

---

# ***EPICS***

## ***Input / Output Controller (IOC)***

### ***Application Developer's Guide***

**Martin R. Kraimer**  
Argonne National Laboratory  
Advanced Photon Source  
June 1998  
EPICS Release 3.13.0beta12

---

/\*\*\*\*\*\*

#### NOTICE

\*\*\*\*\*

This material resulted from work developed under a U.S. Government contract and is subject to the following license: the Government is granted for itself and the public a paid-up, nonexclusive, irrevocable worldwide license in this material to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly.

\*\*\*\*\*

#### DISCLAIMER

\*\*\*\*\*

NEITHER THE UNITED STATES GOVERNMENT NOR ANY AGENCY THEREOF, NOR ANY OF THEIR EMPLOYEES OR OFFICERS MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION, APPARATUS, PRODUCT, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS

\*\*\*\*\*



---

<b>Preface</b> .....	<b>1</b>
. Overview .....	1
. Acknowledgments .....	2
<b>Chapter 1: EPICS Overview</b> .....	<b>5</b>
. What is EPICS? .....	5
. Basic Attributes .....	6
. Hardware - Software Platforms (Vendor Supplied) .....	6
. IOC Software Components .....	7
. Channel Access .....	9
. OPI Tools .....	10
. EPICS Core Software .....	11
. Getting Started .....	12
<b>Chapter 2: Database Locking, Scanning, And Processing</b> .....	<b>13</b>
. Overview .....	13
. Record Links .....	13
. Database Links .....	14
. Database Locking .....	15
. Database Scanning .....	15
. Record Processing .....	16
. Guidelines for Creating Database Links .....	16
. Guidelines for Synchronous Records .....	19
. Guidelines for Asynchronous Records .....	20
. Cached Puts .....	21
. Channel Access Links .....	22
<b>Chapter 3: Database Definition</b> .....	<b>25</b>
. Overview .....	25
. Definitions .....	25
. Breakpoint Tables .....	38
. Menu and Record Type Include File Generation .....	39
. Utility Programs .....	42
<b>Chapter 4: IOC Initialization</b> .....	<b>47</b>
. Overview .....	47
. iocInit .....	48
. Changing iocCore fixed limits .....	49
. TSconfigure .....	49
. initHooks .....	50
. Environment Variables .....	51
. Initialize Logging .....	51
. Get Resource Definitions .....	52
<b>Chapter 5: Access Security</b> .....	<b>53</b>
. Overview .....	53
. Quick Start .....	53
. User's Guide .....	54
. Design Summary .....	59
. Access Security Application Programmer's Interface .....	61
. Database Access Security .....	65
. Channel Access Security .....	67
. Access Control: Implementation Overview .....	68
. Structures .....	70

---

---

<b>Chapter 6: IOC Test Facilities</b> .....	<b>71</b>
. Overview .....	71
. Database List, Get, Put .....	71
. Breakpoints .....	73
. Error Logging .....	74
. Hardware Reports .....	74
. Scan Reports .....	75
. Time Server Report .....	75
. Access Security Commands .....	76
. Channel Access Reports .....	77
. Interrupt Vectors .....	78
. EPICS .....	78
. Database System Test Routines .....	78
. Record Link Routines .....	79
. Old Database Access Testing .....	80
. Routines to dump database information .....	80
<b>Chapter 7: IOC Error Logging</b> .....	<b>83</b>
. Overview .....	83
. Error Message Routines .....	84
. errlog Task .....	85
. Status Codes .....	86
. iocLog .....	87
<b>Chapter 8: Record Support</b> .....	<b>89</b>
. Overview .....	89
. Overview of Record Processing .....	89
. Record Support and Device Support Entry Tables .....	90
. Example Record Support Module .....	91
. Record Support Routines .....	97
. Global Record Support Routines .....	100
<b>Chapter 9: Device Support</b> .....	<b>103</b>
. Overview .....	103
. Example Synchronous Device Support Module .....	104
. Example Asynchronous Device Support Module .....	105
. Device Support Routines .....	107
<b>Chapter 10: Driver Support</b> .....	<b>109</b>
. Overview .....	109
. Device Drivers .....	109
<b>Chapter 11: Static Database Access</b> .....	<b>113</b>
. Overview .....	113
. Definitions .....	113
. Allocating and Freeing DBBASE .....	114
. DBENTRY Routines .....	115
. Read and Write Database .....	116
. Manipulating Record Types .....	117
. Manipulating Field Descriptions .....	118
. Manipulating Record Attributes .....	118
. Manipulating Record Instances .....	119
. Manipulating Menu Fields .....	120
. Manipulating Link Fields .....	121

---

---

. Manipulating MenuForm Fields . . . . .	122
. Find Breakpoint Table . . . . .	123
. Dump Routines . . . . .	123
. Examples . . . . .	124
<b>Chapter 12: Runtime Database Access . . . . .</b>	<b>127</b>
. Overview . . . . .	127
. Database Include Files . . . . .	127
. Runtime Database Access Overview . . . . .	129
. Database Access Routines . . . . .	132
. Runtime Link Modification . . . . .	139
. Channel Access Monitors . . . . .	139
. Lock Set Routines . . . . .	140
. Channel Access Database Links . . . . .	141
<b>Chapter 13: Device Support Library . . . . .</b>	<b>145</b>
. Overview . . . . .	145
. Registering VME Addresses . . . . .	145
. Interrupt Connect Routines . . . . .	146
. Macros and Routines for Normalized Analog Values . . . . .	146
<b>Chapter 14: EPICS General Purpose Tasks . . . . .</b>	<b>149</b>
. Overview . . . . .	149
. General Purpose Callback Tasks . . . . .	149
. Task Watchdog . . . . .	152
<b>Chapter 15: Database Scanning . . . . .</b>	<b>155</b>
. Overview . . . . .	155
. Scan Related Database Fields . . . . .	155
. Software Components That Interact With The Scanning System . . . . .	156
. Implementation Overview . . . . .	159
<b>Chapter 16: Database Structures . . . . .</b>	<b>163</b>
. Overview . . . . .	163
. Include Files . . . . .	163
. Structures . . . . .	164
<b>INDEX . . . . .</b>	<b>167</b>

---

# *Preface*

---

## **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:

**EPICS Overview** An overview of EPICS is presented, showing how the IOC software fits into EPICS. This is the only chapter that discusses OPI software and Channel Access rather than just IOC related topics.

**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 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 Unix and vxWorks 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.

Database Structures A description of the internal database structures.

Other than the first 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 which is, of course, one of the major IOC components.

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

- *EPICS Record Reference Manual*, Philip Stanley, Janet Anderson and Marty Kraimer  
See LANL Web site for latest version.
- *EPICS IOC Applications: Building and Source Release Control*, Marty Kraimer and Janet Anderson,  
See ANL Web site for latest version.
- *vxWorks Programmer's Guide*, Wind River Systems
- *vxWorks Reference Manual*, Wind River Systems

---

## 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 of LANL/GTA. Bob and Jeff also were the principle implementers of the original IOC software. They developed this software (called GTACS) 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 `dbLoadRecords` and `dbLoadTemplate`.

The `build` utility method resulted in the generation of binary files of UNIX that were loaded into IOCs. As new IOC architectures started being supported this caused problems. During 1995, after learning from an abandoned effort now referred to as `EpicsRX`, the build utilities

and binary file (called `default.dctsdrr`) were replaced by all ASCII files. The new method provides architecture independence and a more flexible environment for configuring the record/device/driver support. This principle implementer was Marty Kraimer with many ideas contributed by John Winans and Jeff Hill. Bob Dalesio made sure that we did not go to 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.

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

**Preface**

Acknowledgments

---

# Chapter 1: EPICS Overview

---

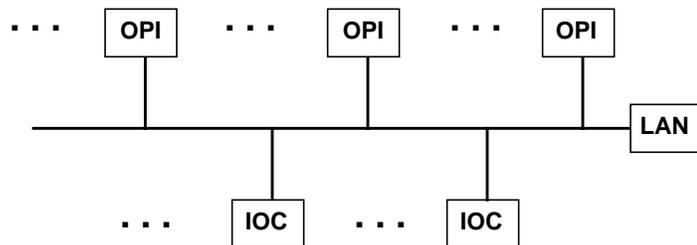
## What is EPICS?

---

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

- **OPI:** Operator Interface. This is a UNIX based workstation which can run various EPICS tools.
- **IOC:** Input/Output Controller. This is a VME/VXI based chassis containing a processor, various I/O modules and VME modules that provide access to other I/O buses such as GPIB.
- **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.

---

## 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.

---

## Hardware - Software Platforms (Vendor Supplied)

---

### OPI

Hardware

- Unix based Workstations: Well supported platforms include SUNOS, SOLARIS, and HP-UX
- Other UNIX platforms have some support, including LINUX
- Limited support is provided for Windows NT and for VMS

Software

- UNIX
- X Windows
- Motif Toolkit

### LAN

Hardware

- Ethernet and FDDI
- ATM in the future

Software

- TCP/IP protocols via sockets

### IOC

Hardware

- VME/VXI bus and crates
- Motorola 68020, 68030, 68040, 68060
- Some support for other processors: Intel, Mips, PowerPC, Sparc, etc.

- Various VME modules (ADCs, DAC, Binary I/O, etc.)
- Allen Bradley Scanner (Most AB I/O modules)
- GPIB devices
- BITBUS devices
- CAMAC
- CANBUS

Software

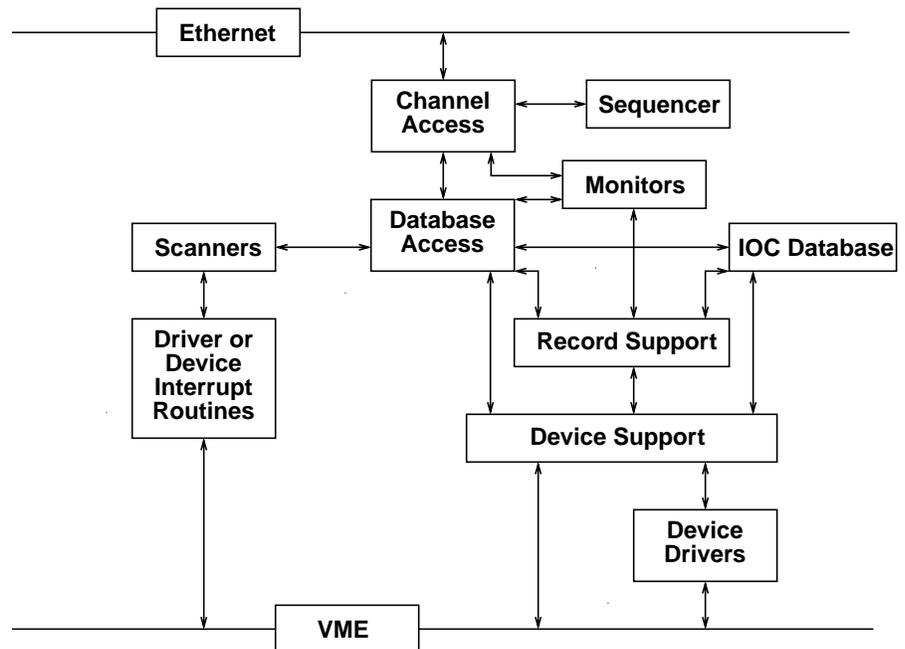
- vxWorks operating system
- Real time kernel
- Extensive “Unix like” libraries

---

## 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.

## 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.

A number of 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.

## 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 12, "Runtime Database Access" on page 127 for details.

## Database Scanning

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

- **Periodic:** A request can be made to process a record periodically. A number of time intervals are supported.
- **Event:** Event scanning is based on the posting of an event by any IOC software component. The actual subroutine call is:  
`post_event(event_num)`
- **I/O Event:** The I/O event scanning system processes records based on external interrupts. An IOC device driver interrupt routine must be available to accept the external interrupts.
- **Passive:** Passive records are processed as a result of linked records being processed or as a result of external changes such as Channel Access puts.
- **Scan Once:** In order to provide for caching puts, The scanning system provides a routine `scanOnce` which arranges for a record to be processed one time.

## Record Support, Device Support and Device Drivers

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

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

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

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

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

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

## Channel Access

Channel Access is discussed in the next section.

## 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.

---

## 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.

## 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.

## Search Server

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

## 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`.

## 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.

---

## 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.

## Channel Access Tools

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

- **MEDM:** Motif version of combined display manager and display editor.
- **DM:** Display Manager. Reads one or more display list files created by EDD, establishes communication with all necessary IOCs, establishes monitors on process variables, accepts operator control requests, and updates the display to reflect all changes.

- **ALH**: Alarm Handler. General purpose alarm handler driven by an alarm configuration file.
- **AR**: Archiver. General purpose tool to acquire and save data from IOCs.
- **Sequencer**: Runs in an IOC and emulates a finite state machine.
- **BURT**: Backup and Restore Tool. General purpose tool to save and restore Channel Access channels. The tool can be run via Unix commands or via a Graphical User Interface.
- **KM**: Knob Manager - Channel Access interface for the sun dials (a set of 8 knobs)
- **PROBE**: Allows the user to monitor and/or change a single process variable specified at run time.
- **CAMATH**: Channel Access interface for Mathematica.
- **CAWINGZ**: Channel Access interface for Wingz.
- **IDL/PVWAVE** Channel Access Interfaces exist for these products.
- **TCL/TK** Channel Access Interface for these products.
- **CDEV** - A library designed to provide a standard API to one or more underlying packages, typically control system interfaces. CDEV provides a Channel Access service.

## Other OPI Tools

- **GDCT**: Graphical Database Configuration Tool. Used to create a run time database for an IOC.
- **EDD**: Display Editor. This tool is used to create a display list file for the Display Manager. A display list file contains a list of static, monitor, and control elements. Each monitor and control element has an associated process variable.
- **SNC**: State Notation Compiler. It generates a C program that represents the states for the IOC Sequencer tool.
- **ASCII Tools** - Tools are provided which generate C include files from menu and record type ASCII definition files.
- **Source/Release**: EPICS provides a Source/Release mechanism for managing EPICS.

---

## 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
- ASCII tools
- Source/Release

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

---

## Getting Started

---

The Document “EPICS IOC Applications: Building and Source Release Control” available via the WWW at [www.aps.anl.gov/asd/controls/epics/EpicsDocumentation/AppDevManuals/iocAppBuildSRcontrol.html](http://www.aps.anl.gov/asd/controls/epics/EpicsDocumentation/AppDevManuals/iocAppBuildSRcontrol.html) gives instructions for building IOC applications. In particular follow the instructions in the section “Quick Start”.

# Chapter 2: Database Locking, Scanning, And Processing

---

## Overview

---

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

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

---

## Record Links

---

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

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

A link to a record in another IOC or a link forced to be a channel access link. Unless the link references the PROC field it is ignored. If it does reference the PROC field a channel access put with a value of 1 is issued.

Links are defined in file `link.h`.

NOTE: This chapter discusses mainly database links.

---

## Database Links

---

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

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

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

### Process Passive

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

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

### Maximize Severity

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

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

## Database Locking

---

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

The following routines are provided for database locking.

```
dbScanLock(precord) ;  
dbScanUnlock(precord) ;
```

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

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

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

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

---

## Database Scanning

---

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

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

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

- **dbScanPassive:** Only record processing routines, `dbGetLink`, `dbPutLink`, and `dbPutField` call `dbScanPassive`. Record processing routines call it for each forward link in the record.
- **dbPutField:** This routine changes the specified field and then, if the field has been declared `process_passive`, calls `dbScanPassive`. Each field of each record type has the attribute `process_passive` declared `TRUE` or `FALSE` in the definition file. This attribute is a global property, i.e. the application developer has no control of it. This use of `process_passive` is used only by `dbPutField`. If `dbPutField` finds the

record already active (this can happen to asynchronous records) and it is supposed to cause it to process, it arranges for it to be processed again, when the current processing completes.

- **dbGetLink:** If the link specifies process passive, this routine calls `dbScanPassive`. Whether or not `dbScanPassive` is called, it then obtains the specified value.
- **dbPutLink:** This routine changes the specified field. Then, if the link specifies process passive, it calls `dbScanPassive`. `dbPutLink` is only called from record processing routines. Note that this usage of `process_passive` is under the control of the application developer. If `dbPutLink` finds the record already active because of a `dbPutField` directed to this record then it arranges for the record to be processed again, when the current processing completes.

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

---

## Record Processing

---

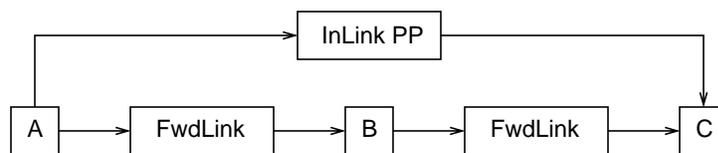
A record is processed as a result of a call to `dbProcess`. Each record support module must supply a routine `process`. This routine does most of the work related to record processing. Since the details of record processing are record type specific this topic is discussed in greater detail in Chapter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

---

## 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:

9. A begins processing. While processing a request is made to process B.
10. B starts processing. While processing a request is made to process C.
11. C starts processing. One of the first steps is to get a value from A via the input link.
12. 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.

13. `C` obtains the value from `A` and completes its processing. Control returns to `B`.
14. `B` completes returning control to `A`
15. `A` completes processing.

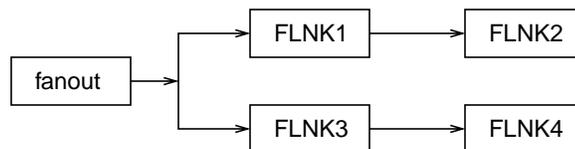
This brief example demonstrates that database links needs more discussion.

## Rules Relating to Database Links

### Processing Order

The processing order is guaranteed to follow the following rules:

1. Forward links are processed in order from left to right and top to bottom. For example the following records are processed in the order `FLNK1`, `FLNK2`, `FLNK3`, `FLNK4` .



2. If a record has multiple input links (calculation and select records) the input is obtained in the natural order. For example if the fields are named `INPA`, `INPB`, ..., `INPL`, then the links are read in the order `A` then `B` then `C`, etc. Thus if obtaining an input results in a record being processed, the processing order is guaranteed.
3. All input and output links are processed before the forward link.

### Lock Sets

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

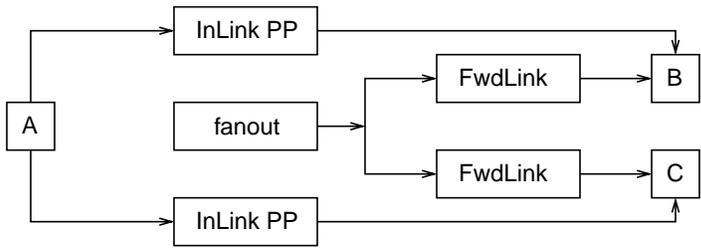
A linked record is not forced to be in the same lock set if all of the following conditions are true.

- The link is an `INLINK` (It is an input link)
- The link is `NPP` (It is no process passive)
- The link is `NMS` (It is no maximize severity)
- The number of elements is `<-1` (The link references a scalar field)

### *PACT* - processing active

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

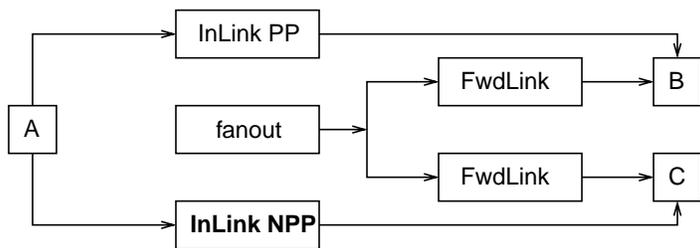
*Process Passive: Link option* Input and output links have an option called process passive. For each such link the application developer can specify process passive TRUE (PP) or process passive FALSE (NPP). Consider the following example



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

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

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



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

*Maximize Severity:  
Link option*

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

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

---

## Guidelines for Synchronous Records

---

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

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

1. A record can be scanned periodically (at one of several rates), via I/O event, or via Event.
2. For each periodic group and for each Event group the phase field can be used to specify processing order.
3. The application programmer has no control over the record processing order of records in different groups.
4. The disable fields (SDIS, DISA, and DISV) can be used to disable records from being processed. By letting the SDIS field of an entire set of records refer to the same input record, the entire set can be enabled or disabled simultaneously. See the Record Reference Manual for details.
5. A record (periodic or other) can be the root of a set of passive records that will all be processed whenever the root record is processed. The set is formed by input, output, and forward links.
6. The `process_passive` option specified for each field of each record determines if a passive record is processed when a `dbPutField` is directed to the field. The application developer must be aware of the possibility of record processing being triggered by external sources if `dbPutFields` are directed to fields that have `process_passive` TRUE.
7. The `process_passive` option for input and output links provides the application developer control over how a set of records are scanned.
8. General link structures can be defined. The application programmer should be wary, however, of defining arbitrary structures without carefully analyzing the processing order.

## Guidelines for Asynchronous Records

---

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

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

9. `pact` is set `TRUE`
10. Data is obtained for all input links
11. Record processing is started
12. The record processing routine returns

The asynchronous completion routine performs the following algorithm:

13. Record processing continues
14. Record specific alarm conditions are checked
15. Monitors are raised
16. Forward links are processed
17. `pact` is set `FALSE`.

A few attributes of the above rules are:

18. Asynchronous record processing does not delay the scanners.
19. Between the time record processing begins and the asynchronous completion routine completes, no attempt will be made to again process the record. This is because `pact` is `TRUE`. The routine `dbProcess` checks `pact` and does not call the record processing routine if it is `TRUE`. Note, however, that if `dbProcess` finds the record active 10 times in succession, it raises a `SCAN_ALARM`.
20. 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.

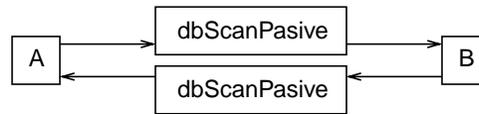
Problems still remain. A few examples are:

### 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.

### Obtain Old Data

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



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

### 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.

### Task Abort

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

---

## 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. Note that it could already be active because it appears twice in a chain of record processing. In this case 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.

---

## Channel Access Links

---

A channel access link is:

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

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

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

For each input link it establishes a channel access monitor. It uses `ca_field_type` and `ca_element_count` when it establishes the monitor. It also monitors the alarm status. Whenever the monitor is invoked the new data is stored in a buffer belonging to `dbCa`. When `iocCore` or the record support module asks for data the data is taken from the buffer and converted to the requested type.

For each output link, a buffer is allocated the first time `iocCore`/record support issues a put and a channel access connection has been made. This buffer is allocated according to `ca_field_type` and `ca_element_count`. Each time `iocCore`/record support issues a put, the data is converted and placed in the buffer and a request is made to `dbCa` to issue a new `ca_put`.

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

Because channel access links imply network activity, they are fundamentally different than database links. For this reason and because channel access does not understand process passive or maximize severity, the semantics of channel access links are not the same as database links. Let's discuss the channel access semantics of `INLINK`, `OUTLINK`, and `FWDLINK` separately.

### **INLINK**

The options for process passive are:

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

Maximize Severity is honored.

## OUTLINK

The options for process passive are:

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

Maximize Severity is not honored.

## FWDLINK

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

The options for process passive are:

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

Maximize Severity is not honored.



# Chapter 3: Database Definition

---

## Overview

---

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

- Menu
- Record Type
- Device
- Driver
- Breakpoint Table
- Record Instance

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

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

This chapter also describes utility programs which operate on these definitions

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

---

## Definitions

---

### Summary

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

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

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

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

driver(drvet_name)
...

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

**#The Following defines a Record Instance**

```
record(record_type,record_name) {
    include "filename"
    field(field_name,"value")
    ...
}
#NOTE: GDCT uses grecord instead of record
```

## General Rules

### *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
breaktable
record
grecord
```

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

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

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

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

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

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

*Escape Sequences* Except for \" the database routines never translate standard C escape sequences, however, a routine `dbTranslateEscape` can be used to translate the standard C escape sequences:

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

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

*dbTranslateEscape* The routine is:

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

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

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

**filename extension** By convention:

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

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

Format:

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

NOTE: In winXX the separator is ; instead of :

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

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

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

The initial path is determined as follows:

- If an initial path is specified, it is used. Else:
- If the environment variable EPICS\_DB\_INCLUDE\_PATH is defined, it is used. Else:
- the default path is ".", i.e. the current directory.

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

## include

Format:

```
include "filename"
```

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

## comment

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

## menu

Format:

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

Where:

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

**choice\_name** - The name placed in the enum generated by dbToMenuH or dbToRecordtypeH

**choice\_value** - The value associated with the choice.

Example:

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

**Record Type**

Format:

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

*rules*

- **asl** - Access Security Level. The default is ASL1. Access Security is discussed in a later chapter. Only two values are permitted for this field (ASL0 and ASL1). Fields which operators normally change are assigned ASL0. Other fields are assigned ASL1. For example, the VAL field of an analog output record is assigned ASL0 and all other fields ASL1. This is because only the VAL field should be modified during normal operations.
- **initial** - Initial Value.
- **promptgroup** - Prompt group to which field belongs. This is for use by Database Configuration Tools. This is defined only for fields that can be given values by database configuration tools. File `guigroup.h` contains all possible definitions. The different groups allow database configuration tools to present the user with groups of fields rather than all prompt fields. I don't know of any tool that currently uses groups.
- **prompt** - A prompt string for database configuration tools. Optional if `promptgroup` is not defined.
- **special** - If specified, then special processing is required for this field at run time.
- **pp** - Should a passive record be processed when Channel Access writes to this field? The default is NO.
- **interest** - Only used by the `dbpr` shell command.
- **base** - For integer fields, a base of DECIMAL or HEX can be specified. The default is DECIMAL.
- **size** - Must be specified for DBF\_STRING fields.
- **extra** - Must be specified for DBF\_NOACCESS fields.
- **menu** - Must be specified for DBF\_MENU fields. It is the name of the associated menu.

*definitions*

- **record\_type** - The unique name of the record type. If duplicates are specified, only the first definition is used.
- **field\_name** - The field name. Only alphanumeric characters are allowed. When include files are generated, the field name is converted to lower case. Previous versions of EPICS required that field name be a maximum of four characters. Although this

restriction no longer exists, problems may arise with some Channel Access clients if longer field names are chosen.

- **field\_type** - This must be one of the following values:
  - DBF\_STRING
  - DBF\_CHAR
  - DBF\_UCHAR
  - DBF\_SHORT
  - DBF\_USHORT
  - DBF\_LONG
  - DBF\_ULONG
  - DBF\_FLOAT
  - DBF\_DOUBLE
  - DBF\_ENUM
  - DBF\_MENU
  - DBF\_DEVICE
  - DBF\_INLINK
  - DBF\_OUTLINK
  - DBF\_FWDLINK
  - DBF\_NOACCESS
- **asl\_level** - This must be one of the following values:
  - ASL0
  - ASL1 (default value)
- **init\_value** - A legal value for data type.
- **prompt\_value** - A prompt value for database configuration tools.
- **gui\_group** - This must be one of the following:
  - GUI\_COMMON
  - GUI\_ALARMS
  - GUI\_BITS1
  - GUI\_BITS2
  - GUI\_CALC
  - GUI\_CLOCK
  - GUI\_COMPRESS
  - GUI\_CONVERT
  - GUI\_DISPLAY
  - GUI\_HIST
  - GUI\_INPUTS
  - GUI\_LINKS
  - GUI\_MBB
  - GUI\_MOTOR
  - GUI\_OUTPUT
  - GUI\_PID
  - GUI\_PULSE
  - GUI\_SELECT
  - GUI\_SEQ1
  - GUI\_SEQ2
  - GUI\_SEQ3

- GUI\_SUB
- GUI\_TIMER
- GUI\_WAVE
- GUI\_SCAN

NOTE: GUI types were invented with the intention of allowing database configuration tools to prompt for groups of fields and when a user selects a group the fields within the group. This feature has never been used and a result is that many record types have not assigned the correct GUI groups to each field.

- **special\_value** must be one of the following:
  - An integer value greater than 103. In this case, the record support special routine is called whenever the field is modified by database access. This feature is present only for compatibility. New support modules should use SPC\_MOD.

The following value disallows access to field.

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

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

- SPC\_SCAN - Scan related field.
- SPC\_ALARMACK - Alarm acknowledgment field.
- SPC\_AS - Access security field.

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

- SPC\_DBADDR - This is set if the record support cvt\_dbaddr routine should be called whenever dbNameToAddr is called, i.e. when code outside record/device support want to access the field.

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

- SPC\_MOD - Notify when modified, i.e. call the record support special routine whenever the field is modified by database access.
- SPC\_RESET - a reset field is being modified.
- SPC\_LINCONV - A linear conversion field is being modified.
- SPC\_CALC - A calc field is being modified.
- **pp\_value** - Should a passive record be processed when Channel Access writes to this field? The allowed values are:
  - NO (default)
  - YES
- **interest\_level** - An interest level for the dbpr command.
- **base** - For integer type fields, the default base. The legal values are:
  - DECIMAL (Default)
  - HEX
- **size\_value** - The number of characters for a DBF\_STRING field.
- **extra\_info** - For DBF\_NOACCESS fields, this is the C language definition for the field. The definition must end with the fieldname in lower case.

*Example*

The following is the definition of the binary input record.

