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

Martin R. Kraimer
Argonne National Laboratory
Advanced Photon Source
November 1994
APS Release 3.12

Table of Contents

Preface

1. Overview

This document describes the core software that resides in an Input/Output Controller (IOC), one of the major components of EPICS. The plan of the book is:

Chapter 1: 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.

Chapter 2: IOC TEST FACILITIES
Test routines that can be executed via the vxWorks shell

Chapter 3: GENERAL PURPOSE FEATURES
General purpose tasks, and error handling conventions.

Chapter 4: DATABASE LOCKING, SCANNING, and PROCESSING
Overview of three closely related IOC concepts. It is given so that later chapters are more meaningful.

Chapter 5: STATIC DATABASE ACCESS
Database access library that works on Unix and vxWorks and on initialized or uninitialized EPICS databases.

Chapter 6: RUNTIME DATABASE ACCESS
The heart of the IOC software is the memory resident database. Rather then describing database structures first, the runtime database access routines are discussed. This is an easier way of understanding the capabilities of the database.

Chapter 7: DATABASE SCANNING
Database scan tasks, i.e. the tasks that request records to be processed.

Chapter 8: RECORD and DEVICE SUPPORT
The concepts of record and device support are discussed. This information is necessary for anyone who wishes to provide customized record and device support.

Chapter 9: DEVICE SUPPORT LIBRARY

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

Chapter 10: IOC DATABASE CONFIGURATION
Various ASCII definition files used by EPICS as well as the build utilities that read the ASCII files and turn them into files understood by EPICS. Anyone writing record and or device support must understand how to modify these ASCII files.

Chapter 11: IOC INITIALIZATION
A great deal happens at IOC initialization. This chapter takes some of the mystery from initialization.

Chapter 12: 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

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 were 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 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.

Since 1991 many improvements and refinements have been made to the EPICS IOC software by people at APS, LANL, and other Sites.

Matt Needes implemented and supplied the description of fast database links and the database debugging tools.

Chapter 1: EPICS Overview

1. What is EPICS?

EPICS consists of a set of software components and tools with which Application Developers can 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.

Figure 1-1 shows the basic physical structure of a control system implemented via EPICS.

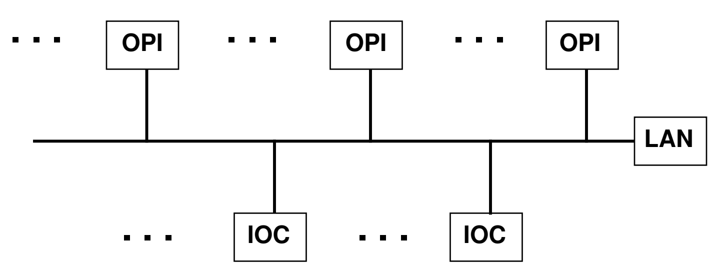


Figure 1-1: EPICS Based Control System

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.

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, currently Sun4s
- Hope to support HP RISC workstation in near future

Software

- UNIX
- X Windows
- Motif Toolkit

LAN

Hardware

- Ethernet and FDDI

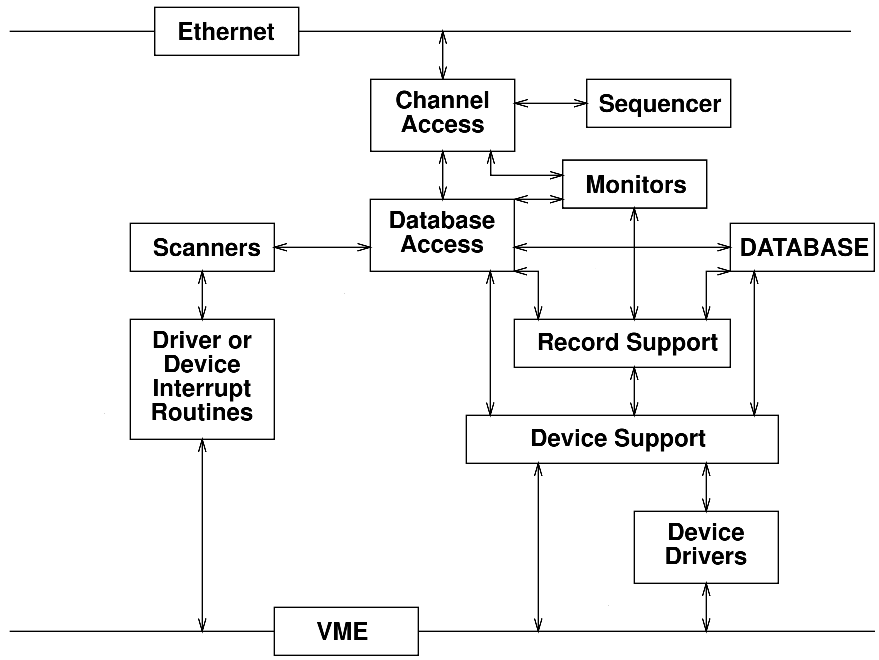


Figure 1-2: System Overview

- ATM in the future

Software

- TCP/IP protocols via sockets

IOC

Hardware

- VME/VXI bus and crates
- Motorola 68020, 68030 and 68040
- Various VME modules (ADCs, DAC, Binary I/O, etc.)
- Allen Bradley Scanner (Most AB I/O modules)
- GPIB devices
- BITBUS devices

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 has 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.

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 "Runtime Database Access" on page 45 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 a later chapter.

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

- **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.
- **XMCA**: Allows the user to monitor and/or change a set of process variables specified at run time.
- **XMSEQ**: GUI which allows the user to prepare sequence programs that can be run on Unix or on an IOC.
- **CAMATH**: Channel Access interface for Mathematica.
- **CAWINGZ**: Channel Access interface for Wingz.
- Other tools are also available.

