EPICS Input / Output Controller (IOC) Application Developer's Guide

Martin R. Kraimer
Argonne National Laboratory
Advanced Photon Source
February 1996
APS Release 3.13 (BETA 3.12.2.2)

Table of Contents

Table of Contents 1

Preface. 1

1. Overview. 1
2. Acknowledgments. 2

Chapter 1: EPICS Overview 5

1. What is EPICS?. 5
2. Basic Attributes. 6
3. Hardware - Software Platforms (Vendor Supplied) 6
4. IOC Software Components. 7
5. Channel Access. 10
6. OPI Tools 11
7. EPICS Core Software 12

Chapter 2: Building IOC Applications 13

1. Overview. 13
2. Quick Start for Beginners 14
3. makeQuickApp 15
4. makeExampleApp. 18
5. Get a new EPICS Release. 19
6. Make Rules 19
7. src/Makefile.Unix 20
8. src/Makefile.Vx 20
9. st.cmd 21
10. Guide to Rest of Application Developers Guide 22

Chapter 3: Database Locking, Scanning, And Processing 23

1. Overview. 23
2. Database Links 23

3. Database Locking 24
4. Database Scanning 25
5. Record Processing 26
6. Guidelines for Creating Database Links 26
7. Guidelines for Synchronous Records 28
8. Guidelines for Asynchronous Records 29
9. Cached Puts 31

Chapter 4: Database Definition 33
1. Overview 33
2. Definitions 33
3. Breakpoint Tables 45
4. Generation of Include Files for Menus and Record Types 46
5. Utility Programs 50

Chapter 5: IOC Initialization 53
1. Overview 53
2. TSconfigure 54
3. initHooks 54
4. dbLoadDatabase, dbLoadRecords, dbLoadTemplate 55
5. Get Resource Definitions 56
6. Initialize Logging 56
7. Start Task Watchdog 57
8. Start Callback Tasks 57
9. Initialize Channel Access Links - Pass 1 57
10. Initialize Driver Support 57
11. Initialize Record Support 57
12. Initialize Device Support 57
13. Initialize Timing System 57
14. Initialize Database 57
15. Initialize Channel Access Links - Pass 2 58
16. Finish Device Support 58
17. Initialize Database Scanners 58
18. Accept Interrupts 58
19. Perform Initial Processing 58
20. Start Channel Access Server 58

Chapter 6: Access Security 59
1. Overview 59
2. User's Guide 59
3. Design Overview 65
4. Access Security Application Programmer's Interface 67
5. Database Access Security 70
6. Channel Access Security 73
7. Access Control: Implementation Overview 74

Chapter 7: IOC Test Facilities 77
1. Overview 77
2. Database List, Get, Put 77
3. Breakpoints 79
4. Hardware Reports 81
5. Scan Reports 81
6. Time Server Report 82
7. Access Security Commands 82

8. Channel Access Reports 83
9. Interrupt Vector. 83
10. Environment Variables 84
11. Database System Test Routines 84
12. Lock Set Routines. 85
13. Old Database Access Testing 85
14. Routines to dump database information 86

Chapter 8: Record Support 89
1. Overview. 89
2. Overview of Record Processing 90
3. Record Support and Device Support Entry Tables. 90
4. Example Record Support Module. 91
5. Record Support Routines 97
6. Global Record Support Routines 100

Chapter 9: Device Support. 105
1. Overview. 105
2. Example Synchronous Device Support Module. 106
3. Example Asynchronous Device Support Module. 107
4. Device Support Routines 110

Chapter 10: Driver Support. 111
1. Overview. 111
2. Device Drivers 112

Chapter 11: Static Database Access 115
1. Overview. 115
2. Definitions. 116
3. Examples. 117
4. Allocating and Freeing DBBASE. 119
5. DBENTRY Routines 120
6. Read and Write Database 120
7. Manipulating Record Descriptions 121
8. Manipulating Record Instances. 122
9. Manipulating Field Descriptions. 123
10. Manipulating Field Values 123
11. Find Breakpoint Table 124
12. Manipulating Menu Fields 124
13. Manipulating Link Fields 125
14. Dump Routines 126

Chapter 12: Runtime Database Access. 127
1. Overview. 127
2. Database Include Files 127
3. Runtime Database Access Overview 130
4. Database Access Routines 132
5. Old Database Access Interface 140

Chapter 13: Device Support Library 141
1. Overview. 141
2. Registering VME Addresses. 141
3. Interrupt Connect Routines. 142
4. Macros and Routines for Normalized Analog Values 143

Chapter 14: EPICS General Purpose Tasks 145
1. Overview 145
2. General Purpose Callback Tasks 146
3. Task Watchdog 148
4. Error Handling 149

Chapter 15: Database Scanning 153
1. Overview 153
2. Scan Related Database Fields 154
3. Software Components That Interact With The Scanning System 154
4. Implementation Overview 157

Chapter 16: Database Structures 163
1. Overview 163
2. Include Files 163
3. Structures 165

Preface

1. Overview

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

The plan of the book is:

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.

IOC Overview
: An overview of what is needed to develop IOC Applications.

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 takes some of the mystery from 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
: Test routines that can be executed via the vxWorks shell.

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, Task Watchdog, and Error Handling.

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, Janet Anderson and Marty Kraimer

EPICS Source/Release Control, Matthew Needes, Janet Anderson, and Marty Kraimer

vxWorks Programmer's Guide, Wind River Systems

vxWorks Reference Manual, Wind River Systems

2. Acknowledgments

The basic model of what an IOC should do and how to do it was developed by Bob Dalesio at LANL/GTA. The principle ideas for Channel Access were developed by Jeff Hill 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.dctsdr) 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.

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

Chapter 1: EPICS Overview

1. What is EPICS?

EPICS consists of a set of software components and tools which 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 Motorola 68xxx 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.

The following figure shows the basic physical structure of a control system implemented via EPICS.

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.

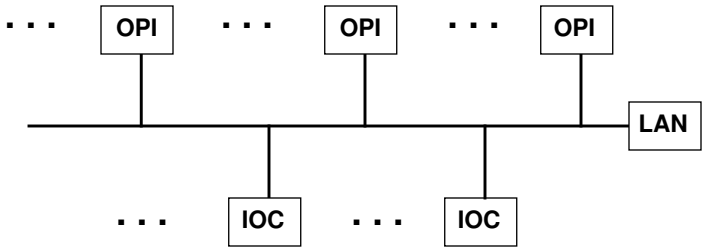


Figure 1-1: EPICS Based Control System

2. Basic Attributes

The basic attributes of EPICS are:

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

3. Hardware - Software Platforms (Vendor Supplied)

OPI

Hardware

- Unix based Workstation, Well supported platforms include SUNOS, SOLARIS, and HP-UX
- Other UNIX platforms have some support. This includes 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 such as Alpha
- 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

4. IOC Software Components

Figure 1-2 contains an overview of the IOC software components and their interactions.

- **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.

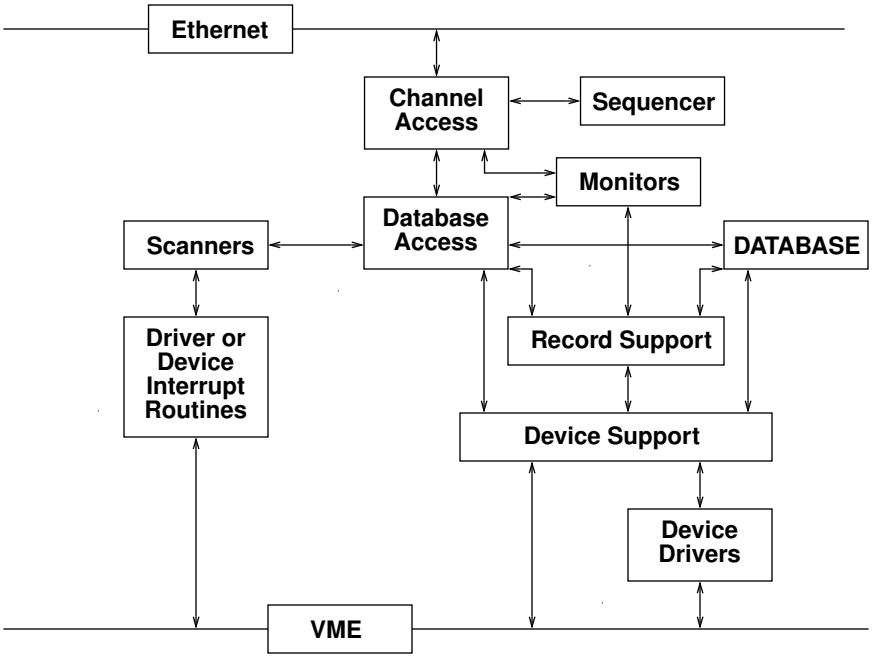


Figure 1-2: System Overview

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 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 "Runtime Database Access" for details.

Database Scanning

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

- **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.

Record Support, Device Support and Device Drivers

In order to remove record specific knowledge from database access, each record type has an associated record support module. Similarly, in order to remove device specific knowledge from record support, each record type can have a set of device support modules. If the method of accessing hardware is complicated, a device driver can be provided to shield the device support modules. Record types *not* associated with hardware do not have device support or 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.

5. Channel Access

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

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 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 process variable values, get and add event, 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. A Channel Access client can communicate with multiple IOCs having differing sets of record types.

Search Server

Channel Access provides a 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 if any of the process variables are located in this IOC, and, if any are found, replies to the sender.

Connection Request Server

For each IOC containing process variables it uses, the Channel Access client issues connection requests. The connection request server accepts the request and establishes a connection to the client. Each such connection is managed by two separate tasks. `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.

6. OPI Tools

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

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 -** CDEV (A library designed to provide a standard API to one or more underlying packages, typically control system interfaces) 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 provides to generate C include files from menu and record type ascii definition files.
- **Source/Release**: EPICS provides a Source/Release mechanism for managing EPICS.

7. EPICS Core Software

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

- Channel Access - Client and Server software
- DATABASE
- Scanners
- Monitors
- GDCT or CAPFAST
- 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.

Chapter 2: Building IOC Applications

1. Overview

This Chapter explains how to create and build ioc applications. This includes:

- IOC databases
- vxWorks startup files
- State Notation Programs
- New record types, device support, drivers
- Other code to be executed in an ioc
- Special Unix code

Several EPICS Application Source/Release systems are available. Your site may have adapted one of them. Consult your EPICS system manager.

This chapter describes procedures based on a simple application environment that can be used at any site. It is not appropriate for large applications but is fine for small applications and for test applications.

It is not a good idea to use this environment for large applications.

If you are new to EPICS or are trying a new release then run `makeExampleApp`, described below. Look at all the files that are generated. They give examples of how to manage each of the ioc components.

User Prerequisites

It is assumed that the reader:

- Understands the C language
- Knows how to use a text editor
- Has at least a superficial knowledge of the make utility

System Prerequisites

Before you can generate EPICS IOC applications your UNIX and/or EPICS system manager must have done the following:

- Installed vxWorks and a board support package
  Consult the vxWorks documentation for details.
- Installed EPICS base
  This can be a specific EPICS release or the master epics tree. Details are given in the manual "EPICS Source/Release Control"

Your EPICS system manager must have installed the extension products `makeQuickApp` and `makeExampleApp` and made them available in your path..

make vs gnumake

EPICS provides an extensive set of make rules. These rules are only guaranteed to work with the GNU version of make, which is supplied by the Free Software Foundation. Thus on most Unix systems the native make will not work. If this is true then this manual assumes that GNU make is invoked by asking for gnumake rather than make. On some systems, e.g. Linux, GNU make may be the default. Ask your EPICS system manager how to invoke GNU make at your site. This chapter always uses gnumake in examples.

2. Quick Start for Beginners

This section explains how to quickly create an example ioc application in a directory called iocapp and located in your home directory.

Create Application

Execute the following commands:

```
cd
makeExampleApp iocapp
cd iocapp
gnumake
```

NOTE: If makeExampleApp fails with a message "No release found" then the default version of EPICS base at your site is not up to date. Ask your EPICS system manager for the full path name of an up to date release of epics and issue the command:

```
makeExampleApp -r <full path name> iocapp
```

boot Parameters

The next step is to set the ioc boot parameters properly via the console serial port on your IOC. Life is much easier if you find out how to connect the serial port to a window on your workstation. See your EPICS system manager for details.