```
recordtype(bi) {
  include "dbCommon.dbd"
  field(INP,DBF_INLINK) {
    prompt("Input Specification")
    promptgroup(GUI_INPUTS)
    special(SPC_NOMOD)
    interest(1)
  }
  field(VAL,DBF_ENUM) {
    prompt("Current Value")
    asl(ASL0)
    pp(TRUE)
  }
  field(ZSV,DBF_MENU) {
    prompt("Zero Error Severity")
    promptgroup(GUI_ALARMS)
    pp(TRUE)
    interest(1)
    menu(menuAlarmSevr)
  }
  field(OSV,DBF_MENU) {
    prompt("One Error Severity")
    promptgroup(GUI_BITS1)
    pp(TRUE)
    interest(1)
    menu(menuAlarmSevr)
  }
  field(COSV,DBF_MENU) {
    prompt("Change of State Svr")
    promptgroup(GUI_BITS2)
    pp(TRUE)
    interest(1)
    menu(menuAlarmSevr)
  }
  field(ZNAM,DBF_STRING) {
    prompt("Zero Name")
    promptgroup(GUI_CALC)
    pp(TRUE)
    interest(1)
    size(20)
  }
  field(ONAM,DBF_STRING) {
    prompt("One Name")
    promptgroup(GUI_CLOCK)
    pp(TRUE)
    interest(1)
    size(20)
  }
  field(RVAL,DBF_ULONG) {
    prompt("Raw Value")
    pp(TRUE)
  }
}
```

```

    }
    field(ORAW,DBF_ULONG) {
        prompt("prev Raw Value")
        special(SPC_NOMOD)
        interest(3)
    }
    field(MASK,DBF_ULONG) {
        prompt("Hardware Mask")
        special(SPC_NOMOD)
        interest(1)
    }
    field(LALM,DBF_USHORT) {
        prompt("Last Value Alarmed")
        special(SPC_NOMOD)
        interest(3)
    }
    field(MLST,DBF_USHORT) {
        prompt("Last Value Monitored")
        special(SPC_NOMOD)
        interest(3)
    }
    field(SIOL,DBF_INLINK) {
        prompt("Sim Input Specifctn")
        promptgroup(GUI_INPUTS)
        special(SPC_NOMOD)
        interest(1)
    }
    field(SVAL,DBF_USHORT) {
        prompt("Simulation Value")
    }
    field(SIML,DBF_INLINK) {
        prompt("Sim Mode Location")
        promptgroup(GUI_INPUTS)
        special(SPC_NOMOD)
        interest(1)
    }
    field(SIMM,DBF_MENU) {
        prompt("Simulation Mode")
        interest(1)
        menu(menuYesNo)
    }
    field(SIMS,DBF_MENU) {
        prompt("Sim mode Alarm Svrty")
        promptgroup(GUI_INPUTS)
        interest(2)
        menu(menuAlarmSevr)
    }
}

```

**device**

This definition defines a single device support module.

```

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

```

*definitions*

- **record\_type** - Record type. The combination of `record_type` and `choice_string` must be unique. If the same combination appears multiple times, the first definition is used.
- **link\_type** - Link type. This must be one of the following:
  - CONSTANT
  - PV\_LINK
  - VME\_IO
  - CAMAC\_IO
  - AB\_IO
  - GPIB\_IO
  - BITBUS\_IO
  - INST\_IO
  - BBGPIB\_IO
  - RF\_IO
  - VXI\_IO
- **dset\_name** - The exact name of the device support entry table without the trailing "DSET". Duplicates are not allowed.
- **choice\_string** - Choice string for database configuration tools. Note that it must be enclosed in "". Note that for a given record type, each `choice_string` must be unique.

*Examples*

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

**driver**

Each driver definition contains the name of a driver entry table. It has the form:

```
driver(drvet_name)
```

*Definitions*

- **drvet\_name** - If duplicates are defined, only the first is used.

*Examples*

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

**breakpoint table**

This defines a breakpoint table.

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

*Definitions*

- **name** - Name of breakpoint table. If duplicates are specified only 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.

*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
}

```

**record instance** Each record instance has the following definition:

```

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

```

*definitions*

- **record\_type** - The record type.
- **record\_name** - The record name. This must be composed of the following characters:  
a-z A-Z 0-9 \_ - : [ ] < > ;  
NOTE: If macro substitutions are used the name must be quoted.  
If duplicate definitions are given for the same record, then the last value given for each field is the value assigned to the field.
- **field\_name** - The field name
- **value** - Depends on field type.
  - DBF\_STRING  
Any ASCII string. If it exceeds the field length, it will be truncated.
  - DBF\_CHAR, DBF\_UCHAR, DBF\_SHORT, DBF\_USHORT, DBF\_LONG, DBF\_ULONG  
A string that represents a valid integer. The standard C conventions are applied, i.e. a leading 0 means the value is given in octal and a leading 0x means that value is given in hex.
  - DBF\_FLOAT, DBF\_DOUBLE  
The string must represent a valid floating point number.
  - DBF\_MENU  
The string must be one of the valid choices for the associated menu.
  - DBF\_DEVICE  
The string must be one of the valid device choice strings.
  - DBF\_INLINK, DBF\_OUTLINK  
The allowed value depends on the bus type of the associated DTYP field. These are as follows:  
NOTE: a DTYP of CONSTANT can be either a constant or a PV\_LINK.
    - CONSTANT  
A constant valid for the field associated with the link.
    - PV\_LINK  
A value of the form:

```

record.field process maximize

```

*field*, *process*, and *maximize* are optional.  
The default value for *field* is VAL.  
*process* can have one of the following values:
      - NPP - No Process Passive (Default)

- PP - Process Passive
- CA - Force link to be a channel access link
- CP - CA and process on monitor
- CPP - CA and process on monitor if record is passive

NOTES:

CP and CPP are valid only for INLINKs.

FWD\_LINKs can be PP or CA. If a FWD\_LINK is a channel access link it must reference the PROC field.

maximize can have one of the following values

- NMS - No Maximize Severity (Default)
- MS - Maximize severity

- VME\_IO

#Ccard Ssignal @parm

where:

card - the card number of associated hardware module.

signal - signal on card

parm - An arbitrary character string of up to 31 characters.

This field is optional and is device specific.

- CAMAC\_IO

#Bbranch Ccrate Nstation Asubaddress Ffunction @parm

branch, crate, station, subaddress, and function should be obvious to camac users. Subaddress and function are optional (0 if not given). Parm is also optional and is device dependent (25 characters max).

- AB\_IO

#Llink Aadapter Ccard Ssignal @parm

link - Scanner, i.e. vme scanner number

adapter - Adapter. Allen Bradley also calls this rack

card - Card within Allen Bradley Chassis

signal - signal on card

parm - An optional character string that is device dependent(27 char max)

- GPIB\_IO

#Llink Aaddr @parm

link - gpib link, i.e. interface

addr - GPIB address

parm - device dependent character string (31 char max)

- BITBUS\_IO

#Llink Nnode Pport Ssignal @parm

link - link, i.e. vme bitbus interface.

node - bitbus node

port - port on the node

signal - signal on port

parm - device specific character string(31 char max)

- INST\_IO

@parm

parm - Device dependent character string(35 char max)

- BBGPIB\_IO

#Llink Bbbaddr Ggpibaddr @parm

link - link, i.e. vme bitbus interface.

bbaddr - bitbus address

gpibaddr - gpib address

parm - optional device dependent character string(31 char max)

- RF\_IO

- #Rcryo Mmicro Ddataset Element
- VXI\_IO
  - #Vframe Cslot Ssignal @parm (Dynamic addressing)
  - or
  - #Vla Signal @parm (Static Addressing)
  - frame - VXI frame number
  - slot - Slot within VXI frame
  - la - Logical Address
  - signal - Signal Number
  - parm - device specific character string(25 char max)
- DBF\_FWDLINK
  - This is either not defined or else is a PV\_LINK. See above for definitions.

### 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")
}
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")
}

```

### record attribute

Each record type can have a set of record attributes. Each attribute is a “psuedo” field that can be accessed via database and channel access. An attribute is given a name the acts like a field name which has the same value for every instance of the record type. Two attributes are generated automatically for each record type: RTYP and VERS. The value for RTYP is the record type name. The default value for VERS is "none specified", which can be changed by record support. Record support can call the following routine to create new attributes or change existing attributes:

```
long dbPutAttribute(char *recordTypename,  
                  char *name, char*value)
```

The arguments are:

`recordTypename` - The name of recordtype.  
`name` - The attribute name, i.e. the psuedo field name.  
`value` - The value assigned to the attribute.

---

## Breakpoint Tables

---

The menu `menuConvert` is handled specially by the `ai` and `ao` records (field is `LINR`). These records allow raw data to be converted to/from engineering units via one of the following:

1. No Conversion.
2. Linear Conversion.
3. Breakpoint table.

Other record types can also use this feature. The first two choices specify no conversion and linear conversion. 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/bpt`
- Add definitions for new breakpoint tables to the end
- Make sure modified `menuConvert.dbd` is loaded into the IOC instead of EPICS version.

Please note that 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:** ASCII string specifying breakpoint table name

- **Low Value Eng:** Engineering Units Value for first breakpoint table entry
- **Low Value Raw:** Raw value for first breakpoint table entry
- **High Value Eng:** Engineering Units: Highest Value desired
- **High Value Raw:** Raw Value for High Value Eng
- **Error:** Allowed error (Engineering Units)
- **First Table:** Engineering units corresponding to first data table entry
- **Last Table:** Engineering units corresponding to last data table entry
- **Delta Table:** Change in engineering units per data table entry

An example definition is:

```
"TypeKdegF" 32 0 1832 4095 1.0 -454 2500 1  
<data table>
```

The breakpoint table can be generated by executing

```
makeBpt bptXXX.data
```

The input file must have the extension of data. The output filename is the same as the input filename with the extension of dbd.

Another way to create the breakpoint table is to include the following definition in a Makefile.Vx:

```
BPTS += bptXXX.dbd
```

NOTE: This requires the naming convention that all data tables are of the form bpt<name>.data and a breakpoint table bpt<name>.dbd.

---

## Menu and Record Type Include File Generation.

---

### Introduction

Given a file containing menus, dbToMenuH generates an include file that can be used by any code which uses the associated menus. Given a file containing any combination of menu definitions and record type definitions, dbToRecordtypeH generates an include file that can be used by any code which uses the menus and record type.

EPICS base uses the following conventions for managing menu and recordtype definitions. Users generating local record types are encouraged to do likewise.

- Each menu that is either for fields in database common (for example menuScan) or is of global use (for example menuYesNo) is defined in a separate file. The name of the file is the same as the menu name with an extension of dbd. The name of the generated include file is the menu name with an extension of h. Thus menuScan is defined in a file menuScan.dbd and the generated include file is named menuScan.h
- Each record type definition is defined in a separate file. In addition, this file contains any menu definitions that are used only by that record type. The name of the file is the same as the recordtype name followed by Record.dbd. The name of the generated include file is the same name with an extension of h. Thus aoRecord is defined in a file aoRecord.dbd and the generated include file is named aoRecord.h. Since aoRecord has a private menu called aoOIF, the dbd file and the generated include file have definitions for this menu. Thus for each record type, there are two source files (xxxRecord.dbd and xxxRecord.c) and one generated file (xxxRecord.h).

Before continuing, it should be mentioned that Application Developers don't have to execute `dbToMenuH` or `dbToRecordtypeH`. If a developer uses the proper naming conventions, it is only necessary to add definitions to their `Makefile.Vx`. The definitions are:

```
MENUS += menuXXX.h (menus)
RECTYPES += xxRecord.h (recordtype & record specific menus)
USER_DBDFLAGS += -I dir
USER_DBDFLAGS += -S macsub
```

Consult the document on building IOC applications for details.

### **dbToMenuH**

This tool is executed as follows:

```
dbToMenuH -Idir -Smacsub menuXXX.dbd
```

It generates a file which has the same name as the input file but with an extension of `h`. Multiple `-I` options can be specified for an include path and multiple `-S` options for macro substitution.

### Example

`menuPriority.dbd`, which contains the definitions for processing priority contains:

```
menu(menuPriority) {
    choice(menuPriorityLOW, "LOW")
    choice(menuPriorityMEDIUM, "MEDIUM")
    choice(menuPriorityHIGH, "HIGH")
}
```

The include file, `menuPriority.h`, generated by `dbToMenuH` contains:

```
#ifndef INCmenuPriorityH
#define INCmenuPriorityH
typedef enum {
    menuPriorityLOW,
    menuPriorityMEDIUM,
    menuPriorityHIGH,
}menuPriority;
#endif /*INCmenuPriorityH*/
```

Any code that needs to use the priority menu values should use these definitions.

### **dbToRecordtypeH**

This tool is executed as follows:

```
dbToRecordtypeH -Idir -Smacsub xxxRecord.dbd
```

It generates a file which has the same name as the input file but with an extension of `h`. Multiple `-I` options can be specified for an include path and multiple `-S` options for macro substitution.

### Example

`aoRecord.dbd`, which contains the definitions for the analog output record contains:

```
menu(aoOIF) {
    choice(aoOIF_Full, "Full")
    choice(aoOIF_Incremental, "Incremental")
}
recordtype(ao) {
    include "dbCommon.dbd"
    field(VAL, DBF_DOUBLE) {
```

```

        prompt("Desired Output")
        asl(ASL0)
        pp(TRUE)
    }
    field(OVAL,DBF_DOUBLE) {
        prompt("Output Value")
    }
    ... (Many more field definitions)
}

```

The include file, aoRecord.h, generated by dbToRecordtypeH contains:

```

#include <vxWorks.h>
#include <semLib.h>
#include "ellLib.h"
#include "fast_lock.h"
#include "link.h"
#include "tsDefs.h"

#ifndef INCaoOIFH
#define INCaoOIFH
typedef enum {
    aoOIF_Full,
    aoOIF_Incremental,
}aoOIF;
#endif /*INCaoOIFH*/
#ifndef INCaoH
#define INCaoH
typedef struct aoRecord {
    char          name[29]; /*Record Name*/
    ... Remaining fields in database common
    double        val;     /*Desired Output*/
    double        oval;    /*Output Value*/
    ... remaining record specific fields
} aoRecord;
#define aoRecordNAME    0
... defines for remaining fields in database common
#define aoRecordVAL     42
#define aoRecordOVAL    43
... defines for remaining record specific fields
#ifdef GEN_SIZE_OFFSET
int aoRecordSizeOffset(dbRecordType *pdbRecordType)
{
    aoRecord *prec = 0;
    pdbRecordType->papFldDes[0]->size=sizeof(prec->name);
    pdbRecordType->papFldDes[0]->offset=
        (short)((char *)&prec->name - (char *)prec);
    ... code to compute size&offset for other fields in dbCommon
    pdbRecordType->papFldDes[42]->size=sizeof(prec->val);
    pdbRecordType->papFldDes[42]->offset=
        (short)((char *)&prec->val - (char *)prec);
    pdbRecordType->papFldDes[43]->size=sizeof(prec->oval);
    pdbRecordType->papFldDes[43]->offset=

```

```
        (short)((char *)&prec->oval - (char *)prec);
    ... code to compute size&offset for remaining fields
    pdbRecordType->rec_size = sizeof(*prec);
    return(0);
}
#endif /*GEN_SIZE_OFFSET*/
```

The analog output record support module and all associated device support modules should use this include file. No other code should use it.

## Discussion of Generated File

Only the analog output record support module and associated device support should include this record definition. Let's discuss the various parts of the file.:

- The enum generated from the menu definition should be used to reference the value of the field associated with the menu.
- The typedef and structure defining the record are used by record support and device support to access fields in an analog output record.
- A #define is present for each field within the record. This is useful for the record support routines that are passed a pointer to a DBADDR structure. They can have code like the following:

```
switch (dbGetFieldIndex(pdbAddr)) {
    case aoRecordVAL :
        ...
        break;
    case aoRecordXXX:
        ...
        break;
    default:
        ...
}
```

The C source routine `aoRecordSizeOffset` is automatically called when a record type file is loaded into an IOC. Thus user code does not have to be aware of this routine except for the following convention: The associate record support module **MUST** include the statements:

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

This convention ensures that the routine is defined exactly once.

---

## Utility Programs

---

### dbExpand

```
dbExpand -I dir -Smacsub file1 file2 ...
```

Multiple `-I` options can be specified for an include path and multiple `-S` options for macro substitution. Note that the environment variable `EPICS_DB_INCLUDE_PATH` can also be used in place of the `-I` options.

NOTE: Host Utility Only

This command reads the input files and then writes, to `stdout`, a file containing ASCII definitions for all information described by the input files. The difference is that comment lines do not appear and all include files are expanded.

This routine is extremely useful if an IOC is not using NFS for the `dbLoadDatabase` commands. It takes more than 2 minutes to load the `base/rec/base.dbd` file into an IOC if NFS is not used. If `dbExpand` creates a local `base.dbd` file, it takes about 7 seconds to load (25 MHZ 68040 IOC).

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

NOTES:

- IOC Only
- Using a path on the ioc does not work very well.
- Both path and substitutions can be null, i.e. they do not have to be given.

This command loads a database file containing any of the definitions given in the summary at the beginning of this chapter.

`dbfile` must be a file containing only *record instances* in standard ASCII format. Such files should have an extension of “.db”.

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

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

Variables are specified in the `dbfile` as `$(variable_name)`. If the substitution string

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

were used, any variables `$(a)`, `$(b)`, `$(c)` would be substituted with the appropriate data.

### EXAMPLE

For example, let `test.db` be:

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

Then issuing the command:

```
dbLoadDatabase("test.db",0,"pre=TEST,STR=test,SCAN=Passive")
```

gives the same results as loading:

```
record(ai,"TESTtestrec1")
record(ai,"TESTtestrec2")
record(stringout,"TESTtestrec3") {
    field(VAL,"test")
    field(SCAN,"Passive")
}
```

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

NOTES:

- IOC Only.
- dbfile must contain only record instances.
- dbLoadRecords is no longer needed. It will probably go away in the future. At the present time dbLoadRecords loads faster than dbLoadDatabase.

### dbLoadTemplate

dbLoadTemplate(char\* template\_def)

dbLoadTemplate reads a template definition file. This file contains rules about loading database instance files, which contain \$(xxx) macros, and performing substitutions.

template\_def contains the rules for performing substitutions on the instance files. For convenience two formats are provided. The format is:

```
file name.db {  
    put Version-1 or Version-2 here  
}
```

Version-1

```
{ set1var1=sub1, set1var2=sub2,..... }  
{ set2var1=sub1, set2var2=sub2,..... }  
{ set3var1=sub1, set3var2=sub2,..... }
```

- or -

Version-2

```
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.db) specifies the record instance input file.

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.

### 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.db {  
    { this=sub1,that=sub2 }  
    { this=sub3,that=sub4 }  
}
```

```
file test.db {  
    pattern{this,that}  
    {sub1,sub2}  
    {sub3,sub4 }
```

Assume that test.db is:

```
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")
}
```

### **dbReadTest**

```
dbReadTest -Idir -Smacsub file.dbd ... file.db ...
```

This utility can be used to check for correct syntax in database definition and database instance files. It just reads all the specified files

Multiple `-I`, and `-S` options can be specified. An arbitrary number of database definition and database instance files can be specified.



# Chapter 4: IOC Initialization

---

## Overview

---

After vxWorks is loaded at IOC boot time, the following commands, normally in a vxWorks startup command file, are issued to load and initialize the control system software:

```
# For many board support packages the following must be added
#cd <full path to target bin directory>
< cdCommands
cd appbin
ld < iocCore
ld < <appname>Lib
cd startup
dbLoadDatabase("<file>.dbd")
dbLoadDatabase("<file>.db")
dbLoadRecords("<file>.db")
    and/or
dbLoadTemplates("<file>.db, "<template_def>")
    . . .
iocInit
```

NOTE: The "IOC Applications: Building and Source/Release Control" manual describes procedures and tools for building IOC applications. This manual should be consulted before creating new startup file.

`cdCommands` defines vxWorks global variables that allow vxWorks `cd` commands for convenient locations. For example in one of my test areas the following `cdCommands` file appears:

```
startup = "/home/phoebus6/MRK/epics/test/iocBoot/iocaccess"
appbin = "/home/phoebus6/MRK/epics/test/bin/mv167"
share = "/home/phoebus6/MRK/iocsys/share"
```

NOTE: This file is automatically generated via make rules.

The first `ld` command loads the core EPICS software. The second command loads the record, device, and driver support plus any other application specific modules.

One or more `dbLoadDatabase` commands load database definition files.

One or more `dbLoadDatabase`, `dbLoadRecords`, and `dbLoadTemplate` commands load record instance definitions.

`iocInit` initializes the various epics components.

---

## **iocInit**

---

`iocInit` performs the following functions:

<b>coreRelease</b>	Prints a messages showing which version of <code>iocCore</code> is being loaded.
<b>getResources</b>	See below. This is obsolete feature.
<b>iocLogInit</b>	Initialize system wide logging facility.
<b>taskwdInit</b>	start the task watchdog task. This task accepts requests to watch other tasks. It runs periodically and checks to see if any of the tasks is suspended. If so it issues an error message. It can also optionally invoke a callback routine
<b>callbackInit</b>	Start the general purpose callback tasks. Three tasks are started with the only difference being scheduling priority.
<b>dbCaLinkInit</b>	Calls <code>dbCaLinkInit</code> . The initializes the task that handles database channel access links.
<b>initDrvSup</b>	<code>InitDrvSup</code> locates each device driver entry table and calls the init routine of each driver.
<b>initRecSup</b>	<code>InitRecSup</code> locates each record support entry table and calls the init routine.
<b>initDevSup</b>	<code>InitDevSup</code> locates each device support entry table and calls the init routine with an argument specifying that this is the initial call.
<b>ts_init</b>	<code>Ts_init</code> initializes the timing system. If a hardware timing board resides in the IOC, hardware timing support is used, otherwise software timing is used. If the IOC has been declared to be a master timer, the initial time is obtained from the UNIX master timer, otherwise the initial time is obtained from the IOC master timer.
<b>initDatabase</b>	<p><code>InitDatabase</code> makes three passes over the database performing the following functions:</p> <ul style="list-style-type: none"> <li>• Pass 1: Initializes following fields: <code>rset</code>, <code>dset</code>, <code>mlis</code>. Calls record support <code>init_record</code> (First pass)</li> <li>• Pass 2: Convert each <code>PV_LINK</code> to <code>DB_LINK</code> or <code>CA_LINK</code></li> <li>• Pass 3: Calls record support <code>init_record</code> (second pass)</li> </ul> <p>After the database is initialized <code>dbLockInitRecords</code> is called. It creates the lock sets.</p>
<b>finishDevSup</b>	<code>InitDevSup</code> locates each device support entry table and calls the init routine with an argument specifying that this is the finish call.
<b>scanInit</b>	The periodic, event, and io event scanners are initialized and started.
<b>interruptAccept</b>	A global variable " <code>interruptAccept</code> " is set <code>TRUE</code> . Until this time no request should be made to process records and all interrupts should be ignored.

---

**initialProcess** dbProcess is called for all records that have PINI TRUE.

**rsrv\_init** The Channel Access server is started

---

## Changing iocCore fixed limits

---

The following commands can be issued after iocCore is loaded to change iocCore fixed limits. The commands should be given before any dbLoad commands are given.

```
callbackSetQueueSize(size)
dbPvdTableSize(size)
scanOnceSetQueueSize(size)
errlogInit(bufferSize)
```

**callbackSetQueueSize** Requests for the general purpose callback tasks are placed in a ring buffer. This command can be used to set the size for the ring buffers. The default is 2000. A message is issued when a ring buffer overflows. It should rarely be necessary to override this default. Normally the ring buffer overflow messages appear when a callback task fails.

**dbPvdTableSize** Record instance names are stored in a process variable directory, which is a hash table. The default number of hash entries is 512. dbPvdTableSize can be called to change the size. It must be called before any dbLoad commands and must be a power of 2 between 256 and 65536. If an IOC contains very large databases (several thousand) then a larger hash table size speeds up searches for records.

**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.

**errlogInit** Thus overrides the default buffer size for the errlog message queue. The default is 1280 bytes.

---

## TSconfigure

---

EPICS supports several methods for an IOC to obtain time so that accurate time stamps can be generated. The default is to obtain NTP time stamps from another computer. The following can be used to change the defaults. If an argument is given the value 0 then the default is applied.

