value argument string is provided it issues a `putProcessRequest` to the named record; if no value is provided it issues a `processGetRequest`. This routine is mainly of interest to system developers for testing database access.

9.14 Record Link Reports

9.14.1 dblsr

Lock Set Report:

```
dblsr(<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.

9.14.2 dbLockShowLocked

Show locked locksets:

```
dbLockShowLocked(<level>)
```

This command generates a report showing all locked locksets, the records they contain, the lockset state and the thread that currently owns the lockset. The `level` argument is passed to `epicsMutexShow` to adjust the information reported about each locked epicsMutex.

9.14.3 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.

9.14.4 dbhcr

Report hardware links. See "Hardware Reports".

9.15 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.

9.15.1 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.

9.15.2 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.

9.15.3 tpn

Test Process Notify:

```
tpn("<record_name.field_name>","<value>")
```

Example:

```
tpn("aitest","5.0")
```

This routine tests the `dbProcessNotify` API when used via the old database access interface. It only supports issuing a `putProcessRequest` to the named record.

9.16 Routines to dump database information

9.16.1 dbDumpPath

Dump Path:

```
dbDumpPath(pdbbase)
```

Example:

```
dbDumpPath(pdbbase)
```

The current path for database includes is displayed.

9.16.2 dbDumpMenu

Dump Menu:

```
dbDumpMenu(pdbbase,"<menu>")
```

Example:

```
dbDumpMenu(pdbbase,"menuScan")
```

If the second argument is 0 then all menus are displayed.

9.16.3 dbDumpRecordType

Dump Record Description:

```
dbDumpRecordType(pdbbase,"<record type>")
```

Example:

```
dbDumpRecordType(pdbbase,"ai")
```

If the second argument is 0 then all descriptions of all records are displayed.

9.16.4 dbDumpField

Dump Field Description:

```
dbDumpField(pdbbase,"<record type>","<field name>")
```

Example:

```
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.

9.16.5 dbDumpDevice

Dump Device Support:

```
dbDumpDevice(pdbbase,"<record type>")
```

Example:

```
dbDumpDevice(pdbbase,"ai")
```

If the second argument is 0 then the device support for all record types is displayed.

9.16.6 dbDumpDriver

Dump Driver Support:

```
dbDumpDriver(pdbbase)
```

Example:

```
dbDumpDriver(pdbbase)
```

9.16.7 dbDumpRecord

Dump Record Instances:

```
dbDumpRecord(pdbbase,"<record type>",level)
```

Example:

```
dbDumpRecords(pdbbase,"ai")
```

If the second argument is 0 then the record instances for all record types are displayed. The third argument determines which fields are displayed just like for the command `dbpr`.

9.16.8 dbDumpBreaktable

Dump breakpoint table

```
dbDumpBreaktable(pdbbase,name)
```

Example:

```
dbDumpBreaktable(pdbbase,"typeKdegF")
```

This command dumps a breakpoint table. If the second argument is 0 all breakpoint tables are dumped.

9.16.9 dbPvdDump

Dump the Process variable Directory:

```
dbPvdDump(pdbbase,verbose)
```

Example:

```
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 recieve 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 their output is sent to the errlog task; unless configured not to, the output will appear on the console as well. Consult any book that describes the standard C library such as "The C Programming Language ANSI C Edition" by Kernighan and Ritchie.

`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 message storms this command can be used to suppress console messages. A argument of 0 suppresses the messages, any other value lets messages 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 means OK, non-0 means an error.
- The header files for most IOC subsystems 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.

IOC routines often return a long integer status value, encoded similar to the vxWorks error status encoding. On some modern architectures a long integer is more than 32 bits wide, but in order to keep the API compatible the status values are still passed as long integers, even though only 32 bits are ever used. The most significant 16 bits indicate the subsystem or module where the error occurred. The least significant 16 bits contain a subsystem-specific status value. In order that status values do not conflict with the vxWorks error status values, all subsystem numbers are greater than 500.

A header file `errMdef.h` defines macros for all the subsystem numbers. For example the database access routines use this module number:

```
#define M_dbAccess  (501 << 16)   /*Database Access Routines*/
```

There are header files 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/libCom/log/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. If the file grows larger than this limit the server will seek back to the beginning of the file and write new messages over the old messages starting from the beginning. 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 to log its messages it must have an environment variable `EPICS_IOC_LOG_INET` set 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.

10.7.4 iocLogPrefix

Many sites run multiple soft IOCs on the same machine. With some log viewers like cmlogviewer it is not possible to distinguish between the log messages from these IOCs since their hostnames are all the same. One solution to this is to add a unique prefix to every log message.

The `iocLogPrefix` command can be run from the startup file during IOC initialization to establish such a prefix that will be prepended to every log message when it is sent to the iocLogServer.

For example, adding the following lines to your `st.cmd` file

```
epicsEnvSet("IOC","sioc-b34-mc10");
iocLogPrefix("fac=LI21 proc=${IOC} ");
```

will categorize all log messages from this IOC as belonging to the facility `LI21` and to the process `sioc-b34-mc10`.

Note that log messages echoed to the IOC's standard output will not show the prefix, it only appears in the version sent to the log server. `iocLogPrefix` should appear fairly early in the startup script so the IOC doesn't try to send any log messages without the prefix. Once the prefix has been set, it cannot be changed without rebooting the IOC. One can determine if a log prefix has been set using `iocLogShow`.

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 is used. 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 its device support modules are the only source files that should include the record specific header files. Thus they will be 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 `struct rset` data structure declared in `recSup.h` and defined by the specific record type. This use of a record support vector table isolates the `iocCore` software from the implementation details of each record type. Thus new record types can be defined without having to modify 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:

```
struct 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 declared 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;
    longstatus;

    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;
    longstatus;
    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. Whevever 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 occured. 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

```
long report(void *precord);
```

This routine is not used by most record types. Any action is record type specific.

11.5.2 Initialize Record Processing

```
long 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

```
long init_record(void *precord, 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

```
long process(void *precord);
```

This routine must follow the guidelines specified previously.

11.5.5 Special Processing

```
long special(struct dbAddr *paddr, int after);
```

This routine implements the record type specific special processing for the field referred to by `dbAddr`. It is called twice when a field is written to from outside the record, once with after=0 before any changes are made to the field, and again with after=1 after the change has been made. The routine can prevent any changes from being made by returning an error status from the first call (after=0). 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 by 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

```
long 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 then 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

```
long 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

```
long 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

```
long 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

```
long 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

```
long 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

```
long 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

```
long 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

```
long get_graphic_double(struct dbAddr *paddr, struct dbr_grDouble *p);
```

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

```
long get_control_double(struct dbAddr *paddr, struct dbr_ctrlDouble  *p);
```

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

```
long get_alarm_double(struct dbAddr *paddr, struct dbr_alDouble *p);
```

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`". Code using these routines should

```
#include <recGbl.h>
```

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 then 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 occured 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.

```
int recGblSetSevr(void *precord,
    short nsta, short nsevr);
```

Returns `TRUE` if it changed `nsta` and/or `nsev`, `FALSE` if it did not change them.

```
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
- `ackt` - do transient alarm need to be acknowledged

These fields are handled by `iocCore` and `recGblResetAlarms` and should not be used by record support. The alarm acknowledgement facility it provided for use by alarm handlers.

11.6.3 Generate Error: Process Variable Name, Caller, Message

SUGGESTION: use `errlogPrintf` instead of this for new code.

```
void 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.

```
void recGblRecordError(
    long   status,
    void   *precord,
    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.

```
void 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

```
void 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

```
void 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

```
void 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

```
void 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

```
void 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 put in field TIME. The routine then returns.
- If TSEL is not a constant link dbGetLink is called and the value put in field TSE.
- If TSE is equal to `epicsTimeEventDeviceTime` (-2) then noting is done, i.e. the routine just returns.
- `epicsTimeGetEvent` is called.

11.6.11 Forward link

```
void 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.

11.6.13 Analog Value Deadband Check

```
void recGblCheckDeadband(
    epicsFloat64 *poldval,
    const epicsFloat64 newval,
    const epicsFloat64 deadband,
    unsigned *monitor_mask,
    const unsigned add_mask);
```

Check if analog (double) value is outside specified deadband, and set bits in monitor mask. This routine is usually called by an analog record's `monitor` (as part of processing) to check if a value is outside a predefined deadband. It also set bits in a monitor mask according to the check result. If `newval` lies outside the specified `deadband`, `newval` is copied into `*poldval`, and `add_mask` is OR'ed into `monitor_mask`.

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 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 then 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` to `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 directly calls the `process` entry of the record support module, which it locates via the `RSET` field in `dbCommon`, rather than calling `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) :
        pcallback = (CALLBACK *)(calloc(1,sizeof(CALLBACK)));
        callbackSetCallback(myCallback,pcallback);
        callbackSetUser(pai,pcallback);
        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. The `interest` value is provided to allow for different kinds of reports, or to control how much detail to display. If a device support module is using a driver, it may choose not 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 before and 1 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 chapter 17 for details.

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). 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 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) 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 form 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 inteface 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:

```
static long report(int level);
static long init(void);
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:

```
static long init(void)
{
    return(ab_driver_init());
}
```

13.2.0.2 report

The report routine is called by the `dbior` 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:

```
static 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 uses 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 modules 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 database. Static database access is described in this chapter, 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 menus, record types, and devices used to define that IOC database 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 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 *pdbbase;
```

NOTE: On an IOC pdbbase is a global variable, which is accessable if you include dbAccess.h

14.2.2 DBENTRY

A typical declaration for a database entry structure is:

```
DBENTRY *pdbentry;
pdbentry=dbAllocEntry(pdbbase);
```

Most static access to a database is via a DBENTRY structure. As many DBENTRYs 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 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.

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.

Most applications should not need to call this routine directly. The dbReadDatabase and dbReadDatabaseFP routines will call it automatically if pdbbase is null. Thus an application normally only has to contain code like the following:

```
DBBASE  *pdbbase=0;
...
status = dbReadDatabase(&pdbbase, dbdfile, search_path, macros);
```

However the static database access library does allow applications to work with multiple databases simultaneously, each referenced via a different DBBASE pointer. Such applications may need to call dbAllocBase directly.

14.3.2 dbFreeBase

```
void dbFreeBase(DBBASE *pdbbase);
```

dbFreeBase frees the entire database reference by pdbbase including the DBBASE structure itself.

14.4 DBENTRY Routines

14.4.1 Alloc/Free DBENTRY

```
DBENTRY *dbAllocEntry(DBBASE *pdbbase);
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 *pdbbase,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 *pdbbase)
{
    DBENTRY dbentry;
    DBENTRY *pdbentry = &dbentry;
    ...
    dbInitEntry(pdbbase,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 *pdbbase. 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 string and the other a FILE * pointer. Thus only one of each type will 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 to the named file.

`dbWriteDevice` writes the description of all devices to the named file.

`dbWriteDriver` writes the description of all drivers to the named file.

`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).

`dbWriteBreaktable` writes the definitions of all breakpoint tables to the named file.

14.5.3 Write Record Instances

```
long dbWriteRecord(DBBASE *pdbbase,char * file,

char *precordTypeName,int level);
long dbWriteRecordFP(DBBASE *pdbbase,FILE *fp,

char *precordTypeName,int level);
```

These routines write record instance data. 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 here all assume that DBENTRY references a record type, i.e. that `dbFindRecordType`, `dbFirstRecordType` or `dbNextRecordType` have 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 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);
char *dbGetPromptGroupNameFromKey(DBBASE *pdbbase, const short key);
short dbGetPromptGroupKeyFromName(DBBASE *pdbbase, const char *name);
```

The `dbGetPrompt` routine returns the character string prompt value, which provides a short description of the field. `dbGetPromptGroup` returns the field's group key.

Conversion between the group key and the group name as a string is provided by two functions: `dbGetPromptGroupNameFromKey` returns a pointer to a static string containing the name of the group, `NULL` for an invalid key. `dbGetPromptGroupKeyFromName` returns the numerical key related to the specified group name string, `0` if the string does not match an existing group name.

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 dbPutRecordAttribute

```
long dbPutRecordAttribute(DBENTRY *pdbentry, const char *name,
    const char *value);
```

This creates or modifies the attribute *name* of the record type referenced by *pdbentry* to *value*. Attribute names should be valid C identifiers, starting with a letter or underscore followed by any number of alphanumeric or underscore characters.

14.8.2 dbGetRecordAttribute

```
long dbGetRecordAttribute(DBENTRY *pdbentry, const char *name);
```

Looks up the attribute *name* of the record type referenced by *pdbentry* and sets the the field pointer in *pdbentry* to refer to this string if it exists. The routine `dbGetString` can be used subsequently to read the current attribute value.

14.9 Manipulating Record Instances

With the exception of `dbFindRecord`, each of the routines described in this section need DBENTRY to reference a valid record type, i.e. that `dbFindRecordType`, `dbFirstRecordType`, or `dbNextRecordType` have been called and returned success.

14.9.1 Get Number of Records

```
int  dbGetNRecords(DBENTRY *pdbentry);
```

Returns the total number of record instances and aliases for the record type that DBENTRY currently references.

14.9.2 Get Number of Record Aliases

```
int dbGetNAliases(DBENTRY *pdbentry)
```

Returns the number of record aliases for the record type that DBENTRY currently references.

14.9.3 Locate Record

```
long dbFindRecord(DBENTRY *pdbentry,char *precordName);
long dbFirstRecord(DBENTRY *pdbentry);
long dbNextRecord(DBENTRY *pdbentry);
```

These routines are used to locate record instances and aliases. If any of these routines returns success, then DBENTRY references a record or a record alias (use `dbIsAlias` to distinguish the two). `dbFindRecord` may be called without DBENTRY referencing a valid record type. `dbFirstRecord` only works if DBENTRY references a record type. The `dbDumpRecords` example given at the end 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` indicates whether the field was found or not. If it was not found, then `dbFindField` must be called before individual fields can be accessed.

14.9.4 Get Record Name

```
char *dbGetRecordName(DBENTRY *pdbentry);
```

This routine only works properly if called after `dbFindRecord`, `dbFirstRecord`, or `dbNextRecord` has returned success. If DBENTRY refers to an alias, the name returned is that of the alias, not of the record it refers to.

14.9.5 Distinguishing Record Aliases

```
int dbIsAlias(DBENTRY *pdbentry)
```

This routine only works properly if called after `dbFindRecord`, `dbFirstRecord`, or `dbNextRecord` has returned success. If DBENTRY refers to an alias it returns a non-zero value, otherwise it returns zero.

14.9.6 Create/Delete/Free Records and Aliases

```
long dbCreateRecord(DBENTRY *pdbentry,char *precordName);
long dbCreateAlias(DBENTRY *pdbentry, const char *paliasName);
long dbDeleteRecord(DBENTRY *pdbentry);
long dbDeleteAliases(DBENTRY *pdbentry);
long dbFreeRecords(DBBASE *pdbbase);
```

`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. `dbCreateAlias` assumes that DBENTRY references a particular record instance and creates an alias for that record. If it returns success, then DBENTRY references the alias just created. `dbDeleteRecord` deletes either a single alias, or a single record instance and all the aliases that refer to it. `dbDeleteAliases` finds and deletes all aliases that refer to the current record. `dbFreeRecords` deletes all record instances.

14.9.7 Copy Record

```
long dbCopyRecord(DBENTRY *pdbentry, char *newRecordName
    int overWriteOK)
```

This routine copies the record instance currently referenced by `DBENTRY` (it fails if DBENTRY references an alias). 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.

14.9.8 Rename Record

```
long dbRenameRecord(DBENTRY *pdbentry, char *newname)
```

This routine renames the record instance currently referenced by DBENTRY (it fails if DBENTRY references an alias). If dbRenameRecord completes successfully, DBENTRY references the renamed record.

14.9.9 Record Visibility

These routines are intended for use by graphical configuration tools.

```
long dbVisibleRecord(DBENTRY *pdbentry);
long dbInvisibleRecord(DBENTRY *pdbentry);
int dbIsVisibleRecord(DBENTRY *pdbentry);
```

Calling dbVisibleRecord makes the record referenced by DBENTRY visible. dbInvisibleRecord makes the record invisible. dbIsVisibleRecord returns TRUE if the record is visible, FALSE otherwise.

14.9.10 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 if no field with the given name could be found, TRUE if the field was located.

14.9.11 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 any database field type except DCT_NOACCESS. dbVerify returns NULL if the string contains a valid value for this field or an error message if not. Note that the strings returned are owned by the DBENTRY, so the next call passing that DBENTRY object that returns a string will overwrite the value returned by a previous call. 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 -1 is returned.

14.10.4 Locate Menu

```
dbMenu *dbFindMenu(DBBASE *pdbbase,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 section.

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.

14.11.4 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 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.

14.12.1 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.

14.12.2 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.

14.12.3 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 item's string value. When changing the string value using `dbPutInfoSting`, the character string provided will be copied, with additional memory being allocated as necessary. Developers 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.

14.12.4 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.

14.12.5 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`, pdbentry must refer to the item to be removed (using `dbFirstInfo`, `dbNextInfo` or `dbFindInfo`). The function returns 0 on success, or an error code.

14.12.6 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.