The vxWorks boot parameters look something like the following:

```
boot device          : xxx
processor number     : 0
host name            : xxx
file name            : <path>/iocapp/bin/<arch>/vxWorks
inet on ethernet (e) : xxx.xxx.xxx.xxx:<netmask>
inet on backplane (b):
```

```
host inet (h)         : xxx.xxx.xxx.xxx
gateway inet (g)      :
user (u)              : xxx
ftp password (pw) (blank = use rsh): xxx
flags (f)             : 0x0
target name (tn)      :
startup script (s)    : st.cmd
other (o)             :
```

The actual values for each field are site dependent. Consult your EPICS system manager for help. Two fields that you can change at will are:

- file name
  This is the name of the vxWorks boot image with a full path name. Thus <path> is the full path name of where the makeExampleApp was executed.
- startup script
  This is a vxWorks script to run after your vxWorks has been loaded and initialized.

In order to determine the file name `cd` to the root of your application directory and execute:

```
cd bin
ls
```

This will display a subdirectory for each architecture that has been built at your site (this also includes your Unix host architecture). For example if your vxWorks board support package is for the MV167 processors a subdirectory mv167 will be listed. Now execute the commands:

```
cd <arch>
pwd
```

This will display the complete path name needed for `file name`.

After correctly setting your boot parameters your ioc should boot properly, load EPICS, load the example database, and initialize EPICS. After this you should have a working IOC.

boot and test

You are now ready to boot your ioc. If your boot parameters are defined properly just press the reset button on your ioc. Again you will find it VERY convient to have the console port of the ioc attached to a scrolling window of your workstation. After booting just issue the following vxWorks shell command:

```
dbl
```

This should display the names of your test database record names.

3. makeQuickApp

Two extension products makeQuickApp and makeExampleApp are available to create a simple application area. They are similar except the makeExampleApp includes example code and databases.

makeQuickApp is executed by issuing the Unix command:

```
makeQuickApp -r <epics base release> <name>
```

Both products create a new directory that contains the following files:

```
   Makefile
   epics@
   quickApp or exampleApp
      src/
         Makefile
*        Makefile.Unix
*        Makefile.Vx
*        local.db
*        epicslocal
*        epicsLIBOBJS
*        st.cmd
*        <example and/or user supplied code>
*        <example and/or user supplied state programs>
*        <user menu, recordtype, device, driver database defs>
      records/
*        <record instance file>
      op/
*        <user created directorys>
   db/
      <installed record type definitions>
   include/
      <generated include files for menus and record types>
   bin/
      <host_arch>
         < installed Unix executables >
      <ioc_arch>
         <installed ioc products>
      ...
```

Files marked with an "*" are user created and/or edited. Each such file is discussed below. The directories are:

epics - A soft link to epics which is created by getrel.
exampleApp - Directory containing example application.
quickApp - Directiory containing quick application.
src - The directory in which all ioc and unix code is created and built
records - The directory in which all record instance files are created and edited.
op - If a user wants to create and manage additional files, such as edd/medm adl files then additional directories should be created and managed in the `op` directory
db - The directory into which record type definition files are installed
include - The directory in which include files generated from menu and record type definitions are installed.
bin - This is the directory in which executables, binaries, etc are installed. The user should NEVER edit or create anything in this directory

If a user wants to create and manage additional files, such as edd or medm adl files, then additional directories should be created and managed in the `op` directory.

local.db epicslocal.db

These files are used to configure epics database definitions for the record, device, and driver support needed by your ioc. `epicslocal.db` contains definitions for support obtained from epics and `local` defines private support created locally. `epicslocal` looks like:

```
include "menuGlobal.db"
include "menuConvert.db"
include "aiRecord.db"
#include "aaiRecord.db"
...
device(ai,CONSTANT,devAiSoft,"Soft Channel")
#device(ai,CONSTANT,devAiSoftRaw,"Raw Soft Channel")
...
#driver(drvXy010)
#driver(drvVxi)
...
```

Thus it has a definition for all record, device, and drivers supplied in epics base. Some record types and ALL hardware device and driver support are preceeded by the comment symbol "#". You are expected to edit this file so that all and only the support you need is uncommented.

File epicslocal.db looks like:

```
include "epicslocal.db"
#include "xxxRecord.db"
#device(xxx,CONSTANT,devXxxSoft,"SoftChannel")
```

The first statement includes everything in epicslocal.db. The remaining are definitions for any locally supplied menu, recordtype, device, driver, or breakpoint tables. Again you must edit this file whenever you add new support. Makefile.Vx includes rules that will cause this file to be expanded into a single file containing all definitions used by the ioc.

epicsLIBOBJS

This file allows you to define all the object modules for record, device, and driver support that your ioc must obtain from epics. It contains definitions as follows:

```
#LIBOBJS += $(EPICS_BASE_BIN)/aaiRecord.o
#LIBOBJS += $(EPICS_BASE_BIN)/aaoRecord.o
LIBOBJS += $(EPICS_BASE_BIN)/aiRecord.o
...
#
# Device Support
#
#LIBOBJS += $(EPICS_BASE_BIN)/devAaiCamac.o
...
#
# Driver support ANSI
#
#LIBOBJS += $(EPICS_BASE_BIN)/drvAb.o
...
```

Thus it has a definition for all record, device, and drivers supplied in epics base. Some record types and ALL hardware device and driver support are preceeded by the comment symbol "#". You are expected to edit this file so that all and only the support you need is uncommented. Makefile.Vx contains rules that will combine all support into a single module called "support"

4. makeExampleApp

makeExampleApp is executed by issuing the Unix command:

```
makeExampleApp -r <epics base release> <name>
```

makeExampleApp creates the following examples:

```
records/
   dbExample.db
src/
   caExample.c
   sncExample.st
   xxxRecord.db
   xxxRecord.c
   devXxxSoft.c
```

dbExample.db This is an example set of records instances. Each name is proceeded by "<userid>:" which is the userid of the person who executed makeExampleApp. The prefix is not shown in the following. The records are:

aiExample This is a passive ai (analog input) record which obtains it's input from record calcExample.

calcExample This is a calc (calculation) record that acts as a counter that continually counts from 0 to 9. It is scanned once a second. It also has a forward link to aiExample. Since aiExample is passive it will also scan once a second.

xxxExample This is a sample record of type xxx, as described below. It is a passive record. You can change its VAL field via a channel access client or via the dbpf ioc command.

caExample.c A Unix application that interfaces to Channel Access. It is executed from a Unix shell by issuing the command:

caExample "pvname"

It issues a channel access get request for the specified process variable and prints the value.

sncExample.st This is a sequencer, i.e. state notation language, example. It prints a message on the ioc console every time the val field of xxxRecord becomes > 5.0 and also everytime it becomes <= 5.0.

xxxRecord.db
xxxRecord.c A skeleton record support module. The record support module is the one described in chapter "Record Support".

devXxxSoft.c A device support module for xxxRecord. The device support module provides synchronous support for the record support.

5. Get a new EPICS Release

If you have an existing application area created by makeQuickApp or makeExampleApp and want to link to a new epics release, issue the following commands at the top of your application tree:

```
gnumake clean uninstall
<full path name>/base/tools/getrel <full path name>
gnumake
```

<full path name> is the complete path name to the epics release desired. For example, at ANL/APS we store all epics releases under /usr/local/epics. Thus to get release 3.12.13 I would give the command:

```
/usr/local/epics/R3.12.13/base/bin/solaris/getrel \
   /usr/local/epics/R3.12.13
```

getrel creates or replaces the following files:

epics
: This is a soft link to the epics release

.current_rel_hist
: This is a history of all epics releases ontained via the getrel command.

6. Make Rules

Make can be executed at the top level application directory, in the *App directory, or in the `src` directory. The same set

top Level Make The most useful commands at the top level directory are:

```
gnumake clean uninstall
```

This command removes everything created by make. If you are confused about the state of the generated files run this and then remake everything. This should also be done whenever you get a new release with the getrel command described below.

The other useful command to run at the top level is:

```
gnumake
```

This makes everything, which for the simple application environment created by makeQuickApp, is the same as executing `gnumake` in the `src` directory.

Application Level Make For the simple application areas described here, this has the same effect as issuing:

`gnumake`

in the `src` directory.

src Make

Running the command:

```
gnumake
```

in the src directory build all Unix and IOC components as described in the files Makefile.Unix and Makefile.Vx. Executing the Unix `ls` command shows that directories `O.<arch>` appear. These are the directories where make actually builds. It is possible to build for a single architecture via the command:

```
gnumake install.<arch>
```

For example if your ioc is an MV167 system then the directory is `O.mv167` and the make command is:

```
gnumake install.mv167
```

Another useful command is:

```
gnumake clean
```

This removes everything generated in the O. directories. Again a ".<arch>" can be appended to invoke clean for a particular architecture.

7. src/Makefile.Unix

This file specifies Unix components. Since this manual is primarily conmcerned with IOC applications, `Makefile.Unix` will not be discussed. The `Makefile.Unix` created by `makeQuickApp` or `makeExampleApp` show how to build a single product consisting of a single source module. For more complicated cases consult the "EPICS Source/Release Control" manual.

8. src/Makefile.Vx

This file specifies IOC components. The following components can be built:

Record Support

In order to build support for a new record type the following definitions have to be added to the makefile:

```
RECTYPES += xxxRecord.h
LIBOBJS += xxxRecord.o
```

Device, Driver, other C modules

For each such module add a definition:

```
LIBOBJS += <name>.o
```

Expanded Database Definition File

makeQuickApp has a definition:

```
DBEXPAND += local.db
```

Normally this is the only such definition needed. This rule results in a single file `local.db` being installed into the application bin directory.

State Notation Programs

For each state notation program add the definition:

```
LIBOBJS += <name>.o
```

The state notation program must be stored in a file named `<name>.st`.

Scripts, etc

A definition of the form:

```
SCRIPTS += <name>
```

results in the file `<name>` being installed from the src directory to the `bin/<arch>` directory. The makefile created by makeQuickApp already has a definition to install `st.cmd`.

vxWorks vxWorks.sym

In order to have definitions for vxWorks and for your startup file in the bin directory, the following must appear:

```
SCRIPTS += st.dmd
SCRIPTS += vxWorks
SCRIPTS += targetbin
```

9. st.cmd

NOTE: The commands dbLoadDatabase, dbExpandInclude, and dbLoadRecords are described in the chapter "Database Definition".

This file, which is maintained in src and installed in bin/<arch>, is the vxWorks startup file. The version supplied by makeQuickApp is:

```
# Example vxWorks startup file
ld < targetbin/iocCore
#if any sequence programs are loaded
#ld < targetbin/seq
ld < support

dbLoadDatabase("local.db")
#dbLoadRecords("../../records/xxx.db","user=USER")

iocLogDisable=1
iocInit

#start sequence programs
#seq &sncxxx
```

The `ld` command loads the core epics components. If any sequence programs are to be loaded the sequencer itself must be loaded. The file support is installed when make is run in the src directory. It contains the executables for all record, device, and driver support.

The `dbLoadDatabase("local.db")` loads the definitions of all menus, record types, device support, driver support, and breakpoint tables needed in this ioc. This is actually an expanded file created by `dbExpandInclude` and installed into `bin/<arch>`.

The command

```
dbLoadRecords("../../records/xxx.db","user=USER")
```

Is an example command for loading record instances. One of these command is supplied for each record instance file.

The `iocLogDisable` command disables logging of error messages. For testing this is normally a VERY good idea.

The `iocInit` command initializes the epics system.

The remaining commands show how to load and start sequence programs.

10. Guide to Rest of Application Developers Guide

Before continuing with building an application you need to become familiar with more of the Application Developer's Guide. All IOC application developers must become familiar with the chapters "Database Locking, Scanning, and Processiong" and "IOC Test Facilities". These two chapters should be read before continuing. The next step is to get some experience building databases.

At this point, if you are an experienced C programmer, you are ready to start writing sequence programs or providing specialized record, device, and driver support. For writing sequence programs consult the sequencer manual. Before writing record, device, and driver support you will have to become familiar with the rest of this manual.

Chapter 3: Database Locking, Scanning, And Processing

1. Overview

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

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

2. Database Links

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

- **INLINK**: Input link, used to fetch data.
- **OUTLINK**: Output link, used to write data.

INLINKs and OUTLINKs can be one of the following: constant, database link, channel access link, or a reference to a hardware signal.

- **FWDLINK**: A forward link refers to a record that should be processed whenever the record containing the forward link completes processing.

NOTE: If a forward link is not a database link it is just ignored.

Links are defined in file `link.h`.

The rest of this chapter only discusses 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 if it refers to a passive record that the application developer wants processed after the record containing the link. 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`.

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.

3. 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` afterwords. Because of database links (Input, Output, and Forward) a modification to one record can cause modification to other records. All records linked together, except possibly for input links declared `NPP` and `NMS`, 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. The lock sets are determined during IOC initialization.