Other OPI Tools

- **DCT**: Database Configuration Tool. Used to create a run time database for an IOC.
- **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.
- **Build Tools**: Tools are available to create the various database components from 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
- DCT or GDCT
- Build 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: 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, format:

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

Examples

```
dbl
dbl "ai"
```

This command prints the names of the 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, format:

```
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 one or more of any character.

dba

Database Address, format:

```
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, format:

```
dbgf ("<record_name.field_name>")
```

Example:

```
dbgf "aitest"
dbgf  "aitest.VAL"
```

This performs a `dbNameToAddr` and then a `dbGetField`. It prints the value of each element of the `dbAddr` structure as well as the field value. If the field name is not specified then `VAL` is assumed (the two examples above are equivalent).

dbpf

Put Field, format:

```
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, format:

```
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. This is used only for pad fields.

dbtr

Test Record, format:

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

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

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, format:

```
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, format:

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

Removes a breakpoint from a record.

dbs

Single Step, format:

```
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, format:

```
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, format:

```
dbp
```

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

dbap

Auto Print, format:

```
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, format:

```
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, format:

```
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 only a list of modules found.

- **1**: Print additional information.
- **2**: Print even more info. The user may be prompted for options.

dbhcr

Hardware Configuration Report, format:

```
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 so on Unix issue the command:

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

`report.sort` should contain the sort order you desire.

5. Scan Reports

scanppl

Print Periodic Lists, format:

```
scanppl
```

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

scanpel

Print Event Lists, format:

```
scanpel
```

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

scanpiol

Print I/O Event Lists, format:

```
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_chanel_status

Format:

```
ca_chanel_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, format:

```
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, format:

```
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, format:

```
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, format:

```
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.

dblls

List Lock Sets, format:

```
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.

dbls

List Structures: This test routine prints a formatted dump of the internal database structures. It is completely menu driven. Only system developers will be normally be interested in this routine because it assumes that the user understands the internal data structures.

12. 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, Format:

```
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, format:

```
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, format:

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

Example:

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

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

Chapter 3: General Purpose Features

1. Overview

This chapter describes two general purpose IOC features:

- General purpose tasks
- Error handling

2. General Purpose Tasks

Callback Tasks

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:
   ```
   callbackRequest(CALLBACK *)
   ```
   This call can be made from interrupt level. 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.

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. Note that things can be simplified if `CALLBACK` is located at the beginning of the structure.

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 in 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 value unique to call to `taskwdAnyInsert`.
5. Remove request for `taskwdAnyInsert:`
   ```
   taskwdAnyRemove(void *userpvt);
   ```
   `userpvt` is the value that was passed to `taskwdAnyInsert`.

3. Error Handling

Overview

The error handling facilities provided by the IOC include 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 routine `errMessage` provides access to a system wide error handling system.
- A global variable `errVerbose` indicates if routines should call `errMessage` for errors belonging to a particular client.

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

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` or `errPrintf` 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.

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.

Chapter 4: 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.

This chapter only discusses database links. Links are defined in file `link.h`.

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 performing any operations that can modify database records and calling `dbScanUnlock` after the modifications are complete. 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 is the mechanism that requests a database record 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 ASCII 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 “Record And Device Support” on page 65.

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 4-1):

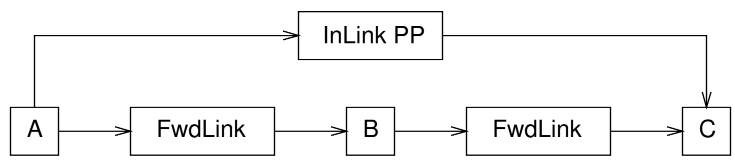


Figure 4-1: Example of Database Links

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 4-2).

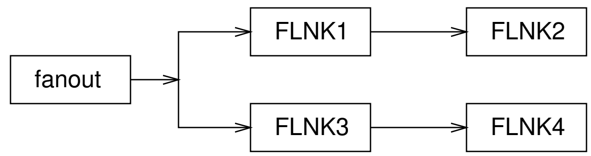


Figure 4-2: Processing Order

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 Section 7 on page 32 and Section 8 on page 33 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 4-3):

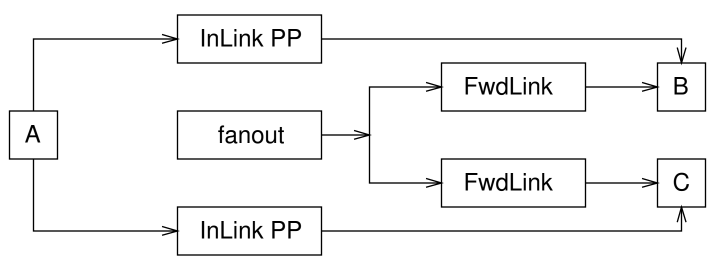


Figure 4-3: Incorrect Link Definition

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 4-4).

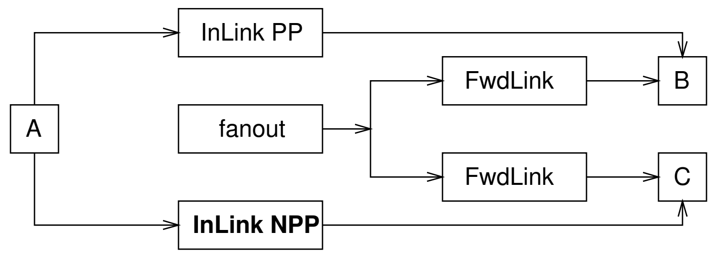


Figure 4-4: Correct Link definition

Process Passive: Field attribute

Each field of each database record type has an attribute called `process_passive`. This attribute is specified in the ASCII 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 via DCT, 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`. See ‘Maximize Severity” on page 28 for details.

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.