```
TSConfigure(master, sync_rate, clock_rate, master_port, slave_port)
```

- **master:** 1=master timing IOC, 0=slave timing, default is slave.
- **sync\_rate:** The clock sync rate in seconds. This rate tells how often the synchronous time stamp support software will confirm that an IOC clock is synchronized. The default is 10 seconds.
- **clock\_rate:** The frequency in hertz of the clock, the default is 1000Hz for the event system. The value will be set to the IOC's internal clock rate when soft timing is used.
- **master\_port:** UDP port for master. The default is 18233
- **slave\_port:** UDP port for slave.

- **time\_out:** UDP information request time out in milliseconds, if zero is entered here, the default will be used which is 250ms.
- **type:** 0=normal operation, 1=force soft timing type

See "Synchronous Time Stamp Support", by Jim Kowalkowski for details. Note that the default is to be a slave. If no master is found the slave will obtain a starting time from Unix.

---

## initHooks

---

NOTE: starting with release 3.13.0beta12 initHooks was changed drastically (thanks to Benjamin Franksen at BESY). Old initHooks.c functions will still work but users are encouraged to switch to the new method.

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

```
typedef enum {
    initHookAtBeginning,
    initHookAfterGetResources,
    initHookAfterLogInit,
    initHookAfterCallbackInit,
    initHookAfterCaLinkInit,
    initHookAfterInitDrvSup,
    initHookAfterInitRecSup,
    initHookAfterInitDevSup,
    initHookAfterTS_init,
    initHookAfterInitDatabase,
    initHookAfterFinishDevSup,
    initHookAfterScanInit,
    initHookAfterInterruptAccept,
    initHookAfterInitialProcess,
    initHookAtEnd
}initHookState;

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

Any new functions that are registered before iocInit reaches the desired state will be called when iocInit reaches that state. The following is skeleton code to use the facility:

```
#include <vxWorks.h>
#include <stdlib.h>
#include <stddef.h>
#include <initHooks.h>

static initHookFunction myHookFunction;

int myHookInit(void)
{
    return(initHookRegister(myHookFunction));
}
```

```
static void myHookFunction(initHookState state)
{
    switch(state) {
        case initHookAfterInitRecSup:
            ...
            break;
        case initHookAfterInterruptAccept:
            ...
            break;
        default:
            break;
    }
}
```

Assuming the code is in file myHook.c, the st.cmd file should contain (before iocInit).

```
ld < bin/myHook.o
myHookInit
```

An arbitrary number of functions can be registered.

---

## Environment Variables

---

The following environment variables are used by iocCore:

```
EPICS_CA_ADDR_LIST
EPICS_CA_CONN_TMO
EPICS_CA_BEACON_PERIOD
EPICS_CA_AUTO_ADDR_LIST
EPICS_CA_REPEATER_PORT
EPICS_CA_SERVER_PORT
EPICS_TS_MIN_WEST
EPICS_TS_NTP_INET
EPICS_IOC_LOG_PORT
EPICS_IOC_LOG_INET
```

These variables can be overridden via the vxWorks putenv function. For example:

```
putenv("EPICS_TS_MIN_WEST=300")
```

Any putenv commands should be issued after iocCore is loaded and before any dbLoad commands.

---

## Initialize Logging

---

Initialize the logging system. See chapter "IOC Error Logging" for details. For initialization just realize that the following can be used if you want to use a private host log file.

```
putenv("EPICS_IOC_LOG_PORT=7004")
```

```
putenv("EPICS_IOC_LOG_INET=164.54.8.12")
```

These command must be given immediately after iocCore is loaded.

If you want to disable logging to the system wide log file just give the command.

```
iocLogDisable = 1
```

This must be given after iocCore is loaded and before any dbLoad commands.

---

## Get Resource Definitions

---

NOTE: This facility is supported for compatibility with previous releases. It should NOT be used for new applications.

iocInit accepts a string argument which is the name of a resource file which can set values of IOC global variables. The resource file contains lines with the following format:

```
global_name  type  value
```

global\_name is the name of the variable to be changed.

type must be one of the following:

```
DBF_STRING  
DBF_SHORT  
DBF_LONG  
DBF_FLOAT  
DBF_DOUBLE
```

value is the value to be assigned to the global variable.

Please note that type **MUST** be set so that it matches the actual type of the global variable because there is no way for GetResources to know the actual type.

# Chapter 5: Access Security

---

## Overview

---

This chapter describes access security, i.e. the system that limits access to IOC databases. It consists of the following sections:

1. Overview - This section
2. Quick start - A summary of the steps necessary to start access security.
3. User's Guide - This explains what access security is and how to use it.
4. Design Summary - Functional Requirements and Design Overview.
5. Application Programmer's Interface
6. Database Access Security - Access Security features for EPICS IOC databases.
7. Channel Access Security - Access Security features in Channel Access
8. 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 via the EPICS WWW documentation

---

## Quick Start

---

In order to "turn on" access security for a particular IOC the following must be done:

- Create the access security file.
- IOC databases may have to be modified
  - Record instances may have to have values assigned to field ASG. If ASG is null the record is in group DEFAULT.
  - Access security files can be reloaded after iocInit via a subroutine record with `asSubInit` and `asSubProcess` as the associated subroutines. Writing the value 1 to this record will cause a reload.
- The vxWorks startup file must contain the following command before iocInit.  
`asSetFilename("accessSecurityFile")`  
The following is an optional command.  
`asSetSubstitutions("var1=sub1,var2=sub2,...")`

The following rules decide if access security is turned on for an IOC:

- If `asSetFilename` is not executed before `iocInit`, access security will NEVER be started..

- If `asSetFile` is given and any error occurs while first initializing access security, then ALL access to that ioc is denied.
- If after successfully starting access security, an attempt is made to restart and an error occurs then the previous access security configuration is maintained.

---

## User's Guide

---

### Features

Access security protects IOC databases from unauthorized Channel Access Clients. Access security is based on the following:

- **Who:** Userid of the channel access client.
- **Where:** Hostid where the user is logged on. This is the host on which the channel access client exists. Thus no attempt is made to see if a user is local or is remotely logged on to the host.
- **What:** Individual fields of records are protected. Each record has a field containing the Access Security Group (ASG) to which the record belongs. Each field has an access security level, which must be 0 or 1. The security level is defined in the ascii record definition file. Thus the access security level for a field is the same for all record instances of a record type.
- **When:** Access rules can contain input links and calculations similar to the calculation record.

### Limitations

An IOC database can be accessed only via Channel Access or via the vxWorks shell. It is assumed that access to the local IOC console is protected via physical security and `telnet/rlogin` access protected via normal Unix and physical security.

No attempt has been made to protect against the sophisticated saboteur. Unix security must be used to limit access to the subnet on which the iocs reside.

### 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

### 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.

#### *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.

### Syntax Definition

In the following description:

```
[ ]   Lists optional elements
|     Separator for alternatives
...   Means that an arbitrary number of definitions may be given.
      Any line beginning with # is a comment
```

```
UAG(<name>) [{ <user> [, <user> ...] }]
...
HAG(<name>) [{ <host> [, <host> ...] }]
...
ASG(<name>) [{
  [ INP<index>(<pvname>)
  ... ]
  RULE(<level>,NONE | READ | WRITE) {
    [UAG(<name> [, <name> ...])]
    [HAG(<name> [, <name> ...])]
    CALC("<calculation>")
  }
  ...
}]
...
```

### Discussion

- **UAG:** User Access Group. This is a list of userids. The list may be empty. The same userid can appear in multiple UAGs. For iocs the userid is taken from the user field of the boot parameters.
- **HAG:** Host Access Group. This is a list of host names. It may be empty. The same host name can appear in multiple HAGs. For iocs the host name is taken from the target name of the boot parameters.
- **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).
    - **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. If the calculation results in (0,1) meaning (FALSE,TRUE) then the rule (doesn't apply, does apply) . The actual test is  $.99 < result < 1.01$ .

Each IOC record contains a field ASG, which specifies the name of the ASG to which the record belongs. If this field is null or specifies a group which is not defined in the access security file then the record is placed in group "DEFAULT".

The access privilege for a channel access client is determined as follows:

1. The ASG associated with the record is searched.
2. Each RULE is checked for the following:
  - a. The field's level must be less than or equal to the level for this RULE.
  - b. If UAG is defined, the user must belong to one of the specified UAGs. If UAG is not defined all users are accepted.
  - c. If HAG is defined, the user's host must belong to one one of the HAGs. If HAG is not defined all hosts are accepted.
  - d. If CALC is specified, the calculation must yield the value 1, i.e. TRUE. If any of the INP fields associated with this calculation are in INVALID alarm severity the calculation is considered false. The actual test for TRUE is  $.99 < result < 1.01$ .
3. The maximum access allowed by step 2 is the access chosen.

Multiple RULEs can be defined for a given ASG, even RULEs with identical levels and access permission.

### **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.

### **IOC Access Security Initialization**

In order to have access security turned on during IOC initialization the following command must appear in the startup file before iocInit is called:

```
asSetFilename("<access security file>")
```

If this command does not appear then access security will not be started by iocInit. If an error occurs when iocInit calls asInit than all access to the ioc is disabled, i.e. no channel access client will be able to access the ioc.

Access security also supports macro substitution just like dbLoadDatabase. The following command specifies the desired substitutions:

```
asSetSubstitutions("var1=sub1,var2=sub2,...")
```

This command must be issued before iocInit.

After an IOC is initialized the access security database can be changed. The preferred way is via the subroutine record described in the next section. It can also be changed by issuing the following command to the vxWorks shell:

```
asInit
```

It is also possible to reissue `asSetFilename` and/or `asSetSubstitutions` before `asInit`. If any error occurs during `asInit` the old access security configuration is maintained. It is **NOT** permissible to call `asInit` before `iocInit` is called.

Restarting access security after `ioc` initialization is an expensive operation and should not be used as a regular procedure.

## Database Configuration

### *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`.

### *Subroutine Record Support*

Two subroutines, which can be attached to a subroutine record, are available (provided with `iocCore`):

```
asSubInit
asSubProcess
```

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`.

### *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.

### **Example:**

Lets design a set of rules for a Linac. Assume the following:

1. Anyone can have read access to all fields at anytime.
2. Linac engineers, located in the injection control or control room, can have write access to most level 0 fields only if the Linac is not in operational mode.
3. Operators, located in the injection control or control room, can have write access to most level 0 fields anytime.
4. The operations supervisor, linac supervisor, and the application developers can have write access to all fields but must have some way of not changing something inadvertently.
5. Most records use the above rules but a few (high voltage power supplies, etc.) are placed under tighter control. These will follow rules 1 and 4 but not 2 or 3.
6. IOC channel access clients always have level 1 write privilege.

Most Linac IOC records will not have the ASG field defined and will thus be placed in ASG "DEFAULT". The following records will have an ASG defined:

- `LI:OPSTATE` and any other records that need tighter control have `ASG="critical"`. One such record could be a subroutine record used to cause a new access configuration file to be loaded. `LI_OPSTATE` has the value (0,1) if the Linac is (not operational, operational).
- `LI:levlpermit` has `ASG="permit"`. In order for the `opSup`, `linacSup`, or an `appDev` to have write privilege to everything this record must be set to the value 1.

The following access configuration satisfies the above rules.

```
UAG(op) {op1,op2,superguy}
UAG(opSup) {superguy}
UAG(linac) {waw,nassiri,grelick,berg,fuja,gsm}
UAG(linacSup) {gsm}
UAG(appDev) {nda,kko}
HAG(icr) {silver,phebos,gaea}
HAG(cr) {mars,hera,gold}
HAG(ioc)
{ioclic1,ioclic2,ioclid1,ioclid2,ioclid3,ioclid4,ioclid5}
ASG(DEFAULT) {
  INPA(LI:OPSTATE)
  INPB(LI:levlpermit)
  RULE(0,WRITE) {
    UAG(op)
    HAG(icr,cr)
    CALC("A=1")
  }
  RULE(0,WRITE) {
    UAG(op,linac,appdev)
    HAG(icr,cr)
    CALC("A=0")
  }
  RULE(1,WRITE) {
    UAG(opSup,linacSup,appdev)
    CALC("B=1")
  }
  RULE(1,READ)
  RULE(1,WRITE) {
    HAG(ioc)
  }
}
ASG(permit) {
  RULE(0,WRITE) {
    UAG(opSup,linacSup,appDev)
  }
  RULE(1,READ)
  RULE(1,WRITE) {
    HAG(ioc)
  }
}
ASG(critical) {
  INPB(LI:levlpermit)
```

```

RULE(1,WRITE) {
    UAG(opSup,linacSup,appdev)
    CALC("B=1")
}
RULE(1,READ)
RULE(1,WRITE) {
    HAG(ioc)
}
}

```

---

## Design Summary

---

### 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

### Additional Requirements

#### *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.

#### *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.

#### *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.

#### *Defaults*

It must be possible to easily define default access rules.

*Access Security is Optional* When an IOC is initialized, access security is optional.

**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.

*Configuration File* User access groups, host access groups, and access security groups are configured via an ASCII file.

*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.

*IOC Database Access Security* The interface between an IOC database and the access security system.

*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`.

Channel access is responsible for implementing the requirement of allowing the user to be changed dynamically.

**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.

**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 <code>iocInit</code>	4,244,520	4,860,840
After <code>iocInit</code>	4,995,416	5,964,904
After <code>caput</code>	5,449,780	6,658,868
After <code>caget</code>	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.

---

## Access Security Application Programmer's Interface

---

### Definitions

```
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);
```

### Initialization

```
long asInitialize(ASINPUTFUNCPtr inputFunction)
long asInitFile(const char *filename,const char *substitutions)
long asInitFP(FILE *fp,const char *substitutions)
```

These routines read an access definition file and perform all initialization necessary. The caller must provide a routine to provide input lines for `asInitialize`. `asInitFile` and `asInitFP` do their own input and also perform macro substitutions.

The initialization routines can be called multiple times. If an access system already exists the old definitions are removed and the new one initialized. Existing members are placed in the new ASGs.

### Group manipulation

*add Member*                    `long asAddMember(ASMEMBERPVT *ppvt, 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.

***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.

***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

***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.

***change Group***

```
long asChangeGroup(ASMEMBERPVT *ppvt, 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.

**Client Manipulation*****add Client***

```
long asAddClient(ASCLIENTPVT *ppvt, ASMEMBERPVT pvt, int asl,  
                char *user, char *host);
```

This routine adds a client to an ASG member. The calling routine must provide storage for `ASCLIENTPVT`. `ASMEMBERPVT` is the value that was set by calling `asAddMember`. `asl` is the access security level.

The caller must provide permanent storage for `user` and `host`.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

*change Client*                    `long asChangeClient(ASCLIENTPVT ppvt,int asl,  
                                  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.

*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.

*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.

*put Client Pvt*                    `void asPutClientPvt(ASCLIENTPVT pvt, void *userPvt);`

This routine is used to set the pointer returned by `asGetClientPvt`.

*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.

*check Get*                        `long asCheckGet(ASCLIENTPVT pvt);`

This routine, actually a macro, returns (`TRUE,FALSE`) if the client (has, doesn't have) get access rights.

*check Put*                        `long asCheckPut(ASCLIENTPVT pvt);`

This routine, actually a macro, returns (`TRUE,FALSE`) if the client (has, doesn't have) put access rights

## **Access Computation**

*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.

***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.

***compute access rights***

```
long asCompute(ASCLIENTPVT pvt);
```

This routine computes the access rights of a client. This routine is normally called by the access library itself rather than use code.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

**Diagnostic*****dump***

```
int asDump(void (*member)(ASMEMBERPVT),  
           void (*client)(ASCLIENTPVT),int verbose);
```

This routine prints 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.

***dump UAG***

```
int asDumpUag(char *uagname)
```

This routine displays the specified UAG or if `uagname` is NULL each UAG defined in the access security database.

***dump HAG***

```
int asDumpHag(char *hagname)
```

This routine displays the specified UAG or if `uagname` is NULL each UAG defined in the access security database.

***dump Rules***

```
int asDumpRules(char *asgname)
```

This routine displays the rules for the specified ASG or if `asgname` is NULL the rules for each ASG defined in the access security database.

***dump member***

```
int asDumpMem(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.

*dump hash table*

```
int asDumpHash(void)
```

This shows the contents of the hash table used to locate UAGs and HAGs,

---

## Database Access Security

---

### Access Level definition

The definition of access level means that a level is defined for each field of each record type.

1. Structure `fldDes` (`dbBase.h`), which describes the attributes of each field, contains a field `access_security_level`. In addition definitions exist for the symbols: ASL0 and ASL1.
2. Each field description in a record description contains a field with the value ASLx.

The meanings of the Access Security Level definitions are as follows:

- ASL0 Assigned to fields used during normal operation
- ASL1 Assigned to fields that may be sensitive to change. Permission to access this level implies permission for ASL0.

Most record types assign ASL as follows: The fields VAL, RES (Reset), and CMD use the value ASL0. All other fields use ASL1.

### Access Security Group definition

`dbCommon` contains the fields ASG and ASP. ASG (Access Security Group) is a character string. The value can be assigned via a database configuration tool or else a utility could be provided to assign values during ioc initialization. ASP is an access security private field. It contains the address of an ASGMEMBER.

### Access Client Definition

Struct `dbAddr` contains a field `asPvt`, which contains the address of an ASGCLIENT. This definition is also added to struct `db_addr` so that old database access also supports access security.

### Database Access Library

Two files `asDbLib.c` and `asCa.c` implement the interface between IOC databases and access control. It contains the following routines:

*Initialization*

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

Calling this routine sets the filename of an access configuration file. The next call to `asInit` uses this file. This routine must be called before `iocInit` otherwise access configuration is disabled. If access security is disabled during `iocInit` it will never be turned on.

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

This routine specifies macro substitutions.

```
int asInit()
```

```
int asInitAsyn(ASDBCALLBACK *pcallback)
```

This routine calls `asInitialize`. If the current access configuration file, as specified by `asSetFilename`, is `NULL` then the routine just returns, otherwise the configuration file is used to create the access configuration database.

This routine is called by `iocInit`. `asInit` can also be called at any time to change the access configuration information.

`asInitAsyn` spawns a task `asInitTask` to perform the initialization. This allows `asInitAsyn` to be called from a subroutine called by the process entry of a subroutine record. `asInitTask` calls `taskwdInsert` so that if it suspends for some reason `taskwd` can detect the failure. After initialization all records in the database are made members of the appropriate access control group.

If the caller provides an `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.

*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.

```
ASMEMBERPVT 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.

*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.

*Subroutines attached to a subroutine record* These routines are provided so that a channel access client can force an ioc to load a new access configuration database.

```
long asSubInit(struct subRecord *prec, int pass)  
long asSubProcess(struct subRecord *prec)
```

These are routines that can be attached to a subroutine record. Whenever a `1` is written to the record, `asSubProcess` calls `asInit`. If `asInit` returns success, it returns with asynchronously. When `asInitTask` calls the completion routine supplied by `asSubProcess`, the return status is used to place the record in alarm.

*Diagnostic Routines*

These routines provide interfaces to the `asDump` routines described in the previous chapter. They do NOT lock before calling the associated routine. Thus they may fail if the access security configuration is changing while they are running. However the danger of the user accidentally aborting a command and leaving the access security system locked is considered a risk that should be avoided.

```
asdbdump(void)
```

This routine calls `asDump` with a member callback and with `verbose TRUE`.

```
aspuag(char *uagname)
```

This routine calls `asDumpUag`.

```
asphag(char *hagname)
```

This routine calls `asDumpHag`.

```
asprules(char *asgname)
```

This routine calls `asDumpRules`.

```
aspmem(char *asgname,int clients)
```

This routine calls `asDumpMem`.

---

## Channel Access Security

---

EPICS Access Security is designed to protect Input Output Controllers (IOCs) from unauthorized access via the Channel Access (CA) network transparent communication software system. This chapter describes the interaction between the 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 communication system and the CA client interface.

**CA Server Interfaces to the Access Security System** The CA server calls `asAddClient()` and `asRegisterClientCallback()` for each of the channels that a client connects to the server. The routine `asRemoveClient()` is called whenever the client clears (removes) a channel or when the client disconnects.

The server maintains storage for the clients host and user names. The initial value of these strings are supplied to the server when the client connects and can be updated at any time by the client. When these strings change then `asChangeClient()` is called for each of the channels maintained by the server for the client.

The server checks for read access when processing gets and for write access when processing puts. If access is denied then an exception message is sent to the client.

The server checks for read access when processing requests to register an event callback (monitor) for the client. If there is read access the server always sends an initial update indicating the current value. If there isn't read access the server sends one update indicating no read access and disables subsequent updates.

The server receives asynchronous notification of access rights change via the callback registered with `asRegisterClientCallback()`. When a channel's access rights change the server communicates the current state to the client library. If read access to a channel is lost and there are events (monitors) registered on the channel then the server sends an update to the client for each of them indicating no access and disables future updates for each event. If read

access is reestablished to a channel and there are events (monitors) registered on the channel then the server re-enables updates and sends an initial update message to the client for each of them.

## Client Interfaces

Additional details on the channel access client side callable interfaces to access security can be obtained from the “Channel Access Reference Manual”.

The client library stores and maintains the current state of the access rights for each channel that it has established. The client library receives asynchronous updates of the current access rights state from the server. It uses this state to check for read access when processing gets and for write access when processing puts. If a program issues a channel access request that is inconsistent with the client library’s current knowledge of the access rights state then access is denied and an error code is returned to the application. The current access rights state as known by the client library can be tested by an applications program with the C macros `ca_read_access()` and `ca_write_access()`.

An application program can also receive asynchronous notification of changes to the access rights state by registering a function to be called back when the client library updates its storage of the access rights state. The application’s call back function is installed for this purpose by calling `ca_replace_access_rights_event()`.

If the access rights state changes in the server after a request is queued in the client library but before the request is processed by the server then it is possible that the request will fail in the server. Under these circumstances then an exception will be raised in the client.

The server always sends one update to the client when the event (monitor) is initially registered. If there isn’t read access then the status in the arguments to the application program’s event call back function indicates no read access and the value in the arguments to the clients event call back is set to zero. If the read access right changes after the event is initially registered then another update is supplied to the application programs call back function.

---

## Access Control: Implementation Overview

---

This chapter provides a few aids for reading the access security code. Include file `asLib.h` describes the control blocks used by the access security library.

## 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.

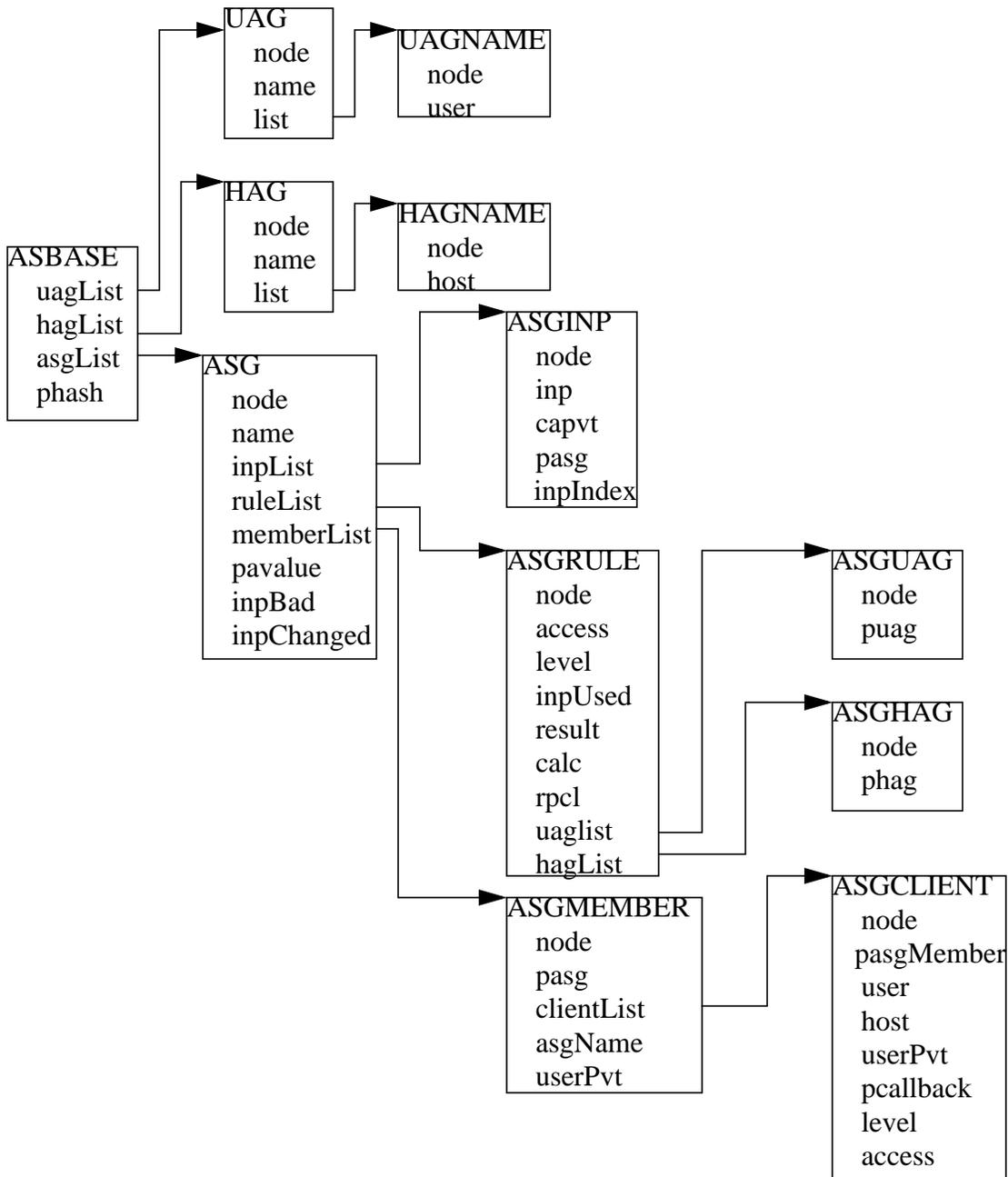
- **asCa.c** This code contains the channel access client code that implements the INP and CALC definitions in an access security database.
- **ascheck.c** The Unix program which performs a syntax check on a configuration file.

## 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.

## Structures



# Chapter 6: IOC Test Facilities

---

## Overview

---

This chapter describes a number of IOC test routines that are of interest to both application developers and system developers. All routines can be executed from the vxWorks shell. The parentheses are optional, but the arguments must be separated by commas. All character string arguments must be enclosed in “”.

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.

---

## Database List, Get, Put

---

### dbl

Database List:

```
dbl ("<record type>", "<filename>")
```

Examples

```
dbl  
dbl "ai"
```

This command prints the names of records in the run time database. If `<record type>` is not specified, all records are listed. If `<record type>` is specified, then only the names of the records of that type are listed.

If `<filename>` is specified the output is written to the specified file (if the file already exists it is overwritten). If this argument is 0 then the output is sent to `stdout`.

### 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.

**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).

**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).

**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.

**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.

**dbtr**

Test Record:

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

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

---

**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. If `all_record_types` is not 0 then all record types are displayed.

---

## 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.

**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.

**dbd**

Remove Breakpoint:

```
dbd ("<record_name>")
```

Removes a breakpoint from a record.

**dfs**

Single Step:

```
dfs ("<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.

**dbc**

Continue:

```
dbc ("<record_name>")
```

Continues execution until another breakpoint is found. This command may also be called without an argument.

**dbp**

Print Fields Of Suspended Record:

```
dbp
```

Prints out the fields of the last record whose execution was suspended.