4. Database Scanning

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

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

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

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

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

5. 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.

6. 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 (Figure 3-1):

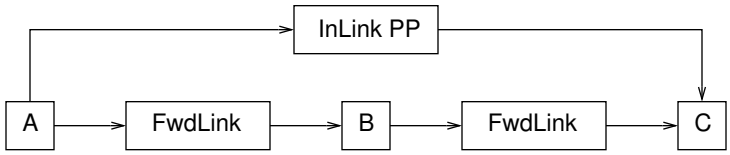


Figure 3-1: Example of Database Links

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

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

This brief example demonstrates that database links needs more discussion.

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`, `FLKN3`, `FLNK4` (Figure 3-2).

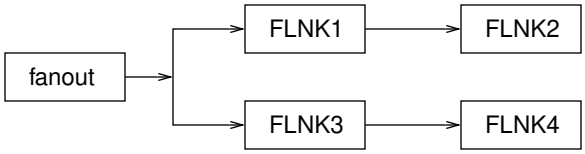


Figure 3-2: Processing Order

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 possibly for `NPP` & `NMS` input links, 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.

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 chapter 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 (Figure 3-3):

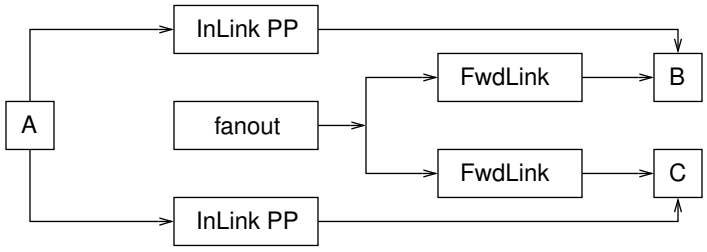


Figure 3-3: Incorrect Link Definition

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 (Figure 3-4).

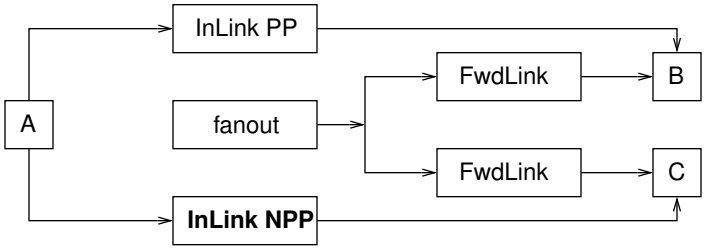


Figure 3-4: Correct Link definition

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`.

7. 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.

8. Guidelines for Asynchronous Records

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

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

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

The asynchronous completion routine performs the following algorithm:

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

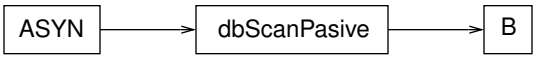
A few attributes of the above rules are:

1. Asynchronous record processing does not delay the scanners.
1. 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.

2. Forward and output links are triggered only when the asynchronous completion routine completes record processing.

With these rules the following works just fine:

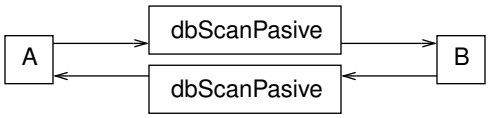


When dbProcess is called for record ASYN, processing will be started but dbScanPassive will not be called. Until the asynchronous completion routine executes any additional attempts to process ASYN are ignored. When the asynchronous callback is invoked the dbScanPassive is performed.

Problems still remain. A few examples are:

Infinite Loop

Infinite processing loops are possible.



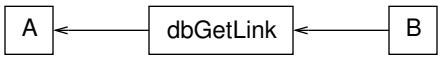
Assume both A and B are asynchronous passive records and a request is made to process A. The following sequence of events occur.

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.

9. 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.

Chapter 4: Database Definition

1. Overview

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

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

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.

2. 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,
   ...
}


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

General Rules

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

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 (Remember that $(...) is used for macro substitution in process variable names). Thus in many cases quotes are not needed.

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.

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 all database files have the extension ".db".

path addpath

The path follows the standard Unix convention, i.e. it is a list of dirtectory names separated by colons.

Format:

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

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.

Utilities which load database files (dbExpandInclude, 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 "/". 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. This only has to be specified if the value desired is ASL0.
- **initial** - Initial Value. This only needs to be specified if the default field value is non zero.

- **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.
- **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 write 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 spscified 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 arrise 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.

- **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 write to this field? The allowed values are:
    - NO (default)
    - YES
- **interest_level -** An interest level for the dbpr and 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.db"
   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: the string $[...] is used for template definitions.
  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. Note that if it exceeds 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

    record.field PP MS

    The field name is optional. PP and MS, which are also optional, can be preceeded by "N" meaning no. Thus (PP,NPP) means (process passive, no process passive) and (MS,NMS) means (maximize severity, no maximize severity). The defaults are NPP and NMS.
  - 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.
  bbadddr - bitbus address
  gpibaddr - gpib address
  parm - optional device dependent character string(31 char max)

- RF_IO

  #Rcryo Mmicro Ddataset Eelement

- VXI_IO

  #Vframe Cslot Ssignal @parm (Dynamic addressing)
  or
  #Vla Signal @parm (Static Addressing)

  frame - VXI frame number
  slot - Slot within VXI frame
  la - Logical Address
  signal - Signal Number
  parm - device specific character string(25 char max)

- DBF_FWDLINK
  This is either not defined or else is a PV_LINK. Process Passive and Maximize are ignored.

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")
}
```

3. 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 liner conversion. The remaining choices are assumed to be the names of breakpoint tables. If a breakpoint table is chosen the record suport modules calls cvtRawToEngBpt or cvtEngToRawBpt. You can look at the ai and ao record support modules for details.

If a user want to add additional breakpoint tables then the following should be done:

- Copy the menuConvert.db file from epics base/rec
- Add definitions for new breakpoint tables to the end
- Make sure modified menuConvert.db is loaded into ioc instead of epics version.

Please note that it is only necessary to load a breakpoint file if a record instance actually choses 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.db. 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 incInit 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 engineeringt 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 1st breakpoint table entry
- **Low Value Raw**: Raw value for 1st 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 file.data
```

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

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

```
BPTS += table.db
```

It is a good idea to name the files the same as the breakpoint table name.

4. Generation of Include Files for Menus and Record Types

Introduction

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

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

- 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 db. The name of the generated include file is the menu name with an extension of h. Thus menuScan is defined in a file menuScan.db and it's 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.db. The name of the generated include file is the same name with an extension of h. Thus aoRecord is defined in a file aoRecord.db and it's generated include file is named aoRecord.h. Since aoRecord has a private menu called aoOIF, the db file and the generated include file have definitions for this menu. Thus for each record type there are two source files (xxxRecord.db 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 (recordtypes & record specific menus)
```

Consult the documentation for the Application Source/Release system you are using for details.

dbToMenuH

This tool is executed as follows:

```
dbToMenuH menuXXX.db
```

It generates a file which has the same name as the input file but with an extension of h.

Example

menuPriority.db, 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 xxxRecord.db
```

It generates a file which has the same name as the input file but with an extension of h.

Example

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

```
menu(aoOIF) {
   choice(aoOIF_Full,"Full")
   choice(aoOIF_Incremental,"Incremental")
}
recordtype(ao) {
   include "dbCommon.db"
   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(dbRecDes *pdbRecDes)
{
    aoRecord *prec = 0;
  pdbRecDes->papFldDes[0]->size=sizeof(prec->name);
  pdbRecDes->papFldDes[0]->offset=
      (short)((char *)&prec->name - (char *)prec);
  ... code to compute size&offset for other fields in dbCommon
  pdbRecDes->papFldDes[42]->size=sizeof(prec->val);
  pdbRecDes->papFldDes[42]->offset=
      (short)((char *)&prec->val - (char *)prec);
  pdbRecDes->papFldDes[43]->size=sizeof(prec->oval);
  pdbRecDes->papFldDes[43]->offset=
      (short)((char *)&prec->oval - (char *)prec);
  ... code to compute size&offset for remaining fields
  pdbRecDes->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

Again remember that only the analog output record support module and it's associated device support should include this record definition. Lets 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 definining the record are used by record suppport 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:
      ...
}GEN_SIZE_OFFSET
```

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 the the routine is defined exactly once.

5. Utility Programs

dbExpand Include

```
dbExpandInclude -I "path" file1 file2 ...
```

NOTE: Unix 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/epics.db file into an ioc if NFS is not used. If dbExpandInclude creates a local epics.db file it takes about 7 seconds to load (25 MHZ 68040 IOC).

dbLoadDatabase

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

NOTE: IOC Only

This command loads a database file containing any of the definitions given in the summary at the beginning of this chapter. It does not accept macro definitions.

dbLoadRecords

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

NOTE: IOC Only
NOTE: This should become part of dbLoadDatabase!!!

db_file must be a file containing only record instances in standard ascii format. Such files should have an extension of ".db".

dbLoadRecords reads dbfile performing the substitutions specified in `substitutions`, which must be a string 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 ioc command:

```
dbLoadRecords("test.db","pre=TEST,STR=\"test\",SCAN=Passive")
```

gives the same results as loading:

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

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 convience 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")
}
```

Chapter 5: IOC Initialization

1. 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:

```
ld  <  targetmv167/iocCore
ld  <  targetmv167/drvSup
ld  <  targetmv167/recSup
ld  <  targetmv167/devSup
#dbPvdTableSize(512)
#ld  <  initHooks.o
# iocLogDisable = 1
# TSconfigure(0)#See below for options

dbLoadDatabase("<file>.db","path")
dbLoadRecords("<.db file>")
   and/or
dbLoadTemplates("<.db file>,"<template_def>")
   . . .
iocInit ("<resource file>")
```

The first four commands load various components of the EPICS software.

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.

`InitHooks.o` is an optional routine that, if supplied, is called after most steps of IOC initialization.

`iocLogDisable` can be set to one to disable logging of error messages generated by the ioc, i.e. error messages will not be send to the system wide error handler.

`TSconfigure` is described in the next section.

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

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

`iocInit` performs the following functions:

```
coreRelease
getResources
iocLogInit
taskwdInit
callbackInit
dbCaLinkInit(1)
initDrvSup
initRecSup
initDevSup
ts_init
initDatabase
dbCaLinkInit(2)
finishDevSup
scanInit()
interruptAccept
initialProcess
rsrv_init
```

2. TSconfigure

`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 is seconds. The default is 10 seconds.
- **clock_rate**: Default is 1000 Hz
- **master_port**: UDP port for master. The default is 18233
- **slave_port**: UDP port for slave.

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.

3. initHooks

This routine, if loaded before `iocInit` is invoked, is called by `iocInit` after each significant initialization step. When called it passes a single argument identifying the step just completed. The argument is defined in `epicsH/initHooks.h`.

The following is the `default initHooks.c` file.

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

```
/* If this function (initHooks) is loaded, iocInit calls it
 * at certain defined points during IOC initialization */

void initHooks  (callNumber)
int   callNumber;
{
   switch (callNumber) {
   case INITHOOKatBeginning :
      break;
   case INITHOOKafterGetResources :
      break;
   case INITHOOKafterLogInit :
      break;
   case INITHOOKafterCallbackInit :
      break;
   case INITHOOKafterCaLinkInit1 :
      break;
   case INITHOOKafterInitDrvSup :
      break;
   case INITHOOKafterInitRecSup :
      break;
   case INITHOOKafterInitDevSup :
      break;
   case INITHOOKafterTS_init :
      break;
   case INITHOOKafterInitDatabase :
      break;
   case INITHOOKafterCaLinkInit2 :
      break;
   case INITHOOKafterFinishDevSup :
      break;
   case INITHOOKafterScanInit :
      break;
   case INITHOOKafterInterruptAccept :
      break;
   case INITHOOKafterInitialProcess :
      break;
   case INITHOOKatEnd :
      break;
   default:
      break;
   }
   return;
}
```

4. dbLoadDatabase, dbLoadRecords, dbLoadTemplate

See the description of Utility Programs in chapter "Database Definition" for details.

5. Get Resource Definitions

`GetResource` gives values to 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.

Another use of `GetResources` is to override the default values for the EPICS related environment variables. These variables are of type DBF_STRING and must begin with the prefix "EPICS_". The variables used in the IOC are:

```
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
```

For example to override thae value for EPICS_TS_MIN_WEST the resource definition file could contain:

```
EPICS_TS_MIN_WEST DBF_STRING 300
```

QUESTION: Why cant we use the envLib facility?

6. 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`.

7. Start Task Watchdog

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.

8. Start Callback Tasks

Start the general purpose callback tasks. Three tasks are started with the only difference being scheduling priority.

9. Initialize Channel Access Links - Pass 1

Calls `dbCaLinkInit` specifying that it is the first call.

10. Initialize Driver Support

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

11. Initialize Record Support

`InitRecSup` locates each record support entry table and calls the init routine.

12. Initialize Device Support

`InitDevSup` locates each device support entry table and calls the init routine with an argument specifying that this is the initial call.

13. Initialize Timing System

`Ts_init` 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.

14. Initialize Database

`InitDatabase` makes four passes over the database performing the following functions:

- Pass 1: Initializes following fields: `rset`, `dset`, `mlis`. Calls record support `init_record` (First pass)
- Pass 2: Attempts to convert `PV_LINKs` to `DB_LINKs`
- Pass 3: Calls record support `init_record` (second pass)
- Pass 4: Determines lock sets

15. Initialize Channel Access Links - Pass 2

Calls `dbCaLinkInit` specifying that it is the second call.

16. Finish Device Support

`InitDevSup` locates each device support entry table and calls the init routine with an argument specifying that this is the finish call.

17. Initialize Database Scanners

The periodic, event, and io event scanners are initialized and started.

18. Accept Interrupts

A global variable "`interruptAccept`" is set `TRUE`. Until this time no request should be made to process records and all interrupts should be ignored.

19. Perform Initial Processing

`dbProcess` is called for all records that have `PINI TRUE`.

20. Start Channel Access Server

The Channel Access server is started.

Chapter 6: Access Security

1. Overview

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

1. User's Guide - This explains what access security is and how to use it.
2. Design Overview - Functional Requirements and Design Overview.
3. Application Programmer's Interface
4. Database Access Security - Access Security features in the EPICS IOC database.
5. Channel Access Security - Access Security features in Channel Access
6. 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.

2. User's Guide

Features

EPICS 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. Note that the security level is defined in the ascii record definition file. Thus the access security level for a field is the same for all instances of records of a given type.
- **When**: Access rules can contain input links and calculations similar to the calculation record, thereby including current values of process variables as part of access privilege.

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>")
   }
   ...
}]
...
```

Rules

- **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 is null or does not have an 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 impels 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. For each channel access client attaching to a record the access privilege is determined as follows:

1. The ASG associated with the record is searched.

2. Each RULE is checked for the following:
   a. Check that this field's level is <= the level for this RULE.
   b. If UAG is defined, is this user a member of one of the specified UAGs. If UAG is not defined all users are accepted.
   c. If HAG is defined, is the user located at one of the HAGs. If HAG is not defined all hosts are accepted.
   d. If CALC is specified, does the calculation 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.

Note that these rules allow multiple RULEs to 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 < "filename"
```

This is a unix command. It displays errors on stdout. If no errors are detected it prints nothing. Please note that this only detects syntax errors not logic errors. Thus it is still possible to get your self in trouble.

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.

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 before asInit. It is **NOT** permissable to call asInit command before iocInit is called.

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 any 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:lev1permit` has ASG="permit". In order for the opSup, linacSup, or an appDev to have write privilege to everything this record must be set to the value 1.

The following access configuration satisfies the above rules.

```
UAG(op) {op1,op2,superguy}
UAG(opSup) {superguy}
UAG(linac) {waw,nassiri,grelick,berg,fuja,gsm}
UAG(linacSup) {gsm}
UAG(appDev) {nda,kko}
HAG(icr) {silver,phebos,gaea}
HAG(cr) {mars,hera,gold}
HAG(ioc)
{ioclic1,ioclic2,ioclid1,ioclid2,ioclid3,ioclid4,ioclid5}
```

```
ASG(DEFAULT) {
   INPA(LI:OPSTATE)
   INPB(LI:lev1permit)
   RULE(0,WRITE) {
      UAG(op)
      HAG(icr,cr)
      CALC(”A=1”)
   }
   RULE(0,WRITE) {
      UAG(op,linac,appdev)
      HAG(icr,cr)
      CALC(”A=0”)
   }
   RULE(1,WRITE) {
      UAG(opSup,linacSup,appdev)
      CALC("B=1")
   }
   RULE(1,READ)
   RULE(1,WRITE) {
      HAG(ioc)
   }
}
ASG(permit) {
   RULE(0,WRITE) {
       UAG(opSup,linacSup,appDev)
   }
   RULE(1,READ)
   RULE(1,WRITE) {
      HAG(ioc)
   }
}
ASG(critical) {
   INPB(LI:lev1permit)
   RULE(1,WRITE) {
      UAG(opSup,linacSup,appdev)
      CALC("B=1")
   }
   RULE(1,READ)
   RULE(1,WRITE) {
      HAG(ioc)
   }
}
```

Performance and Memory Requirements

Performance has not yet been measured but during the tests to measure memory usage no noticeable change in performance during ioc initialization or during Channel Access clients connection was noticed. Unless access privilege is violated the overhead during channel access gets and puts is only an extra comparison.

In order to measure memory usage, the following test was performed:

1. A database consisting of 5000 soft analog records was created.

2. A channel access client (caput) was created that performs ca_puts on each of the 5000 channels. Each time it begins a new set of puts the value increments by 1.
3. A channel access client (caget) was created that has monitors on each of the 5000 channels.

The memory consumption was measured before iocInit, after iocInit, after caput connected to all channels, and after caget connected to all 5000 channels. This was done for APS release 3.11.5 (before access security) and the first version which included access security. The results were:

	R3.11.5	After
Before iocInit	4,244,520	4,860,840
After iocInit	4,995,416	5,964,904
After caput	5,449,780	6,658,868
After caget	8,372,444	9,751,796

Before the database was loaded the memory used was 1,249,692 bytes. Thus most of the memory usage before iocInit resulted from storage for records. The increase since R3.11.5 results from added fields to dbCommon. Fields were added for access security, synchronous time support and for the new caching put support. The other increases in memory usage result from the control blocks needed to support access control. The entire design was based on maximum performance. This resulted in increased memory usage.

3. Design Overview

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.

4. 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(ASINPUTFUNPTR inputFunction)
```

This routine reads an access definition file and performs all initialization necessary.

asInitialize 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,

5. 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
- ASL1Assigned 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 dbAsLib.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. If asSetFilename is never called or is passed a NULL filename then the next call to asInit results in access security being disabled. This routine must be called before iocInit if you want access security to be turned on during ioc initialization.

```
int asInit(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.

An important part of initialization is starting a channel access client which attaches monitors to all INPx fields. This task takes care of recomputing access permissions whenever a monitor is triggered.

asInit actually spawns a task asInitTask to perform the initialization. This allows asInit 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 users callback routine is called via one of the standard callback tasks.

asInit 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 also lock before calling the associated routine and unlock afterwards. Thus they will work correctly even if the access configuration is being changed.

```
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.

6. 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 get request and for write access when processing put requests. 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 its call back 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 get request and for write access when processing put requests. 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.

7. 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. The following is a picture of how the control blocks are related.

Basic Structures

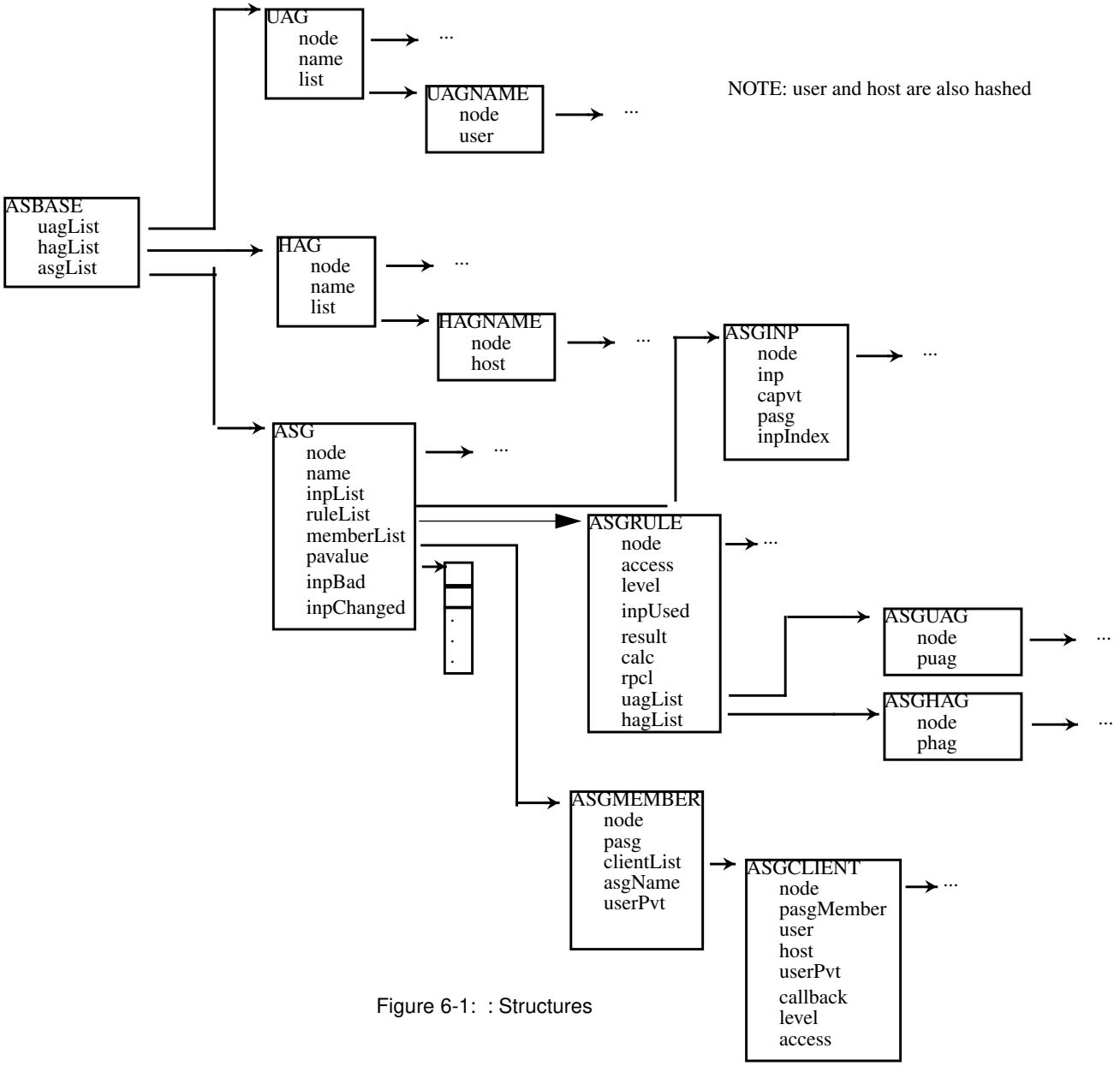


Figure 6-1: : Structures

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.
- a**sLib_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 perform 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.

The access database is locked for the following operations:

1. While initializing. This is done by asInitTask.
2. asCaTask locks while calling asComputeAllAsgs
3. asCa callback routines lock except while initializing.
4. asSetFilename
5. asChangeGroup
   asAddClient
   asChangeClient
   asRemoveClient
6. asdbdump
   aspuag
   asphag
   asprules
   aspmem

Note that the asDump* routines do not lock. This is done for diagnostic purposes. If for some reason an access database is permanently locked, it is still possible to invoke these diagnostic routines.

Chapter 7: IOC Test Facilities

1. 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.

2. Database List, Get, Put

dbl

Database List:

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

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.

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.

3. Breakpoints

A breakpoint facility that allows the user to step through database processing on a per lockset basis. This facility has been constructed in such a way that the execution of all locksets other than ones with breakpoints will not be interrupted. This was done by executing the records in the context of a separate task.

The breakpoint facility records all attempts to process records in a lockset containing breakpoints. A record that is processed through external means, e.g.: a scan task, is called an entrypoint into that lockset. The `dbstat` command described below will list all detected entrypoints to a lockset, and at what rate they have been detected.

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.

dbs

Single Step:

```
dbs ("<record_name>")
```

Steps through execution of records within a lockset. If this command is called without an argument, it will automatically step starting with the last detected breakpoint.

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`."