Lets review 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

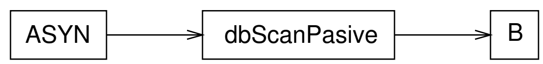
The asynchronous completion routine performs the following algorithm:

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`.

Lets note a few attributes of the above rules:

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

With these rules the following works just fine:

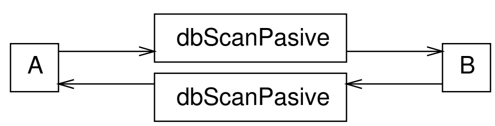


When `dbProcess` is called for record ASYN, processing will be started but `dbScanPassive` will not be called. Until the asynchronous completion routine executes any additional attempts to process ASYN are ignored. When the asynchronous callback is invoked the `dbScanPassive` is performed.

Problems still remain. A few examples are:

Infinite Loop

Infinite processing loops are possible.



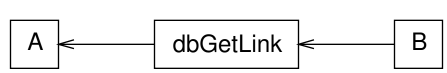
Assume both A and B are asynchronous passive records and a request is made to process A. The following sequence of events occur.

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 `TRUE`.
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 `TRUE`.

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:

ASYN → dbScanPasive → ASYN → dbScanPasive → . . .

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 5: 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. Two flavors of Unix files are supported: a binary file which uses an extension of `.database`, and an ASCII format which uses an extension of `.db` (it is recommended that only the `.db` format be used). A database file is loaded into an IOC via a `dbLoad` command and initialized via the `iocInit` command. A database file contains a number of self defining records which are described later in this manual. 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 access only works on 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. All choice fields are accesses 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.

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 ”`.database`”. Two routines (`dbRead` and `dbWrite`) 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:

```
#include <dbStaticLib.h>
DBBASE  *pdb;

pdb=dbAllocBase();
```

DBENTRY

A typical declaration for a database entry structure is:

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

Most static access to a database is via a `DBENTRY` structure. As many structures as desired can be allocated, each associated with a particular database. The user should NEVER access the fields of `DBENTRY` directly.

Most 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 previous chapter. For static database access a new and simpler set of field types are defined. This allows a simpler interface definition. In addition some database fields can be arrays. For DCT, however, all fields are scalars.

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 ASCII 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 ASCII 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 structure as defined in dbStaticLib.h. Link fields, which have an associated set of static access routines, are described in more detail 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 ASCII versions of the database.

3. EXAMPLE - Dump All Records

The following example demonstrate 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 Structures

```
DBBASE *dbAllocBase(void);
void dbFreeBase(DBBASE *pdbbase);
DBENTRY *dbAllocEntry(DBBASE *pdbbase);
void dbFreeEntry(DBENTRY *pdbentry);
void dbInitEntry(DBBASE *pdbbase,DBENTRY *pdbentry);
void dbFinishEntry(DBENTRY *pdbentry);
DBENTRY *dbCopyEntry(DBENTRY *pdbentry);
```

These routines allocate and free `DBBASE` and `DBENTRY` structures. `dbAllocBase` allocates and initializes a memory resident database. `dbFreebase` frees all memory used by the database. Note that it is possible to work with more then one memory resident database at the same time.

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

The routines `dbInitEntry` and `dbFinishEntry` are provided in case the user wants to allocate a `DBENTRY` structure on the stack.

The routine `dbCopyEntry` allocates a new entry, via a call to `dbAllocEntry`, copies the information from the original entry, and returns the result.

5. Read and Write Database

```
long dbRead(DBBASE *pdbbase,FILE *fp);
long dbWrite(DBBASE *pdbbase,FILE *fpdctsdr,FILE *fp);
```

`dbRead` reads a file containing any combination of self defining records and adds the information to the memory resident database. `dbWrite` writes the memory resident database into a file. `dbWrite` requires two file pointers. The first is a file containing record description information. The second references the output database file.

Although an arbitrary number of database files can be read each must contain the same set of record descriptions. If any mismatch occurs `dbRead` will return an error. If `dbRead` returns a non zero value do not call any of the other routines described in this chapter.

6. Manipulating Record Descriptions

```
long dbFindRecdes(DBENTRY *pdbentry,char *recdesname);
long dbFirstRecdes(DBENTRY *pdbentry);
long dbNextRecdes(DBENTRY *pdbentry);
char *dbGetRecdesName(DBENTRY *pdbentry);
int  dbGetNRecdes(DBENTRY *pdbentry);
long dbCopyRecdes(DBENTRY *from,DBENTRY *to);
```

These routines manipulate the record description. An EPICS database consists of an arbitrary number of record descriptions. The above routines provide access to the following information:

- **Name**: `dbGetRecdesName` returns the record description name, e.g. ”`ai`”
- **Number**: `dbGetNRecdes` returns the number of record descriptions, i.e. the number of record types.

7. Manipulating Record Instances

```
long dbCreateRecord(DBENTRY *pdbentry,char *precordName);
long dbDeleteRecord(DBENTRY *pdbentry);
long dbFindRecord(DBENTRY *pdbentry,char *precordName);
long dbFirstRecord(DBENTRY *pdbentry); /*first of record type*/
long dbNextRecord(DBENTRY *pdbentry);
int  dbGetNRecords(DBENTRY *pdbentry);
char *dbGetRecordName(DBENTRY *pdbentry);
long dbRenameRecord(DBENTRY *pdbentry, char *newname)
long dbCopyRecord(DBENTRY *from,DBENTRY *to);
```

These routines are used to create, delete, locate, etc. record instances. Note that other than `dbFindRecord` all routines assume that one of the record description routines has been used to locate a record type. `dbFindRecord` also calls `dbFindField` if the record name includes an extension.

Routines are provided for accessing the following information:

- **Name**: `dbGetRecordName` returns the record name
- **Number**: `dbGetNRecords` returns the total number of record instances for current record type.