## 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.

## 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.”

---

## Error Logging

---

## eltc

Display error log messages on console:

```
eltc(int noYes)
```

This determines if error messages are displayed on `vxWorks` console. A value of 0 means no and any other value means yes.

---

## Hardware Reports

---

## dbior

I/O Report:

```
dbior ("<driver_name>",<interest level>)
```

This command calls the report entry of the indicated driver. If `<driver_name>` is not specified then the report for all drivers is generated. It 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.

## **dbhcr**

Hardware Configuration Report:

```
dbhcr("filename")
```

This command produces a report of all hardware links. To use it on the IOC, issue the command:

```
dbhcr > report  
or  
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.

---

## **Scan Reports**

---

## **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.

## **scanpel**

Print Event Lists:

```
scanpel(int event_number)
```

This routine prints a list of all records in the event scan list for the specified event number. If event\_number is 0 all event scan lists are shown.

## **scanpiol**

Print I/O Event Lists:

```
scanpiol
```

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

---

## **Time Server Report**

---

## **TSreport**

Format:

```
TSreport
```

This routine prints out information about the Time server. This includes:

- Slave or Master
- Soft or Hardware synchronized

- Clock and Sync rates
- etc.

---

## Access Security Commands

---

### **asSetFilename**

Format:

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

This command defines a new access security file.

### **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.

### **asdbdump**

Format:

```
asdbdump
```

This provides a complete dump of the access security database.

### **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.

### **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.

### **asprules**

Format:

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

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

### **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.

---

## Channel Access Reports

---

### **ca\_channel\_status**

Format:

```
ca_channel_status (taskid)
```

Prints status for each channel in use by specialized vxWorks task.

### **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.

### **dbel**

Format:

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

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

### **dbcar**

Database to Channel Access Report - See "Record Link Reports"

---

## Interrupt Vectors

---

### **veclist**

Format:

```
veclist
```

Print Interrupt Vector List

---

---

## EPICS

---

### **epicsPrtEnvParams**

Format:

```
epicsPrtEnvParams
```

Print Environment Variables

### **epicsRelease**

Format:

```
coreRelease
```

Print release of iocCore.

---

---

## Database System Test Routines

---

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

### **dbt**

Measure Time To Process A Record:

```
dbt ("<record_name>")
```

Times the execution of 100 successive processings of record `record_name`. Note that process passive and forward links within this record may incur the processing of other records in its lockset. This function is a wrapper around the VxWorks `timeXN()` function, and directly displays its output. Therefore one must divide the result by 100 to get the execution time for one processing of `record_name`.

### **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.

### **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.

### **dbtpn**

Test Put Notify:

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

Example:

```
dbtpn "aitest", "5.0"
```

This command performs a `dbNameToAddr`, then calls `dbPutNotify` and has a callback routine that prints a message when it is called. This routine is of interest to system developers for testing database access.

---

## **Record Link Routines**

---

### **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 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.

### **dbcar**

Database to channel access report

```
dbcar(recordname, level)
```

This command generates a report showing database channel access links. If `recordname` is 0 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.

**dbhcr** Report hardware links. See “Hardware Reports”.

---

## 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.

**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.

**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.

**tpn**

Test Put Notify:

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

Example:

```
tpn "aitest", "5.0"
```

This routine tests `dbPutNotify` via the old database access interface.

---

## Routines to dump database information

---

**dbDumpPath**

Dump Path:

```
dbDumpPath(pdbbase)
```

```
dbDumpPath(pdbbase)
```

The current path for database includes is displayed.

### **dbDumpMenu**

Dump Menu:

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

```
dbDumpMenu ( pdbname , "menuScan" )
```

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

### **dbDumpRecordType**

Dump Record Description:

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

```
dbDumpRecordType ( pdbname , "ai" )
```

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

### **dbDumpFldDes**

Dump Field Description:

```
dbDumpFldDes ( pdbname , "<record type>" , "<field name>" )
```

```
dbDumpFldDes ( pdbname , "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.

### **dbDumpDevice**

Dump Device Support:

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

```
dbDumpDevice ( pdbname , "ai" )
```

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

### **dbDumpDriver**

Dump Driver Support:

```
dbDumpDriver ( pdbname )
```

```
dbDumpDriver ( pdbname )
```

### **dbDumpRecords**

Dump Record Instances:

```
dbDumpRecords ( pdbname , "<record type>" , level )
```

```
dbDumpRecords ( pdbname , "ai" )
```

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

### **dbDumpBreaktable**

Dump breakpoint table

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

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

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

#### **dbPvdDump**

Dump the Process variable Directory:

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

```
dbPvdDump (pdbbase , 0 )
```

This command shows how many records are mapped to each hash table entry of the process variable directory. If verbose is not 0 then the command also displays the names which hash to each hash table entry.

# Chapter 7: IOC Error Logging

---

## 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.
- `errlog` Task - A task that displays error messages on the target console and also passes the messages to all registered system wide error logger.
- status codes - EPICS status codes.
- `iocLog`- A system wide error logger supplied with base. It writes all messages to a system wide file.

NOTE: `recGbl` error routines are also provided. They in turn call one of the error message routines.

---

## Error Message Routines

---

### Basic Routines

```
int errlogPrintf(const char *pformat, ...);
int errlogVprintf(const char *pformat,va_list pvar);

int errlogMessage(const char *message);
```

`errlogPrintf` and `errlogVprintf` are like `printf` and `vprintf` provided by the standard C library, except that the output is sent to the `errlog` task. Consult any book that describes the standard C library such as "The C Programming Language ANSI C Edition" by Kernighan and Ritchie.

`errlogMessage` sends message to the `errlog` task

### 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.

`errlogGetSevEnumString` gets the string value of severity.

`errlogSetSevToLog` sets the severity to log. `errlogGetSevToLog` gets the current severity to log.

### 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 has the following format:

```
void errPrintf(long status, __FILE__, __LINE__,
               char *fmtstring <arg1>, ...);
```

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.

## 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.

---

## errlog Task

---

The error message routines can be called by any non-interrupt level code. These routines merely pass the message to the errlog Task.

Task errlog manages the messages. Messages are placed in a message queue, which is read by the errlog task. 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 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.

The error message routines are partially implemented on the host. The host version just calls fprintf or vfprintf instead of using a separate task and a message queue. Thus host messages are NOT sent to a system wide error logger.

## Add and Remove Log Listener

```
typedef void(*errlogListener) (const char *message);  
void errlogAddListener(errlogListener listener);  
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.

## target console routines

```
int eltc(int yesno); /* error log to console (0 or 1) */  
int errlogInit(int bufsize);
```

eltc determines if erlog task writes message to the console. During error messages storms this command can be used to suppress console messages. A argument of 0 suppresses the messages and any other value lets the message go to the console.

errlogInit can be used to initialize the error logging system with a larger buffer. The default is 1280 bytes. An extra MAX\_MESSAGE\_SIZE (currently 256) bytes are allocated but never used. This is a small extra protection against long error messages.

---

## Status Codes

---

EPICS defined status values provide the following features:

- Whenever possible, IOC routines return a status value: (0, non-0) means (OK, ERROR).
- The include files for each IOC subsystem contain macros defining error status symbols and strings.
- Routines are provided for run time access of the error status symbols and strings.
- A global variable `errVerbose` helps code decide if error messages should be generated.

WARNING: During the fall of 1995 a series of tech-talk messages were generated concerning EPICS status values. No consensus was reached.

Whenever it makes sense, IOC routines return a long word status value encoded similar to the vxWorks error status encoding. The most significant short word indicates the subsystem module within which the error occurred. The low order short word is a subsystem status value. In order that status values do not conflict with the vxWorks error status values all subsystem numbers are greater than 500.

A file `epics/share/epicsH/errMdef.h` defines each subsystem number. For example the define for the database access routines is:

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

Directory `epics/share/epicsH` contains an include library for every IOC subsystem that returns standard status values. The status values are encoded with lines of the following format:

```
#define S_xxxxxxx value /*string value*/
```

For example:

```
#define S_dbAccessBadDBR (M_dbAccess|3) \  
/*Invalid Database Request*/
```

For example, when `dbGetField` detects a bad database request type, it executes the statement:

```
return(S_dbAccessBadDBR);
```

The calling routine checks the return status as follows:

```
status = dbGetField( ...);
if(status) { /* Call was not successful */ }
```

---

## **iocLog**

---

This consists of two modules: `iocLogServer` and `iocLogClient`. The client code runs on each ioc and listens for the messages generated by the `erlog` system. It also reports the messages from `vxWorks logMsg`.

### **iocLogServer**

This runs on a host. It receives messages for all enabled `iocLogClients` in the local area network. The messages are written to a file. Epics base provides a startup file "`base/src/util/rc2.logServer`", which is a shell script to start the server. Consult this script for details.

### **iocLogClient**

This runs on each ioc. It is started by default when `iocInit` runs. 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 immediately after `iocCore` is loaded then `iocLogClient` will not even initialize itself.

### **Initialize Logging**

Initialize the logging system. This system traps all `logMsg` calls and sends a copy to a Unix file. Note that this can be disabled by issuing the command `iocLogDisable=1` before issuing `iocInit`.

The following description was supplied by Jeff Hill:

It is possible to configure EPICS so that a log of IOC error messages is stored in a circular ASCII file on a PC or UNIX workstation. Each entry in the log contains the IOC's DNS name, the date and time when the message was received by the log server, and the text of the message generated on the IOC.

All messages generated by the EPICS functions `epicsPrintf()` and `errMessage()` are placed in the log. Messages generated by the `vxWorks` function `logMsg()` are also placed in the log (`logMsg()` can be safely called from interrupt level). Messages generated by `printf()` do not end up in the log and are instead used primarily by diagnostic functions called from the `vxWorks` shell.

To start a log server on a UNIX or PC workstation you must first set the following environment variables and then run the executable "`iocLogServer`" on your PC or UNIX workstation.

**EPICS\_IOC\_LOG\_FILE\_NAME**

The name and path to the log file.

**EPICS\_IOC\_LOG\_FILE\_LIMIT**

The maximum size in characters for the log file (after which it becomes a circular file and writes new messages over old messages at the beginning of the file). If the value is zero then there is no limit on the size of the log file.

**EPICS\_IOC\_LOG\_FILE\_COMMAND**

A shell command string used to obtain the log file path name during initialization and in response to SIGHUP. The new path name will replace any path name supplied in EPICS\_IOC\_LOG\_FILE\_NAME.

Thus, if EPICS\_IOC\_LOG\_FILE\_NAME is "a/b/c.log" and EPICS\_IOC\_LOG\_FILE\_COMMAND returns "A/B" or "A/B/" the log server will be stored at "A/B/c.log"

If EPICS\_IOC\_LOG\_FILE\_COMMAND is empty then this behavior is disabled. This feature was donated to the collaboration by KECK, and it is used by them for switching to a new directory at a fixed time each day. This variable is currently used only by the UNIX version of the log server.

EPICS\_IOC\_LOG\_PORT

THE TCP/IP port used by the log server.

To configure an IOC so that its messages are placed in the log you must set the environment variable EPICS\_IOC\_LOG\_INET to the IP address of the host that is running the log server and EPICS\_IOC\_LOG\_PORT to the TCP/IP port used by the log server.

Defaults for all of the above parameters are specified in the files \$(EPICS\_BASE)/config/CONFIG\_SITE\_ENV and \$(EPICS\_BASE)/config/CONFIG\_ENV.

In base/src/util there is a solaris script for starting the log server. This can be adapted for use on other host architectures.

## 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 putenv command to your IOC startup file. For example

```
ld < iocCore
putenv( "EPICS_IOC_LOG_INET=xxx.xxx.xxx.xxx" )
```

The inet address is for your host workstation.
- On you host start a version of the log server.

# Chapter 8: Record Support

---

## Overview

---

The purpose of this chapter is to describe record support in sufficient detail such that a C programmer can write new record support modules. Before attempting to write new support modules, you should carefully study a few of the existing support modules. If an existing support module is similar to the desired module most of the work will already be done.

From previous chapters, it should be clear that many things happen as a result of record processing. The details of what happens are dependent on the record type. In order to allow new record types and new device types without impacting the core IOC system, the concept of record support and device support has been created. For each record type, a record support module exists. It is responsible for all record specific details. In order to allow a record support module to be independent of device specific details, the concept of device support has been created.

A record support module consists of a standard set of routines that can be called by database access routines. This set of routines implements record specific code. Each record type can define a standard set of device support routines specific to that record type.

By far the most important record support routine is `process`, which `dbProcess` calls when it wants to process a record. This routine is responsible for the details of record processing. In many cases it calls a device support I/O routine. The next section gives an overview of what must be done in order to process a record. Next is a description of the entry tables that must be provided by record and device support modules. The remaining sections give example record and device support modules and describe some global routines useful to record support modules.

The record and device support modules are the only modules that are allowed to include the record specific include files as defined in `base/rec`. Thus they are the only routines that access record specific fields without going through database access.

---

## 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.

## Record Support and Device Support Entry Tables

Each record type has an associated set of record support routines. These routines are located via the data structures defined in `epics/share/epicsH/recSup.h`. The concept of record support routines isolates the `iocCore` software from the details of each record type. Thus new records can be defined and supported without affecting the IOC core software.

Each record type also has zero or more sets of device support routines. Record types without associated hardware, e.g. calculation records, normally do not have any associated device support. Record types with associated hardware normally have a device support module for each device type. The concept of device support isolates IOC core software and even record support from device specific details.

Corresponding to each record type is a set of record support routines. The set of routines is the same for every record type. These routines are located via a Record Support Entry Table (RSET), which has the following structure

```

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 located in dbCommon, contains the address of the DSET. It is given a value by iocInit.

## 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.

### 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();

struct 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};

/* 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 alarm(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.

### **init\_record**

```
static long init_record(void *precord, int pass)
{
    xxxRecord*pxxx = (xxxRecord *)precord;
    xxxdset *pdset;
    long status;
```

```

if(pass==0) return(0);

if((pdset = (xxx_dset *) (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`).

## process

```

static long process(void *precord)
{
    xxxRecord*pxxx = (xxxRecord *)precord;
    xxx_dset *pdset = (xxx_dset *)pxxx->dset;
    long status;
    unsigned char pact=pxxx->pact;

    if( (pdset==NULL) || (pdset->read_xxx==NULL) ) {
        /* leave pact true so that dbProcess doesnt call again*/
        pxxx->pact=TRUE;
        recGblRecordError(S_dev_missingSup,pxxx,"read_xxx");
        return(S_dev_missingSup);
    }

    /* pact must not be set true until read_xxx completes*/
    status=( *pdset->read_xxx)(pxxx); /* read the new value */
    /* return if beginning of asynch processing*/
    if(!pact && pxxx->pact) return(0);
    pxxx->pact = TRUE;
    recGblGetTimeStamp(pxxx);
}

```

```
    /* check for alarms */
    alarm(pxxx);
    /* check event list */
    monitor(pxxx);
    /* process the forward scan link record */
    recGblFwdLink(pxxx);

    pxxx->pact=FALSE;
    return(status);
}
```

The record processing routines are the heart of the IOC software. The record specific process routine is called by `dbProcess` whenever it decides that a record should be processed. Process decides what record processing really means. The above is a good example of what should be done. In addition to being called by `dbProcess` the process routine may also be called by asynchronous record completion routines.

The above model supports both synchronous and asynchronous device support routines. For example, if `read_xxx` is an asynchronous routine, the following sequence of events will occur:

- process is called with `pact FALSE`
- `read_xxx` is called. Since `pact` is `FALSE` it starts I/O, arranges callback, and sets `pact TRUE`
- `read_xxx` returns
- because `pact` went from `FALSE` to `TRUE` process just returns
- Any new call to `dbProcess` is ignored because it finds `pact TRUE`
- Sometime later the callback occurs and process is called again.
- `read_xxx` is called. Since `pact` is `TRUE` it knows that it is a completion request.
- `read_xxx` returns
- process completes record processing
- `pact` is set `FALSE`
- process returns

At this point the record has been completely processed. The next time process is called everything starts all over from the beginning.

### 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.

## Alarm Processing

```

static void alarm(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.

## Raising Monitors

```
static void monitor(xxxRecord *pxxx)
{
    unsigned short    monitor_mask;
    float             delta;

    monitor_mask = recGblResetAlarms(pxxx);
    /* check for value change */
    delta = pxxx->mlst - pxxx->val;
    if(delta<0.0) delta = -delta;
    if (delta > pxxx->mdel) {
        /* post events for value change */
        monitor_mask |= DBE_VALUE;
        /* update last value monitored */
        pxxx->mlst = pxxx->val;
    }
    /* check for archive change */
    delta = pxxx->alst - pxxx->val;
    if(delta<0.0) delta = 0.0;
    if (delta > pxxx->adel) {
        /* post events on value field for archive change */
        monitor_mask |= DBE_LOG;
        /* update last archive value monitored */
        pxxx->alst = pxxx->val;
    }
    /* send out monitors connected to the value field */
    if (monitor_mask){
        db_post_events(pxxx,&pxxx->val,monitor_mask);
    }
    return;
}
```

All record types should call `recGblResetAlarms` as shown. Note that `nsta` and `nsev` will have the value 0 after this routine completes. This is necessary to ensure that alarm checking starts fresh after processing completes. The code also takes care of raising alarm

monitors when a record changes from an alarm state to the no alarm state. It is essential that record support routines follow the above model or else alarm processing will not follow the rules.

Analog type records should also provide monitor and archive hysteresis fields as shown by this example.

db\_post\_events results in channel access issuing monitors for clients attached to the record and field. The call is

```
int db_post_events(void *precord, void *pfield,
                  unsigned int monitor_mask)
```

where:

```
precord - The address of the record
pfield - The address of the field
monitor_mask - A bit mask that can be any combinations of the following:
    DBE_ALARM - A change of alarm state has occurred. This is set by
    recGblResetAlarms.
    DBE_LOG - Archive change of state.
    DBE_VAL - Value change of state
```

**IMPORTANT:** The record support module is responsible for calling db\_post\_event for any fields that change as a result of record processing. Also it should **NOT** call db\_post\_event for fields that do not change.

---

## 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.

### Generate Report of Each Field in Record

```
report(void *precord); /* addr of record*/
```

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

### Initialize Record Processing

```
init(void);
```

This routine is called once at IOC initialization time. Any action is record type specific. Most record types do not need this routine.

### Initialize Specific Record

```
init_record(
    void *precord, /* addr of record*/
    int pass);
```

iocInit calls this routine twice (pass=0 and pass=1) for each database record of the type handled by this routine. It must perform the following functions:

- Check and/or issue initialization calls for the associated device support routines.
- Perform any record type specific initialization.
- During the first pass it can only perform initializations that affect the record referenced by precord.

- During the second pass it can perform initializations that affect other records.

### Process Record

```
process(void *precord); /* addr of record*/
```

This routine must follow the guidelines specified previously.

### Special Processing

```
special(  
    struct dbAddr *paddr,  
    int after); /* (FALSE, TRUE) => (Before, After) Processing*/
```

This routine implements the record type specific special processing for the field referred to by dbAddr. Note that it is called twice. Once before any changes are made to the associated field and once after. File special.h defines special types. This routine is only called for user special fields, i.e. fields with SPC\_XXX >= 100. A field is declared special in the ASCII record definition file. New values should not be added to special.h, instead use SPC\_MOD.

The database access routine, dbGetFieldIndex can be used to determine which field is being modified.

### Get Value

This routine is no longer used. It should be left as a NULL procedure in the record support entry table.

### Convert dbAddr Definitions

```
cvt_dbaddr(struct dbAddr *paddr);
```

This routine is called by dbNameToAddr if the field has special set equal to SPC\_DBADDR. A typical use is when a field refers to an array. This routine can change any combination of the dbAddr fields: no\_elements, field\_type, field\_size, special, and dbr\_type. For example if the VAL field of a waveform record is passed to dbNameToAddr, cvt\_dbaddr would change dbAddr so that it refers to the actual array rather than VAL.

The database access routine, dbGetFieldIndex can be used to determine which field is being modified.

### Get Array Information

```
get_array_info(  
    struct dbAddr *paddr,  
    long *no_elements,  
    long *offset);
```

This routine returns the current number of elements and the offset of the first value of the specified array. The offset field is meaningful if the array is actually a circular buffer.

The database access routine, dbGetFieldIndex can be used to determine which field is being modified.

### Put Array Information

```
put_array_info(  
    struct dbAddr *paddr,  
    long nNew);
```

This routine is called after new values have been placed in the specified array.

The database access routine, dbGetFieldIndex can be used to determine which field is being modified.

### Get Units

```
get_units(  
    struct dbAddr *paddr,  
    char *punits);
```

This routine sets units equal to the engineering units for the field.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

### Get Precision

```
get_precision(
    struct dbAddr *paddr,
    long *precision);
```

This routine gets the precision, i.e. number of decimal places, which should be used to convert the field value to an ASCII string. `recGblGetPrec` should be called for fields not directly related to the value field.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

### Get Enumerated String

```
get_enum_str(
    struct dbAddr *paddr,
    char *p);
```

This routine sets `*p` equal to the ASCII string for the field value. The field must have type `DBF_ENUM`.

Look at the code for the `bi` or `mbbi` records for examples.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

### Get Strings for Enumerated Field

```
get_enum_strs(
    struct dbAddr *paddr,
    struct dbr_enumStrs *p);
```

This routine gives values to all fields of structure `dbr_enumStrs`.

Look at the code for the `bi` or `mbbi` records for examples.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

### Put Enumerated String

```
put_enum_str(
    struct dbAddr *paddr,
    char *p);
```

Given an ASCII string, this routine updates the database field. It compares the string with the string values associated with each enumerated value and if it finds a match sets the database field equal to the index of the string which matched.

Look at the code for the `bi` or `mbbi` records for examples.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

### Get Graphic Double Information

```
get_graphic_double(
    struct dbAddr *paddr,
    struct dbr_grDouble *p); /* addr of return info*/
```

This routine fills in the graphics related fields of structure `dbr_grDouble`. `recGblGetGraphicDouble` should be called for fields not directly related to the value field.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

### Get Control Double Information

```
get_control_double(  
    struct dbAddr *paddr,  
    struct dbr_ctrlDouble *p); /* addr of return info*/
```

This routine gives values to all fields of structure `dbr_ctrlDouble`. `recGblGetControlDouble` should be called for fields not directly related to the value field.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

### Get Alarm Double Information

```
get_alarm_double(  
    struct dbAddr *paddr,  
    struct dbr_alDouble *p); /* addr of return info*/
```

This routine gives values to all fields of structure `dbr_alDouble`.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

---

## 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".

### Alarm Status and Severity

Alarms may be raised in many different places during the course of record processing. The algorithm is to maximize the alarm severity, i.e. the highest severity outstanding alarm is raised. If more than one alarm of the same severity is found then the first one is reported. This means that whenever a code fragment wants to raise an alarm, it does so only if the alarm severity it will declare is greater than that already existing. Four fields (in database common) are used to implement alarms: `sevr`, `stat`, `nsev`, and `nsta`. The first two are the status and severity after the record is completely processed. The last two fields (`nsta` and `nsev`) are the status and severity values to set during record processing. Two routines are used for handling alarms. Whenever a routine wants to raise an alarm it calls `recGblSetSevr`. This routine will only change `nsta` and `nsev` if it will result in the alarm severity being increased. At the end of processing, the record support module must call `recGblResetAlarms`. This routine sets `stat=nsta`, `sevr=nsev`, `nsta=0`, and `nsev=0`. If `stat` or `sevr` has changed value since the last call it calls `db_post_event` for `stat` and `sevr` and returns a value of `DBE_ALARM`. If no change occurred it returns 0. Thus after calling `recGblResetAlarms` everything is ready for raising alarms the next time the record is processed. The example record support module presented above shows how these macros are used.

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

Returns: (TRUE, FALSE) if (did, did not) change nsta and nsev.

```
unsigned short recGblResetAlarms(void *precord);
```

Returns: Initial value for monitor\_mask

### Alarm Acknowledgment

Database common contains two additional alarm related fields: acks (Highest severity unacknowledged alarm) and ackt (does transient alarm need to be acknowledged). These field are handled by iocCore and recGblResetAlarms and are not the responsibility of record support. These fields are intended for use by the alarm handler.

### Generate Error: Process Variable Name, Caller, Message

SUGGESTION: use epicsPrintf instead of this for new code.

```
recGblDbaddrError(
    long status,
    struct dbAddr *paddr,
    char *pcaller_name); /* calling routine name */
```

This routine interfaces with the system wide error handling system to display the following information: Status information, process variable name, calling routine.

### Generate Error: Status String, Record Name, Caller

SUGGESTION: use epicsPrintf instead of this for new code.

```
recGblRecordError(
    long status,
    void *precord, /* addr of record */
    char *pcaller_name); /* calling routine name */
```

This routine interfaces with the system wide error handling system to display the following information: Status information, record name, calling routine.

### Generate Error: Record Name, Caller, Record Support Message

SUGGESTION: use epicsPrintf instead of this for new code.

```
recGblRecsupError(
    long status,
    struct dbAddr *paddr,
    char *pcaller_name, /* calling routine name */
    char *psupport_name); /* support routine name*/
```

This routine interfaces with the system wide error handling system to display the following information: Status information, record name, calling routine, record support entry name.

### Get Graphics Double

```
recGblGetGraphicDouble(
    struct dbAddr *paddr,
    struct dbr_grDouble *pgd);
```

This routine can be used by the get\_graphic\_double record support routine to obtain graphics values for fields that it doesn't know how to set.

### Get Control Double

```
recGblGetControlDouble(
    struct dbAddr *paddr,
    struct dbr_ctrlDouble *pcd);
```

This routine can be used by the get\_control\_double record support routine to obtain control values for fields that it doesn't know how to set.

### Get Alarm Double

```
recGblGetAlarmDouble(
    struct dbAddr *paddr,
```

```
struct dbr_alDouble *pcd);
```

This routine can be used by the `get_alarm_double` record support routine to obtain control values for fields that it doesn't know how to set.

### **Get Precision**

```
recGblGetPrec(  
    struct dbAddr *paddr,  
    long *pprecision);
```

This routine can be used by the `get_precision` record support routine to obtain the precision for fields that it doesn't know how to set the precision.

### **Get Time Stamp**

```
recGblGetTimeStamp(void *precord)
```

This routine gets the current time stamp and puts it in the record

### **Forward link**

```
recGblFwdLink(  
    void *precord);
```

This routine can be used by process to request processing of forward links.

### **Initialize Constant Link**

```
int recGblInitConstantLink(  
    struct link *plink,  
    short dbfType,  
    void *pdest);
```

Initialize a constant link. This routine is usually called by `init_record` (or by associated device support) to initialize the field associated with a constant link. It returns (FALSE, TRUE) if it (did not, did) modify the destination.

# Chapter 9: Device Support

---

## 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.

Database common contains two device related fields:

- **dtyp**: Device Type.
- **dset**: Address of Device Support Entry Table.

The field `dtyp` contains the index of the menu choice as defined by the device ASCII definitions. `iocInit` uses this field and the device support structures defined in `devSup.h` to initialize the field `dset`. Thus record support can locate its associated device support via the `dset` field.