14.13 Find Breakpoint Table

```
brkTable *dbFindBrkTable(DBBASE *pdbbase,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.

14.14 Dump Routines

```
void dbDumpPath(DBBASE *pdbbase)
void dbDumpRecord(DBBASE *pdbbase,char *precordTypeName,int level);
void dbDumpMenu(DBBASE *pdbbase,char *menuName);
void dbDumpRecordType(DBBASE *pdbbase,char *recordTypeName);
void dbDumpField(DBBASE *pdbbase,char *recordTypeName,char *fname);
void dbDumpDevice(DBBASE *pdbbase,char *recordTypeName);
void dbDumpDriver(DBBASE *pdbbase);
void dbDumpRegistrar(DBBASE *pdbbase);
void dbDumpFunction(DBBASE *pdbbase);
```

```
void dbDumpVariable(DBBASE *pdbbase);
void dbDumpBreaktable(DBBASE *pdbbase,char *name);
void dbPvdDump(DBBASE *pdbbase,int verbose);
void dbReportDeviceConfig(DBBASE *pdbbase,FILE *report);
```

These routines are used to dump information about the database. The routines dbDumpRecord, dbDumpMenu, dbDumpDriver, dbDumpRegistrar and dbDumpVariable just call their corresponding dbWriteXxxFP routine, specifying stdout for the file to write to. dbDumpRecordType, dbDumpField, and dbDumpDevice give internal information useful on an ioc. These commands can be executed via iocsh, specifying pdbbase as the first argument.

14.15 Examples

14.15.1 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)
{
    longstatus;
    inti;
    intarg;

    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);
}
```

14.15.2 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");
        while(!status) {
            if (dbIsAlias(pdbentry)
                printf("\n␣␣Alias:%s\n",dbGetRecordName(pdbentry));
            else {
                printf("\n␣␣Record:%s\n",dbGetRecordName(pdbentry));
                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 several 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

15.2 Database Include Files

There are a number of database related include files. Of particular interest to this chapter are:

- dbDefs.h: Miscellaneous database related definitions.
- dbFldTypes.h: Field type definitions.
- dbAccess.h: Runtime database access definitions.
- dbServer.h: API for server implementers.
- 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_STRINGSZ: The number of characters reserved for the record name, including the terminating zero byte.
- PVNAME_SZ: The maximum number of characters allowed in the record name.
- DB_MAX_CHOICES: The maximum number of choices for a choice field.

Note that DB_MAX_CHOICES applies for code using the runtime routines documented in this chapter, but for Channel Access clients the maximum number of choices and their choice string length are different, and are defined in the db_access.h file.

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.

When first loaded the field is always creates as 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 dbServer.h

This header provides an optional API allowing the IOC to display information about the services that are connecting to and using the IOC.

15.2.5 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 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. Other services can similarly be layered on top of the IOC, using database access to manipulate record field values and cause records to be processed.

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 usual way to communicate with an IOC database from outside the IOC is via Channel Access, although other similar network server layers can be added. Most client applications should communicate with the database via Channel Access, not via database access, even if they reside in the same IOC process 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 enhanced in 1991, but Channel Access could not be changed to use the new interface as this would break existing client applications. Instead a module was written which translates the original database access API into the new API. This interface between the old and new style database access calls is not discussed in this manual.

The database access routines are:

- dbNameToAddr: Locate a database variable.
- dbGetField: Get values associated with a database variable (implicit locking).
- 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 (implicit locking).
- dbPutLink: Change value referenced by database link (Macro)
- dbPutLinkValue: Change value referenced by database link (Subroutine)
- dbPut: Routine called by `dbPutxxx` functions.
- dbProcessNotify: Process a record and request notification on completion, together with optional pre-process put or post-process get.
- dbNotifyCancel: Cancel a dbProcessNotify.
- 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.
- 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 the database field data types. DBR_ENUM is used for database fields that represent a set of choices or options. It is used for access to fields of type DBF_ENUM, DBF_DEVICE, and DBF_MENU. The complete set of database field types are defined in dbFldTypes.h. The DBR_PUT_ACKT and DBR_PUT_ACKS requests 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 holds the database request type that most closely matches the database field type. Using this request type will put the smallest load on the IOC.

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 means no, 1 means yes

DBR_PUT_ACKS - The highest alarm severity to acknowledge. If the current alarm severity is less then 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:

1. Call dbNameToAddr once and only once for each field to be accessed.
2. 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 then 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 an IOC process variable name is one of:

```
<record_name>
<record_name>.
<record_name>.<field_name>
<record_name>.<field_name><modifier>
<record_name>.<modifier>
```

For example the value field of a record with record named sample_name is:

"sample_name.VAL".

The *record_name* is case sensitive. The *field_name*s available depend on the record type of the record and usually consist of all upper-case letters. If omitted the field name VAL is used if it exists. Currently the only *modifier* supported is a single dollar sign $ and is only valid on fields which are strings or links. Additional modifiers may be added in future releases.

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,    /* ptr to returned data */
    long *options,    /* ptr to options */
    long *nRequest,   /* ptr to number of elements */
    void *pfl);       /* used by monitor routines */
```

This 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 *plink, /* ptr to link field */
    short dbrType,         /* DBR_xxx */
    void *pbuffer,         /* ptr to returned data */
    long *options,         /* ptr to options */
    long *nRequest);       /* ptr to number of elements desired */
```

NOTES:

- `options` can be NULL if no options are desired.
- `nRequest` can be NULL for a scalar.

`dbGetLink` is implemented as a macro that calls `dbGetLinkValue` and can reference its arguments more than once. 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,  /* ptr to returned data */
    long *options,  /* ptr to options */
    long *nRequest, /* ptr to number of elements */
    void *pfl);     /* used by monitor routines */
```

This routine retrieves the data referenced by `paddr` and converts it to the format specified by `dbrType`.

- options is a read/write field. On entry to dbGet, options holds the desired options. When dbGet returns, options gives the options actually honored. If an option is not honored, the corresponding fields in the 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 has been set to the actual number of elements returned. It is permissible to request zero elements. This is useful when only option data is desired.
- The pfl argument is for use by the Channel Access monitor routines. All other users must pass in 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 then 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, and provides the offset to the current beginning of the array data.

15.4.3 Put Routines

15.4.3.1 dbPutField

Change the value of a process variable, format:

```
long dbPutField(
    struct dbAddr *paddr,
    short dbrType,   /* DBR_xxx */
    void *pbuffer,  /* ptr to 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.

Note that dbPutField implicitly calls dbScanLock or (if the field being modified is a link field) dbScanLockMany. It must therefore not be called by a thread which has already called either of these functions. Call dbPut instead if the record is already locked.

15.4.3.2 dbPutLink and dbPutLinkValue

Change the value referenced by a database link, format:

```
long dbPutLink(
    struct db_link  *plink, /* ptr to link field */
    short dbrType,          /* DBR_xxx */
    void *pbuffer,          /* ptr to data to write */
    long nRequest);         /* number of elements to write */
```

dbPutLink is actually a macro that calls dbPutLinkValue and can reference its arguments more than once. 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,
    shortdbrType,/* DBR_xxx*/
    void*pbuffer,/*addr of data*/
    longnRequest);/*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 Process Notify Subsystem

15.4.4.1 Introduction

The Process Notify subsystem provides the following features:

1. Processes a record with notification of completion.
2. Put after the record is claimed but before before the process.
3. Get after the process but before the record is released.
4. put, then process, then get.

A process request will be issued if any of the following is true:

- The requester has issued a processs request and record is passive.

- The requester is doing a put, the record is passive, and either the field description is process passive or the field is PROC.
- The requester has requested a processGet or a putProcessGet request and the record is passive.

At most one process is performed per `dbProcessNotify` request.

15.4.4.2 dbNotify.h

The dbNotify.h header defines the following:

```
typedef enum {
    processRequest,
    putProcessRequest,
    processGetRequest,
    putProcessGetRequest
} notifyRequestType;

typedef enum {
    putDisabledType,
    putFieldType,
    putType
} notifyPutType;

typedef enum {
    getFieldType,
    getType
} notifyGetType;

typedef enum {
    notifyOK,
    notifyCanceled,
    notifyError,
    notifyPutDisabled
} notifyStatus;

typedef struct processNotify {
    /* Fields for private use by dbNotify implementation: */
    ellCheckNode restartNode;
    void *       pnotifyPvt;
    /* Fields set by dbNotify: */
    notifyStatus status;
    int          wasProcessed; /* (0,1) => (no,yes) */
    /*Fields set by user: */
    notifyRequestType requestType;
    struct dbAddr *paddr;      /* set by dbNameToAddr*/
    int         (*putCallback)(struct processNotify *,notifyPutType type);
    void        (*getCallback)(struct processNotify *,notifyGetType type);
    void        (*doneCallback)(struct processNotify *);
    void *        usrPvt;       /* for private use of user */
} processNotify;

epicsShareFunc void dbProcessNotify(processNotify *pprocessNotify);
epicsShareFunc void dbNotifyCancel(processNotify *pprocessNotify);
```

15.4.4.3 Client Semantics

The client must allocate an instance of `processNotify`, which can be used for an arbitrary number of calls to `dbProcessNotify`. Before calling `dbProcessNotify` the following fields must be given values:

- `requestType` - The request type.
- `paddr` - A struct dbAddr, which is given values by a call to `dbNameToAddr`.
- `putCallback` - If the request is a `putProcessRequest` or a `putProcessGetRequest` this must be given a value. It is called before the record is processed. This routine is expected to issue a database put. The return value should be (0, 1) if the callback operation (was not, was) successful.
- `getCallback` - If the request is a `processGetRequest` or a `putProcessGetrequest` this must be given a value. It is called after the record is processed but before the record is released. This routine is expected to issue a database get.
- `doneCallback` - This must be given a value. It is called after the record is processed and after the optional `getCallback`. This routine may issue a new `dbProcessNotify` if desired.
- `userPvt` - A field for the client and its callback routines to use as needed; this pointer is not used by the processNotify code.

The `notifyPutType` argument to `putCallback` is one of these values:

- `putDisabledType` - Puts are disabled. The client must not issue a put.
- `dbPutFieldType` - The client may issue a `dbPutField` request. This is returned when `paddr` refers to a link field. For link fields the record will never be processed as a result of the `dbProcessNotify`.
- `dbPutType` - The client can issue a dbPut request. The record may or may not be processed after the client callback returns.

The `notifyGetType` argument to `getCallback` will be one of these values:

- `getFieldType` - The client may issue a `dbGetField` request. This is returned when `paddr` refers to a link field. For link fields the record will never be processed as a result of the `dbProcessNotify`.
- `getType` - The client may issue a `dbGet` request.

The `notifyStatus` argument to `doneCallback` is one of these values:

- `notifyOK` - The `dbProcessNotify` request was successful and the record was processed.
- `notifyNoProcessOK` - The `dbProcessNotify` request was successful but the record was not processed.
- `notifyError` - An error occured.

Example code can be found in the routine `dbtpn` which is defined in `base/src/ioc/db/dbNotify.c`. It uses both `putProcesssRequest` and `processGetRequest`.

15.4.4.4 Callback Device Support

EPICS base provides soft device support that uses processNotify for both input and output record types. All use the device type name “Asyn Soft Channel”.

The input types issue a `processGetRequest`:

- `devAiSoftCallback` - Supports aiRecord.
- `devBiSoftCallback` - Supports biRecord.
- `devMbbiSoftCallback` - Supports mbbiRecord.
- `devMbbiDirectSoftCallback` - Supports mbbiDirectRecord.

- devLiSoftCallback - Supports longinRecord.
- devSiSoftCallback - Supports stringinRecord.

The output types issue a channel access ca_put_callback request.

- devAoSoftCallback - Supports aoRecord.
- devBoSoftCallback - Supports boRecord.
- devCalcoutSoftCallback - Supports calcoutRecord.
- devMbboSoftCallback - Supports mbboRecord.
- devMbboDirectSoftCallback - Supports mbboDirectRecord.
- devSiSoftCallback - Supports longoutRecord.

15.4.5 Utility Routines

15.4.5.1 dbBufferSize

Determine the buffer size for a dbGetField request, format:

```
long dbBufferSize(
    shortdbrType,/* DBR_xxx*/
    longoptions,/* options mask*/
    longnRequest);/* 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_xxx*/
```

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

This macro was provided in earlier versions of Base, but has been removed from 3.15 onward. Code that was using it to gain access to the internal components of the the link's dbAddr structure should be converted to make use of the other routines described in this chapter instead.

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` is `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 or some other method that calls `dbPutField`. Link field values cannot be modified using `dbPut` or `dbPutLink`. The following restrictions apply:

- The data type may be `DBR_STRING` or a nil-terminated array of DBR_CHAR or DBR_UCHAR characters
- If a link is being changed to a different hardware link type then the `DTYP` field must be set before the link field.
- The syntax for the string is exactly the same as described for link fields in chapter "Database Definition"

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 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.

15.7.1 Basic Routines

Except for dbCaPutLinkCallback, these routines are normally only called by database access, i.e. they are not called by record support modules.

15.7.1.1 dbCaLinkInit

Called by `iocInit` to initialize the `dbCa` library

```
void dbCaLinkInit(void);
```

15.7.1.2 dbCaAddLink

Add a new channel access link

```
void dbCaAddLink(struct link *plink);
```

15.7.1.3 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.

15.7.1.4 dbCaRemoveLink

Remove channel access link.

```
void dbCaRemoveLink(struct link *plink);
```

15.7.1.5 dbCaGetLink

Get link value

```
long dbCaGetLink(struct link *plink,short dbrType,
    void *pbuffer,unsigned short *psevr,long *nRequest);
```

15.7.1.6 dbCaPutLink

Put link value

```
long dbCaPutLink(struct link *plink,short dbrType,
    void *buffering nRequest);
```

15.7.1.7 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/std/dev/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 asynchrnous 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.

15.7.2 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.

15.7.2.1 dbCaIsLinkConnected

Is Channel Connected

```
int dbCaIsLinkConnected(struct link *plink)
```

This routine returns (TRUE, FALSE) if the link (is, is not) connected.

15.7.2.2 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.

15.7.2.3 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.

15.7.2.4 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.

15.7.2.5 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.

15.7.2.6 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.

15.7.2.7 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.

15.7.2.8 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.

15.7.2.9 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.

15.7.2.10 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.

15.7.2.11 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.

15.8 dbServer API

Software that provides external access to the IOC database is a server layer. It is helpful for the IOC developer to be able to see information about the servers that are making use of the database access routines for purposes of diagnosing problems and reporting usage statistics. The dbServer API is provided to allow the IOC code to display specific kinds of information from the servers connected to it, without having to integrate those servers into the IOC code. Server layers are thus encouraged to register a dbServer structure with the IOC database to allow this information to be retrieved as needed.

15.8.1 Registering a server layer

The dbServer structure is defined in dbServer.h as follows:

```
typedef struct dbServer {
    ELLNODE node;
    const char *name;

    /* Print level-dependent status report to stdout */
    void (* report) (unsigned level);

    /* Get number of channels and clients connected */
    void (* stats) (unsigned *channels, unsigned *clients);

    /* Get identity of client initiating the calling thread */
    /* Must return 0 (OK), or -1 (ERROR) from unknown threads */
    int (* client) (char *pBuf, size_t bufSize);
} dbServer;
```

A server layer should instantiate one of these and pass a pointer to it to dbRegisterServer:

```
void dbRegisterServer(dbServer *psrv);
```

The individual function pointers in the structure are optional, use NULL if a specific routine has not been implemented for this server. Additional function pointers may be added to the end of this structure in future releases, while aiming to keep API-compatibility with older versions. The functions provided by the server layer are used as follows.

- report: This routine should print a status report to stdout. Increasing interest levels should provide additional information.
- stats: This routine returns the current count of the number of channels and clients connected through this service.
- client: When called by one of the server's threads, this routine should fill in the buffer with a short string identifying the specific client (e.g. `user@host`), and return 0. If the calling thread does not belong to this server it should just return -1.

15.8.2 Interacting with Server layers

The dbServer.h header makes the following routines available to the IOC code.

15.8.2.1 dbsr - Server Report

```
void dbsr(unsigned level);
```

This routine scans through the list of registered servers, printing the server's name and then calling its report function if one exists. This is an iocsh command that is intended to replace `casr`, and is only called on demand by the user.

15.8.2.2 dbServerClient - Identifying a client thread

```
int dbServerClient(char *pBuf, size_t bufSize);
```

When the IOC processes a record that has its TPRO field set, this routine is called to obtain a server context for the printed record name. It iterates through all of the registered servers in turn calling their `client()` routines until one of them returns OK or the end of the list is reached. If no server returns OK the routine returns -1.

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 then `printf` calls. Another problem with vxWorks is the `logMsg` facility. `logMsg` generates messages at higher priority then 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 sets of general purpose IOC callback tasks. The only difference between the task sets is their scheduling priority: low, medium or high. The low priority tasks runs at a priority just higher than Channel Access, the medium priority tasks at a priority about equal to the median of the periodic scan tasks, and the high priority tasks at a priority higher than the event scan task. The callback tasks are available for any software component that needs a task under which to run some job either immediately or after some delay. Jobs can also be cancelled during their delay period. The callback tasks register themselves with the task watchdog (described below). They are created with a generous amount of stack space and can thus be used for invoking record processing. For example the I/O event scanner uses the general purpose callback tasks.

The number of general purpose threads per priority level is configurable. On SMP systems with multi-core CPUs, the throughput can be improved and the latency (time between job scheduling and processing) can be lowered by running

multiple parallel callback tasks, which the OS scheduler may assign to different CPU cores. Parallel callback tasks must be explicitly enabled (see 16.2.5 below), as this feature is disabled by default for compatibility reasons.