8. Manipulating Field Descriptions

```
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.

9. 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 ASCII versions of a database.

10. Manipulating Menu Fields

MENU and MENUFORM Fields

```
char **dbGetChoices(DBENTRY *pdbentry);
int  dbGetMenuIndex(DBENTRY *pdbentry);
long dbPutMenuIndex(DBENTRY *pdbentry,int index);
int  dbGetNMenuChoices(DBENTRY *pdbentry);
long dbCopyMenu(DBENTRY *from,DBENTRY *to);
```

These are the routines that should be used for `DCT_MENU` and `DCT_MENUFORM` fields.

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 skeleton 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`.

11. 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.

12. Dump Routines

```
void dbDumpRecords(DBBASE *pdbbase, char *recdesname,int modOnly);
void dbDumpPvd(DBBASE *pdbbase);
void dbReportDeviceConfig(DBBASE *pdbbase,FILE *report)
```

13. Utility Programs

NOTE: The routines described in this section are provided to translate between the old DCT short form reports and `.database` files. Everyone is **STRONGLY** encouraged to start using the GDCT format and use `dbLoadRecords` and/or `dbLoadTemplate` to load databases into IOCs. Thus the `atdb` and `dbta` should be considered temporary commands. See the GDCT document for details.

atdb

ASCII to Database, format:

```
atdb  <dctsdr> <database>
```

e.g.

```
atdb default.dctsdr example.database < example.rpt
```

This program. which accepts its input from `stdin`, creates a new database file and populates it with records defined in the ASCII file. The ASCII file is a file in the old DCT short form format or a file generated by the `dbta` utility with the `-s` option.

It should also be noted that instead of terminating records with a ^L (the old short form report), it is also permissible to terminate records with `$$end`. Thus, in addition to short form reports, the following is valid input to `atdb`:

```
PV: <record name>   Type: <record type>
<field name> <value>
...   <as many fields as desired>
$$end
PV: <record name>   Type: <record type>
<field name> <value>
...    <as many fields as desired>

$$end
...
```

Use `dbta`, described next, on an existing database without the `-s` option to see an example.

dbta

Database to ASCII, format:

```
dbta [-v] [-s] <filename>
```

e.g.

```
dbta  -s example.database > newexample.rpt
```

This utility generates an ASCII file from a database file. If `-v` is specified then all prompt fields are generated, otherwise only fields with non-default values are displayed. If `-s` is specified, then the generated file can be used as input to the old DCT or to `atdb`.

Chapter 6: 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. These routines are used within an IOC to access an initialized database.
- Old Database Access. This is the interface still used by Channel Access and thus by Channel Access clients.

2. Database Include Files

Directory `epics/share/epicsH` contains a number of database related include files. A complete list is:

- **dbDefs.h**: Miscellaneous database related definitions
- **dbBase.h**: Base pointers for database structures
- **choice.h**: Choice Structures (Cvt, Gbl, Rec, and Dev)
- **cvtTable.h**: Conversion Structures
- **dbAccess.h**: Runtime database access definitions
- **dbDefs.h**: Basic database related definitions
- **dbFldTypes.h**: Field type definitions
- **dbRecDes.h**: Record and field description structures
- **dbRecType.h**: Record type structures

- **dbRecords.h**: Database record locations structures
- **devSup.h**: Device support structures
- **drvSup.h**: Driver support structures
- **link.h**: Link structures
- **recSup.h**: Record support structures
- **sdrHeader.h**: Self Defining Record header structures

`dbDefs.h`, `dbFldTypes.h`, and `dbBase.h` are discussed in this section. `dbAccess.h` is discussed with runtime database access. The other include files are of interest to someone needing details about the database structures. See last chapter for details.

Fundamental Database Definitions

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 allowed in a field name. This has the value 4. The process variable directory routines depend on this value because they treat the value as an unsigned long data type.
- **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_GBLCHOICE**: A global choice field
- **DBF_CVTCHOICE**: A conversion choice field
- **DBF_RECCHOICE**: A record specific choice field
- **DBF_DEVCHOICE**: 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_DEVCHOICE 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.

dbBase.h

The database and all its associated structures are located via the following set of variables defined in structure dbBase (defined in dbBase.h):

```
struct dbBase {
    struct choiceSet  *pchoiceCvt;
    struct arrChoiceSet  *pchoiceGbl;
    struct choiceRec  *pchoiceRec;
    struct devChoiceRec  *pchoiceDev;
    struct arrBrkTable  *pcvtTable;
    struct recDes  *precDes;
    struct recType  *precType;
    struct recHeader  *precHeader;
    struct recDevSup  *precDevSup;
    struct drvSup  *pdrvSup;
    struct recSup *precSup;
    struct pvd  *pdbPvd; /* DCT pvd - remove when DCT goes away */
    void  *ppvd;/* pointer to process variable directory*/
    char  *pdbName;/* pointer to database name*/
    struct sdrSum  *psdrSum;/* pointer to default sum */
    long  sdrFileSize; /*size of default.dctsdr file*/
    long  pvtSumFlag; /*internal use only*/
};
```

3. Runtime Database Access

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. It may seem strange that the structure of the IOC database is not described at this point but this would result in confusing detail. 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:

- **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
- **dbPutNotify**: A database put with notification on completion
- **dbNotifyCancel**: Cancel `dbPutNotify`
- **dbPut**: Routine called by `dbPutxxx` and by the `dbCa` functions.

- **dbBufferSize**: Determine number of bytes in request buffer.
- **dbValueSize**: Number of bytes for a value field.

- **dbScanPassive**: Process record if passive
- **dbProcess**: Process Record

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

- **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

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**: returns a `NULL` terminated string
- **DBR_CHAR**: returns a signed char
- **DBR_UCHAR**: returns an unsigned char
- **DBR_SHORT**: returns a short integer
- **DBR_USHORT**: returns an unsigned short integer
- **DBR_LONG**: returns a long integer
- **DBR_ULONG**: returns an unsigned long integer
- **DBR_FLOAT**: returns an IEEE floating point value
- **DBR_DOUBLE**: returns an IEEE double precision floating point value
- **DBR_ENUM**: returns a short which is the enum item

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_DEVCHOICE`, `DBF_CVTCHOICE`, `DBF_GBLCHOICE`, and `DBF_RECCHOICE`. The complete set of database field types are defined in `dbFldTypes.h`.

`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

dbAccess.h

Before describing the routines a few data structures must be described. The structures are defined in `dbAccess.h`. The first structure is `dbAddr`.