Device support modules can be divided into two basic classes: synchronous and asynchronous. Synchronous device support is used for hardware that can be accessed without delays for I/O. Many register based devices are synchronous devices. Other devices, for example all GPIB devices, can only be accessed via I/O requests that may take large amounts of time to complete. Such devices must have associated asynchronous device support. Asynchronous device support makes it more difficult to create databases that have linked records.

If a device can be accessed with a delay of less than a few microseconds then synchronous device support is appropriate. If a device causes delays of greater than 100 microseconds then asynchronous device support is appropriate. If the delay is between these values your guess about what to do is as good as mine. Perhaps you should ask the hardware designer why such a device was created.

If a device takes a long time to accept requests there is another option than asynchronous device support. A driver can be created that periodically polls all its attached input devices. The device support just returns the latest polled value. For outputs, device support just notifies the driver that a new value must be written. the driver, during one of its polling phases, writes the new value. The EPICS Allen Bradley device/driver support is a good example.

## 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};

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) :
            recGblInitConstantLink(&pai->inp,
                DBF_DOUBLE,&pai->val);
            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(status) return(status);
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.

---

## 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 VAL field. `callbackRequest` is an EPICS supplied routine. The watchdog timer routines are supplied by `vxWorks`.
2. It prints a message stating that processing has started, sets `pact` TRUE, and returns. The record processing routine returns without completing processing.
3. When the specified time elapses `myCallback` is called. It locks the record, calls `process`, and unlocks the record. It calls the process entry of the record support module, which it locates via the `rset` field in `dbCommon`, directly rather than `dbProcess`. `dbProcess` would not call `process` because `pact` is TRUE.
4. When `process` executes, it again calls `read_ai`. This time `pact` is TRUE.
5. `read_ai` prints a message stating that record processing is complete and returns a status of 2. Normally a value of 0 would be returned. The value 2 tells the record support routine not to attempt any conversions. This is a convention (a bad convention!) used by the analog input record.
6. When `read_ai` returns the record processing routine completes record processing.

At this point the record has been completely processed. The next time `process` is called everything starts all over.

```
/* Create the dset for devAiTestAsyn */
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;
} devAiTestAsyn={
```

```
        6,  
        NULL,  
        NULL,  
        init_record,  
        NULL,  
        read_ai,  
        NULL};  
  
/* control block for callback*/  
typedef struct myCallback {  
    CALLBACK    callback;  
    struct dbCommon    *precord;  
    WDOG_ID    wd_id;  
}myCallback;  
  
static void myCallback(CALLBACK *pcallback)  
{  
    dbCommon *precord;  
    struct rset*prset;  
  
    callbackGetUser(precord,pcallback);  
    prset = (struct rset *)precord->rset;  
    dbScanLock(precord);  
    *(prset->process)(precord);  
    dbScanUnlock(precord);  
}  
  
static long init_record(void *precord)  
{  
    aiRecord    *pai = (aiRecord *)precord;  
    myCallback    *pcallback;  
  
    /* ai.inp must be a CONSTANT*/  
    switch (pai->inp.type) {  
    case (CONSTANT) :  
        pcallback = (myCallback *) (calloc(1, sizeof(myCallback)));  
        pai->dpvt = (void *) pcallback;  
        callbackSetCallback(myCallback, &pcallback->callback);  
        callbackSetUser(precord, &pcallback->callback);  
        pcallback->precord = (struct dbCommon *) pai;  
        pcallback->wd_id = wdCreate();  
        pai->val = pai->inp.value.value;  
        pai->udf = FALSE;  
        break;  
    default :  
        recGblRecordError(S_db_badField, (void *) pai,  
            "devAiTestAsyn (init_record) Illegal INP field");  
        return(S_db_badField);  
    }  
    return(0);  
}
```

```

static long read_ai(void *precord)
{
    aiRecord    *pai = (aiRecord *)precord;;
    struct callback *pcallback=(struct callback *) (pai->dpvt);
    int    wait_time;

    /* ai.inp must be a CONSTANT*/
    switch (pai->inp.type) {
    case (CONSTANT) :
        if(pai->pact) {
            printf("%s Completed\n",pai->name);
            return(2); /* don't convert*/
        } else {
            wait_time = (int)(pai->val * vxTicksPerSecond);
            if(wait_time<=0) return(0);
            callbackSetPriority(pai->prio,&pcallback->callback);
            printf("%s Starting asynchronous processing\n",
                pai->name);
            wdStart(pcallback->wd_id,wait_time,
                (FUNCPTR)callbackRequest,
                (int)&pcallback->callback);
            pai->pact = TRUE;
            return(0);
        }
    default :
        if(recGblSetSevr(pai,SOFT_ALARM,VALID_ALARM)) {
            if(pai->stat!=SOFT_ALARM) {
                recGblRecordError(S_db_badField, (void *)pai,
                    "devAiTestAsyn (read_ai) Illegal INP field");
            }
        }
    }
    return(0);
}

```

---

## 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.

### Generate Device Report

```

report(
    FILE    fp,    /* file pointer*/
    int    interest);

```

This routine is responsible for reporting all I/O cards it has found. If *interest* is (0,1) then generate a (short, long) report. If a device support module is using a driver, it normally does not have to implement this routine because the driver generates the report.

### Initialize Record Processing

```

init(
    int    after);

```

This routine is called twice at IOC initialization time. Any action is device specific. This routine is called twice: once before any database records are initialized and once after all records are initialized but before the scan tasks are started. `after` has the value (0,1) (before, after) record initialization.

**Initialize Specific Record**

```
init_record(  
    void *precord); /* addr of record*/
```

The record support `init_record` routine calls this routine.

**Get I/O Interrupt Information**

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

This is called by the I/O interrupt scan task. If `cmd` is (0,1) then this routine is being called when the associated record is being (placed in, taken out of) an I/O scan list. See the chapter on scanning for details.

It should be noted that a previous type of I/O event scanning is still supported. It is not described in this document because, hopefully, it will go away in the near future. When it calls this routine the arguments have completely different meanings.

**Other Device Support Routines**

All other device support routines are record type specific.

# Chapter 10: Driver Support

---

## Overview

---

It is not necessary to create a driver support module in order to interface EPICS to hardware. For simple hardware device support is sufficient. At the present time most hardware support has both. The reason for this is historical. Before EPICS there was GTACS. During the change from GTACS to EPICS, record support was changed drastically. In order to preserve all existing hardware support the GTACS drivers were used without change. The device support layer was created just to shield the existing drivers from the record support changes.

Since EPICS now has both device and driver support the question arises: When do I need driver support and when don't I? Lets give a few reasons why drivers should be created.

- The hardware is actually a subnet, e.g. GPIB. In this case a driver should be provided for accessing the subnet. There is no reason to make the driver aware of EPICS except possibly for issuing error messages.
- The hardware is complicated. In this case supplying driver support helps modularized the software. The Allen Bradley driver, which is also an example of supporting a subnet, is a good example.
- An existing driver, maintained by others, is available. I don't know of any examples.
- The driver should be general purpose, i.e. not tied to EPICS. The CAMAC driver is a good example. It is used by other systems, such as CODA.

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".

---

## Device Drivers

---

Device drivers are modules that interface directly with the hardware. They are provided to isolate device support routines from details of how to interface to the hardware. Device drivers have no knowledge of the internals of database records. Thus there is no necessary correspondence between record types and device drivers. For example the Allen Bradley driver provides support for many different types of signals including analog inputs, analog outputs, binary inputs, and binary outputs.

In general only device support routines know how to call device drivers. Since device support varies widely from device to device, the set of routines provided by a device driver is almost completely driver dependent. The only requirement is that routines `report` and `init` must be provided. Device support routines must, of course, know what routines are provided by a driver.

File `drvSup.h` describes the format of a driver support entry table. The driver support module must supply a driver entry table. An example definition is:

```
LOCAL long report();
LOCAL long init();
struct {
    long    number;
    DRVSUPFUN    report;
    DRVSUPFUN    init;
} drvAb={
    2,
    report,
    init
};
```

The above example is for the Allen Bradley driver. It has an associated ascii definition of:

```
driver(drvAb)
```

Thus it is seen that the driver support module should supply two EPICS callable routines: `int` and `report`.

### *init*

This routine, which has no arguments, is called by `iocInit`. The driver is expected to look for and initialize the hardware it supports. As an example the `init` routine for Allen Bradley is:

```
LOCAL long init()
{
    return(ab_driver_init());
}
```

### *report*

The `report` routine is called by the `dbior`, an IOC test routine. It is responsible for producing a report describing the hardware it found at `init` time. It is passed one argument, `level`, which is a hint about how much information to display. An example, taken from Allen Bradley, is:

```
LOCAL long report(int level)
{
    return(ab_io_report(level));
}
```

Guidelines for `level` are as follows:

Level=0	Display a one line summary for each device
Level=1	Display more information
Level=2	Display a lot of information. It is even permissible to prompt for what is wanted.

### *Hardware Configuration*

Hardware configuration includes the following:

- VME/VXI address space
- VME Interrupt Vectors and levels
- Device Specific Information

The information contained in hardware links supplies some but not all configuration information. In particular it does not define the VME/VXI addresses and interrupt vectors. This additional information is what is meant by hardware configuration in this chapter.

The problem of defining hardware configuration information is an unsolved problem for EPICS. At one time configuration information was defined in `module_types.h`. Many existing device/driver support modules still use this method. It should **NOT** be used for any new support for the following reasons:

- There is no way to manage this file for the entire EPICS community.
- It does not allow arbitrary configuration information.
- It is hard for users to determine what the configuration information is.

The fact that it is now easy to include ASCII definitions for only the device/driver support used in each IOC makes the configuration problem much more manageable than previously. Previously if you wanted to support a new VME module it was necessary to pick addresses that nothing in `module_types.h` was using. Now you only have to check modules you are actually using.

Since there are no EPICS defined rules for hardware configuration, the following minimal guidelines should be used:

- Never use `#define` to specify things like VME addresses. Instead use variables and assign default values. Allow the default values to be changed before `iocInit` is executed. The best way is to supply a global routine that can be invoked from the IOC startup file. Note that all arguments to such routines should be one of the following:
  - `int`
  - `char *`
  - `double`
- Call the routines described in chapter “Device Support Library” whenever possible.



# Chapter 11: Static Database Access

---

## Overview

---

An IOC database is created on a Unix system via a Database Configuration Tool and stored in a Unix file. EPICS provides two sets of database access routines: Static Database Access and Runtime Database Access. Static database access can be used on Unix or IOC database files. Runtime database requires an initialized IOC databases. Static database access is described in this chapter and runtime database access in the next chapter.

Static database access provides a simplified interface to a database, i.e. much of the complexity is hidden. `DBF_MENU` and `DBF_DEVICE` fields are accessed via a common type called `DCT_MENU`. A set of routines are provided to simplify access to link fields. All fields can be accessed as character strings. This interface is called static database access because it can be used to access an uninitialized, as well as an initialized database.

Before accessing database records, the files describing menus, record types, and devices must be read via `dbReadDatabase` or `dbReadDatabaseFP`. These routines, which are also used to load record instances, can be called multiple times.

Database Configuration Tools (DCTs) should manipulate an EPICS database only via the static database access interface. An IOC database is created on a Unix system via a database configuration tool and stored in a Unix file with a file extension of ".db". Three routines (`dbReadDatabase`, `dbReadDatabaseFP` and `dbWriteRecord`) access a Unix 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.

---

## Definitions

---

### **DBBASE**

Multiple memory resident databases can be accessed simultaneously. The user must provide definitions in the form:

```
DBBASE *pdbname;
```

### **DBENTRY**

A typical declaration for a database entry structure is:

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

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

## Field Types

Each database field has a type as defined in the next chapter. For static database access a new and simpler set of field types are defined. In addition, at runtime, a database field can be an array. With static database access, however, all fields are scalars. Static database access field types are called DCT field types.

The DCT field types are:

- **DCT\_STRING**: Character string.
- **DCT\_INTEGER**: Integer value
- **DCT\_REAL**: Floating point number
- **DCT\_MENU**: A set of choice strings
- **DCT\_MENUFORM**: A set of choice strings with associated form.
- **DCT\_INLINK**: Input Link
- **DCT\_OUTLINK**: Output Link
- **DCT\_FWDLINK**: Forward Link
- **DCT\_NOACCESS**: A private field for use by record access routines

A DCT\_STRING field contains the address of a NULL terminated string. The field types DCT\_INTEGER and DCT\_REAL are used for numeric fields. A field that has any of these types can be accessed via the dbGetString, dbPutString, dbVerify, and dbGetRange routines.

The field type DCT\_MENU has an associated set of strings defining the choices. Routines are available for accessing menu fields. A menu field can also be accessed via the dbGetString, dbPutString, dbVerify, and dbGetRange routines.

The field type DCT\_MENUFORM is like DCT\_MENU but in addition the field has an associated link field. The information for the link field can be entered via a set of form manipulation fields.

DCT\_INLINK (input), DCT\_OUTLINK (output), and DCT\_FWDLINK (forward) specify that the field is a link, which has an associated set of static access routines described in the next subsection. A field that has any of these types can also be accessed via the dbGetString, dbPutString, dbVerify, and dbGetRange routines.

---

## Allocating and Freeing DBBASE

---

### dbAllocBase

```
DBBASE *dbAllocBase(void);
```

This routine allocates and initializes a DBBASE structure. It does not return if it is unable to allocate storage.

dbAllocBase allocates and initializes a DBBASE structure. Normally an application does not need to call dbAllocBase because a call to dbReadDatabase or dbReadDatabaseFP automatically calls this routine if pdbname is null. Thus the user only has to supply code like the following:

```
DBBASE *pdbname=0;
...
status = dbReadDatabase(&pdbname, "sample.db",
    "<path>", "<macro substitutions>");
```

The static database access library allows applications to work with multiple databases, each referenced via a different (DBBASE \*) pointer. Such applications may find it necessary to call dbAllocBase directly.

dbAllocBase does not return if it is unable to allocate storage.

### dbFreeBase

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

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

---

## DBENTRY Routines

---

### Alloc/Free DBENTRY

```
DBENTRY *dbAllocEntry(DBBASE *pdbname);
void dbFreeEntry(DBENTRY *pdbentry);
```

These routines allocate, initialize, and free DBENTRY structures. The user can allocate and free DBENTRY structures as necessary. Each DBENTRY is, however, tied to a particular database.

dbAllocEntry and dbFreeEntry act as a pair, i.e. the user calls dbAllocEntry to create a new DBENTRY and calls dbFreeEntry when done.

### dbInitEntry dbFinishEntry

```
void dbInitEntry(DBBASE *pdbname, DBENTRY *pdbentry);
void dbFinishEntry(DBENTRY *pdbentry);
```

The routines dbInitEntry and dbFinishEntry are provided in case the user wants to allocate a DBENTRY structure on the stack. Note that the caller **MUST** call dbFinishEntry before returning from the routine that calls dbInitEntry. An example of how to use these routines is:

```
int xxx(DBBASE *pdbname)
{
    DBENTRY dbentry;
    DBENTRY *pdbentry = &dbentry;
    ...
    dbInitEntry(pdbname, pdbentry);
    ...
    dbFinishEntry(pdbentry);
}
```

### 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.

---

## Read and Write Database

---

### 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.

### 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 dbWriteBreaktable(DBBASE *pdbbase,
    const char *filename);
long dbWriteBreaktableFP(DBBASE *pdbbase,FILE *fp);
```

Each of these routines writes files in the same format accepted by `dbReadDatabase` and `dbReadDatabaseFP`. Two versions of each type are provided. The only difference is that one accepts a filename and the other a “FILE\*”. Thus only one of each type has to be described.

`dbWriteMenu` writes the description of the specified menu or, if `menuName` is `NULL`, the descriptions of all menus.

`dbWriteRecordType` writes the description of the specified record type or, if `recordTypeName` is NULL, the descriptions of all record types.

`dbWriteDevice` writes the description of all devices to stdout.

`dbWriteDriver` writes the description of all drivers to stdout.

### Write Record Instances

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

Each of these routines writes files in the same format accepted by `dbReadDatabase` and `dbReadDatabaseFP`. Two versions of each type are provided. The only difference is that one accepts a filename and the other a "FILE \*". Thus only one of each type has to be described.

`dbWriteRecord` writes record instances. If `precordTypeName` is NULL, then the record instances for all record types are written, otherwise only the records for the specified type are written. `level` has the following meaning:

- 0 - Write only prompt fields that are different than the default value.
- 1 - Write only the fields which are prompt fields.
- 2 - Write the values of all fields.

---

## Manipulating Record Types

---

### Get Number of Record Types

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

This routine returns the number of record types in the database.

### 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)
}
```

### 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.

---

## Manipulating Field Descriptions

---

The routines described in this section all assume that DBENTRY references a record type, i.e. that dbFindRecordType, dbFirstRecordType, or dbNextRecordType has returned success or that a record instance has been successfully located.

### Get Number of Fields

```
int dbGetNFields(DBENTRY *pdbentry, int dctonly);
```

Returns the number of fields for the record instance that DBENTRY currently references.

### 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.

### Get Field Type

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

This routine returns the integer value for a DCT field type, see Section on page 114, for a description of the field types.

### 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.

### 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.

### Get Field Prompt

```
char *dbGetPrompt(DBENTRY *pdbentry);  
int dbGetPromptGroup(DBENTRY *pdbentry);
```

The dbGetPrompt routine returns the character string prompt value, which describes the field. dbGetPromptGroup returns the field group as described in guigroup.h.

---

## 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.

### dbPutRecord Attribute

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

This creates or modifies attribute name with value.

---

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

---

## Manipulating Record Instances

---

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

**Get Number of Records**      `int dbGetNRecords(DBENTRY *pdbentry);`

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

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

These routines are used to locate record instances. If any of these routines returns success, then `DBENTRY` references the record. `dbFindRecord` can be called without `DBENTRY` referencing a valid record type. `dbFirstRecord` only works if `DBENTRY` references a record type. The `dbDumpRecords` example given at the beginning of this chapter shows how these routines can be used.

`dbFindRecord` also calls `dbFindField` if the record name includes a field name, i.e. it ends in “.XXX”. The routine `dbFoundField` returns (TRUE, FALSE) if the field (was, was not) found. If it was not found, then `dbFindField` must be called before individual fields can be used.

**Get Record Name**      `char *dbGetRecordName(DBENTRY *pdbentry);`

This routine only works properly if called after `dbFindRecord`, `dbFirstRecord`, or `dbNextRecord` has returned success.

**Create/Delete/Free Record**      `long dbCreateRecord(DBENTRY *pdbentry, char *precordName);`  
                          `long dbDeleteRecord(DBENTRY *pdbentry);`  
                          `long dbFreeRecords(DBBASE *pdbname);`

`dbCreateRecord`, which assumes that `DBENTRY` references a valid record type, creates a new record instance and initializes it as specified by the record description. If it returns success, then `DBENTRY` references the record just created. `dbDeleteRecord` deletes a single record instance/. `dbFreeRecords` deletes all record instances.

**Copy Record**      `long dbCopyRecord(DBENTRY *pdbentry, char *newRecordName`  
                          `int overwriteOK)`

This routine copies the record instance currently referenced by `DBENTRY`. Thus it creates and 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.

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

This routine renames the record instance currently referenced by DBENTRY. If dbRenameRecord completes successfully, DBENTRY references the renamed record.

**Record Visibility**        These routines are for use by graphical configuration tools.

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

dbVisibleRecord sets a record to be visible. dbInvisibleRecord sets a record invisible. dbIsVisibleRecord returns TRUE if a record is visible and FALSE otherwise.

**Find Field**                `long dbFindField(DBENTRY *pdbentry, char *pfieldName);`  
`int dbFoundField(DBENTRY *pdbentry);`

Given that a record instance has been located, dbFindField finds the specified field. If it returns success, then DBENTRY references that field. dbFoundField returns (FALSE, TRUE) if (no field instance is currently available, a field instance is available).

**Get/Put Field Values**     `char *dbGetString(DBENTRY *pdbentry);`  
`long dbPutString(DBENTRY *pdbentry, char *pstring);`  
`char *dbVerify(DBENTRY *pdbentry, char *pstring);`  
`char *dbGetRange(DBENTRY *pdbentry);`  
`int dbIsDefaultValue(DBENTRY *pdbentry);`

These routines are used to get or change field values. They work on all the database field types except DCT\_NOACCESS but should **NOT** be used to prompt the user for information for DCT\_MENU, DCT\_MENUFORM, or DCT\_LINK\_XXX fields. dbVerify returns (NULL, a message) if the string is (valid, invalid). Please note that the strings returned are volatile, i.e. the next call to a routines that returns a string will overwrite the value returned by a previous call. Thus it is the caller's responsibility to copy the strings if the value must be kept.

DCT\_MENU, DCT\_MENUFORM and DCT\_LINK\_XXX fields can be manipulated via routines described in the following sections. If, however dbGetString and dbPutString are used, they do work correctly. For these field types dbGetString and dbPutString are intended to be used only for creating and restoring versions of a database.

---

## 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.

**Get Number of Menu Choices**     `int dbGetNMenuChoices(DBENTRY *pdbentry);`

This routine returns the number of menu choices for menu.

**Get Menu Choice**            `char **dbGetMenuChoices(DBENTRY *pdbentry);`

This routine returns the address of an array of pointers to strings which contain the menu choices.

### Get/Put Menu

```
int dbGetMenuIndex(DBENTRY *pdbentry);  
long dbPutMenuIndex(DBENTRY *pdbentry, int index);  
char *dbGetMenuStringFromIndex(DBENTRY *pdbentry, int index);  
int dbGetMenuIndexFromString(DBENTRY *pdbentry,  
char *choice);
```

NOTE: These routines do not work if the current field value contains a macro definition.

`dbGetMenuIndex` returns the index of the menu choice for the current field, i.e. it specifies which choice to which the field is currently set. `dbPutMenuIndex` sets the field to the choice specified by the index.

`dbGetMenuStringFromIndex` returns the string value for a menu index. If the index value is invalid NULL is returned. `dbGetMenuIndexFromString` returns the index for the given string. If the string is not a valid choice a -1 is returned.

### Locate Menu

```
dbMenu *dbFindMenu(DBBASE *pdbname, char *name);
```

`dbFindMenu` is most useful for runtime use but is a static database access routine. This routine just finds a menu with the given name.

---

## Manipulating Link Fields

---

### Link Types

Links are the most complicated types of fields. A link can be a constant, reference a field in another record, or can refer to a hardware device. Two additional complications arise for hardware links. The first is that field `DTYP`, which is a menu field, determines if the `INP` or `OUT` field is a device link. The second is that the information that must be specified for a device link is bus dependent. In order to shelter database configuration tools from these complications the following is done for static database access.

- Static database access will treat `DTYP` as a `DCT_MENUFORM` field.
- The information for the link field related to the `DCT_MENUFORM` can be entered via a set of form manipulation routines associated with the `DCT_MENUFORM` field. Thus the link information can be entered via the `DTYP` field rather than the link field.
- The Form routines described in the next section can also be used with any link field.

Each link is one of the following types:

- **DCT\_LINK\_CONSTANT**: Constant value.
- **DCT\_LINK\_PV**: A process variable link.
- **DCT\_LINK\_FORM**: A link that can only be processed via the form routines described in the next chapter.

Database configuration tools can change any link between being a constant and a process variable link. Routines are provided to accomplish these tasks.

The routines `dbGetString`, `dbPutString`, and `dbVerify` can be used for link fields but the form routines can be used to provide a friendlier user interface.

### 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.

### 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.

---

## Manipulating MenuForm Fields

---

These routines are used with a DCT\_MENUFORM field (a DTYP field) to manipulate the associated DCT\_INLINK or DCT\_OUTLINK field. They can also be used on any DCT\_INLINK, DCT\_OUTLINK, or DCT\_FWDLINK field.

### Alloc/Free Form

```
int dbAllocForm(DBENTRY *pdbentry)  
long dbFreeForm(DBENTRY *pdbentry)
```

dbAllocForm allocates storage needed to manipulate forms. The return value is the number of elements in the form. If the current field value contains a macro definition, the number of lines returned is 0.

### Get/Put Form

```
char **dbGetFormPrompt(DBENTRY *pdbentry)  
char **dbGetFormValue(DBENTRY *pdbentry)  
long dbPutForm(DBENTRY *pdbentry, char **value)
```

dbGetFormPrompt returns a pointer to an array of pointers to character strings specifying the prompt string. dbGetFormValue returns the current values. dbPutForm, which can use the same array of values returned by dbGetForm, sets new values.

### Verify Form

```
char **dbVerifyForm(DBENTRY *pdbentry, char **value)
```

dbVerifyForm can be called to verify user input. It returns NULL if no errors are present. If errors are present, it returns a pointer to an array of character strings containing error messages. Lines in error have a message and correct lines have a NULL string.

### 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.

### Example

The following is code showing use of these routines:

```
char **value;  
char **prompt;
```

```

char **error;
int n;

...
n = dbAllocForm(pdbentry);
if(n<=0) {<Error>}
prompt = dbGetFormPrompt(pdbentry);
value = dbGetFormValue(pdbentry);
for(i=0; i<n; i++) {
    printf("%s (%s) : \n",prompt[i],value[i]);
    /*The follwing accepts input from stdin*/
    scanf("%s",value[i]);
}
error = dbVerifyForm(pdbentry,value);
if(error) {
    for(i=0; i<n; i++) {
        if(error[i]) printf("Error: %s (%s) %s\n", prompt[i],
            value[i],error[i]);
    }
}else {
    dbPutForm(pdbentry,value)
}
dbFreeForm(pdbentry);

```

All value strings are MAX\_STRING\_SIZE in length.

A set of form calls for a particular DBENTRY, **MUST** begin with a call to dbAllocForm and end with a call to dbFreeForm. The values returned by dbGetFormPrompt, dbGetFormValue, and dbVerifyForm are valid only between the calls to dbAllocForm and dbFreeForm.

---

## Find Breakpoint Table

---

```
brkTable *dbFindBrkTable(DBBASE *pddbbase,char *name)
```

This routine returns the address of the specified breakpoint table. It is normally used by the runtime breakpoint conversion routines so will not be discussed further.

---

## Dump Routines

---

```

void dbDumpPath(DBBASE *pddbbase)
void dbDumpRecord(DBBASE *pddbbase,char *precordTypeName,
    int level);
void dbDumpMenu(DBBASE *pddbbase,char *menuName);
void dbDumpRecordType(DBBASE *pddbbase,char *recordTypeName);
void dbDumpFldDes(DBBASE *pddbbase,char *recordTypeName,
    char *fname);
void dbDumpDevice(DBBASE *pddbbase,char *recordTypeName);

```

```
void dbDumpDriver(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. `dbDumpRecord`, `dbDumpMenu`, and `dbDumpDriver` just call the corresponding `dbWritexxxFP` routine specifying `stdout` for the file. `dbDumpRecDes`, `dbDumpFldDes`, and `dbDumpDevice` give internal information useful on an ioc. Note that all of these commands can be executed on an ioc. Just specify `pdbbase` as the first argument.

---

## Examples

---

### Expand Include

This example is like the `dbExpand` utility, except that it doesn't allow path or macro substitution options. It reads a set of database definition files and writes all definitions to `stdout`. All include statements appearing in the input files are expanded.