4. 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
```

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

```
dbhcr > report
```

The report will probably not be in the sort order desired. The Unix command:

```
sort  report > report.sort
```

should produce the sort order you desire.

5. Scan Reports

scanppl

Print Periodic Lists:

```
scanppl
```

This routine prints a list of all records in the periodic scan lists.

scanpel

Print Event Lists:

```
scanpel
```

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

scanpiol

Print I/O Event Lists:

```
scanpiol
```

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

6. 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.

7. 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.

8. Channel Access Reports

ca_channel_status

Format:

```
ca_channel_status (taskid)
```

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

client_stat

Format:

```
client_stat
```

Channel Access client status

dbel

Format:

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

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

9. Interrupt Vector

veclist

Format:

```
veclist
```

Print Interrupt Vector List

10. Environment Variables

epicsPrtEnvParams

Format:

```
epicsPrtEnvParams
```

Print Environment Variables

11. 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.

12. Lock Set Routines

dblls

List Lock Sets:

```
dblls(lock_set)
```

This command generates a report showing the lock set to which each record belongs. If `lock_set` is 0 all records are shown, otherwise only records in the specified lock set are shown.

dbllsdblinks

List each record in the lock set as well as all database links in each record in the lock set.

```
dbllsdblinks(lock_set)
```

13. 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.

14. Routines to dump database information

dbDumpPath

Dump Path:

```
dbDumpPath(pdbbase)
```

```
dbDumpPath(pdbbase)
```

The current path for database includes is displayed.

dbDumpMenu

Dump Menu:

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

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

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

dbDumpRecDes

Dump Record Description:

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

```
dbDumpRecDes(pdbbase,"ai")
```

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

dbDumpFldDes

Dump Field Description:

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

```
dbDumpFldDes(pdbbase,"ai","VAL")
```

If the second argument is 0 then the field descriptions of all records are displayed. If the third argument is 0 then the description of all fields are diaplayed.

dbDumpDevice

Dump Device Support:

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

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

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

dbDumpDriver

Dump Driver Support:

```
dbDumpDriver(pdbbase)
```

```
dbDumpDriver(pdbbase)
```

dbDumpRecords

Dump Record Instances:

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

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

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

dbPvdDump

Dump the Process variable Directory:

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

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

This command show 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 8: Record Support

1. Overview

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

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

A record support module consists of a standard set of routines 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.

2. Overview of Record Processing

The most important record support routine is `process`. This routine determines what record processing means. Before the record specific "`process`" routine is called, the following has already been done:

- Decision to process a record.
- Check that record is not active, i.e. `pact` must be FALSE.
- Check that the record is not disabled.

The `process` routine, together with its associated device support, is responsible for the following tasks:

- Set record active while it is being processed
- Perform I/O (with aid of device support)
- Check for record specific alarm conditions
- Raise database monitors
- Request processing of forward links

A complication of record processing is that some devices are intrinsically asynchronous. It is **NEVER** permissible to wait for a slow device to complete. Asynchronous records perform the following steps:

1. Initiate the I/O operation and set `pact TRUE`
2. Determine a method for again calling process when the operation completes
3. Return immediately without completing record processing
4. When process is called after the I/O operation complete record processing
5. Set `pact FALSE` and return

The examples given below show how this can be done.

3. Record Support and Device Support Entry Tables

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

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

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

```
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`.

4. Example Record Support Module

This section contains the skeleton of a record support package. The record type is `xxx` and the record has the following fields in addition to the `dbCommon` fields: `VAL`, `PREC`, `EGU`, `HOPR`, `LOPR`, `HIHI`, `LOLO`, `HIGH`, `LOW`, `HHSV`, `LLSV`, `HSV`, `LSV`, `HYST`, `ADEL`, `MDEL`, `LALM`, `ALST`, `MLST`. These fields will have the same meaning as they have for the `ai` record. Consult the Record Reference manual for a description.

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 = (xxxdset *)(pxxx->dset)) == NULL) {
      recGblRecordError(S_dev_noDSET,pxxx,"xxx: init_record");
      return(S_dev_noDSET);
   }
   /* must have read_xxx function defined */
   if( (pdset->number < 5) || (pdset->read_xxx == NULL) ) {
      recGblRecordError(S_dev_missingSup,pxxx,
         "xxx: init_record");
      return(S_dev_missingSup);
   }
   if( pdset->init_record ) {
      if((status=(*pdset->init_record)(pxxx))) return(status);
   }
   return(0);
}
```

This routine, which is called by iocInit twice for each record of type xxx, checks to see if it has a proper set of device support routines and, if present, calls the init_record entry of the DSET.

During the first call to init_record (pass=0) only initializations relating to this record can be performed. During the second call (pass=1) initializations that may refer to other records can be performed. Note also that during the second pass, other records may refer to fields within this record. A good example of where these rules are important is a waveform record. The VAL field of a waveform record actually refers to an array. The waveform record support module must allocate storage for the array. If another record has a database link referring to the waveform VAL field then the storage must be allocated before the link is resolved. This is accomplished by having the waveform record support allocate the array during the first pass (pass=0) and having the link reference resolved during the second pass (pass=1).

process

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

   if( (pdset==NULL) || (pdset->read_xxx==NULL) ) {
```

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

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

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

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

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

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

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

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

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.

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.

5. Record Support Routines

This section describes the routines defined in the RSET. Any routine that does not apply to a specific record type must be declared `NULL`.

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 by added to special.h, instead use one of the existing types. SPC_MOD can always be used. 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 then VAL.

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.

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.

Get Units

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

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

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.

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.

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.

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.

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.

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.

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.

6. Global Record Support Routines

A number of global record support routines are available. These routines are intended for use by the record specific processing routines but can be called by any routine that wishes to use their services.

The name of each of these routines begins with "recGbl".

Alarm Status and Severity

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

```
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

```
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

```
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

```
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.

Get Input Link

```
recGblGetLinkValue(
   struct link*plink,
   void   *precord,
   short  dbrType,
   void   *pdest,
   long   *poptions,
   long   *pnRequest);
```

This routine gets a value from an input link. If the link is a constant this call amounts to a `NOP`.

Put Output Link

```
recGblPutLinkValue(
   struct link  *plink,
   void   *precord,
   short  dbrType,
   void   *pdest,
   long   *pnRequest);
```

This routine writes a value to an output link. If the link is a constant this call amounts to a `NOP`.

Initialize Fast Input Link

```
recGblInitFastInLink(
   struct link  *plink,
   void   *precord,
   short   dbrType,
   char   *ca_string);
```

Initialize a fast input link. This routine should be used if scalar data with options is desired. If the link is not a channel access link or a database link this amounts to a `NOP`. `ca_string` is the uppercase name of the field that Channel Access is to get a value from.

Initialize Constant Link

```
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.

Initialize Fast Output Link

```
recGblInitFastOutLink(
   struct link  *plink,
   void  *precord,
   short  dbrType,
   char  *ca_string);
```

Initialize a fast output link. This routine should be used if scalar data is to be written via the link. If the link is not a channel access link or a database link this amounts to a `NOP`. "`ca_string`" is the uppercase name of the field that Channel Access will take its value from.

Get Fast Input Link

```
recGblGetFastLink(
   struct link  *plink,
   void  *precord,
   void  *pdest);
```

Gets a value from a fast input link. This routine can only be used if the link was initialized via `recGblInitFastInLink`. If the link is a constant link this amounts to a `NOP`.

Put Fast Output Link

```
recGblPutFastLink(
   struct link  *plink,
   void  *precord,
   void   *psource);
```

Puts a value to a fast output link. This routine can only be used if the link was initialized via `recGblInitFastOutLink`. If the link is a constant link this amounts to a `NOP`.

Chapter 9: Device Support

1. Overview

In addition to a record support module, each record type can have an arbitrary number of device support modules. The purpose of device support is to hide hardware specific details from record processing routines. Thus support can be developed for a new device without changing the record support routines.

A device support routine has knowledge of the record definition. It also knows how to talk to the hardware directly or how to call a device driver which interfaces to the hardware. Thus device support routines are the interface between hardware specific fields in a database record and device drivers or the hardware itself.

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 it's associated device support via the dset field.

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

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

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

2. Example Synchronous Device Support Module

```
/* Create the dset for devAiSoft */
long init_record();
long read_ai();
struct {
   long   number;
   DEVSUPFUN   report;
   DEVSUPFUN   init;
   DEVSUPFUN   init_record;
   DEVSUPFUN   get_ioint_info;
   DEVSUPFUN   read_ai;
   DEVSUPFUN   special_linconv;
}devAiSoft={
   6,
   NULL,
   NULL,
   init_record,
   NULL,
   read_ai,
   NULL};

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) :
         status = dbCaAddInLink(&(pai->inp),
            (void *)pai,"VAL");
         if(status) return(status);
         break;
      case (DB_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;
      long  options=0;
      long  nRequest=1;

      status=recGblGetLinkValue(&(pai->inp.value.db_link),
         (void *)pai,DBR_DOUBLE,&(pai->val),
         &options,&nRequest);
      }
      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.

3. Example Asynchronous Device Support Module

This example shows how to write an asynchronous device support routine. It does the following sequence of operations:

1. When first called `pact` is `FALSE`. It arranges for a callback (`myCallback`) routine to be called after a number of seconds specified by the `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*/
struct callback {
   CALLBACK   callback;
   sruct dbCommon   *precord;
   WDOG_ID   wd_id;
};

static void myCallback(struct callback *pcallback)
{
    dbCommon *precord = pcallback->precord;
    struct rset*prset = (struct rset *)precord->rset;

   dbScanLock(precord);
   *(prset->process)(precord);
   dbScanUnlock(precord);
}


static long init_record(void *precord)
{
    aiRecord   *pai = (aiRecord *)precord;
    struct callback*pcallback;

    /* ai.inp must be a CONSTANT*/
   switch (pai->inp.type) {
   case (CONSTANT) :
      pcallback = (struct callback *)
```

```
            (calloc(1,sizeof(struct callback)));
         pai->dpvt = (void *)pcallback;
         callbackSetCallback(myCallback,
            &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);
   }
```

4. Device Support Routines

This section describes the routines defined in the DSET. Any routine that does not apply to a specific record type must be declared `NULL`.

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

1. Overview

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

Since EPICS now has both device and driver support the question arises: When do I need driver support and when dont 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 modulerize 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 dont 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”.

2. Device Drivers

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

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

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

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

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 resposiible for producing a report describing the hardware it found at int 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 uses this method. It should **NOT** be used for any new support for the following reasons:

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

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

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

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

Chapter 11: Static Database Access

1. Overview

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

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

Before accessing database records, the files describing menus, record types, and devices must be read via `dbReadDatabase` or `dbReadDatabaseFP`. These routines, which is 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 `dbWriteRecords`) 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.

2. Definitions

DBBASE

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

```
DBBASE  *pdbbase;
```

DBENTRY

A typical declaration for a database entry structure is:

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

Most static access to a database is via a DBENTRY structure. As many DBENTRYs as desired can be allocated.

The user should NEVER access the fields of DBENTRY directly. They are meant to be used by the static database access library.

Most static access routines accept an argument which contains the address of a DBENTRY. Each routine uses this structure to locate the information it needs and gives values to as many fields in this structure as possible. All other fields are set to NULL.

Database Configuration 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.

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.

Each link is one of the following types:

- **DCT_LINK_CONSTANT**: Constant value.
- **DCT_LINK_FORM**: Constant with associated DCT_MENUFORM field.
- **DCT_LINK_PV**: A process variable link.
- **DCT_LINK_DEVICE**: A device link with associated DCT_MENUFORM field..

Database configuration tools can change any link between being a constant and a process variable link. Routines are provided to accomplish these tasks. A device link can be given values via the form routines described below.

The routines dbGetString, dbPutString, and dbVerify can be used for link fields but should not be used to prompt the user for PV or DEVICE links. They are meant to be used for constant links or to save and restore versions of the database.

3. Examples

dbExpand Include

This example is the dbExpandInclude utility, It reads a set of database definitiion files and writes all definitions to stdout. All include statements appearing in the input files are expanded.

```
/* dbExpandInclude.c */
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <epicsPrint.h>
#include <errMdef.h>
#include <dbStaticLib.h>
#include <dbStaticPvt.h>
#include <dbBase.h>
#include <gpHash.h>

DBBASE *pdbbase = NULL;
```

```
int main(int argc,char **argv)
{
   long     status;
   int      i;
   int      arg,strip;
   char     *path=0;

   /*Look for path, i.e. -I path or -Ipath*/
   for(arg=1; arg<argc; arg++) {
      if(strncmp(argv[arg],"-I",2)!=0) continue;
      if(strlen(argv[arg])==2) {
         path = argv[arg+1];
         strip = 2;
      } else {
         path = argv[arg] + 2;
         strip = 1;
      }
      argc -= strip;
      for(i=arg; i<argc; i++) argv[i] = argv[i + strip];
      break;
   }
   if(argc<2) {
      printf("usage: dbExpandInclude -Ipath file1.db file2.db"\
       "...\n");
      exit(0);
    }
   for(i=1; i<argc; i++) {
      status = dbReadDatabase(&pdbbase,argv[i],path);
      if(!status) continue;
      fprintf(stderr,"For input file %s",argv[i]);
      errMessage(status,"from dbReadDatabase");
   }
   dbWriteMenuFP(pdbbase,stdout,0);
   dbWriteRecDesFP(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. If this is the first time you are reading this manual just quickly look at the example and then come back to it after reading the rest of the chapter. The example shows how to locate each record and display each field.