```
struct dbAddr{
    struct dbCommon *precord:   /* address of record  */
    void  *pfield;          /* address of field  */
    void  *pfldDes;         /* address of struct fldDes  */
    long  no_elements;      /* number of elements (arrays)  */
    short  record_type;     /* type of record being accessed */
    short  field_type;      /* type of database field  */
    short  field_size;      /* size (bytes) of the field being
```

```
                        accessed */
short  special;         /* special processing  */
short  choice_set;      /* index of choiceSet GBLCHOICE &
                        RECCHOICE*/
short  dbr_field_type}/* field type as seen by database
                            request DBR_STRING, ...,
                            DBR_ENUM, DBR_NOACCESS*/
```

- **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 "Database Structures" on page 101 for a description of this structure.
- **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.
- **record_type**: An index specifying the record type. See "Database Structures" on page 101 for how this is used.
- **special**: Some fields require special processing. This specifies the type. Special processing is described later in this manual.
- **choice_set**: For global and record choice fields (described below), this specifies a choice set.
- **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.

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.

Database Access Routines

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`.

dbNameToAddr

Locate a process variable, format:

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

Given a process variable name, this routine locates the process variable and fills in the fields of structure `dbAddr`. The process variable name is of the form "<record_name>.<field_name>". For example the value field of a record with record name '`sample_name`" is "`sample_name.VAL`". Note that the name is case sensitive. All field names are 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.

dbCommonInit

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

dbGetField

Get values associated with a process variable, format:

```
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:

```
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

```
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:

```
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.

dbPutField

Change the value of a process variable, format:

```
dbPutField(
    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 `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:

```
dbPutLink(
    struct  db_link  *pdbLink,      /*addr of database link*/
    struct  dbCommon  *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:

```
dbFastLinkPut(
    struct  link  *plink,
    struct  dbCommon  *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.

dbPutNotify

Put And Notify When Complete, format:

```
typedef struct putNotify{
    /*The following members MUST be set by user*/
    void        (*userCallback)(struct putNotify *);
    struct 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;
    /*The following are private to database access*/
    CALLBACK    callback;
    void        *list;  /*list of records for which to wait*/
    int         nwaiting;
    notifyCmd   cmd;
    unsigned char rescan; /*Should dbPutNotify be called again*/
```

```
}PUTNOTIFY;

long dbPutNotify(PUTNOTIFY *pputnotify);
void dbNotifyCancel(PUTNOTIFY *pputnotify);
```

The following routine is used only by old database access.

```
int dbPutNotifyMapType(PUTNOTIFY *pputnotify,short dbr_type)
```

The status value stored in `PUTNOTIFY` can be one of the following:

- **0**: Success: Callback was already called.
- **S_db_Pending**: Success: Callback will be called later.
- **S_db_Blocked**: The request failed because a `dbPutNotify` conflict occurred.
- **S_xxxx**: The request failed due to some other error.

`dbPutNotify` is a request to notify the caller when all records that are processed as a result of the put complete processing. The complication occurs because of asynchronous records. The following is true:

1. The user supplied callback is called when all processing is complete or when a `S_db_Blocked` is detected. If everything completes synchronously the callback routine will be called BEFORE `dbPutNotify` returns. The user supplied callback routine must not issue any calls that block such as Unix I/O requests.
2. 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 has resulted in a record in the set being already active then the user callback is called with `status=S_db_Blocked`.
3. If a record in the set is found to be active because of a `dbPutField` request then when the record completes a new `dbPutNotify` will be issued.
4. If a record is found to be active for some other reason 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.
5. 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`.

dbPut

Put a value to a database field, format:

```
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.

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

dbScanPassive

Process record if it is passive, format:

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

This request specifies 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 this routine is called by `dbGetLink`, `dbPutField`, and by record processing routines for forward links. In addition to calling `dbProcess` this routine is responsible for creating lists needed for `dbPutNotify`.

dbProcess

Request that a database record be processed, format:

```
dbProcess(struct dbCommom  *precord);  /* addr of record*/
```

Request that record be processed. Record processing is described in detail below.

Database Locking

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

The routines provided are:

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

dbScanLock(struct dbCommon  *precord);/*addr of record*/

dbScanUnlock(struct dbCommon *precord);/*addr of record*/
```

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:

```
dbCaAddInLink(
    struct link  *plink,
    void    *precord,
    char    *pfieldName);

dbCaAddOutLink(
    struct link  *plink,
```

```
        void    *precord,
        char    *pfieldName);

    dbCaGetLink(
        struct link  *plink);

    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

4. 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 7: 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

choiceGbl.ascii

This file contains definitions for a menu related to field `SCAN`. The definitions are of the form:

```
GBL_SCAN    "Passive"
GBL_SCAN    "Event"
GBL_SCAN    "I/O Intr"
GBL_SCAN    "10 second"
...
```

```
GBL_SCAN    ".1 second"
```

The first three definitions 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 `choiceGbl.ascii` and running the `makeSdr` utility. 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.

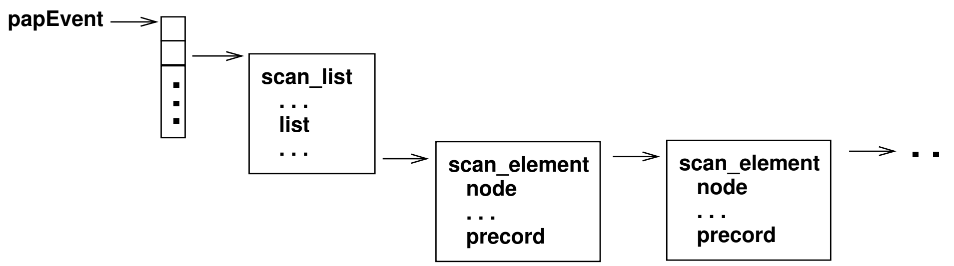


Figure 7-1: Scan List Memory Layout

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

Event Scanning

Event scanning is built around the following definitions:

```
#define MAX_EVENTS 256
#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 7-1.