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. Make calls (in most cases these are actually macros) to initialize the fields in the `CALLBACK`:

```
callbackSetCallback(CALLBACKFUNC func, CALLBACK *pcb);
```

This defines the callback routine to be executed. The first argument is the address of a function that will be given the address of the `CALLBACK` and returns `void`. The second argument is the address of the `CALLBACK` structure.

```
callbackSetPriority(int, CALLBACK *pcb);
```

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 *pcb);
```

This call is used to save a pointer value that can be retrieved again using the macro:

```
callbackGetUser(void *,CALLBACK *pcb);
```

If your callback function exists to process a single record inside calls to `dbScanLock`/`dbScanUnlock`, you can use this shortcut which provides the callback routine for you and sets the other two parameters at the same time (the user parameter here is a pointer to the record instance):

```
callbackSetProcess(CALLBACK *pcb, int prio, void *prec);
```

4. Whenever a callback request is desired just call one of the following:

```
int callbackRequest(CALLBACK *pcb);
int callbackRequestProcessCallback(CALLBACK *pcb, int prio, void *prec);
```

Both 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. The second routine is a shortcut for calling both `callbackSetProcess` and `callbackRequest`. Both return zero in case of success, or an error code (see below).

The following delayed versions wait for the given time before queueing the callback routine for the relevant thread set to execute.

```
callbackRequestDelayed(CALLBACK *pCallback, double seconds);
callbackRequestProcessCallbackDelayed(CALLBACK *pCallback,
int Priority, void *pRec, double seconds);
```

These routines cannot be called from interrupt level code.

16.2.2 Syntax

The following calls are provided:

Notes:

```
void callbackInit(void);
void callbackShutdown(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 callbackSetProcess(CALLBACK *pcallback, int Priority, void *prec);

int callbackRequest(CALLBACK *);
int callbackRequestProcessCallback(
    CALLBACK *pCallback, int Priority, void *prec);
void callbackRequestDelayed(CALLBACK *pCallback, double seconds);
void callbackRequestProcessCallbackDelayed(
    CALLBACK *pCallback, int Priority, void *prec, double seconds);
void callbackCancelDelayed(CALLBACK *pcallback);
int callbackSetQueueSize(int size);
```

- `callbackInit` and `callbackShutdown` are performed automatically at IOC initialization or shutdown, thus user code never calls these functions.
- `callbackSetCallback`, `callbackSetPriority`, `callbackSetUser`, and `callbackGetUser` are actually macros.
- Both `callbackRequest` and `callbackRequestProcessCallback` may be called from interrupt context. Both return zero for success, or one of the following error codes: `S_db_notInit` for a NULL callback pointer, `S_db_badChoice` for an illegal priority value, or `S_db_bufFull` when the associated queue is full.
- The delayed versions of the `callbackRequest` routines wait the given time before queueing the callback.
- `callbackCancelDelayed` can be used to cancel a delayed callback.
- `callbackRequestProcessCallback` issues the calls:

  ```
  callbackSetCallback(ProcessCallback, pCallback);
  callbackSetPriority(Priority, pCallback);
  callbackSetUser(pRec, pCallback);
  callbackRequest(pCallback);
  ```

  The routine `ProcessCallback` was designed for asynchronous device completion and is defined as:

  ```
  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",&pmyStruct->begid[0],

    &pmStruct->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 for each callback priority into a separate ring buffer. These buffers can by default hold up to 2000 requests. This limit can be changed by calling `callbackSetQueueSize` before `iocInit` in the startup file. The syntax is:

```
int callbackSetQueueSize(int size)
```

16.2.5 Parallel Callback Tasks

To enable multiple parallel callback tasks, and set the number of tasks to be started for each priority level, call `callbackParallelThreads` before `iocInit` in the startup file. The syntax is:

```
int callbackParallelThreads(int count, const char *prio)
```

The count argument is the number of tasks to start, with 0 indicating to use the default (number of CPUs), and negative numbers indicating to use the number of CPUs minus the specified amount.

The prio argument specifies the priority level, with "" (empty string), "*", or NULL indicating to apply the definition to all priority levels.

The default value is stored in the variable `callbackParallelThreadsDefault` (initialized to the number of CPUs), which can be changed using the iocShell's `var` command.

16.3 Task Watchdog

EPICS provides a task that acts as a watchdog for other tasks. Any task can request to be watched, and most of the IOC tasks do this. A status monitoring subsystem in the IOC can register to be notified about any changes that occur. The watchdog task runs periodically and checks each task in its task list. If any task is suspended, an error message is displayed and any notifications made. The task watchdog provides the following features:

1. Include module:

   ```
   #include <taskwd.h>
   ```

2. Request by a task to be monitored:

   ```
   taskwdInsert (epicsThreadId tid, TASKWDFUNC callback, VOID *usr);
   ```

   This adds the task with the specified `tid` to the list of tasks to be watched, and makes any requested notifications that a new task has been registered. If `tid` is given as zero, the `epicsThreadId` of the calling thread is used instead. If `callback` is not NULL and the task later becomes suspended, the callback routine will be called with the single argument `usr`.

3. Remove task from list:

   ```
   taskwdRemove(epicsThreadId tid);
   ```

   This routine must be called before the monitored task exits. It makes any requested notifications and removes the task from the list of tasks being watched. If `tid` is given as zero, the `epicsThreadId` of the calling thread is used instead.

4. Request to be notified of changes:

   ```
   typedef struct {
       void (*insert)(void *usr, epicsThreadId tid);
       void (*notify)(void *usr, epicsThreadId tid, int suspended);
       void (*remove)(void *usr, epicsThreadId tid);
   } taskwdMonitor;

   taskwdMonitorAdd(const taskwdMonitor *funcs, void *usr);
   ```

   This call provides a set of callbacks for the task watchdog to call when a task is registered or removed or when any task gets suspended. The `usr` pointer given at registration is passed to the callback routine along with the `tid` of the thread the notification is about. In many cases the `insert` and `remove` callbacks will be called from the context of the thread itself, although this is not guaranteed (the registration could be made by a parent

thread for instance). The `notify` callback also indicates whether the task went into or out of suspension; it is called in both cases, unlike the callbacks registered with `taskwdInsert` and `taskwdAnyInsert`.

5. Rescind notification request:

```
taskwdMonitorDel(const taskwdMonitor *funcs, void *usr);
```

This call removes a previously registered notification. Both `funcs` and `usr` must match the values given to `taskwdMonitorAdd` when originally registered.

6. Print a report:

```
taskwdShow(int level);
```

If `level` is zero, the number of tasks and monitors registered is displayed. For higher values the registered task names and their current states are also shown in tabular form.

7. The following routines are provided for backwards compatibility purposes, but are now deprecated:

```
taskwdAnyInsert(void *key, TASKWDANYFUNC callback, VOID *usr);
```

The callback routine will be called whenever any of the tasks being monitored by the task watchdog become suspended. `key` must have a unique value because the task watchdog system uses this value to determine which entry to remove when `taskwdAnyRemove` is called.

```
taskwdAnyRemove(void *key);
```

`key` is the same 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 standard time intervals are supported and additional periods can be added.
- Event: Event scanning is based on the posting of a named or numbered event by another component of the software.
- 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 (which 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 set from within a database configuration tool. It is quite permissible however to change any of these 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 with the following choices:

`Passive` - Passively scanned.

`Event` - Event Scanned. The field `EVNT` specifies the event name or number.

`I/O Intr` - I/O Event scanned.

`10 Second` - Periodically scanned every 10 seconds

...

`.1 Second` - Periodically scanned every .1 seconds

17.2.2 PHAS - Scan Phase

This 16-bit integer field determines relative processing order for records that are in the same scan set. For example all records periodically scanned at a 2 second rate belong to the same scan set. All Event scanned records with the same `EVNT` belong to 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 - Named or Numbered Events

This field is only used when `SCAN` is set to `Event`, when `EVNT` specifies the associated database event name or number. For named events the `EVNT` field should be set to the event name. Event names are compared using `strcmp()`, so case and leading/trailing spaces must all match. To use the numeric event trigger routine `post_event()` the `EVNT` field must hold an integer in the range 1...255.

17.2.4 PRIO - Scheduling Priority

This field can be used by any software component that needs to specify a scheduling priority. The Event and I/O event scan types use this field.

17.3 Scan Related Software Components

17.3.1 menuScan.dbd

This file holds the definition of the menu used by the field `SCAN`. The default definition is:

```
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 in the order and location shown. The remaining definitions are for the periodic scan rates, which should appear in the order slowest to fastest (the order directly controls the thread priority assigned to the particular scan rate, and faster scan rates should be assigned higher thread priorities). At IOC initialization, the menu choice strings are read while the scan system is being initialized. The number of periodic scan rates and the

period of each rate is determined from the menu choice strings. Thus periodic scan rates can be changed by copying `menuScan.dbd` into the IOC's build directory and modifying the set of choices defined therein. The choice names such as `menuScan10_second` are not used in this case, but must still be unique. Each periodic choice string must begin with a number and be followed by any of the following unit strings:

`second` or `seconds`

`minute` or `minutes`

`hour` or `hours`

`Hz` or `Hertz`

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:

```
#define SCAN_PASSIVE       menuScanPassive
#define SCAN_EVENT         menuScanEvent
#define SCAN_IO_EVENT      menuScanI_O_Intr
#define SCAN_1ST_PERIODIC  (menuScanI_O_Intr + 1)

typedef struct ioscan_head *IOSCANPVT;
typedef struct event_list *EVENTPVT;
typedef void (*io_scan_complete)(void *usr, IOSCANPVT, int prio);
typedef void (*once_complete)(void *usr, struct dbCommon*);

long scanInit(void);
void scanRun(void);
void scanPause(void);
void scanShutdown(void);

EVENTPVT eventNameToHandle(const char* event);
void postEvent(EVENTPVT epvt) EPICS_DEPRECATED;
void post_event(int event);
void scanAdd(struct dbCommon *);
void scanDelete(struct dbCommon *);
double scanPeriod(int scan);
void scanOnce(struct dbCommon *precord);
int scanOnceCallback(struct dbCommon *, once_complete cb, void *usr);
int scanOnceSetQueueSize(int size);

/*print periodic lists*/
epicsShareFunc int scanppl(double rate);

/*print event lists*/
epicsShareFunc int scanpel(const char *event_name);

/*print io_event list*/
epicsShareFunc int scanpiol(void);

void scanIoInit(IOSCANPVT *ppios);
unsigned int scanIoImmediate(IOSCANPVT pios, int prio);
unsigned int scanIoRequest(IOSCANPVT pios);
```

```
void scanIoSetComplete(IOSCANPVT, io_scan_complete, void *usr);
```

The first set of definitions defines the various scan types. The typedefs are used when interfacing with the routines below. The remaining definitions declare the public scan access routines. These are described in the following subsections.

17.3.3 Initializing And Controlling Database Scaning

```
long scanInit(void);
```

The routine `scanInit` is called by `iocInit`. It initializes the scanning system.

```
void scanRun(void);
void scanPause(void);
void scanShutdown(void);
```

These routines start, pause and stop all the scan tasks respectively. They are used by the `iocInit`, `iocRun`, `iocPause` and `iocShutdown` commands.

17.3.4 Adding And Deleting Records From Scan List

The following routines are called each time a record is added to 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 into the correct scan list. The routine `dbPut` calls `scanDelete` and `scanAdd` each time a scan-related field is changed (scan-related fields are 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 the index into the set of enum choices from menuScan.dbd. Most users will pick the value from the `SCAN` field of a database record. The routine returns the scan period in seconds. The result will be 0.0 if scan doesn't refer to a periodic scan rate.

17.3.6 Declaring and Triggering Database Events

Any software component may declare and subsequently trigger a database event. Database events used to be numbered with 8-bit integers and did not have to be declared in advance. Since Base 3.15 though events can now be named, in which case they must be declared to convert the name into an event object.

```
EVENTPVT eventNameToHandle(const char* event);
```

This routine must be called from task context (i.e. not from an interrupt service routine) to convert an event's name into an associated `EVENTPVT` handle. The first time each name is seen a handle will be created for it; subsequent calls to `eventNameToHandle` with the same name will return the same handle.

A database event is triggered by calling one of:

```
void postEvent(EVENTPVT eventObj);
void post_event(int eventNum) EPICS_DEPRECATED;
```

The original integer `post_event` routine is now deprecated in favor of the new routine `postEvent` that takes an event handle instead of the event number. These event-posting routines may be called by virtually any IOC software component, including from an interrupt service routine on VxWorks or RTEMS. For example sequence programs can call them. The record support module for the `eventRecord` calls `postEvent`.

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 I/O event source the following must be done:
   (a) Declare an `IOSCANPVT` variable, e.g.

   ```
   static IOSCANPVT ioscanpvt;
   ```

   (b) Call `scanIoInit` during initialization, e.g.

   ```
   scanIoInit(&ioscanpvt);
   ```

3. Provide the device support `get_ioint_info` routine. This routine has the prototype:

   ```
   long get_ioint_info(int cmd, struct dbCommon *precord,
       IOSCANPVT *ppvt);
   ```

   This routine will be called each time the record pointed to by `precord` is added to or deleted from an I/O Event scan list. The `cmd` argument will be zero if the record is being added to an I/O event list, 1 if it is being deleted from the list. This routine must set `*ppvt` to the `IOSCANPVT` variable associated with this record.

4. Whenever an I/O event is detected, the device software must call `scanIoRequest` or `scanIoImmediate`, e.g.

   ```
   scanIoRequest(ioscanpvt);
   scanIoImmediate(ioscanpvt, priorityLow);
   ```

   The routine `scanIoRequest()` may be safely called from interrupt level. A request is queued and will be handled by one of the standard callback threads. There are three sets of callback threads fed from three queues, one for each priority level (see section 16.2); the `PRIO` field of a record determines which queue will be used for processing this record after `scanIoRequest()` has been called.

   The routine `scanIoImmediate()` may not be called from interrupt level. Instead of queuing a request, this routine directly processes records on the current thread. Unlike `scanIoRequest`, this routine only scans records with the given priority level. It must therefore be called three times, once for each priority level.

   `scanIoRequest()` returns a bit pattern indicating which priority queues the request was added to. A return value of zero means that no records are currently configured to use this interrupt source for I/O Interrupt scanning.

5. Device or driver support that needs to implement flow control can set up a completion callback by calling `scanIoSetComplete`, e.g.

   ```
   static void myCallback(void *arg, IOSCANPVT pvt, int prio) {
       ...
   }

   scanIoSetComplete(ioscanpvt, myCallback, (void *)arg);
   ```

The completion callback will be run from one of the callback threads, once per priority actually used (bits set in the return value of `scanIoRequest`), after the list of records with that priority level has been processed. Note that for records with asynchronous device support, record processing might not have completed when the callback is run.

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`. This section gives an overview of how this code is organized.

17.4.1 Definitions And Routines Common To All Scan Types

Everything is built around two basic structures:

```
typedef struct scan_list {
    epicsMutexId lock;
    ELLLIST      list;
    short        modified;
};

typedef struct scan_element{
    ELLNODE         node;
    scan_list       *pscan_list;
    struct dbCommon *precord;
}
```

Each `scan_list.list` is the head of a list of `scan_element` nodes pointing to records that all belong to the same scan set. For example, all records that are periodically scanned at the 1 second rate are in the same scan set. The libCom `ellLib` routines are used to access the list. The `scan_element.node` field contains the next and previous links. Each record that appears in a `scan_list` has an associated `scan_element`. The `SPVT` field which appears in `dbCommon` points to 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 when 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 Database Event Scanning

Event scanning is built around the following definitions:

```
typedef struct event_list {
    CALLBACK           callback[NUM_CALLBACK_PRIORITIES];
    scan_list          scan_list[NUM_CALLBACK_PRIORITIES];
    struct event_list *next;
```

```
    char                event_name[MAX_STRING_SIZE];
} event_list;

static event_list * volatile pevent_list[256];
static epicsMutexId event_lock;
```

Event scanning uses the general purpose callback tasks to perform record processing, i.e. no extra threads are spawned for this. When a named event is declared by a call to `eventNameToHandle()` an `event_list` will be created for that named event. Every `event_list` contains a `scan_list` for each of the 3 priorities. The `next` member is used to keep a singly-linked list of all the `event_list` objects, with the first item on that list pointed to by `pevent_list[0]`. `pevent_list` is an array of pointers to numbered `event_list` objects, and is used when an event name is an integer in the range 1..255. It provides fast access to 255 numbered events, i.e. one for each possible numeric database event.

17.4.2.1 postEvent

```
void postEvent(event_list *pel);
```

This routine is called to request an event scan for a named event handle. It may be called from interrupt level. It looks at each `scan_list` in the `event_list` (one for each callback priority) and if any nodes are present in the list it makes a `callbackRequest` to process that set of records. The appropriate callback task calls routine `eventCallback`, which just calls `scanList`.

17.4.2.2 post_event

```
void post_event(int eventNum) EPICS_DEPRECATED;
```

This routine is called to request an event scan for a numbered event. It may be called from interrupt level. It looks up the `event_list` indicated by the given event number and calls `postEvent` with that handle.

17.4.3 I/O Event Scanning

I/O event scanning is built around the following definitions:

```
typedef struct io_scan_list {
    CALLBACK callback;
    scan_list scan_list;
} io_scan_list;

typedef struct ioscan_head {
    struct ioscan_head *next;
    struct io_scan_list iosl[NUM_CALLBACK_PRIORITIES];
    io_scan_complete cb;
    void *arg;
} ioscan_head;

static ioscan_head *pioscan_list = NULL;
static epicsMutexId ioscan_lock;
```

I/O event scanning uses the general purpose callback tasks to perform record processing, i.e. no extra threads are spawned for this. 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

```
void scanIoInit(IOSCANPVT *ppios)
```

This routine is called by device or driver support. It must be called once for each interrupt source. `scanIoInit` allocates and initializes an `ioscan_head` object which contains an `io_scan_list` for each callback priority. It puts the address of the allocated object in `ppios`.

When `scanAdd` or `scanDelete` are called, they call the device support routine `get_ioint_info` which returns `ppios`. The `scan_element` is then added to or deleted from the correct scan list.

17.4.3.2 scanIoRequest

```
unsigned int scanIoRequest(IOSCANPVT pios)
```

This routine is called by device or driver support to request a specific I/O event scan. It may be called from interrupt level. It looks at each `io_scan_list` referenced by `pios` (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 `ioscanCallback`, which calls `scanList` followed by any completion callback that was registered with `pios`.

The `scan_element` is then added to or deleted from the correct scan list.

17.4.3.3 scanIoImmediate

```
unsigned int scanIoImmediate(IOSCANPVT pios, int prio);
```

Scans any records in the given IOSCANPVT with the given priority. Record processing is done using the current thread. This is intended to allow device or driver support to implement private scanning threads. However links in these records may result in other records also being processed using the same thread.

Such device or driver support should call `scanIoImmediate` for all priority levels. For maximum throughput these calls can be made concurrently.

17.4.4 Periodic Scanning

Periodic scanning is built around the following definitions:

```
typedef struct periodic_scan_list {
    scan_list           scan_list;
    double              period;
    const char          *name;
    unsigned long       overruns;
    volatile enum ctl   scanCtl;
    epicsEventId        loopEvent;
} periodic_scan_list;

static int nPeriodic;
static periodic_scan_list **papPeriodic;
static epicsThreadId *periodicTaskId;
```

The `nPeriodic` variable holds the number of periodic scan rates configured. `papPeriodic` points to an array of pointers to `periodic_scan_lists`. There is an array element for each scan rate. A periodic scan task is created for each scan rate.

The following routines implement periodic scanning:

17.4.4.1 initPeriodic

```
void initPeriodic(void);
```

This routine first determines how many periodic scan rates are to be created from the definition of the `menuScan` menu. The array of pointers referenced by `papPeriodic` is allocated. For menu choice a `periodic_scan_list` is allocated and initialized. It parses the choice string for that choice to obtain the scan period for the scan.

17.4.4.2 periodicTask

```
periodicTask (struct scan_list *psl);
```

In outline this task runs an infinite loop, calling `scanList` and then waiting until the start of the next scan interval, allowing for the time it took to scan the list. If a periodic scan list takes longer to process than its defined scan period, its next scan will be delayed by half a scan period, with a maximum of 1 second delay. This does not limit what scan rates can actually be implemented, as long as all the records in the list can be processed within the requested period. Persistent over-runs (more than 10 times in a row) will result in a warning message being logged. The total number of over-runs is counted by each scan thread and can be displayed using the `scanppl` command.

17.4.5 Scan Once

17.4.5.1 scanOnce

```
typedef void (*once_complete)(void *usr, struct dbCommon*);
int scanOnce (dbCommon *precord);
int scanOnceCallback(struct dbCommon *, once_complete cb, void *usr)
```

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 may be called from interrupt level.

The `scanOnceCallback` variant also takes a callback function and user pointer; the completion function is invoked from the `onceTask` after the record has been processed.

These functions return zero when a request is successfull queued, and a non-zero error code if a request can't be queued.

17.4.5.2 SetQueueSize

`scanOnce` places its requests into a ring buffer. This is set by default to be 1000 entries long. The size can be changed by executing the following command in the startup script before `iocInit`:

```
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 expansion

Lines of input not beginning with a comment character (#) are searched for macro references in the form ${name} or $(name). The documentation for the macLib facility (chapter 19) describes some possible syntax variations for macro references. Such references are replaced with the value of the environment variable they name before any other processing takes place. Macro 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 substitution of the following variables, which would otherwise be performed before the `epicsEnvSet` command was run.

Macro references that appear inside single-quotes are not expanded.

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 takes place after the macro expansion described above has been performed, and cannot 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 (including backslashes) within the quotes. A single quote may occur between double quotes and a double quote may occur between single quotes. Note that commands called from the shell may perform additional unescaping and macro expansion on their argument strings.

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 capabilties 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 library routines 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 Definition) and chapter 9 (IOC Test Facilities) among others. The commands described in the sequencer documentation will also be recognized if the sequencer is included.

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 this displays a list of all commands.
#	A '#' as the first non-whitespace character on a line marks the beginning of a comment, which continues to the end of the line (some older versions of Base may require a space after the '#' character to properly recognize it as a comment). If the '#' character is immediately followed by a '-', the commented line will not be echoed with the IOC shell output.
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 directory	Change working directory to directory.
pwd	Print the name of the working directory.
var [name [value]]	If both arguments are present, assign the value to the named variable.If only the name 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 variable keyword as described in Section6.9 on page104.
show [-level] [task ...]	Show information about specified tasks. If no task arguments are present, show information on all tasks. The level argument controls the amount of information printed. The default level is 0. The task arguments can be task names or task i.d. numbers.
system command_string	Send command_string to the system command interpreter for execution. This command is present only if some application database definition file contains registrar(iocshSystemCommand) and if the system provides a suitable command interpreter (vxWorks does not).
epicsEnvSet name value	Set environment variable name to the specified value.
epicsEnvShow [name]	If no name is specified the names and values of all environment variables will be shown. If a name 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 sec	Pause execution of IOC shell for sec 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_HISTSIZE	Number of previous command lines to remember. If the IOCSH_HISTSIZE 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.

18.2.7 Conditionals

The IOC shell does not provide operaters for conditionally executing commands but the effect can be simulated using macro expansion. The simplest technique is to preceed a command with a macro that expands to either '#' or '' (or ' '). The following startup script line shows how this can be done:

```
...
$(LOAD_DEBUG=#) $(DEBUG) dbLoadRecords("db/debugRec.db", "P=$(P),R=debug")
...
```

Starting the IOC in the normal fashion will result in the above line being commented out and the debugRec.db file being omitted:

```
./st.cmd
```

Setting the `LOAD_DEBUG` environment variable to an empty string before starting the IOC will result in the debugRec.db file being loaded:

```
LOAD_DEBUG="" ./st.cmd
```

A similar technique can be used to execute external scripts conditionally. The startup command file contains code like:

```
epicsEnvSet PILATUS_ENABLED "$(PILATUS_ENABLED=NO)"
...
< pilatus-$(PILATUS_ENABLED).cmd
```

with one set of conditional code in a file named pilatus-YES.cmd and the other set of conditional code in a file named pilatus-NO.cmd This technique can be expanded to a form similar to a C 'switch' statement for the example above by providing additional pilatus-*XXX*.cmd scripts.

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 prototypes for calling the IOC shell command interpreter are:

```
int iocsh(const char *pathname);
int iocshLoad(const char *pathname, const char *macros);
int iocshCmd(const char *cmd);
int iocshRun(const char *cmd, const char *macros);
```

The pathname argument to the `iocsh` function 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, using

```
iocsh "script"
```

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. The `iocshLoad` function is an extension of the `iocsh` command that takes an additional string consisting of a set of macro definitions. This invokation of the IOC shell will then treat these macros as additional environment variables during execution, but will not persist after the shell exits.

The `iocshCmd` function takes a single IOC shell command and executes it. The `iocshRun` function executes the command with additional macro replacement defined by the user in the second parameter. These functions may be called from any thread, but many of the commands are not necessarily thread-safe so this should only be used with care. These functions are most useful to execute a single IOC shell command from a vxWorks startup script or command line, like this:

```
iocshCmd "iocsh command string"
iocshRun "iocsh command string" "VAR=VAL"
```

The stdio stream redirection and environment variable expansion processes described above are performed on the string as part of the execution process.

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
iocshArgInt	The argument will be converted to an integer value.
iocshArgDouble	The argument will be converted to a double-precision floating point value.
iocshArgString	The argument will be left as a string. The memory used to hold the string is 'owned' by iocsh and will be reused once the handler function returns.
iocshArgPersistentString	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 free().
iocshArgPdbbase	The argument must be pdbbase.
iocshArgArgv	An arbitrary number of arguments is expected. Subsequent iocshArg 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
iocshArgInt	int	args[i].ival
iocshArgDouble	double	args[i].dval
iocshArgString	char *	args[i].sval
iocshArgPersistentString	char *	args[i].sval
iocshArgPdbbase	void *	args[i].vval
iocshArgArgv	int	args[i].aval.ac
	char **	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 Registrar Command Registration

Commands are normally registered with the IOC shell in a registrar function. The application's 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 its commands with the iocsh.

The following code fragments shows how this can be performed for an example driver.

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

```
/* drvXxx code, FuncDef and CallFunc definitions ... */

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.

18.3.4 Automatic Command Registration

A C++ static constructor can also 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);
}

static int doRegister(void)
{
    iocshRegister(epicsThreadSleepFuncDef, epicsThreadSleepCallFunc);
    return 1;
}
static int done = doRegister();
```

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 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 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 if only the other is needed. The least signficant 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. 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 will be 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 or the keyword `VAL` which refers to the previous result of this calculation. 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, and `VAL` refers to the the VAL field (which can be overwritten from outside the record via Channel Access or a database link).

19.2.1.4 Variable Assignment Operator

Recently added is the ability to assign the result of a sub-expression to any of the single letter variables, which can then be used in another sub-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 sub-expression that is not an assignment, which can appear anywhere in the string. Sub-expressions within the string are separated by a semi-colon character.

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, while 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 parameters inside parentheses. The parameter seperator is a comma.

- Absolute value: `abs(a)`
- Exponential e^a: `exp(a)`
- Logarithm, base 10: `log(a)`
- Natural logarithm (base e): `ln(a)` *or* `loge(a)`
- *n* parameter maximum value: `max(a, b, ...)`
- *n* parameter 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, so 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 any non-numeric values: `isnan(a, ...)`
- Test for all finite, numeric values: `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. Unlike in C though, ^ 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 precence 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 requirea 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.

19.3.1 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 through C++ template containers which are not used in EPICS. If all you need from there is `std::min()`, `std::max()` and/or `std::swap()` your code may compile faster if you include `epicsAlgorithm.h` and use `epicsMin()`, `epicsMax()` and `epicsSwap()` instead.

Template Function	Meaning
epicsMin(a, b)	Returns the smaller of a or b compared using a<b. Handles NaNs correctly.
epicsMax(a, b)	Returns the larger of a or b compared using a<b. Handles NaNa correctly.
epicsSwap(a, b)	Swaps the values of a and b; the data type must support both copy-construction and assignment.

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` routines, which also provides a pointer 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 implement `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 ending normally, i.e. when the thread function returns.

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 various C++ template headers and APIs:

- `epicsGuard.h` - Mutex guard classes
- `epicsSingleton.h` - Single object enforcement
- `resourceLib.h` - Hash tables
- `tsDLList.h` - Double Linked Lists
- `tsFreeList.h` - Free List for efficient new/delete
- `tsSLList.h` - Single Linked Lists

Currently these are only being used by Channel Access Clients and the portable Channel Access Server.

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 repeatedly call malloc() followed soon afterwards by a call to free() the temporarily allocated storage.
- Between those calls to malloc() and free(), an additional call to malloc() is made that does NOT have an associated free().

In some environments, e.g. vxWorks 5.x, 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
dbmfInit()	Initialize the facility. Each time malloc() must be called size*chunkItems bytes are allocated. size is the maximum size request from dbmfMalloc() that will be allocated from the dbmf pool. If dbmfInit() was not called before one of the other routines then it is automatically called with size=64 and chuckItems=10.
dbmfMalloc()	Allocate memory. If bytes is > size then malloc() is used to allocate the memory.
dbmfFree()	Free the memory allocated by dbmfMalloc().
dbmfFreeChunks()	Free all chunks that have contain only free items.
dbmfShow()	Show the status of the dbmf 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 void (*FREEFUNC)(void *);

typedef struct ELLLIST {
  ELLNODE  node;
  int   count;
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 ellFree2 (ELLLIST *pList, FREEFUNC freeFunc);
void ellFree (ELLLIST *pList);    // Use only if freeFunc is free()
void ellVerify (ELLLIST *pList);
```

19.9 epicsRingBytes

`epicsRingBytes.h` describes a C facility for a commonly used type of ring buffer.

19.9.1 C interface

EpicsRingBytes provides methods for creating and using ring buffers (first in first out circular buffers) that store bytes. The unlocked variant is designed so that one writer thread and one reader thread can access the ring simultaneously without requiring mutual exclusion. The locked variant uses an epicsSpinLock, and works with any numbers of writer and reader threads.

```
epicsRingBytesId epicsRingBytesCreate(int nbytes);
epicsRingBytesId epicsRingBytesLockedCreate(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 nbytes. The returned epicsRingBytesId is passed to the other ring methods.
epicsRingBytesLockedCreate()	Same as epicsRingBytesCreate, but create the spin lock secured variant of the ring buffer.
epicsRingBytesDelete()	Delete the ring buffer and free any associated memory.
epicsRingBytesGet()	Move up to nbytes from the ring buffer to value. The number of bytes actually moved is returned.
epicsRingBytesPut()	Move nbytes from value to the ring buffer if there is enough free space available to hold them. The number of bytes actually moved is returned, which will be zero if insufficient space exists.
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., nbytes 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.

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. The unlocked variant is designed so that one writer thread and one reader thread can access the ring simultaneously without requiring mutual exclusion. The locked variant uses an epicsSpinLock, and works with any numbers of writer and reader threads.

```
template <class T>
class epicsRingPointer {
public:
    epicsRingPointer(int size, bool locked);
    ~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. If locked is true, the spin lock secured variant is created.
~epicsRingPointer()	Destructor.

push()	Push a new entry on the ring. It returns (false,true) is (failure, success). Failure means the ring was full.
pop()	Take a element off the ring. It returns 0 (null) if the ring was empty.
flush()	Remove all elements from the ring. If this operation is performed on a ring buffer of the unsecured variant, 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 size 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);
epicsRingPointerId epicsRingPointerLockedCreate(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.

epicsRingPointerCreate() creates the unsecured variant, epicsRingPointerLockedCreate() creates the spin lock secured variant of the ring buffer.

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 `return(noRestart);` A periodic timer returns with a statement like `return(restart,10.0);` 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 and 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. This method should not be called from within a C++ static constructor, since the queue thread requires that a current time provider be available and the last-resort time provider is not guaranteed to have been registered until all constructors have run. 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 then 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*/
} GPHENTRY;

struct gphPvt;

/*tableSize must be power of 2 in range 256 to 65536*/
void gphInitPvt(struct gphPvt **ppvt, int tableSize);
GPHENTRY *gphFind(struct gphPvt *pvt, const char *name, void *pvtid);
GPHENTRY *gphAdd(struct gphPvt *pvt, const char *name, void *pvtid);
void gphDelete(struct gphPvt *pvt, const char *name, void *pvtid);
void gphFreeMem(struct gphPvt *pvt);
void gphDump(struct gphPvt *pvt);
void gphDumpFP(FILE *fp, struct gphPvt *pvt);
```

where

pvt - For internal use by the gpHash library. Caller must provide storage for a `struct gphPvt *pvt`

name - The character string that will be hashed and added to table.

pvtid - The name plus the 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.

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 using the environment variables `EPICS_IOC_LOG_INET` and `EPICS_IOC_LOG_PORT` as inputs to `logClientCreate` and also registers the 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 ussue, 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 */
);

/*Expand string using environment variables alongside macro definitions */
epicsShareFunc char *         /* expanded string; NULL if any undefined macros */
epicsShareAPI macDefExpand(
```

```
        const char *str,            /* string to be expanded */
        MAC_HANDLE *macros          /* opaque handle; can be NULL if default */
                                    /* special characters are to be used */
    );
```

NOTE: The directory <base>/src/libCom/macLib contains two files `macLibNOTES` and `macLibREADME` that explain this library.

19.17 epicsThreadPool

`epicsThreadPool.h` implements general purpose threaded work queue. Pieces of work (jobs) are submitted to a queue which is shared by a group of worker threads. After jobs are placed on the queue they may be executed in any order.

The thread pool library will never implicitly destroy pools or jobs. Such cleanup is always the responsibility of user code.

19.17.1 Configure a pool

An `epicsThreadPool` instance can be obtained in two ways. The preferred method is the use an existing shared thread pool. Alternately, a new pool may be explicitly created. In both cases `NULL` may be passed to use the configuration, or a non-default configuration may be prepared in the following way.