```
/* dbExpand.c */
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <epicsPrint.h>
#include <dbStaticLib.h>

DBBASE *pdbbase = NULL;

int main(int argc, char **argv)
{
    long    status;
    int     i;
    int     arg;

    if(argc<2) {
        printf("usage: expandInclude file1.db file2.db...\n");
        exit(0);
    }
    for(i=1; i<argc; i++) {
        status = dbReadDatabase(&pdbbase, argv[i], NULL, NULL);
        if(!status) continue;
        fprintf(stderr, "For input file %s", argv[i]);
        errMessage(status, "from dbReadDatabase");
    }
    dbWriteMenuFP(pdbbase, stdout, 0);
    dbWriteRecordTypeFP(pdbbase, stdout, 0);
    dbWriteDeviceFP(pdbbase, stdout);
    dbWriteDriverFP(pdbbase, stdout);
    dbWriteRecordFP(pdbbase, stdout, 0, 0);
    return(0);
}
```

---

```
}
```

## dbDumpRecords

NOTE: This example is similar but not identical to the actual dbDumpRecords routine.

The following example demonstrates how to use the database access routines. The example shows how to locate each record and display each field.

```
void dbDumpRecords(DBBASE *pdbbase)
{
    DBENTRY *pdbentry;
    long status;

    pdbentry = dbAllocEntry(pdbbase);
    status = dbFirstRecordType(pdbentry);
    if(status) {printf("No record descriptions\n");return;}
    while(!status) {
        printf("record type: %s",dbGetRecordTypeName(pdbentry));
        status = dbFirstRecord(pdbentry);
        if(status) printf(" No Records\n");
        else printf("\n Record:%s\n",dbGetRecordName(pdbentry));
        while(!status) {
            status = dbFirstField(pdbentry,TRUE);
            if(status) printf(" No Fields\n");
            while(!status) {
                printf(" %s:%s",dbGetFieldName(pdbentry),
                    dbGetString(pdbentry));
                status=dbNextField(pdbentry,TRUE);
            }
            status = dbNextRecord(pdbentry);
        }
        status = dbNextRecordType(pdbentry);
    }
    printf("End of all Records\n");
    dbFreeEntry(pdbentry);
}
```



# Chapter 12: Runtime Database Access

---

## Overview

---

This chapter describes routines for manipulating and accessing an initialized IOC database.

This chapter is divided into the following sections:

- Database related include files. All of interest are listed and those of general interest are discussed briefly.
- Runtime database access overview.
- Description of each runtime database access routine.
- Runtime modification of link fields.
- Lock Set Routines
- Database to Channel Access Routines
- Old Database Access. This is the interface still used by Channel Access and thus by Channel Access clients.

---

## Database Include Files

---

Directory `base/include` contains a number of database related include files. Of interest to this chapter are:

- **dbDefs.h**: Miscellaneous database related definitions
- **dbFldTypes.h**: Field type definitions
- **dbAccess.h**: Runtime database access definitions.
- **link.h**: Definitions for link fields.

### dbDefs.h

This file contains a number of database related definitions. The most important are:

- **PVNAME\_SZ**: The number of characters allowed in the record name.
- **FLDNAME\_SZ**: The number of characters formerly allowed in a field name. This restriction no longer applies in any base software except `dbCaLink.c`. THIS SHOULD BE FIXED. It is unknown what effect removing this restriction will have on Channel Access Clients.
- **MAX\_STRING\_SIZE**: The maximum string size for string fields or menu choices.
- **DB\_MAX\_CHOICES**: The maximum number of choices for a choice field.

### 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**: ASCII character string
- **DBF\_CHAR**: Signed character
- **DBF\_UCHAR**: Unsigned character
- **DBF\_SHORT**: Short integer
- **DBF\_USHORT**: Unsigned short integer
- **DBF\_LONG**: Long integer
- **DBF\_ULONG**: Unsigned long integer
- **DBF\_FLOAT**: Floating point number
- **DBF\_DOUBLE**: Double precision float
- **DBF\_ENUM**: An enumerated field
- **DBF\_MENU**: A menu choice field
- **DBF\_DEVICE**: A device choice field
- **DBF\_INLINK**: Input Link
- **DBF\_OUTLINK**: Output Link
- **DBF\_FWDLINK**: Forward Link
- **DBF\_NOACCESS**: A private field for use by record access routines

A field of type `DBF_STRING`, ..., `DBF_DOUBLE` can be a scalar or an array. A `DBF_STRING` field contains a `NULL` terminated ascii string. The field types `DBF_CHAR`, ..., `DBF_DOUBLE` correspond to the standard C data types.

`DBF_ENUM` is used for enumerated items, which is analogous to the C language enumeration. An example of an enum field is field `VAL` of a multi bit binary record.

The field types `DBF_ENUM`, `DBF_MENU`, and `DBF_DEVICE` all have an associated set of ASCII strings defining the choices. For a `DBF_ENUM`, the record support module supplies values and thus are not available for static database access. The database access routines locate the choice strings for the other types.

`DBF_INLINK` and `DBF_OUTLINK` specify link fields. A link field can refer to a signal located in a hardware module, to a field located in a database record in the same IOC, or to a field located in a record in another IOC. A `DBF_FWDLINK` can only refer to a record in the same IOC. Link fields are described in a later chapter.

`DBF_INLINK` (input), `DBF_OUTLINK` (output), and `DBF_FWDLINK` (forward) specify that the field is a link structure as defined in `link.h`. There are three classes of links:

1. Constant - The value associated with the field is a floating point value initialized with a constant value. This is somewhat of a misnomer because constant link fields can be modified via `dbPutField` or `dbPutLink`.
2. Hardware links - The link contains a data structure which describes a signal connected to a particular hardware bus. See `link.h` for a description of the bus types currently supported.
3. Process Variable Links - This is one of three types:
  - a. `PV_LINK`: The process variable name.
  - b. `DB_LINK`: A reference to a process variable in the same IOC.
  - c. `CA_LINK`: A reference to a variable located in another IOC.

DCT always creates a `PV_LINK`. When the IOC is initialized each `PV_LINK` is converted either to a `DB_LINK` or a `CA_LINK`.

`DBF_NOACCESS` fields are for private use by record processing routines.

## 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`.

## link.h

This header file describes the various types of link fields supported by EPICS.

---

## Runtime Database Access Overview

---

With the exception of record and device support, all access to the database is via the channel or database access routines. Even record support routines access other records only via database or channel access. Channel Access, in turn, accesses the database via database access.

Perhaps the easiest way to describe the database access layer is to list and briefly describe the set of routines that constitute database access. This provides a good look at the facilities provided by the database.

Before describing database access, one caution must be mentioned. The only way to communicate with an IOC database from outside the IOC is via Channel Access. In addition, any special purpose software, i.e. any software not described in this document, should communicate with the database via Channel Access, not database access, even if it resides in the same IOC as the database. Since Channel Access provides network independent access to a database, it must ultimately call database access routines. The database access interface was changed in 1991, but Channel Access was never changed. Instead a module was written which translates old style database access calls to new. This interface between the old and new style database access calls is discussed in the last section of this chapter.

The database access routines are:

- **dbNameToAddr**: Locate a database variable.
  
- **dbGetField**: Get values associated with a database variable.
- **dbGetLink**: Get value of field referenced by database link (Macro)
- **dbGetLinkValue**: Get value of field referenced by database link (Subroutine)
- **dbGet**: Routine called by dbGetLinkValue and dbGetField
  
- **dbPutField**: Change the value of a database variable.
- **dbPutLink**: Change value referenced by database link (Macro)
- **dbPutLinkValue**: Change value referenced by database link (Subroutine)
- **dbPut**: Routine called by dbPutxxx functions.
- **dbPutNotify**: A database put with notification on completion
- **dbNotifyCancel**: Cancel dbPutNotify
- **dbNotifyAdd**: Add a new record for to notify set.
- **dbNotifyCompletion**: Announce that put notify is complete.
  
- **dbBufferSize**: Determine number of bytes in request buffer.
- **dbValueSize**: Number of bytes for a value field.
- **dbGetRset** Get pointer to Record Support Entry Table
- **dbIsValueField** Is this field the VAL field.
- **dbGetFieldIndex** Get field index. The first field in a record has index 0.
- **dbGetNelement** Get number of elements in the field
- **dbIsLinkConnected** Is the link field connected.
- **dbGetPdbAddrFromLink** Get address of DBADDR.
- **dbGetLinkDBFtype** Get field type of link.
  
- **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.

### Database Request Types and Options

Before describing database access structures, it is necessary to describe database request types and request options. When dbPutField or dbGetField are called one of the arguments is a database request type. This argument has one of the following values:

- **DBR\_STRING**: Value is a NULL terminated string

- **DBR\_CHAR**: Value is a signed char
- **DBR\_UCHAR**: Value is an unsigned char
- **DBR\_SHORT**: Value is a short integer
- **DBR\_USHORT**: Value is an unsigned short integer
- **DBR\_LONG**: Value is a long integer
- **DBR\_ULONG**: Value is an unsigned long integer
- **DBR\_FLOAT**: Value is an IEEE floating point value
- **DBR\_DOUBLE**: Value is an IEEE double precision floating point value
- **DBR\_ENUM**: Value is a short which is the enum item
- **DBR\_PUT\_ACKT**: Value is an unsigned short for setting the ACKT.
- **DBR\_PUT\_ACKS**: Value is an unsigned short for global alarm acknowledgment.

The request types `DBR_STRING`, ..., `DBR_DOUBLE` correspond exactly to valid data types for database fields. `DBR_ENUM` corresponds to database fields that represent a set of choices or options. In particular it corresponds to the fields types `DBF_ENUM`, `DBF_DEVICE`, and `DBF_MENU`. The complete set of database field types are defined in `dbFldTypes.h`. `DBR_PUT_ACKT` and `DBR_PUT_ACKS` are used to perform global alarm acknowledgment.

`dbGetField` also accepts argument options which is a mask containing a bit for each additional type of information the caller desires. The complete set of options is:

- **DBR\_STATUS**: returns the alarm status and severity
- **DBR\_UNITS**: returns a string specifying the engineering units
- **DBR\_PRECISION**: returns a long integer specifying floating point precision.
- **DBR\_TIME**: returns the time
- **DBR\_ENUM\_STRS**: returns an array of strings
- **DBR\_GR\_LONG**: returns graphics info as long values
- **DBR\_GR\_DOUBLE**: returns graphics info as double values
- **DBR\_CTRL\_LONG**: returns control info as long values
- **DBR\_CTRL\_DOUBLE**: returns control info as double values
- **DBR\_AL\_LONG**: returns alarm info as long values
- **DBR\_AL\_DOUBLE**: returns alarm info as double values

## Options Example

The file `dbAccess.h` contains macros for using options. A brief example should show how they are used. The following example defines a buffer to accept an array of up to ten float values. In addition it contains fields for options `DBR_STATUS` and `DBR_TIME`.

```
struct buffer {
    DBRstatus
    DBRtime
    float value[10];
} buffer;
```

The associated `dbGetField` call is:

```
long options, number_elements, status;
...
options = DBR_STATUS | DBR_TIME;
number_elements = 10;
status =
dbGetField(paddr, DBR_FLOAT, &buffer, &options, &number_elements);
```

Consult `dbAccess.h` for a complete list of macros.

Structure `dbAddr` contains a field `dbr_field_type`. This field is the database request type that most closely matches the database field type. Using this request type will put the smallest load on the IOC.

Channel Access provides routines similar to `dbGetField`, and `dbPutField`. It provides remote access to `dbGetField`, `dbPutField`, and to the database monitors described below.

## **ACKT and ACKS**

The request types `DBR_PUT_ACKT` and `DBR_PUT_ACKS` are used for global alarm acknowledgment. The alarm handler uses these requests. For each of these types the user (normally channel access) passes an unsigned short value. This value represents:

`DBR_PUT_ACKT` - Do transient alarms have to be acknowledged? (0,1) means (no, yes).

`DBR_PUT_ACKS` - The highest alarm severity to acknowledge. If the current alarm severity is less then or equal to this value the alarm is acknowledged.

---

## **Database Access Routines**

---

### **dbNameToAddr**

Locate a process variable, format:

```
long dbNameToAddr(  
    char *pname, /*ptr to process variable */  
    struct dbAddr *paddr);
```

The most important goal of database access can be stated simply: Provide quick access to database records and fields within records. The basic rules are:

- Call `dbNameToAddr` once and only once for each field to be accessed.
- Read field values via `dbGetField` and write values via `dbPutField`.

The routines described in this subsection are used by channel access, sequence programs, etc. Record processing routines, however, use the routines described in the next section rather than `dbGetField` and `dbPutField`.

Given a process variable name, this routine locates the process variable and fills in the fields of structure `dbAddr`. The format for a process variable name is:

```
"<record_name>.<field_name>"
```

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

```
"sample_name.VAL".
```

The record name is case sensitive. Field names always consist of all upper case letters.

`dbNameToAddr` locates a record via a process variable directory (PVD). It fills in a structure (`dbAddr`) describing the field. `dbAddr` contains the address of the record and also the field. Thus other routines can locate the record and field without a search. Although the PVD allows the record to be located via a hash algorithm and the field within a record via a binary search, it still takes about 80 microseconds (25MHz 68040) to located a process variable. Once located the `dbAddr` structure allows the process variable to be accessed directly.

## Get Routines

### *dbGetField*

Get values associated with a process variable, format:

```
long dbGetField(
    struct dbAddr *paddr,
    short dbrType, /* DBR_XXX */
    void *pbuffer, /*addr of returned data */
    long *options, /*addr of options */
    long *nRequest, /*addr of number of elements */
    void *pfl); /*used by monitor routines */
```

Thus routine locks, calls dbGet, and unlocks.

### *dbGetLink*

#### *dbGetLinkValue*

Get value from the field referenced by a database link, format:

```
long dbGetLink(
    struct db_link *pdbLink, /*addr of database link*/
    short dbrType, /* DBR_XXX*/
    void *pbuffer, /*addr of returned data*/
    long *options, /*addr of options*/
    long *nRequest); /*addr of number of elements desired*/
```

#### NOTES:

- 1) options can be NULL if no options are desired.
- 2) nRequest can be NULL for a scalar.

dbGetLink is actually a macro that calls dbGetLinkValue. The macro skips the call for constant links. User code should never call dbGetLinkValue.

This routine is called by database access itself and by record support and/or device support routines in order to get values for input links. The value can be obtained directly from other records or via a channel access client. This routine honors the link options (process and maximize severity). In addition it has code that optimizes the case of no options and scalar.

### *dbGet*

Get values associated with a process variable, format:

```
long dbGet(
    struct dbAddr*paddr,
    short dbrType, /* DBR_XXX*/
    void *pbuffer, /*addr of returned data
    long *options, /*addr of options*/
    long *nRequest, /*addr of number of elements*/
    void *pfl); /*used by monitor routines*/
```

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

”options” is a read/write field. Upon entry to dbGet, options specifies the desired options. When dbGetField returns, options specifies the options actually honored. If an option is not honored, the corresponding fields in buffer are filled with zeros.

"nRequest" is also a read/write field. Upon entry to dbGet it specifies the maximum number of data elements the caller is willing to receive. When dbGet returns it equals the actual number of elements returned. It is permissible to request zero elements. This is useful when only option data is desired.

"pfl" is a field used by the Channel Access monitor routines. All other users must set pfl=NULL.

dbGet calls one of a number of conversion routines in order to convert data from the DBF types to the DBR types. It calls record support routines for special cases such as arrays. For example, if the number of field elements is greater than 1 and record support routine get\_array\_info exists, then it is called. It returns two values: the current number of valid field elements and an offset. The number of valid elements may not match dbAddr.no\_elements, which is really the maximum number of elements allowed. The offset is for use by records which implement circular buffers.

## Put Routines

### *dbPutField*

Change the value of a process variable, format:

```
long dbPutField(  
    structdbAddr *paddr,  
    short dbrType, /* DBR_xxx*/  
    void *pbuffer, /*addr of data*/  
    long nRequest); /*number of elements to write*/
```

This routine is responsible for accepting data in one of the DBR\_xxx formats, converting it as necessary, and modifying the database. Similar to dbGetField, this routine calls one of a number of conversion routines to do the actual conversion and relies on record support routines to handle arrays and other special cases.

It should be noted that routine dbPut does most of the work. The actual algorithm for dbPutField is:

1. If the DISP field is TRUE then, unless it is the DISP field itself which is being modified, the field is not written.
2. The record is locked.
3. dbPut is called.
4. If the dbPut is successful then:  
If this is the PROC field or if both of the following are TRUE: 1) the field is a process passive field, 2) the record is passive.
  - a. If the record is already active ask for the record to be reprocessed when it completes.
  - b. Call dbScanPassive after setting putf TRUE to show the process request came from dbPutField.
5. The record is unlocked.

### *dbPutLink*

### *dbPutLinkValue*

Change the value referenced by a database link, format:

```
long dbPutLink(  
    structdb_link *pdbLink, /*addr of database link*/  
    short dbrType, /* DBR_xxx*/  
    void *pbuffer, /*addr of data to write*/  
    long nRequest); /*number of elements to write*/
```

`dbPutLink` is actually a macro that calls `dbPutLinkValue`. The macro skips the call for constant links. User code should never call `dbPutLinkValue`.

This routine is called by database access itself and by record support and/or device support routines in order to put values into other database records via output links.

For Channel Access links it calls `dbCaPutLink`.

For database links it performs the following functions:

1. Calls `dbPut`.
2. Implements maximize severity.
3. If the field being referenced is `PROC` or if both of the following are true: 1) `process_passive` is `TRUE` and 2) the record is passive then:
  - a. If the record is already active because of a `dbPutField` request then ask for the record to be reprocessed when it completes.
  - b. otherwise call `dbScanPassive`.

### *dbPut*

Put a value to a database field, format:

```
long dbPut(
    struct dbAddr *paddr,
    short dbrType, /* DBR_XXX*/
    void *pbuffer, /*addr of data*/
    long nRequest); /*number of elements to write*/
```

This routine is responsible for accepting data in one of the `DBR_XXX` formats, converting it as necessary, and modifying the database. Similar to `dbGet`, this routine calls one of a number of conversion routines to do the actual conversion and relies on record support routines to handle arrays and other special cases.

### **Put Notify Routines**

`dbPutNotify` is a request to notify the caller when all records that are processed as a result of a put complete processing. The complication occurs because of record linking and asynchronous records. A put can cause an entire chain of records to process. If any record is an asynchronous record then record completion means asynchronous completion.

The following rules are implemented:

1. If a `putNotify` is already active on the record to which the put is directed, `dbPutNotify` just returns `S_db_Blocked` without calling the callback routine.

In all other cases, i.e. the cases for the following rules, the callback routine will be always be called unless `dbNotifyCancel` is called.

2. The user supplied callback is called when all processing is complete or when an error is detected. If everything completes synchronously the callback routine will be called BEFORE `dbPutNotify` returns.
3. The user supplied callback routine must not issue any calls that block such as Unix I/O requests.
4. In general a set of records may need to be processed as a result of a single `dbPutNotify`. If database access detects that another `dbPutNotify` request is active on any record in the set, other then the record referenced by the `dbPutNotify`, then the `dbPutNotify` request will restarted
5. If a record in the set is found to be active because of a `dbPutField` request then when that record completes the `dbPutNotify` will be restarted.

6. If a record is found to already be active because of the original `dbPutNotify` request then nothing is done. This is what is done now and any attempt to do otherwise could easily cause existing databases to go into an infinite processing loop.

It is expected that the caller will arrange a timeout in case the `dbPutNotify` takes too long. In this case the caller can call `dbNotifyCancel`

### *dbPutNotify*

Perform a database put and notify when record processing is complete.

Format:

```
long dbPutNotify(PUTNOTIFY *pputnotify);
```

where PUTNOTIFY is

```
typedef struct putNotify{
    void      (*userCallback)(struct putNotify *);
    DBADDR   *paddr;      /*dbAddr set by dbNameToAddr*/
    void      *pbuffer;   /*address of data*/
    long      nRequest;   /*number of elements to be written*/
    short     dbrType;    /*database request type*/
    void      *usrPvt;    /*for private use of user*/
    /*The following is status of request.Set by dbPutNotify*/
    long      status;
    /*fields private to database access*/
    ...
}PUTNOTIFY;
```

The caller must allocate a PUTNOTIFY structure and set the fields:

`userCallback` - Routine that is called upon completion  
`paddr` - address of a DBADDR  
`pbuffer` - address of data  
`nRequest` - number of data elements  
`dbrType` - database request type  
`usrPvt` - a void \* field that caller can use as needed.

The status value returned by `dbPutNotify` is either:

- **S\_db\_Pending**: Success: Callback may already have been called or will be called later.
- **S\_db\_Blocked**: The request failed because a `dbPutNotify` is already active in the record to which the put is directed.

When the user supplied callback is called, the status value stored in PUTNOTIFY is one of the following:

- **0**: Success
- **S\_xxxx**: The request failed due to some other error.

The user callback is always called unless `dbPutNotify` returns `S_db_Blocked` or `dbNotifyCancel` is called before the put notify competes.

### *dbNotifyCancel*

Cancel an outstanding `dbPutNotify`.

Format:

```
void dbNotifyCancel(PUTNOTIFY *pputnotify);
```

This cancels an active dbPutNotify.

*dbNotifyAdd* This routine is called by database access itself. It should never be called by user code.

*dbNotifyCompletion* This routine is called by database access itself. It should never be called by user code.

## Utility Routines

*dbBufferSize* Determine the buffer size for a dbGetField request, format:

```
long dbBufferSize(  
    short dbrType, /* DBR_XXX*/  
    long options, /* options mask*/  
    long nRequest); /* number of elements*/
```

This routine returns the number of bytes that will be returned to dbGetField if the request type, options, and number of elements are specified as given to dbBufferSize. Thus it can be used to allocate storage for buffers.

**NOTE:** This should become a Channel Access routine

*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

*dbGetRest* 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.

*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.

*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).

*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`.

*dbIsLinkConnected* Is the link connected.

Format:

```
int dbIsLinkConnected(struct link *plink);
```

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

*dbGetPdbAddrFromL* Get address of DBADDR from link.

*ink*

Format:

```
DBADDR *dbGetPdbAddrFromLink(struct link *plink);
```

This macro returns the address of the DBADDR for a database link and NULL for all other link types.

*dbGetLinkDBFtype* Get field type of a link.

Format:

```
int dbGetLinkDBFtype(struct link *plink);
```

## Attribute Routine

*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")
```

## Process Routines

*dbScanPassive* Process record if it is passive, format:

*dbScanLink*

*dbScanFwdLink*

```
long dbScanPassive(
    struct dbCommon *pfrom,
    struct dbCommon *pto); /* addr of record*/
long dbScanLink(
```

```
    struct dbCommon *pfrom,  
    struct dbCommon *pto);  
void dbScanFwdLink(struct link *plink);
```

dbScanPassive and dbScanLink are given the record requesting the scan, which may be NULL, and the record to be processed. If the record is passive and `pact=FALSE` then dbProcess is called. Note that these routine are called by dbGetLink, dbPutField, and by recGblFwdLink.

dbScanFwdLink is given a link that must be a forward link field. It follows the rules for scanning a forward link. That is for DB\_LINKS it calls dbScanPassive and for CA\_LINKS it does a dbCaPutLink if the PROC field of record is being addressed.

### *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.

---

## Runtime Link Modification

---

Database links can be changed at run time but only via a channel access client, i.e. via calls to dbPutField but not to dbPutLink. The following restrictions apply:

- Only DBR\_STRING is allowed.
- If a link is being changed to a different hardware link type then the DTYP field must be modified before the link field.
- The syntax for the string field is exactly the same as described for link fields in chapter “Database Definition”

NOTE: For this release modification to/from hardware links has not been tested. In addition modification to record/device support will be needed in order to properly support dynamic modification of hardware links.

---

## 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.

---

## Lock Set Routines

---

User code only calls dbScanLock and dbScanUnlock. All other routines are called by iocCore.

### *dbScanLock*

Lock a lock set:

```
long void dbScanLock(struct dbCommon *precord);
```

Lock the lock set to which the specified record belongs.

### *dbScanUnlock*

Unlock a lock set:

```
long void dbScanUnlock(struct dbCommon *precord);
```

Lock the lock set to which the specified record belongs

- dbLockGetLockId* Get lock set id:
- ```
long dbLockGetLockId(struct dbCommon *precord);
```
- Each lock set is assigned a unique ID. This routine retrieves it. This is most useful to determine if two records are in the same lock set.
- dbLockInitRecords* Determine lock sets for each record in database.
- ```
void dbLockInitRecords(dbBase *pdbbase);
```
- Called by `iocInit`.
- dbLockSetMerge* Merge records into same lock set.
- ```
void dbLockSetMerge(struct dbCommon *pfirst,
                    struct dbCommon *psecond);
```
- If specified records are not in same lock set the lock sets are merged. Called by `dbLockInitRecords` and also when links are modified by `dbPutField`.
- dbLockSetSplitSl* Recompute lock sets for given lock set
- ```
void dbLockSetSplit(struct dbCommon *psource);
```
- This is called when `dbPutField` modifies links.
- dbLockSetGblLock* Global lock for modifying links.
- ```
void dbLockSetGblLock(void);
```
- Only one task at a time can modify link fields. This routine provides a global lock to prevent conflicts.
- dbLockSetGblUnlock* Unlock the global lock.
- ```
void dbLockSetGblUnlock(void);
```
- dbLockSetRecordLock* If record is not already scan locked lock it.
- ```
void dbLockSetRecordLock(struct dbCommon *precord);
```

---

## 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.

**Basic Routines**

These routines are normally only called by database access, i.e. they are not called by record support modules.

*dbCaLinkInit*

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

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

*dbCaAddLink*

Add a new channel access link

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

*dbCaRemoveLink*

Remove channel access link.

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

*dbCaGetLink*

Get link value

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

*dbCaPutLink*

Put link value

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

*dbGetNelements*

Get Number of Elements

```
long dbCaGetNelements(struct link *plink,long *nelements);
```

This call, which returns an error if link is not connected, sets the native number of elements.

*dbCaGetSevr*

Get Alarm Severity

```
long dbCaGetSevr(struct link *plink,short *severity);
```

This call, which returns an error if link is not connected, sets the alarm severity.

*dbCaIsLinkConnected*Is Channel Connected

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

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



# Chapter 13: Device Support Library

---

## Overview

---

Include file `devLib.h` provides definitions for a library of routines useful for device and driver modules. These are a new addition to EPICS and are not yet used by all device/driver support modules. Until they are, the registration routines will not prevent addressing conflicts caused by multiple device/drivers trying to use the same VME addresses.