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 appropriate `scan_list`.

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:

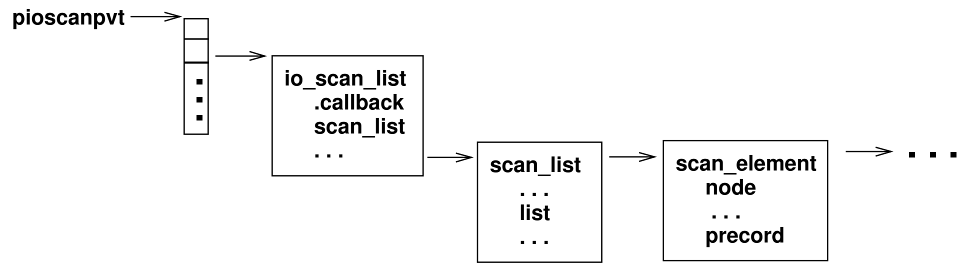


Figure 7-2: Interrupt Source Structure

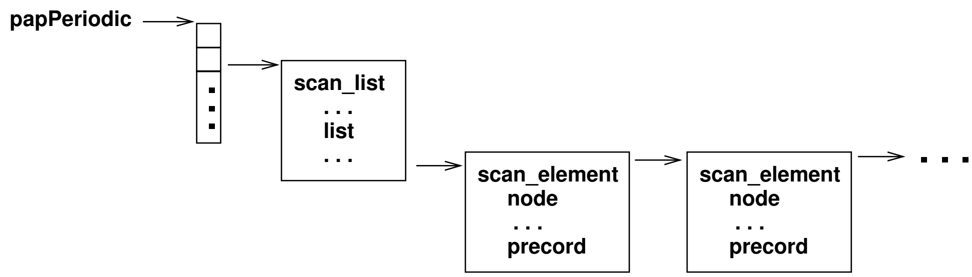


Figure 7-3: Structure after iocInit

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 7-2:

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

scanIoRequest

```
scanIoRequest (IOSCANPVT pioscanpvt)
```

This routine is called to request I/O event scanning. It can be called from interrupt level. It looks at each `io_scan_list` referenced by `pioscanpvt` (one for each callback priority) and if any elements are present in the `scan_list` a `callbackRequest` is issued. The appropriate callback task calls routine `ioeventCallback`, which just calls `scanList`.

Periodic Scanning

Periodic scanning is built around the following definitions:

```
static int nPeriodic;
static struct scan_list **papPeriodic;
static int *periodicTaskId;
```

`nPeriodic`, which is determined at `iocInit` time, is the number of periodic rates. `papPeriodic` is a pointer to an array of pointers to `scan_lists`. There is an array element for each scan rate. Thus the structure illustrated in Figure 7-3 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 8: Record And Device Support

1. Overview

The purpose of this chapter is to describe record and device support in sufficient detail so that a C programmer can write new record and/or device 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 the previous discussion, 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 all 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 `epics/share/epicsH/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. This section describes the overall model followed by record processing. Before the record specific "`process`" routine is called, the following has already been done:

- Decision to process a record.
- Check that record is not already active (`pact TRUE`).
- 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. The method to follow is to 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;   /* get value field */
        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 item */
        RECSUPFUN   get_enum_strs;   /* get all enum strings */
        RECSUPFUN   put_enum_str;   /* put enum item 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 support for particular record */
        DEVSUPFUN   get_ioint_info;   /* get io interrupt information */
        /* 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
static long get_value();
#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 */
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;
};

/* forward declaration for internal routines*/
static void alarm();
static void monitor();
```

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(pxxx,pass)
    struct xxxRecord   *pxxx;
    int  pass;
{
    struct xxxdset *pdset;
    long  status;

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

    if((pdset = (struct 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(pxxx)
    struct xxxRecord   *pxxx;
{
    struct xxxdset   *pdset = (struct xxxdset *)(pxxx->dset);
    long   status;
    unsigned char   pact=pxxx->pact;

    if( (pdset==NULL) || (pdset->read_xxx==NULL) ) {
        /* leave pact true so that dbProcess doesnt keep calling*/
        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 */
    if(!pact && pxxx->pact) return(0); /* return if beginning of asynch processing*/
    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_value(pxxx,pvdes)
    struct xxxRecord  *pxxx;
    struct valueDes  *pvdes;
{
    pvdes->field_type = DBF_FLOAT;
    pvdes->no_elements=1;
    (float *)(pvdes->pvalue) = &pxxx->val;
    return(0);
}

static long get_units(paddr,units)
    struct dbAddr *paddr;
    char   *units;
{
    struct xxxRecord  *pxxx=(struct xxxRecord *)paddr->precord;

    strncpy(units,pxxx->egu,sizeof(pxxx->egu));
    return(0);
}

static long get_graphic_double(paddr,pgd)
    struct dbAddr  *paddr;
    struct dbr_grDouble  *pgd;
{
    struct xxxRecord  *pxxx=(struct xxxRecord *)paddr->precord;

    if(paddr->pfield == (void *)(&pxxx->val)) {
        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 Section 5 on page 72.

Alarm Processing

```
static void alarm(pxxx)
    struct 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. The above algorithm ensures that the alarm being reported will not change unless the value changes by the hysteresis value.