```
typedef struct {
    unsigned int initialThreads;
    unsigned int maxThreads;
    unsigned int workerStack;
    unsigned int workerPriority;
} epicsThreadPoolConfig;

void epicsThreadPoolConfigDefaults(epicsThreadPoolConfig *);
```

Pool configuration should always be initialized with `epicsThreadPoolConfigDefault()` before modification by user code. This will initialize configuration parameters to sensible defaults, which can then be overwritten as needed by user code.

Note that no epicsThreadPool functions will ever retain a pointer to the user provided `epicsThreadPoolConfig` instance.

initialThreads A suggestion for the number of worker threads to start immediately when the pool is created. When this number is less than maxThreads additional workers may be started later as needed. Defaults to zero.

maxThreads Upper limit on the number of worker threads in this pool. Defaults to number of CPU cores.

workerStack Worker thread stack size. Defaults to size of `epicsThreadStackSmall`.

workerPriority Worker thread priority. Defaults to just above `epicsThreadPriorityCAServerHigh`

These defaults are choosen to be suitable for CPU intensive background work by drivers.

19.17.2 Create a shared pool

```
epicsThreadPool* epicsThreadPoolGetShared(epicsThreadPoolConfig *opts);
void epicsThreadPoolReleaseShared(epicsThreadPool *pool);
```

A shared thread pool is obtained by calling `epicsThreadPoolGetShared()`. A global list of shared pools is examined. If an existing pool matches the requested configuration, then it is returned. Otherwise a new pool is created, added to the global list, then returned. `epicsThreadPoolGetShared()` may return NULL in situations of memory exhaustion.

Note that `NULL` may be passed to use the default configuration.

As for example:

```
void usercode() {
  epicsThreadPoolConfig myconf;
  epicsThreadPoolConfigDefault(&myconf);
  /* overwrite defaults if needed */
  myconf.workerPriority = epicsThreadPriorityLow;
  ... = epicsThreadPoolGetShared(&myconf);

  /* or to use the defaults */
  ... = epicsThreadPoolGetShared(NULL);
}
```

The user provided configuration may be altered to ensure that the maxThreads is greater than or equal to the number of threads the host system can run in parallel. In addition, when a existing shared pool is returned, the user supplied config is overwritten with the pool's actual config.

If a thread pool will not be used further it must be released, which may cause it to be free'd when no other references exist. It is advisable to ensure that all queued jobs have completed as queued jobs may still run if the other references to the queue remain.

When matching a requested configuration against the configuration of a existing shared pool, the following conditions must be meet for an existing shared queue to be used.

- workerPriority must match exactly.
- maxThreads and workerStack of the pool must be greater than or equal to the corresponding parameters of the request.

Note that the initialThreads option is ignored when requesting a shared pool.

19.17.3 Creating an exclusive pool

```
epicsThreadPool* epicsThreadPoolCreate(epicsThreadPoolConfig *opts);
void epicsThreadPoolDestroy(epicsThreadPool *);
```

Unshared thread pools are created/destroyed in a similar fashion.

Note that `NULL` may be passed to use the default configuration.

When a pool is destroyed it will freeze the queue to prevent new jobs from being added, then block until any previously queued jobs complete.

19.17.4 Basic job operations

```
/* job modes */
typedef enum {
    epicsJobModeRun,
    epicsJobModeCleanup
} epicsJobMode;
typedef void (*epicsJobFunction)(void* arg, epicsJobMode mode);
```

```
#define EPICSJOB_SELF ...
epicsJob* epicsJobCreate(epicsThreadPool* pool,
                         epicsJobFunction cb,
                         void* user);
void epicsJobDestroy(epicsJob*);
int epicsJobQueue(epicsJob*);
int epicsJobUnqueue(epicsJob*);
```

The normal lifecycle of a job is for it to be created, queued some number of times, then destroyed. Like with an `epicsThreadPool*`, the lifecycle of an `epicsJob*` is completely controlled by user code. Jobs will never be implicitly destroyed. When created, a pool, work function, and user argument are specified. The special user argument `EPICSJOB_SELF` may be passed to set the user argument to the `epicsJob*` returned by `epicsJobCreate()`.

A job may be queued at any time. The queuing process can fail (return non-zero) if:

- The job is not currently associated with a pool.
- The associated pool is not allowing jobs to be queued.

A job may be unqueued with `epicsJobUnqueue()`. This function will return 0 if the job was successfully removed from the queue or non-zero if this is not possible. A job can not be unqueued if it was not queued to begin with, is running, or has completed.

A job may also be destroyed at any time, including while its job function is running. In this case destruction is deferred until the job function returns.

If a thread pool is destroyed before all of its jobs are destroyed, then each job function is called one final time with the mode `epicsJobModeCleanup` to provide an opportunity to call `epicsJobDestroy`. If this is not done, then the job is disassociated from the pool. It is always the responsibility of user code to explicitly call `epicsJobDestroy`.

19.17.5 Writing job functions

```
typedef struct {
  epicsJob* job;
  ...
} myWork;

static
void myfunction(void* arg,
                epicsJobMode mode)
{
  myWork *priv=arg;
  if(mode==epicsJobModeCleanup) {
    epicsJobDestroy(priv->job);
    free(priv);
    return;
  }
  /* do normal work */
}

static
void somewhere(...)
{
   epicsThreadPool *pool;
   myWork *priv = ...; /* allocate somehow */
```

```
    pool = epicsThreadPoolCreate(NULL);
    assert(pool!=NULL && priv!=NULL);
    priv->job = epicsJobCreate(pool, &myfunction, priv);
    assert(priv->job!=NULL);
    epicsJobQueue(priv->job);
    epicsThreadPoolDestroy(pool);
}
```

Some restrictions apply to job functions. Only the following epicsThreadPool functions may be called from a job function. When using a shared pool, no modification should be made to the worker threads (eg. don't change priority). If such modifications are needed, then an exclusively owned pool should be created.

- `epicsJobQueue()`
- `epicsJobUnqueue()`
- `epicsJobCreate()`
- `epicsJobDestroy()`

No internal locks are held while a job function runs. So a job function may lock arbitrary mutexes without causing a deadlock. When in a job function, care must be taken to only call those function explicitly marked as safe to call from a running job function as these functions are written to avoid corrupting the internal state of the pool.

19.17.6 Moving jobs between pools

It may be desirable to move epicsJob instances between pools, or to have jobs not associated with any pool. This is supported with the caveat that the `epicsJobMove()` function must not run concurrently with any other epicsThreadPool functions operating on the same job. In addition to functions operating explicitly on this job, this also includes `epicsThreadPoolDestroy()`

A job may be created with no pool association by passing NULL to the `epicsJobCreate()` function instead of an explicit `epicsThreadPool*` pointer. The association can be changed at runtime with the `epicsJobMove()` function.

19.17.7 Pool control

```
typedef enum {
    epicsThreadPoolQueueAdd,
    epicsThreadPoolQueueRun
} epicsThreadPoolOption;
void epicsThreadPoolControl(epicsThreadPool* pool,
                            epicsThreadPoolQueueOption opt,
                            unsigned int val);
int epicsThreadPoolWait(epicsThreadPool* pool, double timeout);
```

It may be useful to manipulate the queue of a thread pool at runtime (eg. unittests). Currently defined options are:

epicsThreadPoolQueueAdd Set to 0 to prevent additional jobs from being queued. Set to 1 to resume normal operation.

epicsThreadPoolQueueRun Set to 0 to prevents workers from taking jobs from the queue. Set to 1 for normal operation.

These options may be combined with `epicsThreadPoolWait()` to block until the queue is empty.

`epicsThreadPoolWait()` accepts a timeout in seconds. A timeout value less than 0.0 never times out, a value of exactly 0.0 will not block, and values greater than 0.0 will block for the requested time at most.

This function returns 0 if the queue was emptied and no jobs are running at any moment during the timeout period, or non-zero if the timeout period ellapses and jobs remain in the queue or are running.

19.18 misc

19.18.1 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 (usually decimal) numbers separated by periods.

2. xxxxxxxx:p

   The Internet address number of the host, specified as a single (usually hexadecimal) 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. The numbers are normally interpreted in base 16 if they begin with '0x' or '0X', in base 8 if they begin with '0', and in base 10 otherwise. However the numeric forms are interpreted by the operating system's gethostbyname() function, thus the acceptable bases may be OS-specific.

19.18.2 adjustment

`adjustment.h` describes a single function:

```
size_t adjustToWorstCaseAlignment(size_t size);
```

adjustToWorstCaseAlignment() returns a value >= *size* that an exact multiple of the worst case alignment for the architecture on which the routine is executed.

19.18.3 cantProceed

`cantProceed.h` declares 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()` accepts a printf format string and variable number of arguments; it displays 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()`.

19.18.4 dbDefs

`dbDefs.h` includes the C header `stddef.h` and then defines several generally-useful macros if they have not already been defined:

- `TRUE` - 1
- `FALSE` - 0
- `NELEMENTS(array)` - number of elements in array.
- `CONTAINER(pointer, structure, member)` - returns a pointer to the parent structure given a pointer to a member. The `structure` argument is a type name, `member` is the name of the member in that structure that `pointer` refers to.
- `LOCAL` - synonym for `static`, deprecated.
- `OFFSET(structure, member)` - synonym for `offsetof`, deprecated.

19.18.5 epicsConvert

`epicsConvert.h` currently describes:

```
float epicsConvertDoubleToFloat(double value);
```

`epicsConvertDoubleToFloat` converts a double to a float. If the double value is outside the range that can be represented as a float the value assigned will be FLT_MIN or FLT_MAX with the appropriate matching sign. A floating exception is never raised.

19.18.6 epicsString

`epicsString.h` currently describes:

```
int epicsStrnRawFromEscaped(char *dst, size_t dstlen, const char *src,
    size_t srclen);
int epicsStrnEscapedFromRaw(char *outbuf, size_t outsize, const char *inbuf,
    size_t inlen);
size_t epicsStrnEscapedFromRawSize(const char *src, size_t srclen);
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 epicsStrGlobMatch(const char *str, const char *pattern);
char *epicsStrtok_r(char *s, const char *delim, char **lasts);
unsigned int epicsStrHash(const char *str, unsigned int seed);
unsigned int epicsMemHash(const char *str, size_t length,
    unsigned int seed);
```

`epicsStrnRawFromEscaped` copies up to `strlen` characters from the string `src` into a buffer `dst` of size `dstlen`, converting C-style escape sequences into their binary form. A zero byte terminates the input string. The

resulting string will be zero-terminated as long as dstlen is non-zero. The return value is the number of characters that were actually written into dst, not counting characters that would not fit or the zero terminator. Since the output string can never be longer than the source, it is legal for src and dst to point to the same buffer and strlen and dstlen to have the same value, thus performing the character translation in-place.

epicsStrnEscapedFromRaw does the opposite of epicsStrnRawFromEscaped: It tries to copy strlen characters from the string src into a buffer dst of size dstlen, converting non-printable characters into C-style escape sequences. A zero byte will *not* terminate the input string. The output string will be zero-terminated as long as dstlen is non-zero. No more than dstlen characters will actually be written into the output buffer, although all the characters in the input string will be read. The return value is the number of characters that would have been stored in the output buffer if it were large enough, or a negative value if dst == src. In-place translations are not allowed since the escaped results will usually be larger than the input string.

The following escaped character constants will be used in the output:

```
\a  \b  \f  \n  \r  \t  \v  \\  \'  \"
```

All other non-printable characters appear as octal escapes in form \ooo where ooo are three octal digits (0-7). Non-printable characters are determined by the C runtime library's isprint() function.

epicsStrnEscapedFromRawSize scans up to strlen characters of the string src that may contain non-printable characters, and returns the size of the output buffer that would be needed to escape that string. The terminating zero-byte needed in the output buffer is not included in the count, so must be allowed for by the caller. This routine is faster than calling epicsStrnEscapedFromRaw with a zero length output buffer; both should return the same result.

epicsStrPrintEscaped prints the contents of its input buffer, substituting escape sequences for non-printable characters.

epicsStrCaseCmp and epicsStrnCaseCmp implement the strcasecmp and strncasecmp functions respectively, which are not available on all supported operating systems. They operate like strcmp and strncmp, but are case insensitive, using the C locale.

epicsStrDup implements strdup, which is not available on all supported operating systems. It allocates sufficient memory for the string, copies it and returns a pointer to the new copy. The pointer should eventually be passed to the function free(). If insufficient memory is available cantProceed() is called.

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.

epicsStrHash calculates a hash of a zero-terminated string str, while epicsMemHash uses the same algorithm on a fixed-length memory buffer that may contain zero bytes. In both cases an initial seed value may be provided which permits multiple strings or buffers to be combined into a single hash result. The final result should be masked to achieve the desired number of bits in the hash value.

19.18.7 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 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.

19.18.8 locationException

A C++ template for use as an exception object, used inside Channel Access. Not documented here.

19.18.9 shareLib.h

This is the header file for the "decorated names" that appear in header files, e.g.

```
#define epicsExportSharedSymbols
epicsShareFunc int epicsShareAPI a_func(int arg);
```

Thse are needed to properly create DLLs on windows. Read the comments in the shareLib.h file for a detailed description of where they should be used. Note that the `epicsShareAPI` decorator is deprecated for all new EPICS APIs and is being removed from APIs that are only used within the IOC.

19.18.10 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.

19.18.11 unixFileName.h

Defines macros OSI_PATH_LIST_SEPARATOR and OSI_PATH_SEPARATOR

19.18.12 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 generating human-readable output. 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);

typedef int (*TESTFUNC)(void);
epicsShareFunc void testHarness(void);
epicsShareFunc void runTestFunc(const char *name, TESTFUNC func);

#define runTest(func) runTestFunc(#func, func)
```

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 when writing test programs. The individual testPass() and testFail() routines can be used when the test program takes a different path on success than on failure, but one or other must always be called for any particular test. 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 fully implemented, these tests may still be executed, wrapped between calls to testTodoBegin() and testTodoEnd(). testTodoBegin() takes a constant string indicating why these tests are not expected to succeed. This modifies the counting of the results so the wrapped tests will not be recorded as failures.

Additional information can be supplied using the testDiag() routine, which displays the relevent information as a comment in the result output. None of the printable 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.

On vxWorks and RTEMS, an alternative test harness can be used to run a series of tests in order and summarize the results from them all at the end just like the Perl harness does. The routine testHarness() is called once at the beginning of the test harness program. Each test program is run by passing its main routine name to the runTest() macro which expands into a call to the runTestFunc() routine. The last test program or the harness program itself must finish by calling epicsExit() which triggers the test summary mechanism to generate its result outputs (from an epicsAtExit callback routine).

Some tests require the context of an IOC to be run. This conflicts with the idea of running multiple tests within a test harness, as iocInit() is only allowed to be called once, and some parts of the full IOC (e.g. the rsrv CA server) can not be shut down cleanly. The function iocBuildIsolated() allows to start an IOC without its Channel Access parts, so that it can be shutdown quite cleany using iocShutdown(). This feature is only intended to be used from test programs. Do not use it on productional IOCs. After building the IOC using iocBuildIsolated() or iocBuild(), it has to be started by calling iocRun(). The suggested call sequence in a test program that needs to run the IOC without Channel Access is:

```
#include "iocInit.h"

MAIN(iocTest)
{
    iocBuildIsolated() || iocRun();

    /* ... test code ... */

    iocShutdown();
    dbFreeBase(pdbbase);
    registryFree();
    pdbbase = NULL;
    return testDone();
}
```

The part from iocBuildIsolated() to iocShutdown() can be repeated to execute multiple tests within one executable or harness.

To make it easier to create a single test program that can be built for both the embedded and workstation operating system harnesses, the header file `testMain.h` provides a convenience macro MAIN() that adjusts the name of the test program according to the platform it is running on: main() on workstations and a regular function name on embedded systems.

The following is a simple example of a test program using the epicsUnitTest routines:

```
#include <math.h>
#include "epicsUnitTest.h"
#include "testMain.h"

MAIN(mathTest)
{
    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-specific APIs for facilities that are different on the various systems. These APIs are called Operating System Independent, or OSI, and are part of libCom. The OSI APIs have multiple implementations, which are Operating System Dependent or OSD. Some APIs are implemented using the features of a particular compiler that is supported on multiple operating systems. For example the GNU Compiler Collection (GCC) is used for compiling many targets and provides a common GCC-specific API for some features. Base now makes it possible to use compiler-specific as well as OS-specific code to implement the OSI APIs.

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/
   *.h
   *.c
   *.cpp
   compiler/
       borland/
       clang/
       default/
       gcc/
       msvc/
       solStudio/
   os/
       Linux/
       Darwin/
       RTEMS/
       WIN32/
       default/
       posix/
       solaris/
       vxWorks/
```

Code for additional compilers and operating systems may also be present.

20.1.2 Rules for building OSI code

The `src/libCom/osi` directory contains source and header files that provide common definitions for the OSI APIs. The directories under `osi/compiler` contain compiler-specific files that implement some of the OSI APIs. The directories under `osi/os` contain operating-system-specific files that implement the remaining OSI APIs.