---

## Registering VME Addresses

---

### Definitions of Address Types

```
typedef enum {
    atVMEA16,
    atVMEA24,
    atVMEA32,
    atLast /* atLast must be the last enum in this list */
} epicsAddressType;

char *epicsAddressTypeName[]
= {
    "VME A16",
    "VME A24",
    "VME A32"
};

int EPICStovxWorksAddrType[]
= {
    VME_AM_SUP_SHORT_IO,
    VME_AM_STD_SUP_DATA,
    VME_AM_EXT_SUP_DATA
};
```

### Register Address

```
long devRegisterAddress(
    const char *pOwnerName,
    epicsAddressType addrType,
    void *baseAddress,
    unsigned size,
    void **pLocalAddress);
```

This routine is called to register a VME address. This routine keeps a list of all VME addresses requested and returns an error message if an attempt is made to register any addresses that are already being used. `*pLocalAddress` is set equal to the address as seen by the caller.

**Unregister Address**      `long devUnregisterAddress(  
                                epicsAddressType  addrType,  
                                void  *baseAddress,  
                                const char *pOwnerName);`

This routine releases addresses previously registered by a call to `devRegisterAddress`.

---

## Interrupt Connect Routines

---

### Definitions of Interrupt Types

```
typedef enum {intCPU, intVME, intVXI} epicsInterruptType;
```

### Connect

```
long devConnectInterrupt(  
    epicsInterruptType  intType,  
    unsigned  vectorNumber,  
    void  (*pFunction)(),  
    void  *parameter);
```

### Disconnect

```
long devDisconnectInterrupt(  
    epicsInterruptType  intType,  
    unsigned  vectorNumber);
```

### Enable Level

```
long devEnableInterruptLevel(  
    epicsInterruptType  intType,  
    unsigned  level);
```

### Disable Level

```
long devDisableInterruptLevel(  
    epicsInterruptType  intType,  
    unsigned  level);
```

---

## Macros and Routines for Normalized Analog Values

---

**Normalized GetField**      `long devNormalizedGblGetField(  
                                long  rawValue,  
                                unsigned  nbits,  
                                DBREQUEST  *pdbrequest,  
                                int  pass,  
                                CALLBACK  *pcallback);`

This routine is just like `recGblGetField`, except that if the request type is `DBR_FLOAT` or `DBR_DOUBLE`, the normalized value of `rawValue` is obtained, i.e. `rawValue` is converted to a value in the range  $0.0 \leq \text{value} < 1.0$

---

**Convert Digital Value to a Normalized Double Value**

```
#define devCreateMask(NBITS)((1<<(NBITS))-1)
#define devDigToNml(DIGITAL,NBITS) \
    (((double)(DIGITAL))/devCreateMask(NBITS))
```

**Convert Normalized Double Value to a Digital Value**

```
#define devNmlToDig(NORMAL,NBITS) \
    (((long)(NORMAL)) * devCreateMask(NBITS))
```



# Chapter 14: EPICS General Purpose Tasks

---

## 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 error messages that are never seen. In addition the vxWorks implementation of `fprintf` requires much more stack space than `printf` calls. Another problem with vxWorks is the `logMsg` facility. `logMsg` generates messages at higher priority than all other tasks except the shell. EPICS solves all of these problems via an error message handling facility. Code can call any of the routines `errMessage`, `errPrintf`, or `epicsPrintf`. 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.

---

## General Purpose Callback Tasks

---

### Overview

EPICS provides three general purpose IOC callback tasks. The only difference between the tasks is scheduling priority: Low, Medium, and High. The low priority task runs at a priority just higher than Channel Access, the medium at a priority about equal to the median of the periodic scan tasks, and the high at a priority higher than the event scan task. The callback tasks provide a service for any software component that needs a task under which to run. The callback tasks use the task watchdog (described below). They use a rather generous stack and can thus be used for invoking record processing. For example the I/O event scanner uses the general purpose callback tasks.

The following steps must be taken in order to use the general purpose callback tasks:

1. Include callback definitions:

```
#include <callback.h>
```

2. Provide storage for a structure that is a private structure for the callback tasks:

```
CALLBACK mycallback;
```

It is permissible for this to be part of a larger structure, e.g.

```
struct {
    ...
    CALLBACK mycallback;
    ...
} ...
```

3. Call routines (actually macros) to initialize fields in CALLBACK:

```
callbackSetCallback(VOIDFUNCPTR, CALLBACK *);
```

This defines the callers callback routine. The first argument is the address of a function returning VOID. The second argument is the address of the CALLBACK structure.

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

The first argument is the priority, which can have one of the values: `priorityLow`, `priorityMedium`, or `priorityHigh`. These values are defined in `callback.h`. The second argument is again the address of the CALLBACK structure.

```
callbackSetUser(VOID *, CALLBACK *);
```

This call is used to save a value that can be retrieved via a call to:

```
callbackGetUser(VOID *, CALLBACK *);
```

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

```
callbackRequest(CALLBACK *);
callbackRequestProcessCallback(CALLBACK *);
```

Either can be called from interrupt level code. The callback routine is passed a single argument, which is the same argument that was passed to `callbackRequest`, i.e., the address of the CALLBACK structure.

**Syntax**

The following calls are provided:

```
long callbackInit(void);

void callbackSetCallback(void *pcallbackFunction,
    CALLBACK *pcallback);
void callbackSetPriority(int priority, CALLBACK *pcallback);
void callbackSetUser(void *user, CALLBACK *pcallback);

void callbackRequest(CALLBACK *);
void callbackRequestProcessCallback(CALLBACK *pCallback,
    int Priority, void *pRec);
```

```
void callbackGetUser(void *user, CALLBACK *pcallback);
```

Notes:

- callbackInit is performed automatically when EPICS initializes and IOC. Thus user code never calls this function.
- callbackSetCallback, callbackSetPriority, callbackSetUser, and callbackGetUser are actually macros.
- callbackRequest and callbackRequestProcessCallback can both be called at interrupt level.
- callbackRequestProcessCallback is designed for the completion phase of asynchronous record processing. It issues the calls:

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

ProcessCallback, which is designed for asynchronous device completion applications, consists of the following code:

```
static void ProcessCallback(CALLBACK *pCallback)
{
    dbCommon      *pRec;
    struct rset   *prset;

    callbackGetUser(pRec, pCallback);
    prset = (struct rset *)pRec->rset;
    dbScanLock(pRec);
    (*prset->process)(pRec);
    dbScanUnlock(pRec);
}
```

## 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],
        &pmyStruct->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.

## Callback Queue

The callback requests put the requests on a vxWorks ring buffer. This buffer is set by default to hold 2000 requests. This value can be changed by calling `callbackSetQueueSize` before `incInit` in the startup file. The syntax is:

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

---

## Task Watchdog

---

EPICS provides an IOC task that is a watchdog for other tasks. Any task can make a request to be watched. The task watchdog runs periodically and checks each task in its task list. If any task is suspended, an error message is issued and, optionally, a callback task is invoked. The task watchdog provides the following features:

1. Include module:

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

2. Insert request:

```
taskwdInsert (int tid, VOIDFUNCPTR callback,  
             VOID *userarg);
```

This is the request to include the task with the specified `tid` in the list of tasks to be watched. If `callback` is not `NULL` then if the task becomes suspended, the callback routine will be called with a single argument `userarg`.

3. Remove request:

```
taskwdRemove(int tid);
```

This routine would typically be called from the callback routine invoked when the original task goes into the suspended state.

4. Insert request to be notified if any task suspends:

```
taskwdAnyInsert(void *userpvt, VOIDFUNCPTR callback,  
              VOID *userarg);
```

The callback routine will be called whenever any of the tasks being monitored by the task watchdog task suspends. `userpvt` must have a non `NULL` unique value

taskwdAnyInsert, because the task watchdog system uses this value to determine who to remove if taskwdAnyRemove is called.

5. Remove request for taskwdAnyInsert :

```
taskwdAnyRemove(void *userpvt);
```

userpvt is the value that was passed to taskwdAnyInsert.



# Chapter 15: Database Scanning

---

## Overview

---

Database scanning is the mechanism for deciding when to process a record. Five types of scanning are possible:

- **Periodic:** A record can be processed periodically. A number of time intervals are supported.
- **Event:** Event scanning is based on the posting of an event by another component of the software via a call to the routine `post_event`.
- **I/O Event:** The original meaning of this scan type is a request for record processing as a result of a hardware interrupt. The mechanism supports hardware interrupts as well as software generated events.
- **Passive:** Passive records are processed only via requests to `dbScanPassive`. This happens when database links (Forward, Input, or Output), which have been declared "Process Passive" are accessed during record processing. It can also happen as a result of `dbPutField` being called (This normally results from a Channel Access put request).
- **Scan Once:** In order to provide for caching puts, The scanning system provides a routine `scanOnce` which arranges for a record to be processed one time.

This chapter explains database scanning in increasing order of detail. It first explains database fields involved with scanning. It next discusses the interface to the scanning system. The last section gives a brief overview of how the scanners are implemented.

---

## Scan Related Database Fields

---

The following fields are normally defined via DCT. It should be noted, however, that it is quite permissible to change any of the scan related fields of a record dynamically. For example, a display manager screen could tie a menu control to the SCAN field of a record and allow the operator to dynamically change the scan mechanism.

### SCAN

This field, which specifies the scan mechanism, has an associated menu of the following form:

- Passive:** Passively scanned.
- Event:** Event Scanned. The field `EVNT` specifies event number
- I/O Event scanned.**
- 10 Second:** Periodically scanned - Every 10 seconds
- ...
- .1 Second:** Periodically scanned - Every .1 seconds

**PHAS**

This field determines processing order for records that are in the same scan set. For example all records periodically scanned at a 2 second rate are in the same scan set. All Event scanned records with the same EVNT are in the same scan set, etc. For records in the same scan set, all records with PHAS=0 are processed before records with PHAS=1, which are processed before all records with PHAS=2, etc.

In general it is not a good idea to rely on PHAS to enforce processing order. It is better to use database links.

**EVNT - Event Number**

This field only has meaning when SCAN is set to Event scanning, in which case it specifies the event number. In order for a record to be event scanned, EVNT must be in the range 0,...255. It should also be noted that some EPICS software components will not request event scanning for event 0. One example is the eventRecord record support module. Thus the application developer will normally want to define events in the range 1,...,255.

**PRIO - Scheduling Priority**

This field can be used by any software component that needs to specify scheduling priority, e.g. the event and I/O event scan facility uses this field.

## Software Components That Interact With The Scanning System

**menuScan.ascii**

This file contains definitions for a menu related to field SCAN. The definitions are of the form:

```
menu(menuScan) {
    choice(menuScanPassive, "Passive")
    choice(menuScanEvent, "Event")
    choice(menuScanI_O_Intr, "I/O Intr")
    choice(menuScan10_second, "10 second")
    choice(menuScan5_second, "5 second")
    choice(menuScan2_second, "2 second")
    choice(menuScan1_second, "1 second")
    choice(menuScan_5_second, ".5 second")
    choice(menuScan_2_second, ".2 second")
    choice(menuScan_1_second, ".1 second")
}
```

The first three choices must appear first and in the order shown. The remaining definitions are for the periodic scan rates, which must appear in order of decreasing rate. At IOC initialization, the menu values are read by scan initialization. The number of periodic scan rates and the value of each rate is determined from the menu values. Thus periodic scan rates can be changed by changing menuScan.ascii and loading this version via dbLoadAscii. The only requirement is that each periodic definition must begin with the value and the value must be in units of seconds.

**dbScan.h**

All software components that interact with the scanning system must include this file.

The most important definitions in this file are:

```
/* Note that these must match the first four definitions*/
/* in choiceGbl.ascii*/
#define SCAN_PASSIVE 0
```

```
#define SCAN_EVENT 1
#define SCAN_IO_EVENT 2
#define SCAN_1ST_PERIODIC 3

/*definitions for SCAN_IO_EVENT */
typedef void * IOSCANPVT;
extern int interruptAccept;

long scanInit(void);
void post_event(int event);
void scanAdd(struct dbCommon *);
void scanDelete(struct dbCommon *);
void scanOnce(void *precord);
int scanOnceSetQueueSize(int size);
int scanppl(void); /*print periodic lists*/
int scanpel(void); /*print event lists*/
int scanpiol(void); /*print io_event list*/
void scanIoInit(IOSCANPVT *);
void scanIoRequest(IOSCANPVT);
```

The first set of definitions defines the various scan types. The next two definitions (IOSCANPVT and interruptAccept) are for interfacing with the I/O event scanner. The remaining definitions define the public scan access routines. These are described in the following subsections.

### Initializing Database Scanners

```
scanInit(void);
```

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

### Adding And Deleting Records From Scan List

The following routines are called each time a record is added or deleted from a scan list.

```
scanAdd(struct dbCommon *);
scanDelete(struct dbCommon *);
```

These routines are called by scanInit at IOC initialization time in order to enter all records created via DCT into the correct scan list. The routine dbPut calls scanDelete and scanAdd each time a scan related field is changed (each scan related field is declared to be SPC\_SCAN in dbCommon.ascii). scanDelete is called before the field is modified and scanAdd after the field is modified.

### Declaring Database Event

Whenever any software component wants to declare a database event, it just calls:

```
post_event(event)
```

This can be called by virtually any IOC software component. For example sequence programs can call it. The record support module for eventRecord calls it.

### Interfacing to I/O Event Scanning

Interfacing to the I/O event scanner is done via some combination of device and driver support.

1. Include <dbScan.h>
2. For each separate event source the following must be done:
  - a. Declare an IOSCANPVT variable, e.g.

```
static IOSCANPVT ioscanpvt;
```
  - b. Call scanIoInit, e.g.

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

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

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

This routine is called each time the record pointed to by `precord` is added or deleted from an I/O event scan list. `cmd` has the value (0,1) if the record is being (added to, deleted from) an I/O event list. This routine must give a value to `*ppvt`.

4. Whenever an I/O event is detected call `scanIoRequest`, e.g.

```
scanIoRequest(ioscanpvt)
```

This routine can be called from interrupt level. The request is actually directed to one of the standard callback tasks. The actual one is determined by the `PRIO` field of `dbCommon`.

The following code fragment shows an event record device support module that supports I/O event scanning:

```
#include <vxWorks.h>  
#include <types.h>  
#include <stdioLib.h>  
#include <intLib.h>  
#include <dbDefs.h>  
#include <dbAccess.h>  
#include <dbScan.h>  
#include <recSup.h>  
#include <devSup.h>  
#include <eventRecord.h>  
/* Create the dset for devEventXXX */  
long init();  
long get_ioint_info();  
struct {  
    long number;  
    DEVSUPFUN report;  
    DEVSUPFUN init;  
    DEVSUPFUN init_record;  
    DEVSUPFUN get_ioint_info;  
    DEVSUPFUN read_event;  
} devEventTestIoEvent={  
    5,  
    NULL,  
    init,  
    NULL,  
    get_ioint_info,  
    NULL};  
static IOSCANPVT ioscanpvt;  
static void int_service(IOSCANPVT ioscanpvt)  
{  
    scanIoRequest(ioscanpvt);  
}  
  
static long init()  
{  
    scanIoInit(&ioscanpvt);  
    intConnect(<vector>, (FUNCPTR)int_service, ioscanpvt);  
}
```

```

        return(0);
    }
    static long get_ioint_info(
        int    cmd,
        struct eventRecord    *pr,
        IOSCANPVT    *ppvt)
    {
        *ppvt = ioscanpvt;
        return(0);
    }

```

---

## Implementation Overview

---

The code for the entire scanning system resides in `dbScan.c`, i.e. periodic, event, and I/O event. This section gives an overview of how the code in `dbScan.c` is organized. The listing of `dbScan.c` must be studied for a complete understanding of how the scanning system works.

### Definitions And Routines Common To All Scan Types

Everything is built around two basic structures:

```

struct scan_list {
    FAST_LOCK    lock;
    ELLLIST    list;
    short    modified;
    long    ticks;    /*used only for periodic scan sets*/
};

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

```

Later we will see how `scan_lists` are determined. For now just realize that `scan_list.list` is the head of a list of records that belong to the same scan set (for example, all records that are periodically scanned at a 1 second rate are in the same scan set). The node field in `scan_element` contain the list links. The normal `vxWorks lstLib` routines are used to access the list. Each record that appears in some scan list has an associated `scan_element`. The `SPVT` field which appears in `dbCommon` holds the address of the associated `scan_element`.

The `lock`, `modified`, and `pscan_list` fields allow `scan_elements`, i.e. records, to be dynamically removed and added to scan lists. If `scanList`, the routine which actually processes a scan list, is studied it can be seen that these fields allow the list to be scanned very efficiently if no modifications are made to the list while it is being scanned. This is, of course, the normal case.

The `dbScan.c` module contains several private routines. The following access a single scan set:

- **printList:** Prints the names of all records in a scan set.

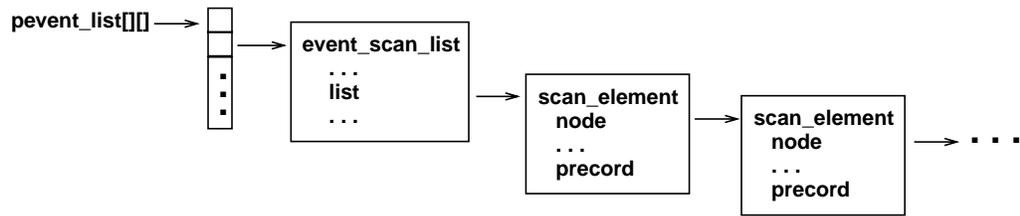


Figure 15-1: Scan List Memory Layout

- **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.

## Event Scanning

Event scanning is built around the following definitions:

```
#define MAX_EVENTS 256
typedef struct event_scan_list {
    CALLBACK      callback;
    scan_list     scan_list;
}event_scan_list;
static event_scan_list
    *pevent_list[NUM_CALLBACK_PRIORITIES][MAX_EVENTS];
```

`pevent_list` is a 2d array of pointers to `scan_lists`. Note that the array allows for 256 events, i.e. one for each possible event number. In other words, each event number and priority has its own scan list. No `scan_list` is actually created until the first request to add an element for that event number. The event scan lists have the memory layout illustrated in Figure 15-1.

### *post\_event*

```
post_event(int event)
```

This routine is called to request event scanning. It can be called from interrupt level. It looks at each `event_scan_list` referenced by `pevent_list[*][event]` (one for each callback priority) and if any elements are present in the `scan_list` a `callbackRequest` is issued. The appropriate callback task calls routine `eventCallback`, which just calls `scanList`.

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

```
struct io_scan_list {
    CALLBACK      callback;
    struct scan_list scan_list;
    struct io_scan_list *next;
```

```

    }
    static struct io_scan_list
        *iosl_head[NUM_CALLBACK_PRIORITIES]
        = {NULL, NULL, NULL};

```

The array `iosl_head` and the field `next` are only kept so that `scanpiol` can be implemented and will not be discussed further. I/O event scanning uses the general purpose callback tasks to perform record processing, i.e. no task is spawned for I/O event. The callback field of `io_scan_list` is used to communicate with the callback tasks.

The following routines implement I/O event scanning:

*scanIoInit*

```

scanIoInit (IOSCANPVT *ppioscanpvt)

```

This routine is called by device or driver support. It is called once for each interrupt source. `scanIoInit` allocates and initializes an array of `io_scan_list` structures; one for each callback priority and puts the address in `pioscanpvt`. Remember that three callback priorities are supported (low, medium, and high). Thus for each interrupt source the structures are illustrated in Figure 15-2:

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

*scanIoRequest*

```

scanIoRequest (IOSCANPVT pioscanpvt)

```

This routine is called to request I/O event scanning. It can be called from interrupt level. It looks at each `io_scan_list` referenced by `pioscanpvt` (one for each callback priority) and if any elements are present in the `scan_list` a `callbackRequest` is issued. The appropriate callback task calls routine `ioeventCallback`, which just calls `scanList`.

**Periodic Scanning**

Periodic scanning is built around the following definitions:

```

static int nPeriodic;
static struct scan_list **papPeriodic;
static int *periodicTaskId;

```

`nPeriodic`, which is determined at `iocInit` time, is the number of periodic rates. `papPeriodic` is a pointer to an array of pointers to `scan_lists`. There is an array element for each scan rate. Thus the structure illustrated in Figure 15-3 exists after `iocInit`.

A periodic scan task is created for each scan rate. The following routines implement periodic scanning:

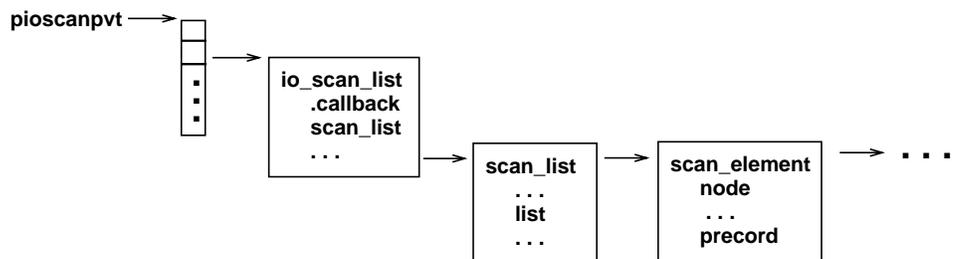


Figure 15-2: Interrupt Source Structure

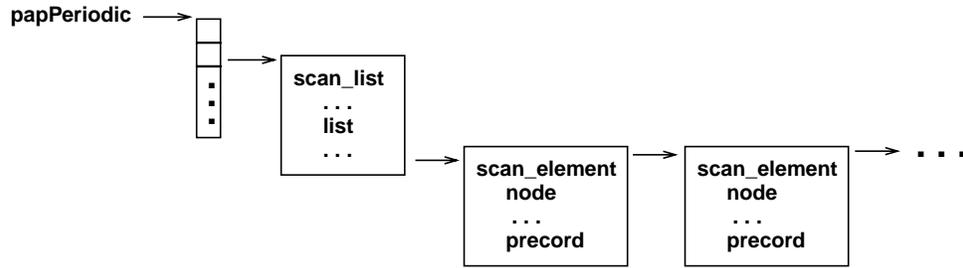


Figure 15-3: Structure after iocInit

*initPeriodic*

```
initPeriodic()
```

This routine first determines the scan rates. It does this by accessing the `SCAN` field of the first record it finds. It issues a call to `dbGetField` with a `DBR_ENUM` request. This returns the menu choices for `SCAN`. From this the periodic rates are determined. The array of pointers referenced by `papPeriodic` is allocated. For each scan rate a `scan_list` is allocated and a `periodicTask` is spawned.

*periodicTask*

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

This task just performs an infinite loop of calling `scanList` and then calling `taskDelay` to wait until the beginning of the next time interval.

**Scan Once**

*scanOnce*

```
void scanOnce (void *precord)
```

A task `onceTask` waits for requests to issue a `dbProcess` request. The routine `scanOnce` puts the address of the record to be processed in a ring buffer and wakes up `onceTask`.

This routine can be called from interrupt level.

*SetQueueSize*

`scanOnce` places its request on a `vxWorks` ring buffer. This is set by default to 1000 entries. It can be changed by executing the following command in the `vxWorks` startup file.

```
int scanOnceSetQueueSize(int size);
```

# Chapter 16: Database Structures

---

## Overview

---

This chapter describes the internal structures describing an IOC database. It is of interest to EPICS system developers but serious application developers may also find it useful. This chapter is intended to make it easier to understand the IOC source listings. It also gives a list of the header files used by IOC Code.

---

## Include Files

---

This section lists the files in base/include that are of most interest to IOC Application Developers:

**alarm.h alarmString.h** - These files contain definitions for all alarm status and severity values.

**caDef.h caErr.h caEventMask.h** - These files are of interest to anyone writing channel access clients.

**callback.h** - The definitions for the General Purpose callback system.

**dbAccess.h** - Definitions for the runtime database access routines.

**dbBase.h** - Definitions for the structures used to store an EPICS database.

**dbDefs.h** - A catchall file for definitions that have no other reasonable place to appear.

**dbFldTypes.h** - Definitions for DBF\_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

**ellLib.h** - A library that provides the same functions as the vxWorks `lstLib`. All routines start with `ell` instead of `lst`. The `ellLib` routines work on both IOCs and on UNIX.