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

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

4. 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 pdbbase is null. Thus the user only has to supply code like the following:

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

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 *pdbbase);
```

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

5. DBENTRY Routines

Alloc/Free DBENTRY

```
DBENTRY *dbAllocEntry(DBBASE *pdbbase);
void dbFreeEntry(DBENTRY *pdbentry);
DBENTRY *dbCopyEntry(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 caller calls dbAllocEntry to create a new DBENTRY and calls dbFreeEntry when done.

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.

dbInitEntry dbFinishEntry

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

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

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

6. Read and Write Database

Read Database File

```
long dbReadDatabase(DBBASE **ppdbbase,const char *filename,
      char *path);
long dbReadDatabaseFP(DBBASE **ppdbbase,FILE *fp,
      char *path);
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

onl;y 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 File

```
long dbWriteRecord(DBBASE *pdbbase,char * file,
   char *precdesname,int level);
long dbWriteRecordFP(DBBASE *pdbbase,FILE *fp,
   char *precdesname,int level);
void dbWriteMenu(DBBASE *pdbbase,char *filename,
   char *menuName);
void dbWriteMenuFP(DBBASE *pdbbase,FILE *fp,char *menuName);
void dbWriteRecDes(DBBASE *pdbbase,char *filename,
   char *recdesName);
void dbWriteRecDesFP(DBBASE *pdbbase,FILE *fp,
   char *recdesName);
void dbWriteDevice(DBBASE *pdbbase,char *filename);
void dbWriteDeviceFP(DBBASE *pdbbase,FILE *fp)
void dbWriteDriver(DBBASE *pdbbase,char *filename);
void dbWriteDriverFP(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.

`dbWriteRecord` writes record instances. If `precdesname` 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.

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

`dbWriteRecDes` writes the description of the specified record type or, if `recdesName` is NULL, the descriptions of all menus.

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

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

7. Manipulating Record Descriptions

Get Number of Record Types

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

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

Locate Record Descriptions

```
long dbFindRecdes(DBENTRY *pdbentry,char *recdesname);
long dbFirstRecdes(DBENTRY *pdbentry);
long dbNextRecdes(DBENTRY *pdbentry);
```

dbFindRecDes locates the description of a particular record type. dbFirstRecDes locates the description for the first, in alphabetical order, record type. Given that DBENTRY points to a particular record type, dbNextRecdes 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 = dbFirstRecdes(pdbentry);
while(!status) {
   /*Do something*/
status = dbNextRecdes(pdbentry)
}
```

Get Record Type Name

```
char *dbGetRecdesName(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 dbFindRecdes, dbFirstRecdes, or dbNextRecdes. It returns NULL if DBENTRY does not point to a record description.

8. Manipulating Record Instances

With the exception of dbFindRecord, each of the routines described in this section require that DBENTRY references a valid record description, i.e. that dbFindRecdes, dbFirstRecdes, or dbNextRecdes has been called and returned success.

Create/Delete Record

```
long dbCreateRecord(DBENTRY *pdbentry,char *precordName);
long dbDeleteRecord(DBENTRY *pdbentry);
```

dbCreateRecord, which assumes that DBENTRY references a valid record type, creates a new record instances and initializes it as specified by the record description. If it returns sucess then DBENTRY references the record just created.

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);
int  dbFoundField(DBENTRY *pdbentry);
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.

Rename Record

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

This routine renames the record instance currently referenced by DBENTRY.

9. Manipulating Field Descriptions

The routines described in this section all assume that DBENTRY references a record instance, i.e. that `dbFindRecord`, `dbFirstRecord`, or `dbNextRecord` has returned success.

```
long dbFindField(DBENTRY *pdbentry,char *pfieldName);
long dbFirstFielddes(DBENTRY *pdbentry,int dctonly);
long dbNextFielddes(DBENTRY *pdbentry,int dctonly);
int  dbGetFieldType(DBENTRY *pdbentry);
int  dbGetNFields(DBENTRY *pdbentry,int dctonly);
char *dbGetFieldName(DBENTRY *pdbentry);
char *dbGetPrompt(DBENTRY *pdbentry);
int   dbGetPromptGroup(DBENTRY *pdbentry);
```

These routines manipulate the field descriptions for a particular record type. Note that if a record instance has previously been located they also update the location of the field itself.

10. Manipulating 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 they are intended to be used only for creating and restoring versions of a database.

11. Find Breakpoint Table

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

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

12. Manipulating Menu Fields

MENU and MENUFORM Fields

```
dbMenu *dbFindMenu(DBBASE *pdbbase,char *name);
char **dbGetChoices(DBENTRY *pdbentry);
int  dbGetMenuIndex(DBENTRY *pdbentry);
long dbPutMenuIndex(DBENTRY *pdbentry,int index);
int  dbGetNMenuChoices(DBENTRY *pdbentry);
```

dbFindMenu is most useful for runtime use but is a static database access routine.

The other routines should only be used for DCT_MENU and DCT_MENUFORM fields. Thus they should only be called if dbFindField, dbFirstFieldDes, or dbNextFieldDes has returned success and the field type is DCT_MENU or DCT_MENUFORM.

MENUFORM Fields

These routines are used with a DCT_MENUFORM field (a DTYP field) but actually manipulate an associated field DCT_INLINK or DCT_OUTLINK field.

```
int dbAllocForm(DBENTRY *pdbentry)
long dbFreeForm(DBENTRY *pdbentry)
char **dbGetFormPrompt(DBENTRY *pdbentry)
char **dbGetFormValue(DBENTRY *pdbentry)
long dbPutForm(DBENTRY *pdbentry, char **value)
char **dbVerifyForm(DBENTRY *pdbentry,char **value)
```

dbAllocForm allocates storage needed to manipulate forms. The return value is the number of elements in the form. 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. 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. 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]);
   scanf("%s",value[i]);
}
if(dbPutForm(pdbentry,value)) {
   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]);
   }
}
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.

13. Manipulating Link Fields

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_FORM, DCT_LINK_PV, or DCT_LINK_DEVICE.

Constant and Process Variable Links

```
long dbCvtLinkToConstant(DBENTRY *pdbentry);
long dbCvtLinkToPvlink(DBENTRY *pdbentry);
long dbPutPvlink(DBENTRY *pdbentry,int pp,int ms,
   char *pvname);
long dbGetPvlink(DBENTRY *pdbentry,int *pp,int *ms,
   char *pvname);
```

These routines should be used for modifying DCT_LINK_CONSTANT or DCT_LINK_PV links. They should not be used for DCT_LINK_FORM or DCT_LINK_DEVICE links, which should be processed via the associated DCT_MENUFORM field described above.

14. Dump Routines

```
void dbDumpPath(DBBASE *pdbbase)
void dbDumpRecord(DBBASE *pdbbase,char *precdesname,
   int level);
void dbDumpMenu(DBBASE *pdbbase,char *menuName);
void dbDumpRecDes(DBBASE *pdbbase,char *recdesName);
void dbDumpFldDes(DBBASE *pdbbase,char *recdesName,
   char *fname);
void dbDumpDevice(DBBASE *pdbbase,char *recdesName);
void dbDumpDriver(DBBASE *pdbbase);
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 spefiying 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.

Chapter 12: Runtime Database Access

1. Overview

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

This chapter is divided into the following sections:

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

2. Database Include Files

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

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

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 applys 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.

3. Runtime Database Access Overview

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

Perhaps the easiest way to describe the database access layer is to list and briefly describe the set of routines that constitute database access. This provides a good look at the facilities provided by the database. The structure of the IOC database is not described at this point. The structure is explained in the last chapter.

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:

- **dbScanLockInit**: Initialize scan locking
- **dbScanLock**: Lock database
- **dbScanUnlock**: Unlock database

- **dbCommonInit** Initialize database common

- **dbNameToAddr**: Locate a database variable.

- **dbGetField**: Get values associated with a database variable.
- **dbGetLink**: Get value of field referenced by database link
- **dbFastLinkGet**: Fast get value of field referenced by database link
- **dbGet**: Routine called by `dbGetLink` and `dbGetField` and `dbCa` functions

- **dbPutField**: Change the value of a database variable.
- **dbPutLink**: Change value referenced by database link
- **dbFastLinkPut**: Fast change value referenced by database link
- **dbPut**: Routine called by `dbPutxxx` and by the `dbCa` 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.

- **dbScanPassive**: Process record if it is passive.
- **dbScanLink**: Process record referenced by link if it is passive.
- **dbProcess**: Process Record

- **dbCaAddInLink**: Initialize a channel access database input link
- **dbCaAddOutLink**: Initialize a channel access database output link
- **dbCaGetLink**: Get the current value for a channel access database input link
- **dbCaPutLink**: Put a value to a channel access database output link
- **dbCaLinkInit**: Initialize a Channel Access 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 acknowledgement.

`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.

4. Database Access Routines

Database Locking

Database locking is described in another chapter. For now lets just state that a database record must be locked before it is modified and unlocked afterwards.

The routines provided are:

```
void dbScanLockInit(/* called only by iocInit*/
   int  nset);  /* number of lock sets*/

void dbScanLock(struct dbCommon  *precord);

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

dbCommonInit

This routine is called by iocInit to initialize fields in database common. It is only included for completeness.

dbNameToAddr

Locate a process variable, format:

```
long dbNameToAddr(
```

```
    char  *pname, /*ptr to process variable name */
    struct dbAddr  *paddr);
```

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

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

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

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

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

```
long dbGetLink(
   struct db_link  *pdbLink,/*addr of database link*/
   struct dbCommon  *pdest,/*addr of destination record*/
   short  dbrType,/* DBR_xxx*/
   void  *pbuffer,/*addr of returned data*/
   long  *options,/*addr of options*/
   long  *nRequest);/*addr of number of elements desired*/
```

This routine is called by database access itself and by record support and/or device support routines in order to get values from other database records via input links. It calls dbGet to obtain data and also implements the process passive and maximize severity link options.

dbFastLinkGet

```
long dbFastLinkGet(
   struct   link  *plink,
   struct   dbCommon  *precord,
   void     *pdest);
```

This routine gets a value from an input link to pdest. Do not call this routine unless you have a properly initialized Channel Access or database link. This routine is not intended to be called directly by record support, use recGblGetFastLink() instead.

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 then 1 and record support routine get_array_info exists, then it is called. It returns two values: the current number of valid field elements and an offset. The number of valid elements may not match dbAddr.no_elements, which is really the maximum number of elements allowed. The offset is for use by records which implement circular buffers.

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

Change the value referenced by a database link, format:

```
long dbPutLink(
   structdb_link  *pdbLink,/*addr of database link*/
   structdbCommon  *psource,/*addr of source record*/
   short dbrType, /* DBR_xxx*/
   void  *pbuffer,/*addr of data to write*/
   long  nRequest);/*number of elements to write*/
```

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. 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.

dbFastLinkPut

Fast putLink, forward:

```
long dbFastLinkPut(
   structlink  *plink,
   structdbCommon  *precord,
   void  *psource);
```

This routine puts the value from `psource` to an output link. Do not call this routine unless you have a properly initialized Channel Access or database link. This routine is not intended to be called directly by record support, use `recGblPutFastLink`() instead.

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 (normall via a switch statement).

Process Routines

dbScanPassive
dbScanLink

Process record if it is passive, format:

```
long dbScanPassive(
   struct dbCommon  *pfrom,
   struct dbCommon  *pto);   /* addr of record*/
long dbScanLink(
   struct dbCommon *pfrom,
   struct dbCommon *pto);
```

These routines 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`.

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.

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.

The routines provided are:

```
void dbCaLinkInit(int pass);
long dbCaAddInLink(
   struct link  *plink,
   void  *precord,
   char  *pfieldName);

long dbCaAddOutLink(
   struct link  *plink,
   void  *precord,
   char  *pfieldName);

long dbCaGetLink(
   struct link  *plink);

long dbCaPutLink(
   struct link  *plink);
```

For a description of these routines see:

Links in a Distributed database: Theory and Implementation,
Nicholas T. Karonis and Martin R. Kraimer, December 1991

5. Old Database Access Interface

Channel Access has not yet been modified to support the database access routines described above. The database access interface was changed because as more database field types and request options were defined the previous database access interface become harder and harder to modify. In order to make the transition to the new database access without obsoleting all software that used Channel Access an interface module was written. Thus module translates old database calls to new. Several of the Channel Access arguments directly map to database access arguments. Thus existing Channel Access clients use the old database access interface.