Header files residing under `src/libCom/osi` are installed into `include` as follows:

- Header files in the `osi` directory are installed into `include`
- Header files from an OS-specific directory `osi/os/<os>` are installed into `include/os/<os>`. The search order for locating the specific file to be installed is:
  1. `osi/os/<os>`
  2. `osi/os/posix` — if the target uses Posix APIs
  3. `osi/os/default`
- Header files from a compiler-specific directory `osi/compiler/<cmplr>` are installed into the directory `include/compiler/<cmplr>`. The search order for locating the specific file to be installed is:
  1. `osi/compiler/<cmplr>`
  2. `osi/compiler/default`

The search order for locating OSD source files is:

1. `osi/compiler/<cmplr>`
2. `osi/compiler/default`
3. `osi/os/<os>`
4. `osi/os/posix`
5. `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:

1. `include/compiler/<cmplr>`
2. `include/os/<os>`
3. `include`

20.2 epicsAssert

This is an enhanced version of ANSI C's `assert` facility. To use this, replace:

```
#include <assert.h>
```

with

```
#include "epicsAssert.h"
```

20.2.1 Runtime Assertion Checks

The same `assert(expression)` macro is used as with the ANSI header to test a run-time assertion.

If a run-time assertion check finds the expression false, it calls `errlog` indicating the program's author, file name, and line number. Each OS provides 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; on posix there are instructions about saving the core file. After printing the message, the calling thread is suspended.

To provide the author's name for the message, before including the `epicsAssert.h` file define a preprocessor macro named `epicsAssertAuthor` as a string containing the name and email address to be contacted.

20.2.2 Compile-time Assertion Checks

The C or C++ compiler can be used to evaluate and check a static expression at compile-time. Static assertions may only be placed where a variable declaration is valid, and can only test certain kinds of constant expressions. A static assertion looks like this:

```
STATIC_ASSERT(expression);
```

If the expression evaluates as false, the compiler will be presented with an illegal variable declaration using the name `static_assert_failed_at_line_<n>` and so should halt compilation at that point.

20.3 epicsAtomic

This is an operating system and compiler independent interface to an operating system and or compiler dependent implementation of several atomic primitives. Currently, only increment, decrement, add, subtract, compare-and-swap, and test-and-set primitives have been implemented as appropriate for the C primitive types of `int`, `size_t`, and `void *` pointer.

These primitives can be safely used in a multithreaded programs on symmetric multiprocessing (SMP) systems. Where possible the primitives are implemented with compiler intrinsic wrappers for archetecture specific instructions. Otherwise they are implemeted with OS specific functions and otherwise, when lacking a sufficently capable OS specific interface, then in some rare situations a mutual exclusion primitive is used for synchronization.

In operating systems environments which allow C code to run at interrupt level the implementation must use interrupt level invokable CPU instruction primitives.

All C++ functions are implemented in the namespace `atomics` which is nested inside of namespace `epics`.

```
#include <epicsAtomic.h>
```

20.3.1 C Callable Interface

```
void epicsAtomicReadMemoryBarrier ();
void epicsAtomicWriteMemoryBarrier ();

size_t epicsAtomicIncrSizeT ( size_t * pTarget );
int epicsAtomicIncrIntT ( int * pTarget );

size_t epicsAtomicDecrSizeT ( size_t * pTarget );
int epicsAtomicDecrIntT ( int * pTarget );
```

```
size_t epicsAtomicAddSizeT ( size_t * pTarget, size_t delta );
size_t epicsAtomicSubSizeT ( size_t * pTarget, size_t delta );
int epicsAtomicAddIntT ( int * pTarget, int delta );

void epicsAtomicSetSizeT  ( size_t * pTarget, size_t newValue );
void epicsAtomicSetIntT ( int * pTarget, int newValue );
void epicsAtomicSetPtrT ( EpicsAtomicPtrT * pTarget, EpicsAtomicPtrT newValue );

size_t epicsAtomicGetSizeT ( const size_t * pTarget );
int epicsAtomicGetIntT ( const int * pTarget );
EpicsAtomicPtrT epicsAtomicGetPtrT ( const EpicsAtomicPtrT * pTarget );

size_t epicsAtomicCmpAndSwapSizeT ( size_t * pTarget,
                                    size_t oldVal, size_t newVal );
int epicsAtomicCmpAndSwapIntT ( int * pTarget,
                                    int oldVal, int newVal );
EpicsAtomicPtrT epicsAtomicCmpAndSwapPtrT (
                                    EpicsAtomicPtrT * pTarget,
                                    EpicsAtomicPtrT oldVal,
                                    EpicsAtomicPtrT newVal );

int epicsAtomicTestAndSet ( int * pTarget );
```

C Function	C++ Function	Meaning
epicsAtomicReadMemoryBarrier	readMemoryBarrier	Load target into cache.
epicsAtomicWriteMemoryBarrier	writeMemoryBarrier	Push cache version of target into target.
epicsAtomicIncrXxxx	increment	Lock out other SMP processors from accessing the target, load the target into cache, add one to the target, flush cache to the target, allow other SMP processors to access the target, and return the new value of the target as modified by this operation.
epicsAtomicDecrXxxx	decrement	Lock out other smp processors from accessing the target, load the target into cache, subtract one from the target, flush cache to the target, allow other SMP processors to access the target, and return the new value of target as modified by this operation.
epicsAtomicAddXxxx	add	Lock out other SMP processors from accessing the target, load the target into cache, add delta to the target, flush the cache to the target, allow other SMP processors to access the target, return new value of target as modified by this operation.
epicsAtomicSubXxxx	subtract	Lock out other SMP processors from accessing the target, load the target into cache, subtract delta from the target, flush the cache to the target, allow other SMP processors to access the target, return new value of target as modified by this operation.
epicsAtomicSetXxxx	set	Set the cached version of target, and flush the cache to the target.
epicsAtomicGetXxxx	get	Fetch the target into cache, and return the cached value.

epicsAtomicCmpAndSwapXxxx	compareAndSwap	Lock out other SMP processors from accessing the target, load the target into cache, if the target is equal to oldVal set the target to newVal, flush cache to the target, allow other SMP processors to access the target, and return the original value stored in the target.
epicsAtomicTestAndSet	testAndSet	Lock out other SMP processors from accessing the target, load the target into cache, if the target value is logical false (zero) set the target to be logical true, flush cache to the target, allow other SMP processors to access the target, and return the original value stored in the target.

20.4 epicsEndian

`epicsEndian.h` provides an operating-system independent means of discovering the native byte order of the CPU which the compiler is targeting, and works in both C and C++ code. It defines the following preprocessor macros, the values of which are integers:

- `EPICS_ENDIAN_LITTLE`
- `EPICS_ENDIAN_BIG`
- `EPICS_BYTE_ORDER`
- `EPICS_FLOAT_WORD_ORDER`

The latter two macros are defined to be one or other of the first two, and may be compared with them to determine conditional compilation or execution of code that performs byte or word swapping as necessary.

```
#if (EPICS_BYTE_ORDER == EPICS_ENDIAN_BIG)
    /* ... */
#else /* EPICS_ENDIAN_LITTLE */
    /* ... */
#endif /* EPICS_BYTE_ORDER */
```

20.5 epicsEvent

`epicsEvent.h` defines the C++ and C APIs for a simple binary semaphore. If multiple threads are waiting on the same event, only one of them will be woken when the event is signalled.

20.5.1 C++ Interface

```
typedef enum {
    epicsEventOK = 0,
    epicsEventWaitTimeout,
    epicsEventError
} epicsEventStatus;

/* Backwards compatibility */
#define epicsEventWaitStatus epicsEventStatus
```

```
#define epicsEventWaitOK epicsEventOK
#define epicsEventWaitError epicsEventError

typedef enum {
    epicsEventEmpty,
    epicsEventFull
} epicsEventInitialState;

class epicsEvent {
public:
    epicsEvent ( epicsEventInitialState initial = epicsEventEmpty );
    ~epicsEvent ();
    void trigger ();
    void signal () { this->trigger(); }
    void wait ();                    /* blocks until full */
    bool wait ( double timeOut );    /* false if still empty at time out */
    bool tryWait ();                 /* false if empty */
    void show ( unsigned level ) const;

    class invalidSemaphore;          /* exception payload */
private:
    ...
};
```

Method	Meaning
epicsEvent	An epicsEvent can be created empty or full. If it is created empty then a wait issued before a trigger will block. If created full then the first wait will always succeed. Multiple triggers may be issued between waits but have the same effect as a single trigger.
~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.
trigger	Trigger 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.
signal	A synonym for trigger, provided for backwards compatibility.
wait()	Wait for the event to be triggered.
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 true) if the event was triggered, false if it timed out.
tryWait()	Similar to wait except that if event has not already been triggered the call completes immediately. The return value is true if an unused event has already been signalled, false if not.
show	Display information about the epicsEvent. The information displayed is architecture dependent.

The primary use of an event semaphore is for thread 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->trigger();
```

20.5.2 C Interface

```
epicsEventId epicsEventCreate(epicsEventInitialState initialState);
epicsEventId epicsEventMustCreate(epicsEventInitialState initialState);
void epicsEventDestroy(epicsEventId id);

epicsEventStatus epicsEventTrigger(epicsEventId id);
void epicsEventMustTrigger(epicsEventId id);
#define epicsEventSignal(ID) epicsEventMustTrigger(ID)

epicsEventStatus epicsEventWait(epicsEventId id);
void epicsEventMustWait(epicsEventId id);
epicsEventStatus epicsEventWaitWithTimeout(epicsEventId id, double timeOut);
epicsEventStatus epicsEventTryWait(epicsEventId id);

void epicsEventShow(epicsEventId id, unsigned int level);
```

The C routines shown above generally correspond to one of the C++ methods. The `epicsEventSignal` macro provides backwards compatibility. The routines `epicsEventMustCreate`, `epicsEventMustTrigger` and `epicsEventMustWait` do not return if they fail.

20.6 epicsFindSymbol

`epicsFindSymbol.h` contains the following definitions:

```
void * epicsFindSymbol(const char *name);
void * epicsLoadLibrary(const char *name);
const char *epicsLoadError(void);
```

Function	Meaning
epicsFindSymbol	Return the address of the named variable
epicsLoadLibrary	Load named shared library
epicsLoadError	Returns an error string if a library load fails

The registry, described in another chapter, provides a way to find and return the address referred to by a registered symbol. Symbols that have not been explicitly registered may not be found. If the registry is asked for a name that has not been registered, it calls `epicsFindSymbol`. If `epicsFindSymbol` can locate the symbol, it returns the associated address, otherwise it returns null.

On vxWorks `epicsFindSymbol` calls `symFindByName`. On Linux, Darwin and Solaris it calls `dlsym`. The default version just returns null, i.e. it always fails.

The `epicsLoadLibrary` routine can be used to load a named library. Note that the library name may be different on different operating systems. This routine is implemented on vxWorks, Linux, Darwin, Windows and Solaris.

If `epicsLoadLibrary` fails, the routine `epicsLoadError` can be used to fetch a printable string that describes the reason for the failure. Note that this library loading API is not thread-safe and should not be used in circumstances where multiple threads might try to use it at the same time.

20.7 epicsGeneralTime

The generalTime framework provides a mechanism for several time providers to be present within the system. There are two types of provider, one type for the current time and one type for providing Time Event times. Each time provider has a priority, and installed providers are queried in priority order whenever a time is requested, until one returns successfully. Thus there is a fallback from higher priority providers (smaller value of priority) to lower priority providers (larger value of priority) if the higher priority ones fail. Each architecture has a "last resort" provider, installed at priority 999, usually based on the system clock, which is used in the absence of any other provider.

Targets running vxWorks and RTEMS have an NTP provider installed at priority 100.

Registered providers may also add an interrupt-safe routine to be called from the `epicsTimeGetCurrentInt()` or `epicsTimeGetEventInt()` API routines. These interfaces do not check the priority queue, and will only succeed if the last-used provider has registered a suitable routine.

There are two interfaces to this framework, `epicsGeneralTime.h` for consumers that wish to get a time and query the framework, and `generalTimeSup.h` for providers that supply timestamps.

20.7.1 Consumer interface

The `epicsGeneralTime.h` header contains the following:

```
void generalTime_Init(void);
int installLastResortEventProvider(void);
void generalTimeResetErrorCounts(void);
int generalTimeGetErrorCounts(void);

const char * generalTimeHighestCurrentName(void);
const char * generalTimeCurrentProviderName(void);
const char * generalTimeEventProviderName(void);

/* Original names, for compatibility */
#define generalTimeCurrentTpName generalTimeCurrentProviderName
#define generalTimeEventTpName generalTimeEventProviderName

long generalTimeReport(int interest);
```

Function	Meaning
generalTime_Init	Initialise the framework. This is called automatically by any function that requires the framework. It does not need to be called explicitly.
installLastResortEventProvider	Install a Time Event time provider that returns the current time for any Time Event number. This is optional as it is site policy whether the last resort for a Time Event time in the absence of any working provider should be a failure, or the current time.
generalTimeResetErrorCounts	Reset the internal counter of the number of times the time returned was earlier than when previously requested. Used by device support for bo record with DTYP = "General Time" and OUT = "@RSTERRCNT"

generalTimeGetErrorCounts	Return the internal counter of the number of times the time returned was earlier than when previously requested. Used by device support for longin record with DTYP = "General Time" and INP = "@GETERRCNT"
generalTimeCurrentProviderName	Return the name of the provider that last returned a valid current time, or NULL if none. Used by stringin device support with DTYP = "General Time" and INP = "@BESTTCP"
generalTimeEventProviderName	Return the name of the provider that last returned a valid Time Event time, or NULL if none. Used by stringin device support with DTYP = "General Time" and INP = "@BESTTEP"
generalTimeHighestCurrentName	Return the name of the registered current time provider that has the highest priority. Used by stringin device support with DTYP = "General Time" and INP = "@TOPTCP"
generalTimeReport	Provide information about the installed providers and their current best times.

20.7.2 Time Provider Interface

The `generalTimeSup.h` header for time providers contains the following:

```
typedef int (*TIMECURRENTFUN)(epicsTimeStamp *pDest);
typedef int (*TIMEEVENTFUN)(epicsTimeStamp *pDest, int event);

int generalTimeRegisterCurrentProvider(const char *name,
    int priority, TIMECURRENTFUN getTime);
int generalTimeRegisterEventProvider(const char *name,
    int priority, TIMEEVENTFUN getEvent);

/* Original names, for compatibility */
#define generalTimeCurrentTpRegister generalTimeRegisterCurrentProvider
#define generalTimeEventTpRegister generalTimeRegisterEventProvider

int generalTimeAddIntCurrentProvider(const char *name,
    int priority, TIMECURRENTFUN getTime);
int generalTimeAddIntEventProvider(const char *name,
    int priority, TIMEEVENTFUN getEvent);

int generalTimeGetExceptPriority(epicsTimeStamp *pDest, int *pPrio,
        int ignorePrio);
```

Function	Meaning
generalTimeRegisterCurrentProvider	Register a current time provider with the framework. The getTime routine must return epicsTimeOK if it provided a valid time, or epicsTimeERROR if it could not.
generalTimeRegisterEventProvider	Register a provider of Time Event times with the framework. The getEvent routine must return epicsTimeOK if it provided a valid time for the requested Time Event, or epicsTimeERROR if it could not. It is an implemetation decision by the provider whether a request for an Event that has never happened should return an error and/or a valid timestamp.
generalTimeAddIntCurrentProvider	Add or replace an interrupt-safe provider routine for an already-registered current time provider with the given name and priority.
generalTimeAddIntEventProvider	Add or replace an interrupt-safe provider routine for an already-registered event time provider with the given name and priority.

generalTimeGetExceptProirity — Request the current time from the framework, but exclude providers with priority ignorePrio. This allows a provider without an absolute time source to synchronise itself with other providers that do provide an absolute time. pPrio returns the priority of the provider that supplied the result, which may be higher or lower than ignorePrio.

If multiple providers are registered at the same priority, they will be queried in the order in which they were registered until one is able to provide the time when requested.

Some providers may start a task that periodically synchronizes themselves with a higher priority provider, using `generalTimeGetExceptPriority()` to ensure that they are themselves excluded from this time request.

Interrupt-safe providers are optional, but an IOC that needs to request the time from interrupt context must be using a current or event time source that supports the appropriate request because only the most recently successful provider will be used (the priority list cannot be traversed for requests made from interrupt context). The result returned by is not protected against backwards movement.

20.7.3 Internal Interface

The generalTime framework also now provides the implementations of the following routines that are declared in `epicsTime.h`:

```
int epicsTimeGetCurrent(epicsTimeStamp *pDest);
int epicsTimeGetEvent(epicsTimeStamp *pDest, int eventNumber);
int epicsTimeGetCurrentInt(epicsTimeStamp *pDest);
int epicsTimeGetEventInt(epicsTimeStamp *pDest, int eventNumber);
```

If `epicsTimeGetEvent()` is called with an event number of 0 (`epicsTimeEventCurrentTime`) it will return the time from the best available current time provider. Thus providers do not need to provide event times if they do not implement an event system.

20.7.4 Example

Soft device support is provided for ai, bo, longin and stringin records. A typical example is:

```
record(ai, "$(IOC):GTIM_CURTIME") {
  field(DESC, "Get Time")
  field(DTYP, "General Time")
  field(INP,  "@TIME")
}

record(bo, "$(IOC):GTIM_RSTERR") {
  field(DESC, "Reset ErrorCounts")
  field(DTYP, "General Time")
  field(OUT,  "@RSTERRCNT")
}

record(longin, "$(IOC):GTIM_ERRCNT") {
  field(DESC, "Get ErrorCounts")
  field(DTYP, "General Time")
  field(INP,  "@GETERRCNT")
}
```

```
record(stringin, "$(IOC):GTIM_BESTTCP") {
  field(DESC, "Best Time-Current-Provider")
  field(DTYP, "General Time")
  field(INP,  "@BESTTCP")
}

record(stringin, "$(IOC):GTIM_BESTTEP") {
  field(DESC, "Best Time-Event-Provider")
  field(DTYP, "General Time")
  field(INP,  "@BESTTEP")
}
```

20.8 epicsInterrupt

`epicsInterrupt.h` contains the following:

20.8.1 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.

20.8.2 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.

20.9 epicsMath

epicsMath.h includes math.h and also ensures that isnan and isinf are defined.

20.10 epicsMessageQueue

`epicsMessageQueue.h` describes a C++ and a C facility for interlocked communication between threads.

20.10.1 C++ Interface

EpicsMessageQueue provides methods for sending messages between threads on a first in, first out basis. It is designed so that a single message queue can be used with multiple writer and reader threads.