**epicsPrint.h errMdef.h** - EPICS error handling system

**fast\_lock.h** - The FASTLOCK routines.

**freeList.h** - A general purpose free list facility

**gpHash.h** - A general purpose hash library.

**guigroup.h** - The guigroup definitions.

**initHooks.h** - Definitions used by `initHooks.c` routines.

**link.h** - Link definitions

**module\_types.h** - VME hardware configuration. **SHOULD NOT BE USED BY NEW SUPPORT.**

**recSup.h** - The record global routines.

**special.h** - Definitions for special fields, i.e. `SPC_XXX`.

**task\_params.h** - Definitions for task priorities, stack space, etc.

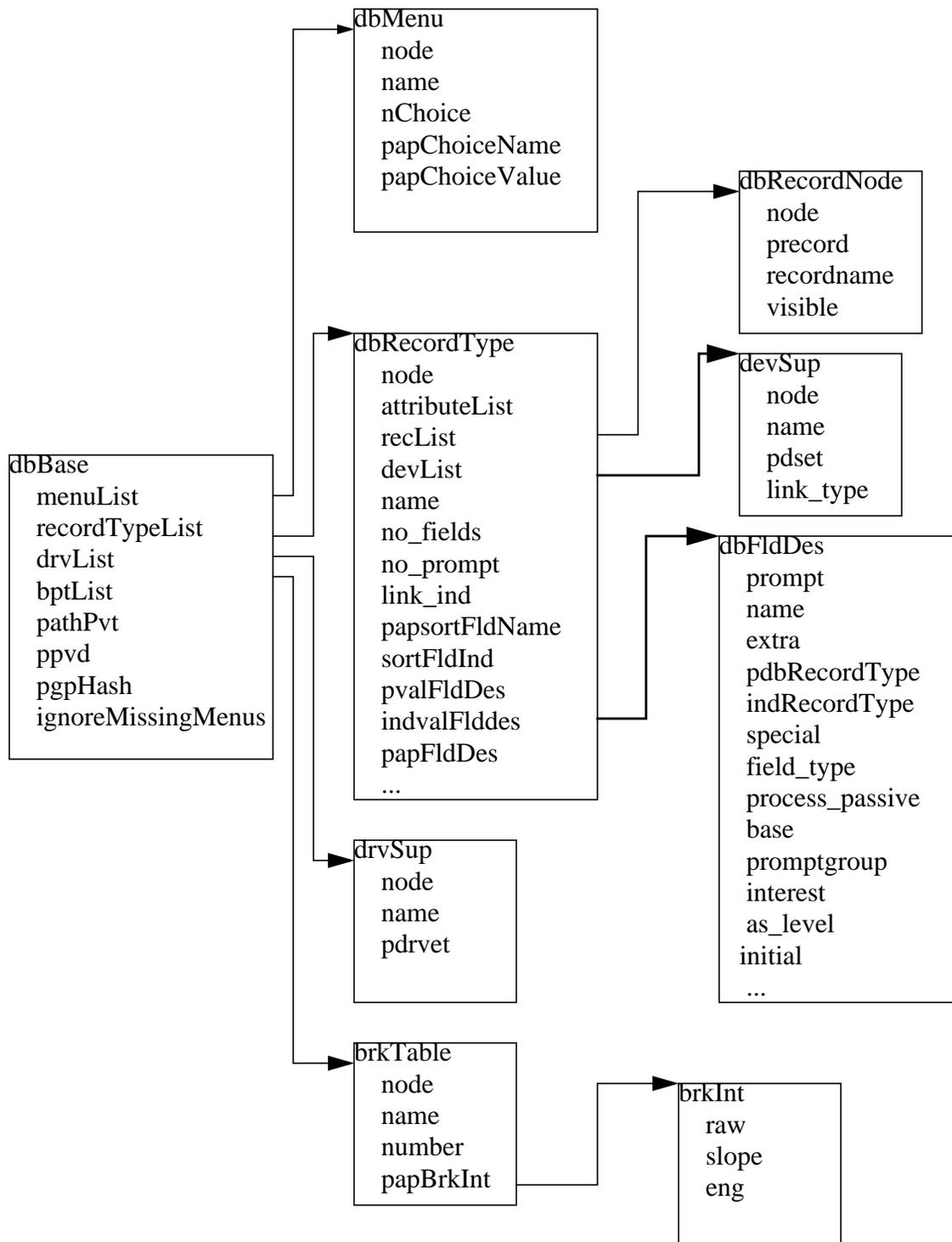
**taskwd.h** - Task Watchdog System

**tsDefs.h** - Time stamp routines. Will also have to look at `base/src/libCom/tsSubr.c`

---

## Structures

---







|                                     |            |                                              |          |
|-------------------------------------|------------|----------------------------------------------|----------|
| database definitions . . . . .      | 25         | DBF_OUTLINK . . . . .                        | 128      |
| Database Format - Summary . . . . . | 25         | DBF_SHORT . . . . .                          | 128      |
| database link . . . . .             | 13         | DBF_UCHAR . . . . .                          | 128      |
| Database Link Guidelines . . . . .  | 16         | DBF_ULONG . . . . .                          | 128      |
| Database Links . . . . .            | 13         | DBF_USHORT . . . . .                         | 128      |
| Database Locking . . . . .          | 15         | DBF_xxx Definitions of Field types . . . . . | 128      |
| Database Scanning . . . . .         | 15         | dbFindBrkTable . . . . .                     | 123      |
| DB_MAX_CHOICES . . . . .            | 127        | dbFindField . . . . .                        | 120      |
| db_post_events . . . . .            | 139        | dbFindMenu . . . . .                         | 121      |
| dba . . . . .                       | 72         | dbFindRecord . . . . .                       | 119      |
| dbAccess.h . . . . .                | 127        | dbFindRecordType . . . . .                   | 117      |
| dbAdd . . . . .                     | 129        | dbFinishEntry . . . . .                      | 115      |
| dbAddPath . . . . .                 | 116        | dbFirstField . . . . .                       | 118      |
| DBADDR . . . . .                    | 129        | dbFirstRecord . . . . .                      | 119      |
| dbAllocBase . . . . .               | 114        | dbFirstRecordType . . . . .                  | 117      |
| dbAllocEntry . . . . .              | 115        | dbFldTypes.h . . . . .                       | 127      |
| dbAllocForm . . . . .               | 122        | dbFoundField . . . . .                       | 118, 120 |
| dbap . . . . .                      | 74         | dbFreeBase . . . . .                         | 115      |
| dbAsciiToMenuH . . . . .            | 40         | dbFreeEntry . . . . .                        | 115      |
| dbAsciiToRecordtypeH . . . . .      | 40         | dbFreeForm . . . . .                         | 122      |
| dbb . . . . .                       | 73         | dbGet . . . . .                              | 133      |
| dbBufferSize . . . . .              | 137        | dbGetDefaultName . . . . .                   | 118      |
| dbc . . . . .                       | 73         | dbGetField . . . . .                         | 133      |
| dbCaAddLink . . . . .               | 142        | dbGetFieldIndex . . . . .                    | 137      |
| dbCaGetLink . . . . .               | 142        | dbGetFieldName . . . . .                     | 118      |
| dbCaGetSevr . . . . .               | 142        | dbGetFieldType . . . . .                     | 118      |
| dbCaLinkInit . . . . .              | 48, 142    | dbGetFormPrompt . . . . .                    | 122      |
| dbCaPutLink . . . . .               | 142        | dbGetFormValue . . . . .                     | 122      |
| dbcar . . . . .                     | 77, 79     | dbGetLink . . . . .                          | 133      |
| dbCaRemoveLink . . . . .            | 142        | dbGetLinkDBFtype . . . . .                   | 138      |
| dbCopyEntry . . . . .               | 116        | dbGetLinkField . . . . .                     | 122      |
| dbCopyEntryContents . . . . .       | 116        | dbGetLinkType . . . . .                      | 122      |
| dbCopyRecord . . . . .              | 119        | dbGetMenuChoices . . . . .                   | 120      |
| dbCreateRecord . . . . .            | 119        | dbGetMenuIndex . . . . .                     | 121      |
| dbCvtLinkToConstant . . . . .       | 122        | dbGetMenuIndexFromString . . . . .           | 121      |
| dbCvtLinkToPvlink . . . . .         | 122        | dbGetMenuStringFromIndex . . . . .           | 121      |
| dbd . . . . .                       | 73         | dbGetNelements . . . . .                     | 138      |
| dbDefs.h . . . . .                  | 127        | dbGetNFields . . . . .                       | 118      |
| dbDeleteRecord . . . . .            | 119        | dbGetNLinks . . . . .                        | 122      |
| dbDumpBreaktable . . . . .          | 124        | dbGetNMenuChoices . . . . .                  | 120      |
| dbDumpDevice . . . . .              | 81, 123    | dbGetNRecords . . . . .                      | 119      |
| dbDumpDriver . . . . .              | 81, 124    | dbGetNRecordTypes . . . . .                  | 117      |
| dbDumpFldDes . . . . .              | 81, 123    | dbGetPdbAddrFromLink . . . . .               | 138      |
| dbDumpMenu . . . . .                | 80–81, 123 | dbGetPrompt . . . . .                        | 118      |
| dbDumpPath . . . . .                | 123        | dbGetPromptGroup . . . . .                   | 118      |
| dbDumpRecord . . . . .              | 123        | dbGetRange . . . . .                         | 120      |
| dbDumpRecords . . . . .             | 81, 125    | dbGetRecordAttribute . . . . .               | 119      |
| dbDumpRecordType . . . . .          | 81, 123    | dbGetRecordName . . . . .                    | 119      |
| DBE_ALARM . . . . .                 | 97         | dbGetRecordTypeName . . . . .                | 117      |
| DBE_LOG . . . . .                   | 97         | dbGetRset . . . . .                          | 137      |
| DBE_VAL . . . . .                   | 97         | dbGetString . . . . .                        | 120      |
| dbel . . . . .                      | 77         | dbgf . . . . .                               | 72       |
| dbExpand . . . . .                  | 42, 124    | dbgrep . . . . .                             | 71       |
| DBF_CHAR . . . . .                  | 128        | dbhcr . . . . .                              | 75, 80   |
| DBF_DEVICE . . . . .                | 128        | dbInitEntry . . . . .                        | 115      |
| DBF_DOUBLE . . . . .                | 128        | dbInvisibleRecord . . . . .                  | 120      |
| DBF_ENUM . . . . .                  | 128        | dbior . . . . .                              | 74       |
| DBF_FLOAT . . . . .                 | 128        | dbIsDefaultValue . . . . .                   | 120      |
| DBF_FWDLINK . . . . .               | 37, 128    | dbIsLinkConnected . . . . .                  | 138      |
| DBF_INLINK . . . . .                | 128        | dbIsValueField . . . . .                     | 137      |
| DBF_LONG . . . . .                  | 128        | dbIsVisibleRecord . . . . .                  | 120      |
| DBF_MENU . . . . .                  | 128        | dbl . . . . .                                | 71       |
| DBF_NOACCESS . . . . .              | 128        | dbLoadDatabase . . . . .                     | 43       |
|                                     |            | dbLoadRecords . . . . .                      | 43       |

|                                                      |         |                                              |     |
|------------------------------------------------------|---------|----------------------------------------------|-----|
| dbLoadTemplate . . . . .                             | 44      | dbScan.h . . . . .                           | 156 |
| dbLockGetLockId . . . . .                            | 141     | dbScanFwdLink . . . . .                      | 139 |
| dbLockInitRecords . . . . .                          | 141     | dbScanLink . . . . .                         | 138 |
| dbLockSetGblLock . . . . .                           | 141     | dbScanLock . . . . .                         | 140 |
| dbLockSetGblUnlock . . . . .                         | 141     | dbScanPassive . . . . .                      | 138 |
| dbLockSetMerge . . . . .                             | 141     | dbScanUnlock . . . . .                       | 140 |
| dbLockSetRecordLock . . . . .                        | 141     | dbstat . . . . .                             | 74  |
| dbLockSetSplitSl . . . . .                           | 141     | dbt . . . . .                                | 78  |
| dblsr . . . . .                                      | 79      | dbtgf . . . . .                              | 78  |
| dbNameToAddr . . . . .                               | 132     | dbToMenuH . . . . .                          | 39  |
| dbNextField . . . . .                                | 118     | dbToRecordtypeH . . . . .                    | 39  |
| dbNextRecord . . . . .                               | 119     | dbtpf . . . . .                              | 79  |
| dbNextRecordType . . . . .                           | 117     | dbtpn . . . . .                              | 79  |
| dbNotifyAdd . . . . .                                | 137     | dbtr . . . . .                               | 72  |
| dbNotifyCancel . . . . .                             | 136     | dbTranslateEscape . . . . .                  | 27  |
| dbNotifyCompletion . . . . .                         | 137     | dbValueSize . . . . .                        | 137 |
| dbnr . . . . .                                       | 73      | dbVerify . . . . .                           | 120 |
| dbp . . . . .                                        | 73      | dbVerifyForm . . . . .                       | 122 |
| dbPath . . . . .                                     | 116     | dbVisibleRecord . . . . .                    | 120 |
| dbpf . . . . .                                       | 72      | dbWriteBreaktable . . . . .                  | 116 |
| dbpr . . . . .                                       | 72      | dbWriteBreaktableFP . . . . .                | 116 |
| dbProcess . . . . .                                  | 139     | dbWriteDevice . . . . .                      | 116 |
| dbPut . . . . .                                      | 135     | dbWriteDeviceFP . . . . .                    | 116 |
| dbPutAttribute . . . . .                             | 38, 138 | dbWriteDriver . . . . .                      | 116 |
| dbPutField . . . . .                                 | 134     | dbWriteDriverFP . . . . .                    | 116 |
| dbPutForm . . . . .                                  | 122     | dbWriteMenu . . . . .                        | 116 |
| dbPutLink . . . . .                                  | 134     | dbWriteMenuFP . . . . .                      | 116 |
| dbPutMenuIndex . . . . .                             | 121     | dbWriteRecord . . . . .                      | 117 |
| dbPutNotify . . . . .                                | 135–136 | dbWriteRecordFP . . . . .                    | 117 |
| dbPutRecordAttribute . . . . .                       | 118     | dbWriteRecordType . . . . .                  | 116 |
| dbPutString . . . . .                                | 120     | dbWriteRecordTypeFP . . . . .                | 116 |
| dbPvdDump . . . . .                                  | 82, 124 | DCT_FWDLINK . . . . .                        | 114 |
| dbPvdTableSize . . . . .                             | 49      | DCT_INLINK . . . . .                         | 114 |
| DBR_AL_DOUBLE . . . . .                              | 131     | DCT_INTEGER . . . . .                        | 114 |
| DBR_AL_LONG . . . . .                                | 131     | DCT_LINK_CONSTANT . . . . .                  | 121 |
| DBR_CHAR . . . . .                                   | 131     | DCT_LINK_DEVICE . . . . .                    | 121 |
| DBR_CTRL_DOUBLE . . . . .                            | 131     | DCT_LINK_FORM . . . . .                      | 121 |
| DBR_CTRL_LONG . . . . .                              | 131     | DCT_LINK_PV . . . . .                        | 121 |
| DBR_DOUBLE . . . . .                                 | 131     | DCT_MENU . . . . .                           | 114 |
| DBR_ENUM . . . . .                                   | 131     | DCT_MENUFORM . . . . .                       | 114 |
| DBR_ENUM_STRS . . . . .                              | 131     | DCT_NOACCESS . . . . .                       | 114 |
| dbr_field_type in DBADDR . . . . .                   | 129     | DCT_OUTLINK . . . . .                        | 114 |
| DBR_FLOAT . . . . .                                  | 131     | DCT_REAL . . . . .                           | 114 |
| DBR_GR_DOUBLE . . . . .                              | 131     | DCT_STRING . . . . .                         | 114 |
| DBR_GR_LONG . . . . .                                | 131     | devConnectInterrupt . . . . .                | 146 |
| DBR_LONG . . . . .                                   | 131     | devCreateMask . . . . .                      | 147 |
| DBR_PRECISION . . . . .                              | 131     | devDisableInterruptLevel . . . . .           | 146 |
| DBR_PUT_ACKS . . . . .                               | 131–132 | devDisconnectInterrupt . . . . .             | 146 |
| DBR_PUT_ACKT . . . . .                               | 131–132 | devEnableInterruptLevel . . . . .            | 146 |
| DBR_SHORT . . . . .                                  | 131     | device . . . . .                             | 26  |
| DBR_STATUS . . . . .                                 | 131     | device - database definition . . . . .       | 33  |
| DBR_TIME . . . . .                                   | 131     | Device Support Entry Table . . . . .         | 91  |
| DBR_UCHAR . . . . .                                  | 131     | devNmlToDig . . . . .                        | 147 |
| DBR_ULONG . . . . .                                  | 131     | devNormalizedGblGetField . . . . .           | 146 |
| DBR_UNITS . . . . .                                  | 131     | devRegisterAddress . . . . .                 | 145 |
| DBR_USHORT . . . . .                                 | 131     | devUnregisterAddress . . . . .               | 146 |
| DBR_xxx Database Request Types and Options . . . . . | 130     | driver . . . . .                             | 26  |
| dbReadDatabase . . . . .                             | 116     | driver - database definition . . . . .       | 34  |
| dbReadDatabaseFP . . . . .                           | 116     | Driver Support Entry Table Example . . . . . | 110 |
| dbReadTest . . . . .                                 | 45      | drvnet_name - driver definition . . . . .    | 34  |
| dbRenameRecord . . . . .                             | 120     | DSET . . . . .                               | 91  |
| dbReportDeviceConfig . . . . .                       | 124     | dset - dbCommon . . . . .                    | 103 |
| dbt . . . . .                                        | 73      | dset_name - device definition . . . . .      | 34  |
|                                                      |         | dtyp - dbCommon . . . . .                    | 103 |

---

**E**

---

elc. . . . . 74, 86  
 Environment Variables . . . . . 51  
 EPICS . . . . . 1, 5  
     Basic Attributes. . . . . 6  
     Hardware/Software Platforms. . . . . 6  
     Overview. . . . . 1  
 EPICS\_CA\_ADDR\_LIST . . . . . 51  
 EPICS\_CA\_AUTO\_ADDR\_LIST . . . . . 51  
 EPICS\_CA\_BEACON\_PERIOD. . . . . 51  
 EPICS\_CA\_CONN\_TMO . . . . . 51  
 EPICS\_CA\_REPEATER\_PORT . . . . . 51  
 EPICS\_CA\_SERVER\_PORT . . . . . 51  
 EPICS\_IOC\_LOG\_FILE\_COMMAND. . . . . 87  
 EPICS\_IOC\_LOG\_FILE\_LIMIT. . . . . 87  
 EPICS\_IOC\_LOG\_FILE\_NAME . . . . . 87  
 EPICS\_IOC\_LOG\_INET . . . . . 51  
 EPICS\_IOC\_LOG\_PORT. . . . . 51, 88  
 EPICS\_TS\_MIN\_WEST . . . . . 51  
 EPICS\_TS\_NTP\_INET . . . . . 51  
 epicsAddressType . . . . . 145  
 epicsAddressTypeName . . . . . 145  
 epicsInterruptType . . . . . 146  
 epicsPrintf . . . . . 85, 101  
 epicsPrtEnvParams . . . . . 78  
 epicsRelease . . . . . 78  
 EPICSstovxWorksAddrType . . . . . 145  
 epicsVprintf. . . . . 85  
 errlog Task . . . . . 85  
 errlogAddListener. . . . . 86  
 errlogFatal. . . . . 84  
 errlogGetSevEnumString . . . . . 84  
 errlogGetSevToLog. . . . . 84  
 errlogInfo . . . . . 84  
 errlogInit . . . . . 49, 86  
 errlogListener . . . . . 86  
 errlogMajor . . . . . 84  
 errlogMessage. . . . . 84  
 errlogMinor. . . . . 84  
 errlogPrintf . . . . . 84  
 errlogRemoveListener. . . . . 86  
 errlogSetSevToLog . . . . . 84  
 errlogSevEnum . . . . . 84  
 errlogSevPrintf . . . . . 84  
 errlogSevVprintf . . . . . 84  
 errlogVprintf. . . . . 84  
 errMessage . . . . . 84  
 errPrintf. . . . . 84-85  
 Escape Sequence. . . . . 27  
 Event . . . . . 155  
 Event - Scan Type. . . . . 155  
 Event Scanning . . . . . 160  
 EVNT - Scan Related Field . . . . . 156  
 extra - field definition rules. . . . . 29  
 extra\_info - field definition . . . . . 31

---

**F**

---

field . . . . . 26  
 field\_name - field definition . . . . . 29

field\_name - record instance definition. . . . . 35  
 field\_size in DBADDR. . . . . 129  
 field\_type in DBADDR. . . . . 129  
 filed\_type - field definition . . . . . 30  
 filename extension conventions . . . . . 27  
 FLDNAME\_SZ. . . . . 127  
 FWDLINK . . . . . 13

---

**G**

---

get\_alarm\_double Record Support Routine . . 100  
 get\_array\_info - Record Support Routine. . . . . 98  
 get\_control\_double - Record Support Routine 100  
 get\_enum\_str - record Support Routine . . . . . 99  
 get\_enum\_strs - record Support Routine . . . . . 99  
 get\_graphic\_double - example . . . . . 94  
 get\_graphic\_double - Record Support Routine. 99  
 get\_joint\_info . . . . . 159  
 get\_joint\_info - device support routine. . . . . 108  
 get\_precision - Record Support Routine. . . . . 99  
 get\_units - example . . . . . 94  
 get\_units - Record Support Routine . . . . . 98  
 gft . . . . . 80  
 GPIB\_IO . . . . . 36  
 grecord . . . . . 26  
 gui\_group - field definition . . . . . 30  
 Guidelines for Asynchronous Records . . . . . 20  
 Guidelines for Synchronous Records . . . . . 19

---

**H**

---

HAG . . . . . 54-56

---

**I**

---

I/O Event - Scan Type. . . . . 155  
 I/O Event scanned. . . . . 155  
 I/O Event Scanning . . . . . 157, 160  
 include. . . . . 26  
 include - Database Definitions . . . . . 28  
 Include File Generation. . . . . 39  
 init - device support routine . . . . . 107  
 init - Record Support Routine. . . . . 97  
 init\_record - device support routine . . . . . 108  
 init\_record - example . . . . . 92  
 init\_record - Record Support Routine. . . . . 97  
 init\_value - field definition . . . . . 30  
 InitDatabase . . . . . 48  
 InitDevSup . . . . . 48  
 InitDrvSup. . . . . 48  
 initHookFunction . . . . . 50  
 initHookRegister. . . . . 50  
 initHooks. . . . . 50  
 initHookState . . . . . 50  
 initial - field definition rules . . . . . 29  
 Initialize Logging . . . . . 51  
 initPeriodic . . . . . 162  
 InitRecSup. . . . . 48

INLINK ..... 13  
 INP ..... 55  
 Input/Output Controller ..... 1  
   Hardware/Software Platforms..... 6  
   Software Components..... 7  
 INST\_IO ..... 36  
 interest - field definition rules ..... 29  
 interest\_level - field definition ..... 31  
 interruptAccept ..... 48  
 IOC ..... 5  
   See Input/Out Controller  
 IOC Error Logging ..... 83  
 iocInit ..... 48  
 iocLogClient ..... 87  
 iocLogDisable ..... 87  
 iocLogServer..... 87

---

**K**

---

Keywords ..... 26

---

**L**

---

LAN ..... 5  
 link.h ..... 127  
 LINK\_ALARM..... 14  
 link\_type - device definition ..... 34  
 Local Area Network  
   Hardware/Software Platforms..... 6  
 logMsg ..... 87

---

**M**

---

Macro Substitution ..... 27  
 MAX\_STRING\_SIZE ..... 127  
 Maximize Severity ..... 14  
 menu ..... 26  
 menu - Database Definition ..... 28  
 menu - field definition rules ..... 29  
 menuScan.ascii ..... 156  
 monitor - example..... 96  
 MS..... 14  
 Multiple Definitions ..... 27

---

**N**

---

name - breakpoint table..... 34  
 NMS ..... 14  
 no\_elements in DBADDR ..... 129  
 NPP..... 14

---

**O**

---

Operator Interface  
   Hardware/Software Platforms..... 6

OPI ..... 5  
 OUTLINK..... 13  
 Overview of Record Processing ..... 89

---

**P**

---

Passive..... 155  
 Passive - Scan Type..... 155  
 path ..... 26  
 path - Database Definitions..... 27  
 Periodic - Scan Type..... 155  
 Periodic Scanning..... 161  
 periodicTask ..... 162  
 pfield in DBADDR ..... 129  
 pfldDes in DBADDR ..... 129  
 pft ..... 80  
 PHAS - Scan Related Field..... 156  
 post\_event ..... 157, 160  
 PP ..... 14  
 pp - field definition rules ..... 29  
 pp\_value - field definition ..... 31  
 precord - DBADDR ..... 129  
 PRIO - Scan Related Field ..... 156  
 process - example ..... 93  
 process - Record Support Routine ..... 98  
 process - record support routine ..... 16  
 Process Passive ..... 14  
 prompt - field definition rules ..... 29  
 prompt\_value - field definition ..... 30  
 ..... 29  
 Psuedo Field ..... 37  
 put\_array\_info - Record Support Routine..... 98  
 put\_enum\_str - Record Support Routine ..... 99  
 putenv ..... 51  
 PUTNOTIFY ..... 136  
 PV\_LINK ..... 35  
 PVNAME\_SZ..... 127

---

**Q**

---

Quoted String ..... 27

---

**R**

---

recGblDbaddrError ..... 101  
 recGblFwdLink..... 102  
 recGblGetAlarmDouble ..... 101  
 recGblGetControlDouble ..... 101  
 recGblGetGraphicDouble ..... 101  
 recGblGetPrec..... 102  
 recGblGetTimeStamp ..... 102  
 recGblInitConstantLink ..... 102  
 recGblRecordError ..... 101  
 recGblRecsupError ..... 101  
 recGblResetAlarms ..... 101  
 recGblSetSevr ..... 100  
 record ..... 26  
 record attribute ..... 37

---

|                                                    |     |
|----------------------------------------------------|-----|
| record instance - database definition . . . . .    | 35  |
| Record Instance File . . . . .                     | 25  |
| Record Processing . . . . .                        | 16  |
| Record Support Entry Table . . . . .               | 90  |
| record type - Database Definition . . . . .        | 29  |
| record_name - record instance definition . . . . . | 35  |
| record_type - device definition . . . . .          | 34  |
| record_type - record instance definition . . . . . | 35  |
| record_type - record type definition . . . . .     | 29  |
| recordtype . . . . .                               | 26  |
| report - device support routine . . . . .          | 107 |
| report - Record Support Routine . . . . .          | 97  |
| Resource Definitions . . . . .                     | 52  |
| RF_IO . . . . .                                    | 36  |
| RSET . . . . .                                     | 90  |
| RSET - example . . . . .                           | 91  |
| RULE . . . . .                                     | 55  |
| rules                                              |     |
| field definition . . . . .                         | 29  |

---

**S**

---

|                                              |         |
|----------------------------------------------|---------|
| S_db_Blocked . . . . .                       | 136     |
| S_db_Pending . . . . .                       | 136     |
| SCAN - Scan Related Field . . . . .          | 155     |
| Scan Once - Scan Type . . . . .              | 155     |
| Scan Related Database Fields . . . . .       | 155     |
| SCAN_1ST_PERIODIC . . . . .                  | 157     |
| scanAdd . . . . .                            | 157     |
| scanDelete . . . . .                         | 157     |
| scanInit . . . . .                           | 157     |
| scanIoInit . . . . .                         | 161     |
| scanIoRequest . . . . .                      | 161     |
| scanOnce . . . . .                           | 162     |
| scanOnceSetQueueSize . . . . .               | 49, 162 |
| scanpel . . . . .                            | 75      |
| scanpiol . . . . .                           | 75      |
| scanppl . . . . .                            | 75      |
| size - field definition rules . . . . .      | 29      |
| size_value - field definition . . . . .      | 31      |
| SPC_ALARMACK . . . . .                       | 31      |
| SPC_AS . . . . .                             | 31      |
| SPC_CALC . . . . .                           | 31      |
| SPC_DBADDR . . . . .                         | 31      |
| SPC_LINCONV . . . . .                        | 31      |
| SPC_MOD . . . . .                            | 31      |
| SPC_NOMOD . . . . .                          | 31      |
| SPC_RESET . . . . .                          | 31      |
| SPC_SCAN . . . . .                           | 31      |
| special - field definition rules . . . . .   | 29      |
| special - Record Support Routine . . . . .   | 98      |
| special in DBADDR . . . . .                  | 129     |
| special_value - field definition . . . . .   | 31      |
| status codes . . . . .                       | 86      |
| struct dbAddr . . . . .                      | 129     |
| struct putNotify . . . . .                   | 136     |
| synchronous device support example . . . . . | 104     |

---

| <hr/>                                        |       |
|----------------------------------------------|-------|
| <b>T</b>                                     |       |
| <hr/>                                        |       |
| taskwd.h . . . . .                           | 152   |
| taskwdAnyInsert . . . . .                    | 152   |
| taskwdAnyRemove . . . . .                    | 153   |
| taskwdInsert . . . . .                       | 152   |
| taskwdRemove . . . . .                       | 152   |
| timexN . . . . .                             | 78    |
| tpn . . . . .                                | 80    |
| Ts_init . . . . .                            | 48    |
| TSConfigure . . . . .                        | 49    |
| TSconfigure . . . . .                        | 49    |
| TSreport . . . . .                           | 75    |
| <hr/>                                        |       |
| <b>U</b>                                     |       |
| UAG . . . . .                                | 54–55 |
| Unquoted String . . . . .                    | 27    |
| <hr/>                                        |       |
| <b>V</b>                                     |       |
| value - record instance definition . . . . . | 35    |
| veclist . . . . .                            | 78    |
| VME_AM_EXT_SUP_DATA . . . . .                | 145   |
| VME_AM_STD_SUP_DATA . . . . .                | 145   |
| VME_AM_SUP_SHORT_IO . . . . .                | 145   |
| VME_IO . . . . .                             | 36    |
| VXI_IO . . . . .                             | 37    |
| vxWorks startup command file . . . . .       | 47    |