Since this manual concentrates on IOC software, this is not the place to describe the old database interface. Other documents describe it. The header file `db_access.h` also provides descriptive information.

Chapter 13: Device Support Library

1. 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.

2. 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(
    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);
```

This routine releases addresses previously registered by a call to devRegisterAddress.

3. 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);
```

4. 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<=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

1. Overview

This chapter describes three sets of EPICS supplied general purpose tasks: 1) Callback, 2) Task Watchdog, and 2) Error message handling.

Often when writing code for an IOC there is no obvious task under which to execute. A good example is completion code for an asynchronous device support module. EPICS supplies the callback tasks for such code.

If an IOC tasks "crashes" there is normally no one monitoring the vxWorks shell to detect the problem. EPICS provides a task watchdog task which periodically checks the state of other tasks. If it finds that a monitored task has terminated or suspended it issues an error message and can also call other routines which can take additional actions. For example a subroutine record can arrange to be put into alarm if a monitored task crashes.

Since IOCs normally run autonomously, i.e. no one is monitoring the vxWorks shell, IOC code that issues printf calls generates errors messages that are never seen. In addition the vxWorks implementation of fprintf requires much more stack space then printf calls. Another problem with vxWorks is the logMsg facility. logMsg generates messages at higher priority then all other tasks except the shell. EPICS solves all of these problems via an error message handling facility. Code can call any of the routines errMessage, errPrintf, or 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.

2. 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 = NULL;
   struct rset *prset = (struct rset *)pRec->rset;

   callbackGetUser(pRec, pCallback);
   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],
      &pmStruct->endid[0]);
}
example(char *pbegid, char*pendid)
{
   strcpy(&myStruct.begid[0],pbegid);
   strcpy(&myStruct.endid[0],pendid);
   callbackSetCallback(myCallback,&myStruct.callback);
   callbackSetPriority(priorityLow,&myStruct.callback);
   callbackSetUser(&myStruct,&myStruct.callback);
   callbackRequest(&myStruct.callback);
}
```

The example can be tested by issuing the following command to the vxWorks shell:

```
example("begin","end")
```

This simple example shows how to use the callback tasks with your own structure that contains the `CALLBACK` structure at an arbitrary location.

3. Task Watchdog

EPICS provides an IOC task that is a watchdog for other tasks. Any task can make a request to be watched. The task watchdog runs periodically and checks each task in its task list. If any task is suspended, an error message is issued and, optionally, a callback task is invoked. The task watchdog provides the following features:

1. Include module:

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

2. Insert request:

   ```
   taskwdInsert (int tid, 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`.

4. Error Handling

Overview

The EPICS error handling facilities provide the following features:

- Three routines epicsPrintf, errMessage, and errPrintf provide access to the EPICS error handling system. NOTE: Other routines such as recGblRecordError call these routines.
- The error handling system provides a separate low priority task that formats and processes the error message. The caller has to wait until the message is processed but higher priority tasks are not delayed.
- The error message can be sent to a system wide error handling system, which currently is just a system wide log file.
- errMessage and errPrintf provide the ability to provide a set of EPICS defined status values that 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.

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.

Return Status Values

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 */}
```

Interface to System Wide Error Handling System

Either `errMessage`, `errPrintf, or epicsPrintf` can be used as an interface to the system wide error handling system. At the present time, they end up calling `logMsg`. Facilities have been added to EPICS to trap `logMsg` calls and direct them to a system wide log file. In the future, a more generalized system wide error handling system, which allows an error handling program to receive error messages from all or selected IOCs, can be provided.

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

errMessage

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.

errPrintf

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.

epicsPrintf

Routine `epicsPrintf` has the following format:

```
int epicsPrintf(const char  *fmtstring, ...);
int epicsVprintf (const char *pFormat, va_list pvar);
```

These routines are like the printf and vprintf provided by the standard C library. Consult any book that describes the standard C library such as "The C Programming Language ANSI C Edition" by Kernighan and Ritchie.

Chapter 15: Database Scanning

1. Overview

Database scanning is the mechanism for deciding when to process a record. Four 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.

2. Scan Related Database Fields

The following fields are normally defined via DCT. It should be noted, however, that it is quite permissible to change any of the scan related fields of a record dynamically. For example, a display manager screen could tie a menu control to the SCAN field of a record and allow the operator to dynamically change the scan mechanism.

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 Intr: 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 I/O event scan facility uses this field.

3. 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 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);
}
```

4. Implementation Overview

The code for the entire scanning system resides in `dbScan.c`, i.e. periodic, event, and I/O event. This section gives an overview of how the code in `dbScan.c` is organized. The listing of `dbScan.c` must be studied for a complete understanding of how the scanning system works.

Definitions And Routines Common To All Scan Types

Everything is built around two basic structures:

```
struct scan_list {
   FAST_LOCK  lock;
   ELLLIST  list;
```

```
    short   modified;
    long   ticks;   /*used only for periodic scan sets*/
};

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

Later we will see how `scan_lists` are determined. For now just realize that `scan_list.list` is the head of a list of records that belong to the same scan set (for example, all records that are periodically scanned at a 1 second rate are in the same scan set). The node field in `scan_element` contain the list links. The normal vxWorks `lstLib` routines are used to access the list. Each record that appears in some scan list has an associated `scan_element`. The `SPVT` field which appears in `dbCommon` holds the address of the associated `scan_element`.

The `lock`, `modified`, and `pscan_list` fields allow `scan_elements`, i.e. records, to be dynamically removed and added to scan lists. If `scanList`, the routine which actually processes a scan list, is studied it can be seen that these fields allow the list to be scanned very efficiently if no modifications are made to the list while it is being scanned. This is, of course, the normal case.

The `dbScan.c` module contains several private routines. The following access a single scan set:

- **printList**: Prints the names of all records in a scan set.
- **scanList**: This routine is the heart of the scanning system. For each record in a scan set it does the following:
  ```
  dbScanLock(precord);
  dbProcess(precord);
  dbScanUnlock(precord);
  ```
  It also has code to recognize when a scan list is modified while the scan set is being processed.
- **addToList**: This routine adds a new element to a scan list.
- **deleteFromList**: This routine deletes an element from a scan list.

Event Scanning

Event scanning is built around the following definitions:

```
#define MAX_EVENTS 256
#define EVENT_QUEUE_SIZE 1000
static struct scan_list *papEvent[MAX_EVENTS];
static SEM_ID eventSem;
static RING_ID eventQ;
static int eventTaskId;
```

`papEvent` is an array of pointers to `scan_lists`. Note that the array has 256 elements, i.e. one for each possible event number. In other words, each event number 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.

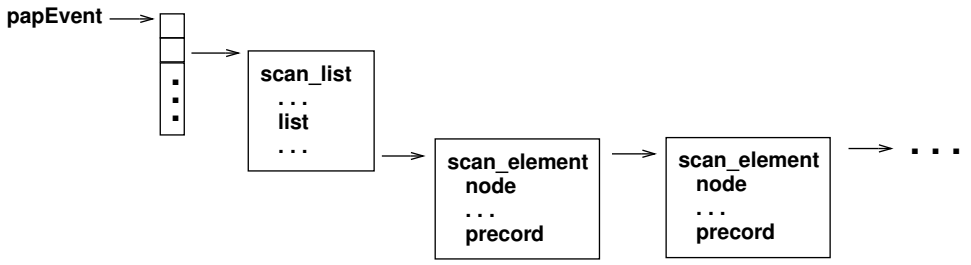


Figure 15-1: Scan List Memory Layout

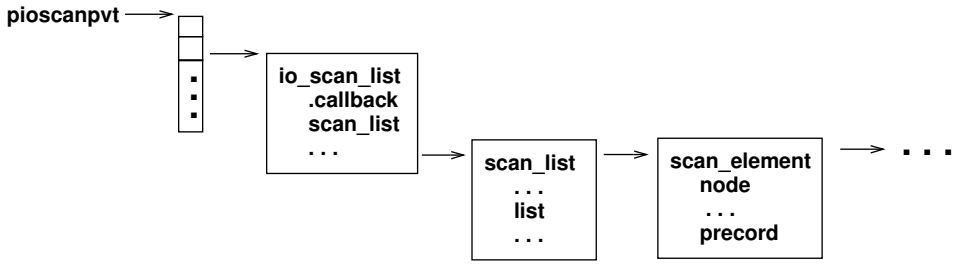


Figure 15-2: Interrupt Source Structure

At `iocInit` time a task "`eventTask`" is spawned. It waits on semaphore `eventSem`. When `post_event` is called it puts the event number on the ring buffer `eventQ` and issues a `semGive` for `eventSem`. This wakes up `eventTask` which calls `scanList` for the

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:

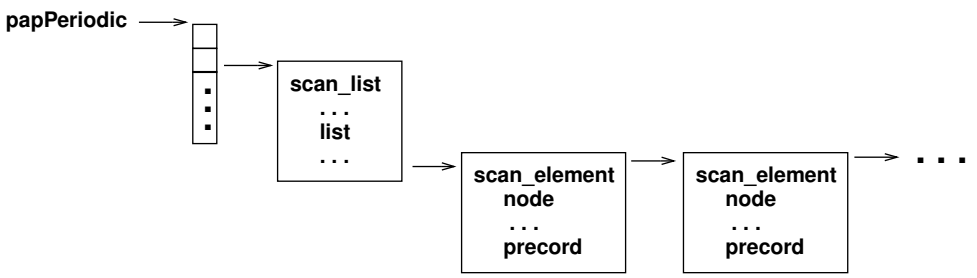


Figure 15-3: Structure after iocInit

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:

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`.

Chapter 16: Database Structures

1. Overview

This chapter describes the internal structures describing an IOC database. It is of interest to EPICS system developers but serious application developers may also find it useful. This chapter is intended to make it easier to understand the IOC source listings. It also gives a list of the header files used by IOC Code.

2. Include Files

This section lists the files in base/include that are of most interest to IOC Application Developers:

alarm.h alarmString.h - These files contain definitions for all alarm status and severity values.

cadef.h caerr.h caeventmask.h - These files are of interest to anyone writing channel access clients.

callback.h - The definitions for the General Purpose callback system.

dbAccess.h - Definitions for the runtime database access routines.

dbBase.h - Definitions for the structures used to store an EPICS database.

dbDefs.h - A catchall file for definitions that have no other reasonable place to appear.

dbFldTypes.h - Definitions for DBF_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 is 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.

i**nitHooks.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.

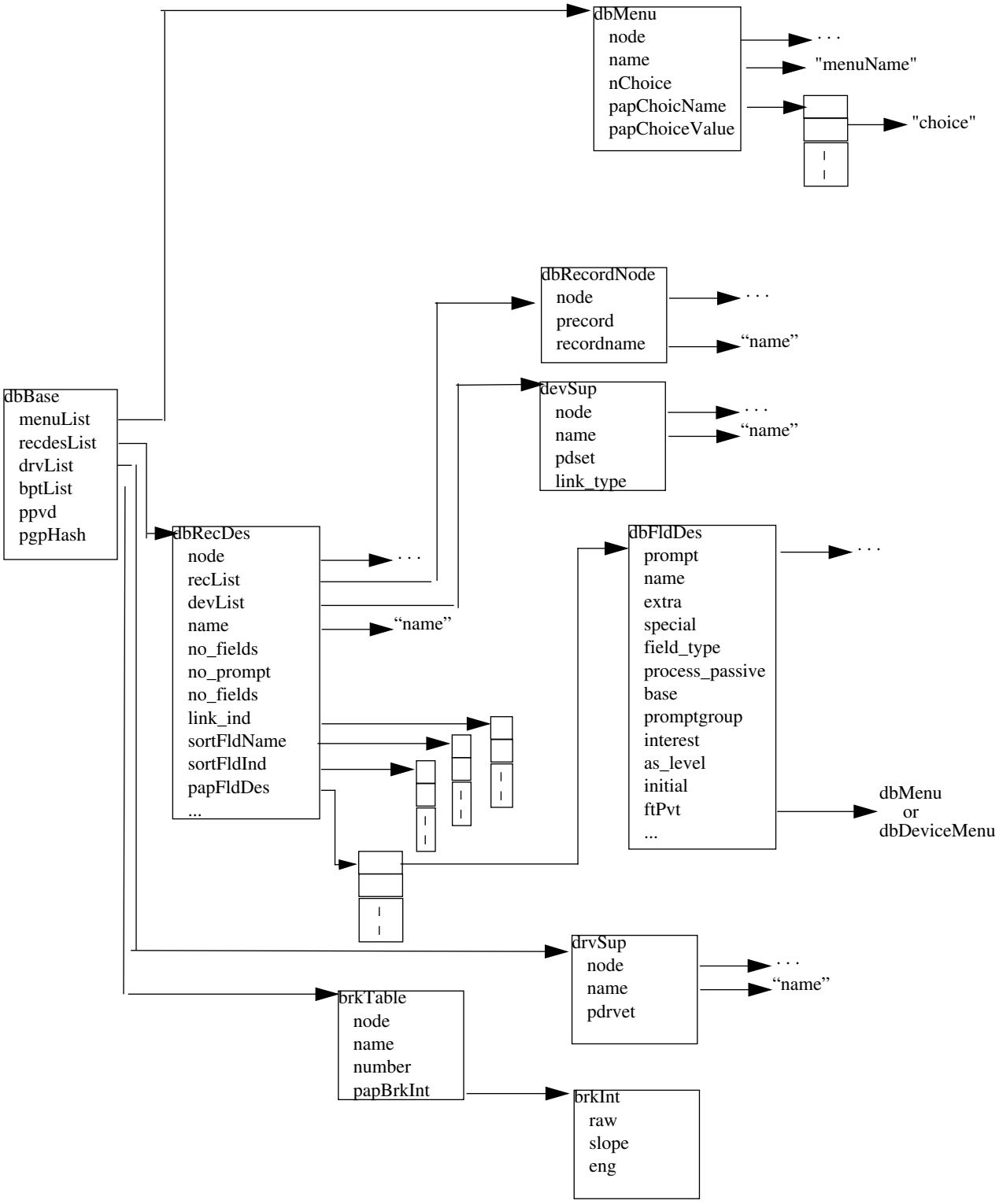
special.h - Definitions for special fields, i.e. SPC_xxx.

task_params.h - Definitions for task priorities, stack space, etc.

taskwd.h - Task Watchdog System

tsDefs.h - Time stamp routines. Will also have ti look at base/src/libCom/tsSubr.c

3. Structures



A

alarm - example. 95
asdbdump 82
asInit 82
asl - field definition rules. 36
asl_level - field definition 37
asphag 82
aspmem 83
asprules 83
aspuag 82
asPvt in DBADDR 129
asSetFilename 82
asynchronous device support example 107

B

base - field definition. 39
base - field definition rules 37
breakpoint table - database definition 42
Breakpoint Tables 45
Breakpoints 79

C

ca_channel_status 83
Cached Puts. 31
CALLBACK 146
callbackGetUser 147
callbackInit 147
callbackRequest 147
callbackRequestProcessCallback 147
callbackSetCallback 147
callbackSetPriority 147
callbackSetUser. 147
Channel Access. 1, 5
choice_string - device definition. 41
client_stat 83
comment - Database Definitions. 36
cvt_dbaddr - Record Support Routine 98

D

Database
  Locking 1–2
  Processing 1–2
  Runtime Access 2
  Scanning 1–2
  Static Access 2
  Structures 2
database access routines - List of 130
database definitions. 33
Database Format - Summary 33
Database Link Guidelines. 26
Database Links 23
Database Locking 24
Database Scanning 25
dba. 78
dbAccess.h 127
dbAdd 129
DBADDR 129
dbAllocEntry. 120
dbAllocForm. 124
dbap. 80
dbAsciiExpand 50
dbAsciiRead 120
dbAsciiReadFP 120
dbAsciiToMenuH 47
dbAsciiToRecordtypeH. 48
dbb. 79
dbBufferSize 138
dbc. 80
dbCaAddOutLink 139
dbCaGetLink. 139
dbCaLinkInit 57–58
dbCopyEntry 120
dbCreateRecord. 122
dbCvtLinkToConstant. 125
dbCvtLinkToPvlink. 125
dbd. 79
dbDefs.h 127
dbDeleteRecord. 122
dbDumpDevice 87, 126
dbDumpDriver 87, 126
dbDumpFldDes. 86, 126
dbDumpMenu 86, 126
dbDumpRecDes 86, 126
dbDumpRecord. 126
dbDumpRecords 87
dbel 83
DBF_xxx Definitions of Field types. 128
dbFastLinkGet. 134
dbFastLinkPut. 135
dbFindBrkTable 124
dbFindField. 123
dbFindMenu 124
dbFindRecdes 122
dbFindRecord 122
dbFinishEntry 120
dbFirstFielddes 123
dbFirstRecdes 122
dbFirstRecord 122
dbFldTypes.h. 127
dbFoundField 122
dbFreeEntry. 120
dbFreeForm. 124
dbGet. 134
dbGetChoices 124
dbGetField. 133
dbGetFieldIndex 138
dbGetFieldName. 123
dbGetFieldType. 123
dbGetFormPrompt 124
dbGetFormValue. 124
dbGetLink 133
dbGetLinkField. 125
dbGetLinkType 125
dbGetMenuIndex 124
dbGetNFields 123
dbGetNLinks. 125
dbGetNMenuChoices 124

dbGetNRecdes 121
dbGetNRecords 122
dbGetPrompt 123
dbGetPromptGroup 123
dbGetPvlink 125
dbGetRange 123
dbGetRecdesName 122
dbGetRecordName 123
dbGetRset 138
dbGetString 123
dbgf 78
dbGrep 78
dbhcr 81
dbInitEntry 120
dbior 81
dbIsDefaultValue 123
dbIsValueField 138
dbl 77
dblls 85
dbllsdblinks 85
dbLoadRecords 50, 55
dbLoadTemplate 51
dbLoadTemplates 55
dbNameToAddr 132
dbNextFielddes 123
dbNextRecdes 122
dbNextRecord 122
dbNotifyAdd 137
dbNotifyCancel 137
dbNotifyCompletion 137
dbp 80
dbpf 78
dbpr 78
dbProcess 139
dbPut 136
dbPutField 134
dbPutForm 124
dbPutLink 135
dbPutMenuIndex 124
dbPutNotify 137
dbPutPvlink 125
dbPutString 123
dbPvdDump 87, 126
dbr_field_type in DBADDR 129
DBR_xxx Database Request Types and Options . 131
dbRenameRecord 123
dbReportDeviceConfig 126
dbs 80
dbScan.h 155
dbScanLock 132
dbScanLockInit 132
dbScanPassive 139
dbScanUnlock 132
dbstat 80
dbt 84
dbtgf 84
dbtpf 84
dbtpn 84
dbtr 79
dbValueSize 138
dbVerify 123
dbVerifyForm 124
dbWriteDevice 121
dbWriteDeviceFP 121
dbWriteDriver 121
dbWriteDriverFP 121
dbWriteMenu 121
dbWriteMenuFP 121
dbWriteRecDesFP 121
dbWriteRecord 121
dbWriteRecordFP 121
devConnectInterrupt 142
devCreateMask 143
devDisableInterruptLevel 142
devDisconnectInterrupt 142
devEnableInterruptLevel 142
device - database definition 41
Device Support 1
  Library 2
Device Support Entry Table 91
devNmlToDig 143
devNormalizedGblGetField 143
devRegisterAddress 142
devUnregisterAddress 142
driver - database definition 41
Driver Support Entry Table Exampl 112
drvet_name - driver definition 42
DSET 91
dset - dbCommon 105
dset_name - device definition 41
dtyp - dbCommon 105

E

e 112
EPICS 1, 5
  Basic Attributes 6
  Hardware/Software Platforms 6
  Overview 1
epicsAddressType 141
epicsAddressTypeName 141
epicsInterruptType 142
epicsPrintf 151
epicsPrtEnvParams 84
EPICStovxWorksAddrType 141
errMessage 151
Error Handling 2
errPrintf 151
Event - Scan Type 153
EVNT - Scan Related Field 154
extra - field definition rules 37
extra_info - field definition 39

F

field_name - field definition 37
field_name - record instance definition 42
field_size in DBADDR 129
field_type in DBADDR 129
filed_type - field definition 37
finish device support 58
FWDLINK 23

G

Get Resource Definitions 56
get_alarm_double Record Support Routine . . 100
get_array_info - Record Support Routine. 98
get_control_double - Record Support Routine . 99
get_enum_str - record Support Routine 99
get_enum_strs - record Support Routine 99
get_graphic_double - example 95
get_graphic_double - Record Support Routine. 99
get_ioint_info - device support routine. 110
get_precision - Record Support Routine. 99
get_units - .example 94
get_units - Record Support Routine 98
gft 85
gui_group - field definition 37
Guidelines for Asynchronous Records 29
Guidelines for Synchronous Records 28

I

I/O Event - Scan Type. 153
I/O Event Scanning. 156
include - Database Definitions 35
init - device support routine 110
init - Record Support Routine. 97
init database scanning 58
init_record - device support routine 110
init_record - example 93
init_record - Record Support Routine. 97
init_value - field definition 37
InitDevSup 57–58
InitDrvSup. 57
initHooks. 54
initial - field definition rules 36
InitRecSup. 57
INLINK. 23
Input/Output Controller 1
  Hardware/Software Platforms. 7
  Initialization 1
  Software Components. 7
interest - field definition rules 37
interest_level - field definition 39
interruptAccept 58
IOC 5
  See Input/Out Controller. 1
iocInit 54
iocLogDisable 56

L

LAN 5
link.h 127
LINK_ALARM. 24
link_type - device definition 41
Local Area Network
  Hardware/Software Platforms. 7
logMsg 56

M

Maximize Severity 24
menu - Database Definition 36
menu - field definition rules 37
menuScan.ascii 154
monitor - example 96
MS. 24

N

name - breakpoint table. 42
NMS 24
no_elements in DBADDR 129
NPP 24

O

Old Database Access. 140
Operator Interface 1
  Hardware/Software Platforms. 6
OPI 5
  See Operator Interface 1
OUTLINK. 23
Overview of Record Processing 90

P

Passive - Scan Type. 153
path - Database Definitions. 35
Periodic - Scan Type 153
pfield in DBADDR 129
pfldDes in DBADDR 129
pft 85
PHAS - Scan Related Field. 154
post_event 156
PP 24
pp - field definition rules. 37
pp_value - field definition 39
precord - DBADDR 129
PRIO - Scan Related Field 154
process - example 93
process - Record Support Routine 98
process - record support routine 26
Process Passive 24
prompt - field definition rules 37
prompt_value - field definition 37
. 37
put_array_info - Record Support Routine. 98
put_enum_str - Record Support Routine 99
PUTNOTIFY 137

R

recdord type - Database Definition 36

recGblDbaddrError 101
recGblFwdLink 102
recGblGetAlarmDouble 101
recGblGetControlDouble 101
recGblGetFastLink 103
recGblGetGraphicDouble 101
recGblGetLinkValue 102
recGblGetPrec 102
recGblGetTimeStamp 102
recGblInitConstantLink 103
recGblInitFastInLink 102
recGblInitFastOutLink 103
recGblPutFastLink 103
recGblPutLinkValue 102
recGblRecordError 101
recGblRecsupError 101
recGblResetAlarms 100
recGblSetSevr 100
record instance - database definition 42
Record Processing 26
Record Support 1
Record Support Entry Table 90
record_name - record instance definition 42
record_type - device definition 41
record_type - record instance definition 42
record_type - record type definition 37
report - device support routine 110
report - Record Support Routine 97
RSET 90
RSET - example 91

S

SCAN - Scan Related Field 154
Scan Once - Scan Type 153
Scan Related Database Fields 154
scanAdd 156
scanDelete 156
scanInit 155
scanOnce 161
scanpel 81
scanpiol 81
scanppl 81
Shell 1
size - field definition rules 37
size_value - field definition 39
special - field definition rules 37
special - Record Support Routine 98
special in DBADDR 129
special_value - field definition 38
struct dbAddr 129
struct putNotify 137
synchronous device support example 106

T

taskwd.h 148
taskwdAnyInsert 149
taskwdAnyRemove 149
taskwdInsert 148
taskwdRemove 149
timexN 84
tpn 86
Ts_init 57
TSConfigure 54
TSreport 82

V

value - record instance definition 42
veclist 83
VME_AM_EXT_SUP_DATA 142
VME_AM_STD_SUP_DATA 142
VME_AM_SUP_SHORT_IO 141
VMS 2
vxWorks 1
vxWorks startup command file 53