```
class epicsMessageQueue {
public:
    epicsMessageQueue(unsigned int capacity, unsigned int maximumMessageSize);
    ~epicsMessageQueue();
    int trySend(void *message, unsigned int messageSize);
    int send(void *message, unsigned int messageSize);
    int 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
epicsMessageQueue()	Constructor. The capacity is the maximum number of messages, each containing 0 to maximumMessageSize bytes, that can be stored in the message queue.
~epicsMessageQueue()	Destructor.

trySend()	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.
send()	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.
tryReceive()	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 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.
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 dispalyed.
pending()	Returns the number of messages presently in the queue.

20.10.2 C interface

```
typedef struct epicsMessageQueueOSD *epicsMessageQueueId;

epicsMessageQueueId epicsMessageQueueCreate(unsigned int capacity,
    unsigned int maximumMessageSize);
void epicsMessageQueueDestroy(epicsMessageQueueId id);
int epicsMessageQueueTrySend(epicsMessageQueueId id,
    void *message, unsigned int size);
int epicsMessageQueueSend(epicsMessageQueueId id,
    void *message, unsigned int size);
int epicsMessageQueueSendWithTimeout(epicsMessageQueueId id,
    void *message, unsigned int size, double timeout);
int epicsMessageQueueTryReceive(epicsMessageQueueId id,
    void *message, unsigned int size);
int epicsMessageQueueReceive(epicsMessageQueueId id,
    void *message, unsigned int size);
int epicsMessageQueueReceiveWithTimeout(epicsMessageQueueId id,
    void *message, unsigned int size, double timeout);
void epicsMessageQueueShow(epicsMessageQueueId id);
int epicsMessageQueuePending(epicsMessageQueueId id);
```

Each C function corresponds to one of the C++ methods.

20.11 epicsMutex

`epicsMutex.h` contains both C++ and C descriptions for a mutual exclusion semaphore.

20.11.1 C++ Interface

```
typedef enum {
    epicsMutexLockOK, epicsMutexLockTimeout, epicsMutexLockError
} epicsMutexLockStatus;

#define newEpicsMutex new epicsMutex(__FILE__,__LINE__)

class epicsMutex {
public:
    epicsMutex ();
    epicsMutex ( const char *pFileName, int lineno );
    ~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 `newEpicsMutex` macro simplifies debugging by letting the mutex store the source filename and line-number where it was created, using the second form of the constructor. The C interface also stores this information. The `epicsMutex::show()` routine can display that source location.

The typical use of a mutual exclusion semaphore is:

```
epicsMutex *plock = newEpicsMutex;
...
...
plock->lock();
/* process resource */
plock->unlock();
```

20.11.2 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.

20.11.3 Implementation Notes

The implementation:

- Must implement recursive locking
- May implement priority inheritance and be deletion safe

A posix version is implemented via pthreads.

20.12 epicsSpin

`epicsSpin.h` contains definitions for a spin lock semaphore.

20.12.1 C Interface

```
typedef struct epicsSpin *epicsSpinId;

epicsSpinId epicsSpinCreate();
void epicsSpinDestroy(epicsSpinId);

void epicsSpinLock(epicsSpinId);
int epicsSpinTryLock(epicsSpinId);
void epicsSpinUnlock(epicsSpinId);
```

Method	Meaning
epicsSpinCreate	Create a spin lock, allocate required resources, and initialize it to an unlocked state.
epicsSpinDestroy()	Remove the spin lock and any resources it uses. Any further use of the spin lock may result in unknown (most certainly bad) behavior. The results are also undefined if epicsSpinDestroy is used when a thread holds the lock.
epicsSpinLock()	Wait (by spinning, i.e. busy waiting) until the resource is free. After a successful lock, no additional, i.e. recursive, locking attempts may be issued.
epicsSpinTryLock()	Similar to lock except that, if the resource is owned by another thread, the call completes immediately. The return value is 0 if the resource is owned by the caller.

epicsSpinUnlock() Release the spin lock resource which was locked by epicsSpinLock or epicsSpinTryLock. The results are undefined if the lock is not held by the calling thread, or if epicsSpinUnlock is used on an uninitialized spin lock.

20.12.2 Implementation Notes

The default implementation uses POSIX spin locks on POSIX compliant systems, and epicsMutex for non-POSIX environments.

20.13 epicsStdlib

`epicsStdlib.h` includes `stdlib.h` and `epicsTypes.h` and provides declarations for a series of string to number conversion functions and macros.

20.13.1 Integer Parsing

These routines convert a string into an integer of the indicated type and number base. The units pointer argument may be NULL, but if not it will be left pointing to the first non-whitespace character following the numeric string, or to the terminating zero byte.

```
int epicsParseLong(const char *str, long *to, int base, char **units);
int epicsParseULong(const char *str, unsigned long *to, int base,
    char **units);
int epicsParseLLong(const char *str, long long *to, int base, char **units);
int epicsParseULLong(const char *str, unsigned long long *to, int base,
    char **units);

int epicsParseInt8(const char *str, epicsInt8 *to, int base, char **units);
int epicsParseUInt8(const char *str, epicsUInt8 *to, int base, char **units);

int epicsParseInt16(const char *str, epicsInt16 *to, int base, char **units);
int epicsParseUInt16(const char *str, epicsUInt16 *to, int base, char **units);

int epicsParseInt32(const char *str, epicsInt32 *to, int base, char **units);
int epicsParseUInt32(const char *str, epicsUInt32 *to, int base, char **units);

int epicsParseInt64(const char *str, epicsInt64 *to, int base, char **units);
int epicsParseUInt64(const char *str, epicsUInt64 *to, int base, char **units);
```

The return value from these routines is a status code, zero meaning OK.

20.13.2 Floating-point Parsing

These routines convert a string into a floating-point type.

```
double epicsStrtod(const char *str, char **endp);

int epicsParseDouble(const char *str, double *to, char **units);
```

```
int epicsParseFloat(const char *str, float *to, char **units);

#define epicsParseFloat32(str, to, units) epicsParseFloat(str, to, units)
#define epicsParseFloat64(str, to, units) epicsParseDouble(str, to, units)
```

`epicsStrtod()` has the same semantics as the C99 function `strtod()` and is provided because some architectures do not fully conform to the standard. It is implemented as a simple macro on those architectures that do conform.

`epicsParseDouble` and `epicsParseFloat` convert a string into a variable of the indicated type. The units pointer argument may be NULL, but if not it will be left pointing to the first non-whitespace character following the numeric string, or to the terminating zero byte.

The return value from these routines is a status code, zero meaning OK.

20.13.3 Replacements for scanf()

The following routines are implemented as macros that call routines described above. They return 1 for a successful conversion, 0 on failure, and can be used to replace equivalent calls to `sscanf()`.

```
#define epicsScanLong(str, to, base) !epicsParseLong(str, to, base, NULL)
#define epicsScanULong(str, to, base) !epicsParseULong(str, to, base, NULL)
#define epicsScanLLong(str, to, base) !epicsParseLLong(str, to, base, NULL)
#define epicsScanULLong(str, to, base) !epicsParseULLong(str, to, base, NULL)
#define epicsScanFloat(str, to) !epicsParseFloat(str, to, NULL)
#define epicsScanDouble(str, to) !epicsParseDouble(str, to, NULL)
```

`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.

20.14 epicsStdio

The `epicsStdio.h` first includes the operating systems's `stdio.h` header, then provides 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);

enum TF_RETURN {TF_OK = 0, TF_ERROR = 1};
enum TF_RETURN truncateFile(const char *pFileName, unsigned size );

/* Stream redirection of standard streams */
#define stdin epicsGetStdin()
#define stdout epicsGetStdout()
#define stderr epicsGetStderr()

#define printf epicsStdoutPrintf
#define puts epicsStdoutPuts
#define putchar epicsStdoutPutchar
```

```
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, ...);
int epicsStdoutPuts(const char *str);
int epicsStdoutPutchar(int c);
```

`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, while others implement them incorrectly. Standardized as a C99 function, `snprintf` acts like `sprintf` except that the `size` argument gives the maximum number of characters (including the trailing zero byte) that may be placed in `str`. Similarly `vsnprintf` is a size-limited version of `vsprintf`. In both cases the return values are supposed to be the number characters (not counting the terminating zero byte) that would be written to `str` if it was large enough to hold them all; the output has been truncated if the return value is `size` or more.

On some operating systems though the implementations of these functions do not always return the correct value. If the OS implementation does not correctly return the number of characters that would have been written when the output gets truncated, it is not worth trying to fix this as long as they return `size-1` instead; the resulting string must always be correctly terminated with a zero byte.

On operating systems such as Solaris which follow the Single Unix Specification V2, the `epicsSnprintf` and `epicsVsnprintf` implementations may not provide correct C99 semantics for the return value when `size` is given as zero. On these systems `epicsSnprintf` and `epicsVsnprintf` can return an error (a value less than zero) when a buffer length of zero is passed in, so callers should not use that technique to calculate the length of the buffer required.

`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 `epicsSetThreadStdin/Stdout/Stderr` routines allow the standard file steams to be redirected on a per thread basis, e.g. calling `epicsSetThreadStdout` will affect only the thread which calls it. To cancel a stream redirection, pass a NULL argument in another call to the same redirection routine that was used to set it.

The routines `epicsGetStdin/Stdout/Stderr` and `epicsStdoutPrintf/Puts/Putchar` are not normally named directly in user code. They are provided for the following macros that redefine the well-known C identifiers:

- `stdin` becomes `epicsGetStdin()`
- `stdout` becomes `epicsGetStdout()`
- `stderr` becomes `epicsGetStderr()`
- `printf` becomes `epicsStdoutPrintf`
- `puts` becomes `epicsStdoutPuts`
- `putchar` becomes `epicsStdoutPutchar`

This is done so that any I/O through these standard streams can be redirected, usually to or from a file. The primary use of this facility is iocsh, which allows redirection of the input and/or output streams when running commands. In order for the redirection to work, all modules involved in I/O must include `epicsStdio.h` instead of the system header `stdio.h`.

20.15 epicsTempFile

`epicsTempFile.h` provides definitions for the following functions:

```
void epicsTempName(char *pbuf, size_t buflen);
FILE * epicsTempFile(void);
```

`epicsTempName` and `epicsTempFile` can be called to get unique filenames and open `FILE *` pointers. Note that `epicsTempName` cannot guarantee that the filenames it returns will not be created by some other thread or process after it has returned, although it does check that the filename it generates does not already exist. This security hole is why POSIX.1-2008 marked `tempnam()` as obsolete. The `epicsTempName` function will probably be deprecated in a future release of Base.

20.16 epicsThread

`epicsThread.h` contains C++ and C descriptions for a thread.

20.16.1 C Interface

```
typedef void (*EPICSTHREADFUNC)(void *parm);

#define epicsThreadPriorityMax          99
#define epicsThreadPriorityMin           0

/* some generic values */
#define epicsThreadPriorityLow          10
#define epicsThreadPriorityMedium       50
#define epicsThreadPriorityHigh         90

/* some iocCore specific values */
#define epicsThreadPriorityCAServerLow  20
#define epicsThreadPriorityCAServerHigh 40
#define epicsThreadPriorityScanLow      60
#define epicsThreadPriorityScanHigh     70
#define epicsThreadPriorityIocsh        91
#define 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);

/* (epicsThreadId)0 is guaranteed to be an invalid thread id */
typedef struct epicsThreadOSD *epicsThreadId;

typedef int epicsThreadOnceId;
```

```
#define EPICS_THREAD_ONCE_INIT 0

void epicsThreadOnce(epicsThreadOnceId *id, EPICSTHREADFUNC, void *arg);

void epicsThreadExitMain(void);

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);
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);
int epicsThreadGetCPUs(void);

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 (*EPICS_THREAD_HOOK_ROUTINE)(epicsThreadId id);
int epicsThreadHookAdd(EPICS_THREAD_HOOK_ROUTINE hook);
int epicsThreadHookDelete(EPICS_THREAD_HOOK_ROUTINE hook);
void epicsThreadHooksShow(void);
void epicsThreadMap(EPICS_THREAD_HOOK_ROUTINE func);

typedef void * epicsThreadPrivateId;
epicsThreadPrivateId epicsThreadPrivateCreate(void);
void epicsThreadPrivateDelete(epicsThreadPrivateId id);
void epicsThreadPrivateSet(epicsThreadPrivateId,void *);
void * epicsThreadPrivateGet(epicsThreadPrivateId);
```

Routine	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:

```
void myInitFunc(void * arg) {
    ...
}
epicsThreadOnceId onceFlag =
    EPICS_THREAD_ONCE_INIT;
...
epicsThreadOnce(&onceFlag, myInitFunc,
    (void *)myParm);
```

For each unique epicsThreadOnceId, epicsThreadOnce guarantees that 1) myInitFunc will only be called only once. 2) myInitFunc will have returned before any other epicsThreadOnce call returns. 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.

epicsThreadIsEqual

Compares two threadIds and returns (0,1) if they (are not, are) the same.

epicsThreadIsSuspended

How and why a thread can be suspended is implementation dependent. A thread calling epicsThreadSuspendSelf should result in this routine returning true for that thread, but a thread may also be suspended for other reasons.

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 intime 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 value of 0 means that no thread was found with the specified name.
epicsThreadGetCPUs	Get the number of CPUs (logical cores) available to the IOC. On systems that provide Hyper-Threading, this may be more the number of physical CPU cores.
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.
epicsThreadHookAdd	Register a routine to be called by every new thread before the thread function gets run. Hook routines will often register a thread exit routine with epicsAtThreadExit to release thread-specific resources they have allocated.
epicsThreadHookDelete	Remove routine from the list of hooks run at thread creation time.
epicsThreadHooksShow	Print the current list of hook function pointers.
epicsThreadMap	Call func once for every known thread.
epicsThreadPrivateCreate	Thread private variables are intended for use by legacy libraries written for a single threaded environment and which used 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 and store the index returned. Each thread can later call epicsThreadPrivateGet and epicsThreadPrivateSet with that index to access a thread-specific pointer store.
epicsThreadPrivateDelete	Delete a thread private variable.
epicsThreadPrivateSet	Set the value for a thread private pointer.
epicsThreadPrivateGet	Get the value of a thread private pointer. The value returned is the last value given to epicsThreadPrivateSet by the same thread. If called before epicsThreadPrivateSet the pointer's value is NULL.

The epicsThread API 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 rely on the existence of multiple thread 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 - Every epicsThread has an Id which gets returned by epicsThreadCreate and is valid as long as that thread still exists. A value of 0 always means no thread. If a threadId is used after the thread has terminated, the results are not defined (but will normally lead to bad things happening). Thus code that looks after other threads MUST be aware of threads terminating.

20.16.2 C++ Interface

```
class epicsThreadRunable {
public:
    virtual void run() = 0;
    virtual void stop();
    virtual void show(unsigned int level) const;
};

class epicsShareClass epicsThread {
public:
    epicsThread (epicsThreadRunable &,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 a wrapper around the C interface. Two differences are the method `start` and the class `epicsThreadRunable`.

The `start` method must not be called until after the `epicsThread` constructor has returned. Calling the `start` method allows the `run` method of the `epicsThreadRunable` object to be executed in the context of the new thread.

Code using the C++ API must provide a class that derives from `epicsThreadRunable`. For example:

```
class myThread: public epicsThreadRunable {
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()
{
    // ...
}
```

20.17 epicsTime

`epicsTime.h` contains C++ and C descriptions for time.

20.17.1 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:
  - `lui` - Number of seconds since 1900 (The NTP epoch)
  - `luf` - Fraction of a second. For example 0x800000000 represents 1/2 second.
- `local_tm_nano_sec` and `gm_tm_nano_sec` - Defined by epics. These just add 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.

20.17.2 C++ Interface

```
class epicsTime;

class epicsTimeEvent {
public:
    epicsTimeEvent (const int &number);
    operator int () const;
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 and from ANSI Cs "struct tm" (with nano seconds)
    // adjusted for GM time (UTC)
    operator gm_tm_nano_sec () const;
    epicsTime ( const gm_tm_nano_sec & );
    epicsTime & operator = ( const gm_tm_nano_sec & );

    // 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;

private:
    ...
};
```

20.17.3 class epicsTimeEvent

```
class epicsShareClass epicsTimeEvent
{
public:
    epicsTimeEvent (const int &number);
    operator int () const;
private:
    int eventNumber;
};
```

Method	Meaning

Convert to/from integer	Does not currently check that the range of the integer is valid, although it might one day.

20.17.4 class epicsTime

epicsTime Method	Meaning
epicsTime()	The default constructor sets the time to the beginning of the epics epoch.
epicsTime(const epicsTime &t)	Copy constructor, copies the time from its argument.
static getEvent	Returns an epicsTime indicating when the associated event last occurred. See the description of the C routine epicsTimeGetEvent below for details.
static getCurrent	Returns an epicsTime containing the current time. For example: `epicsTime now = epicsTime::getCurrent();`
convert to/from epicsTimeStamp	Three epicsTime methods interface with epicsTimeStamp values: A constructor, an assignment operator, and a conversion operator. For example: `epicsTimeStamp ts1 = {12345, 67890}, ts2;` `epicsTime t1(ts1), t2;` // *constructor* `t2 = ts1;` // *assignment* `ts2 = t1;` // *conversion operator*

convert to/from time_t	The structure time_t_wrapper is used instead of time_t directly to avoid undesired conversions to integer types. Three methods are provided for ANSI time_t: A constructor, an assignment operator, and a conversion operator. Assume the following definitions:
	`time_t tt;`
	`time_t_wrapper ttw;`
	`epicsTime time;`
	An example of the copy constructor is:
	`ttw.ts = tt;`
	`epicsTime time1(ttw);`
	An example of the assignment operator is:
	`time = ttw;`
	An example of the time_t_wrapper operator is:
	`ttw = time;`
	`tt = ttw.ts;`

convert to/from struct tm in local timezone	The structure local_tm_nano_sec is used instead of struct tm directly to add a nanoseconds field and indicate the time is expressed in the local timezone. Three methods are provided for local_tm_nano_sec: A constructor, an assignment operator, and a conversion operator. Assume the following definitions:
	`local_tm_nano_sec ltn;`
	`epicsTime time;`
	An example of the copy constructor is:
	`epicsTime time1(ltn);`
	An example of the assignment operator is:
	`time = ltn;`
	An example of the local_tm_nano_sec operator is:
	`ltn = time;`

epicsTime Method	Meaning

convert to/from struct tm in UTC	The structure local_tm_nano_sec is used instead of struct tm directly to add a nanoseconds field and indicate the time is expressed in UTC. Three methods are provided for gm_tm_nano_sec: A constructor, an assignment operator, and a conversion operator. Assume the following definitions:
	`gm_tm_nano_sec gtn;`
	`epicsTime time;`
	An example of the copy constructor is:
	`epicsTime time1(gtn);`
	An example of the assignment operator is:
	`time = gtn;`
	An example of the gm_tm_nano_sec operator is:
	`gtn = time;`

convert to/from struct timespec	Three methods are provided for the POSIX struct timespec: A constructor, an assignment operator, and a conversion operator. Assume the following definitions:
	`struct timespec tts;`
	`epicsTime time;`
	An example of the copy constructor is:
	`epicsTime time1(tts);`
	An example of the assignment operator is:
	`time = tts;`
	An example of the conversion operator is:
	`tts = time;`

convert to/from struct timeval	Three methods are provided for BSD's struct timeval: A constructor, an assignment operator, and a conversion operator. Assume the following definitions:
	`struct timeval ttv;`
	`epicsTime time;`
	An example of the copy constructor is:
	`epicsTime time1(ttv);`
	An example of the assignment operator is:
	`time = ttv;`
	An example of the conversion operator is:
	`ttv = time;`

convert to/from NTP timestamps	Three methods are provided for the NTP timestamp structure: A constructor, an assignment operator, and a conversion operator. Assume the following definitions:
	`l_fp ntp;`
	`epicsTime time;`
	An example of the copy constructor is:
	`epicsTime time1(ntp);`
	An example of the assignment operator is:
	`time = ntp;`
	An example of the conversion operator is:
	`ntp = time;`

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:
	`epicsTime time, time1, time2;`
	`double t1,t2,t3;`
	`...`
	`t1 = time2 - time1;`
	`time = time1 + 4.5;`
	`time = time2 - t3;`
	`time2 += 6.0;`

epicsTime Method	Meaning

Comparison operators ==, |=, <=, <, >=, >	Two epics times can be compared:
	`epicsTime time1, time2;`
	`...`
	`if (time1 <= time2)`
	`...`

strftime	This method is an extension of the standard C library routine strftime. See your OS documentation for details about the standard routine. The epicsTime method adds support for the printing the fractional portion of the time. It searches the format string for the sequence `%0nf` where n is the desired precision, and uses this format to convert the fractional seconds to the requested precision. For example:
	`epicsTime time = epicsTime::getCurrent();`
	`char buf[30];`
	`time.strftime(buf, 30, "%Y-%m-%d %H:%M:%S.%06f");`
	`printf("%s\n", buf);`
	This will print the current time in the format:
	`2001-01-26 20:50:29.813505`

show	Shows the date/time.

20.17.5 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

/* These are implemented in the "generalTime" framework */
int epicsTimeGetCurrent (epicsTimeStamp *pDest);
int epicsTimeGetEvent (epicsTimeStamp *pDest, int eventNumber);

/* These are callable from an Interrupt Service Routine */
int epicsTimeGetCurrentInt(epicsTimeStamp *pDest);
int epicsTimeGetEventInt(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);
int epicsTimeFromGMTM (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.

Note that the `epicsTimeGetCurrent()` and `epicsTimeGetEvent()` routines and their ISR-callable equivalents `epicsTimeGetCurrentInt()` and `epicsTimeGetEventInt()` are now implemented in epicsGeneralTime.c

20.18 osiPoolStatus

`osiPoolStatus.h` contains the following description:

```
int osiSufficentSpaceInPool(void);
```

Method	Meaning
osiSufficentSpaceInPool	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.

20.19 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 otherwise documented.

20.20 Ignoring Posix Signals

`epicsSignal.h` contains the following commented declarations:

```
/*
 * Required to avoid problems with soft IOCs getting SIGHUPs
 * when a Channel Access client disconnects:
 */
void epicsSignalInstallSigHupIgnore ( void );

/*
 * Required to avoid terminating a process which is blocking
 * in a socket send() call when a SIGPIPE signal is generated
 * by the OS:
 */
void epicsSignalInstallSigPipeIgnore ( void );

/*
 * Required only if shutdown() and close() do not interrupt
 * a thread blocking in a socket system call:
 */
```

```
void epicsSignalInstallSigAlarmIgnore ( void );
void epicsSignalRaiseSigAlarm ( struct epicsThreadOSD * );
```

Not otherwise documented.

20.21 OS-Independent Socket API

The header file `osiSock.h` provides wrappers around the different socket APIs provided by the supported operating systems. This API was designed to make it possible to write network applications that will compile and run on any OS. See the comments and declarations in the header file for details.

20.22 epicsMMIO

`epicsMMIO.h` provides a set of calls to perform safe access to Memory Mapped I/O regions. This is the typical means to access VME or PCI bus devices.

The following are the equivalant signatures of the MMIO read and write calls. The actual implementations may use macros.

```
epicsUInt8      ioread8 (void* addr);
epicsUInt16 nat_ioread16(void* addr);
epicsUInt16  be_ioread16(void* addr);
epicsUInt16  le_ioread16(void* addr);
epicsUInt32 nat_ioread32(void* addr);
epicsUInt32  be_ioread32(void* addr);
epicsUInt32  le_ioread32(void* addr);

void     iowrite8 (void* addr, epicsUInt8  val);
void nat_iowrite16(void* addr, epicsUInt16 val);
void  be_iowrite16(void* addr, epicsUInt16 val);
void  le_iowrite16(void* addr, epicsUInt16 val);
void nat_iowrite32(void* addr, epicsUInt32 val);
void  be_iowrite32(void* addr, epicsUInt32 val);
void  le_iowrite32(void* addr, epicsUInt32 val);
```

The 16 and 32-bit calls have three variants: `nat_`, `be_`, and `le_` which specify the byte ordering of the MMIO register being access as having: CPU Native, Big Endian, or Little Endian byte order. The specification will be used to re-order the bytes read/written into the CPU native integer format.

Determining which of these variants to use in a specific case requires knowledge of the underlying hardware (bus and/or device). This document can present only a general rule, which is that the `nat_` variant will be used for VME devices as the common bus bridges do automatic byte lane swapping. PCI devices will generally use of one `be_` or `le_`, although some devices have been known to have a mix of BE and LE registers.

These calls do not perform any checking of address alignment.

All of these calls have CPU, OS, and/or compiler specific definitions which try to preserve all MMIO operations by defeating instruction reordering and operation splitting/combining optimizations by the compiler and CPU.

20.23 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.

20.23.1 Overview

`devLib.h` provides definitions for a library of routines useful for device and driver modules, which are primarily indended 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.

20.23.2 Location Probing

20.23.2.1 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.

20.23.2.2 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.

20.23.2.3 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.

20.23.3 Registering VME Addresses

20.23.3.1 Definitions of Address Types

```
typedef enum {
    atVMEA16, atVMEA24, atVMEA32,
    atISA,
    atVMECSR,
```

```
    atLast /* atLast is the last enum in this list */
} epicsAddressType;

char  *epicsAddressTypeName[] = {
    "VME␣A16", "VME␣A24", "VME␣A32",
    "ISA", "VME␣CSR"
};
```

20.23.3.2 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.

20.23.3.3 Print Address Map

```
long  devAddressMap(void)
```

This routine displays the table of registered VME address ranges, including the owner of each registered address.

20.23.3.4 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`.

20.23.3.5 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.

20.23.4 Interrupt Connection Routines

20.23.4.1 Connect

```
long  devConnectInterruptVME(
    unsigned vectorNumber,
    void (*pFunction)(void *),
    void  *parameter);
```

Connect ISR `pFunction` up to the VME interrupt `vectorNumber`.

20.23.4.2 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.

20.23.4.3 Check If Used

```
int devInterruptInUseVME(
    unsigned vectorNumber);
```

Determines if a VME interrupt vector is in use, returning a boolean value.

20.23.4.4 Enable

```
long devEnableInterruptLevelVME(
    unsigned level);
```

Enable the given VME interrupt level onto the CPU.

20.23.4.5 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.

20.23.5 Macros for Normalized Analog Values

20.23.5.1 Convert Digital Value to a Normalized Double Value

```
#define devCreateMask(NBITS)((1<<(NBITS))-1)
#define devDigToNml(DIGITAL,NBITS) \
    (((double)(DIGITAL))/devCreateMask(NBITS))
```

20.23.5.2 Convert Normalized Double Value to a Digital Value

```
#define devNmlToDig(NORMAL,NBITS) \
    (((long)(NORMAL)) * devCreateMask(NBITS))
```

20.23.6 Deprecated Interrupt Routines

20.23.6.1 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.

20.23.6.2 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.

20.23.6.3 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.

20.23.6.4 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.

20.23.6.5 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.

20.24 vxWorks Specific routines and Headers

The routines described in this section are included in a application by the Makfile command:

```
<appl>_OBJS_vxWorks += $(EPICS_BASE_BIN)/vxComLibrary
```

20.24.1 veclist

This routine shows the vxWorks interrupt vector table, but only works properly on 68K family CPUs.

20.24.2 logMsgToErrlog

This traps all calls to logMsg and sends them to errlogPrintf.

20.24.3 camacLib.h

This was included with 3.13.

20.24.4 epicsDynLink

This provides the routines `symFindByNameEPICS` and `symFindByNameAndTypeEPICS`. It is only provided for modules that have not been converted to use `epicsFindSymbol`. These routines are deprecated.

20.24.5 module_types.h

This is only provided for device/driver support that have not been converted to use OSI features of base. This header is deprecated. Instead of using this, drivers should register a configuration command to obtain the information originally provided by `module_types.h`

20.24.6 task_params.h

This is only provided for device/driver support that have not been converted to use OSI features of base. This header is deprecated.

20.24.7 vxComLibrary

This routine causes 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. However on most other systems this routine is not functional, so a registry facility is provided to implement the binding. Any item that is looked up by name at runtime must be registered for it to be visible to other code.

At compile time a perl script reads the IOCs database definition file and produces a C source file containing a routine which registers all record/device/driver/function support defined in that 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 implements the symbol table. Each different type of symbol has its own unique ID. Everything to be registered is stored in the same gpHash table.

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);
```

Provides addresses for both the record support entry table and the routine which computes the size and offset of each field.

21.3 registryDeviceSupport.h

```
int registryDeviceSupportAdd(const char *name, struct dset *pdset)
struct dset *registryDeviceSupportFind(const char *name);
```

This provides addresses for device support entry tables.

21.4 registryDriverSupport.h

```
int registryDriverSupportAdd(const char *name, struct drvet *pdrvet);
struct drvet *registryDriverSupportFind(const char *name);

int registerRecordDeviceDriver(DBBASE *pdbbase);
```

This provides addresses for driver support tables.

21.5 registryFunction.h

```
typedef void (*REGISTRYFUNCTION)(void);

typedef struct registryFunctionRef {
    const char *name;
    REGISTRYFUNCTION addr;
} registryFunctionRef;

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.

If you use these routines to register functions directly instead of using a `function()` statement in a database definition file, the registered functions will not appear in the output from the `dbDumpFunction` command.

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. The following steps are take as part of the standard Make rules:

- Execute this script using a dbd file created by dbExpand
- Compile the resulting C++ file
- Include the object file in the IOC executable

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 was intended to make it easier to understand the IOC source listings, but the information in it is likely to be outdated. It also lists some of the header files provided for interfacing to 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 - Definitions for alarm status and severity values.

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_xxx` and `DBR_xxx` 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

initHooks.h - Definitions used by `initHooks.c` routines.

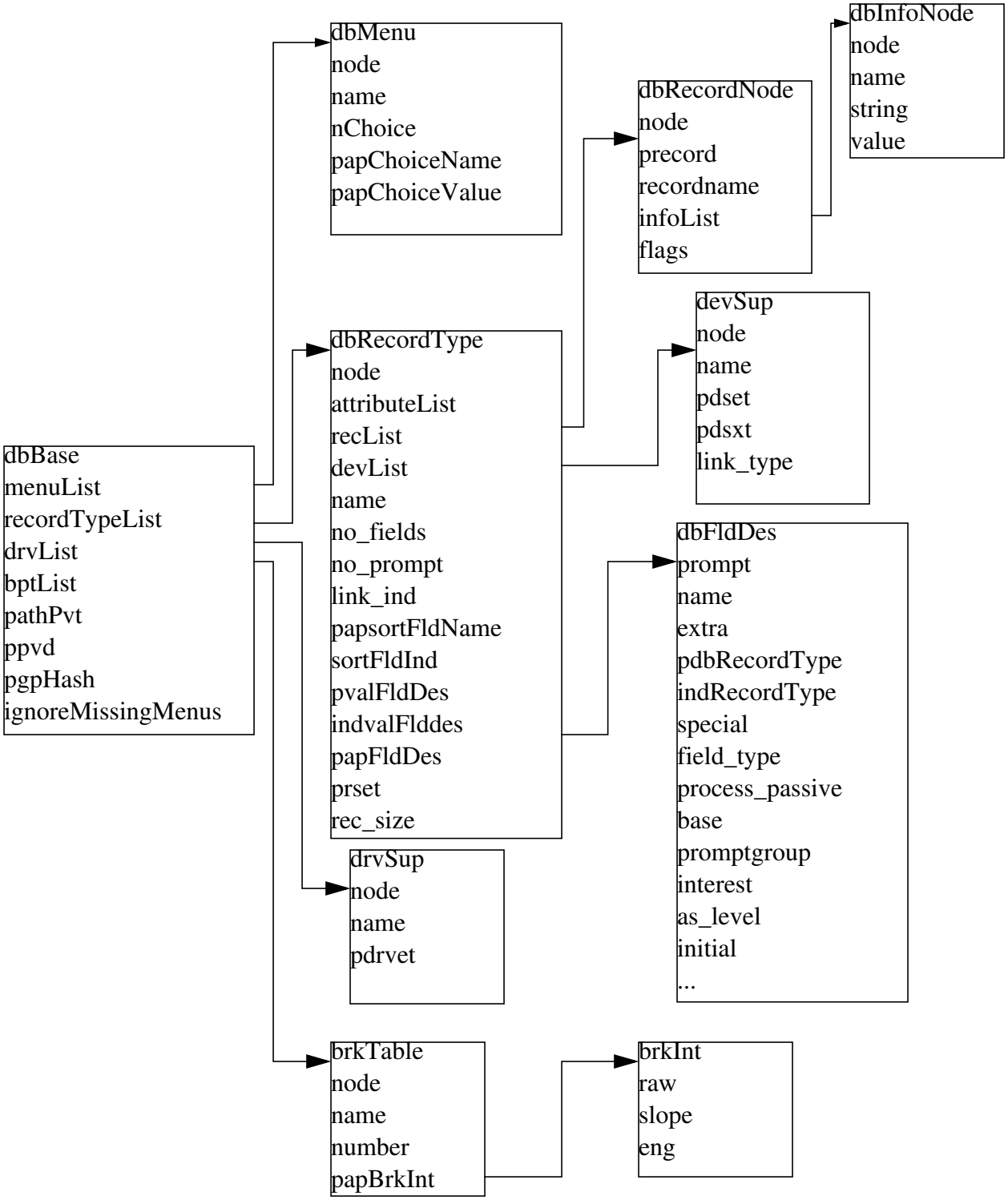
link.h - Link definitions

recSup.h - The record global routines.

special.h - Definitions for special fields, i.e. `SPC_xxx`.

taskwd.h - Task Watchdog System

22.3 Structures



Index