Raising Monitors

```
static void monitor(pxxx)
    struct 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;
}
```

The first part of the code will be common to most record types. 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.

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. 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` 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 at some future time.

Generate Error: Process Variable Name, Caller, Message

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

This routine can be called whenever an error is returned from a call to dbNameToAddr, dbGetxxx, or dbPutxxx. It 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.

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 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`.

6. 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. A field is defined to be user special in the ASCII record definition.

Get Value

```
get_value(
    void   *precord,   /* addr of record*/
    struct valueDes   *p);   /*addr of value description struct*/
```

This routine returns a description of the `VAL` field of the record. The structure `valueDes`, which is defined in `recSup.h`, is defined as follows:

```
struct valueDes {
    int    field_type,
    long   no_elements,
    void   *pvalue}
```

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`, and `special`. 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   old_offset,
    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. Note that `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`.

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`.

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.

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`. Note that `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`. Note that `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`.

7. Example Device Support Modules

In addition to a record support module, each record type has an arbitrary number of device support modules. The purpose of device support is to hide device specifics 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 the device support routines are the interface between hardware specific fields in a database record and device drivers or the hardware itself.

The common portion of every database record contains two device related fields:

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

The field `dtyp` is filled in by DCT. It contains the index of the menu choice as defined in `devSup.ascii`. `iocInit` uses this field and the device support structures defined in `devSup.h` to initialize the field `dset`.

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(pai)
    struct aiRecord  *pai;
{

    long status;
    /* ai.inp must be a CONSTANT or a PV_LINK or a DB_LINK or a CA_LINK*/
    switch (pai->inp.type) {
        case (CONSTANT) :
            pai->val = pai->inp.value.value;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(pai)
    struct aiRecord*pai;
{
    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.

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.
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(pcallback)
    struct callback *pcallback;
{
    struct dbCommon   *precord=pcallback->precord;
    struct rset   *prset=(struct rset *)(precord->rset);

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

static long init_record(pai)
    struct aiRecord  *pai;
{
   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);
        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(pai)
    struct aiRecord   *pai;
{
    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);
                printf("%s Starting asynchronous processing\n",pai->name);
                wdStart(pcallback->wd_id,wait_time,callbackRequest,(int)pcallback);
                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);
}
```

8. 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 the database records are initialized and once after. `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.

9. 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 particular device driver.

File `drvSup.h` describes the format of a driver support entry table. File `drvSup.ascii` defines the supported device drivers.

Chapter 9: 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 10: IOC Database Configuration

1. Overview

This chapter describes the ASCII files that must be modified and/or created in order to provide new record support, device support, and/or device drivers. Before the ASCII files are described, an overview of database configuration and the concept of Self Defining Record (SDR) files is presented. Although it is not necessary for the application developer to understand these concepts, the discussion should clear up the mystery of what happens to the ASCII definition files.

The serious reader should obtain a listing of all the ASCII files in `base/src/ascii`.

2. Overview of IOC Database Configuration

NOTE: Everyone is **STRONGLY** encouraged to start using the GDCT ASCII database format and `dbLoadRecords` and `dbLoadTemplates`. See the GDCT document for details.

The IOC database is a memory resident database plus assorted data structures. Many of the data structures are configured via ASCII definition files. Let's briefly discuss the steps involved up to and including initialization of an IOC database.

1. Create ASCII Files. Each configuration component has one or more associated ASCII definition files. The components and related ASCII files are:
   - **dbRecType.ascii**: The allowable record types
   - **choiceGbl.ascii**: Global choices, i.e. options common to multiple record types.
   - **choiceRec.ascii**: Record specific choices.
   - **cvt*.ascii**: A group of ASCII files for defining conversion options.
   - **devSup.ascii**: Device support.
   - **drvSup.ascii**: Driver support.

- **dbCommon.ascii**: Definition of fields common to all record types.
- ***Record.ascii**: Record specific field definitions.

2. Create a DCT SDR File. Build utilities are provided to process the ASCII definition files. The build utilities convert each ASCII file to a self defining record (SDR) file. An SDR file contains a set of one or more self defining records. Two or more SDR files can be concatenated to create a file that is again an SDR file. A script file ”makesdr” executes the appropriate build utilities and concatenates the output files so that an SDR file appropriate for input to DCT is generated.
3. OLD STYLE:
   Create a database via DCT. DCT reads the SDR file generated by makesdr and accepts user input. The user creates and/or modifies an arbitrary number of records. When done DCT generates a file <name>.database. This file, which is also in SDR format, contains all the SDR records from the input SDR file as well as SDR records for the actual database records.
   NEW STYLE
   Create the database file via GDCT. It saves a file with the extension .db
4. OLD STYLE:
   After an IOC is booted and iocCore is loaded, the commands:
   ```
   dbLoad(”<database>”)
   ...
   ```
   NEW STYLE:
   ```
   dbLoad("default.dctsdr")
   dbLoadRecords("<file>.db")
   ...
   ```
   OLD AND NEW
   ```
   ...
   iocInit(”<resource file>”
   ```
   are executed. dbLoad reads the SDR file containing the database.

The Source/Release control manual describes the details of creating the SDR files. This manual merely describes the contents of the ASCII files.

3. Self Defining Records

Self defining records provide the following features:

1. Many different types of information can be stored in the same file.
2. Two or more files containing self defining records can be combined with the Unix cat command to form a new file that is also in self defining record format.
3. Record structures can contain pointer fields. In files all pointers are kept as offsets. When sdrLoad reads a self defining record all offsets are automatically converted to addresses.

Each self defining record consists of a header (sdrHeader) followed by data. The header has the following format:

```
struct sdrHeader {
    long  magic;  /* magic number */
    long  nbytes;  /*number of bytes of data which follow header*/
    short  type;  /* sdr record type*/
    short  pad;
    long  create_date;  /* creation date in standard unix format*/
}
```

The allowable types are:

- **SDR_DB_RECTYPE**: Record Types
- **SDR_DB_RECORDS**: The actual database records
- **SDR_DB_RECDES**: Record and field descriptions
- **SDR_CHOICEGBL**: Global choices
- **SDR_CHOICECVT**: Conversion choices
- **SDR_CHOICEREC**: Record specific choices
- **SDR_CHOICEDEV**: Device support choices
- **SDR_DEVSUP**: Device support description structures
- **SDR_CVTTABLE**: Conversion tables
- **SDR_DRVSUP**: Driver support structures
- **SDR_RECSUP**: Record support structures

NOTE: `SDR_DB_RECTYPE` is needed to decipher many of the other SDRs. If needed it must always be the first SDR in a file.

Each type of self defining record is created by one of the build utilities or by DCT. Two subroutines are provided for use by any programs that want to use self defining records. The two routines are `sdrLoad` and `sdrUnload`.

4. Ascii Definition Files

dbRectype

This file, which defines the valid record types, has the format:

```
"<record type>"
...
```

Choice

choiceGbl

The global choice table has the following format:

```
<choice_set>  "<choice_string>"
...
```

The values for `<choice_set>` are defined in `choiceGbl.h`.

choiceRec

This file contains choices special to particular record types. For each record type the following definitions are accepted:

```
"record type" <choice_set> "<choice_string>"
...
```

The choice sets are defined in various record specific include files.

cvtTable

Raw data can be converted to engineering units via one of the following:

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

A conversion file consists of a set of definitions. The first two lines define no conversion and linear conversion. The remainder of the file defines breakpoint tables.

There are two methods of preparing breakpoint tables. The first method is to directly provide the breakpoint table. The second is to provide a table of raw values corresponding to equally spaced engineering values.

The format for directly defining a breakpoint table is as follows:

```
"<name>"  BreakTable
<raw value> <eng value>
...
ENDTABLE
```

The format for generating a breakpoint table from a data table of raw values corresponding to equally spaced engineering values is:

```
<header line>
<data table>
ENDTABLE
```

The header line contains the following information:

- **Name**: ASCII string
- **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:

```
"NO CONVERSION"
"LINEAR"
"TypeKdegF" 32 0 1832 4095 1.0 -454 2500 1
<data table>
ENDTABLE
"Example Breakpoint" BreakTable
0  0
1000  .1
2000  2.5
3000  3.9
4000  5.5
4096  8.0
ENDTABLE
```

devSup

This file defines the device support for each record type. For each record type the following definitions are accepted:

```
"record type"  <link_type> "<dset_name>" "<choice_string>"
    ...
```

where

- **" record type"**: Name of the record type
- **<link_type>**: Link type as defined in link.h
- **<dset_name>**: ASCII name of the device support entry table.

- **<choice_string>**: String value for this choice.

drvSup

This file contains the name of each driver entry table. It has the form:

```
"<drvet_name>"
  ...
```

5. Record Description Files

An ASCII definition file must exist for each record type (for example `ai.ascii`). This file describes each field of the record except the fields defined by `db_common.ascii`.

Preceding the field definitions is a line of the form

```
RECTYPE   "<type>"
```

Each field is defined by a number of definitions. The following definitions appear for all fields:

- **prompt**: Prompt string enclosed in double quotes
- **fldname**: Field Name string
- **special**: Special Processing
- **aslev**: Access Security Level
- **field_type**: Field Type as specified in `db_fldtypes.h`
- **process_passive**: Should `dbPutField` cause passive record to be processed
- **interest**: Interest level

The remaining definitions depend of the `field_type`.

- **field_type**: `DBF_STRING`
  - **size**: Field Size
  - **promptflag**: `YES` or `NO`
- **field_type**: `DBF_UCHAR`, `DBF_SHORT`, `DBF_LONG`, `DBF_ULONG`, `DBF_FLOAT`, `DBF_DOUBLE`, or `DBF_ENUM`.
  - **initial**: Initial Value
  - **promptflag**: 0 or 1

  If promptflag is >=1 then the following are defined:
  - **lowfl**: `CON` or `VAR`
  - **range1**: Field name(`VAR`) or Value(`CON`) for low operating range
  - **highfl**: `CON` or `VAR`
  - **range2**: Field name(`VAR`) or Value(`CON`) for high operating range

  If field_type is `DBF_UCHAR`, `DBF_SHORT`, `DBF_LONG`, or `DBF_ULONG`:
  - **cvt_type**: `CT_DECIMAL` or `CT_HEX`
- **field_type**: `DBF_GBLCHOICE`, or `DBF_RECCHOICE`.
  - **initial**: Initial Value
  - **choice_set**: Index of choice set
  - **promptflag**: `YES` or `NO`
- **field_type**: `DBF_CVTCHOICE`, or `DBF_DEVCHOICE`.
  - **initial**: Initial Value
  - **promptflag**: `YES` or `NO`
- **field_type**: `DBF_INLINK`, `DBF_OUTLINK`, or `DBF_FWDLINK`.

  - **promptflag**: `YES` or `NO`
- **field_type**: `DBF_NOACCESS`.
  - **size**: Field Size
  - **xxx**: Code to be inserted in the `.h` file.

6. ASCII Build Utilities

The ASCII definition files are not used directly by IOC software or by the Database Configuration Tool (DCT). Instead they are translated by one of a set of ”`Build`” utility programs. This section lists each build utility, the ASCII input files it accepts, and the SDR file it generates. It also lists the input and output for DCT.

bldCvtTable

This program reads file `cvtTable.ascii` and generates two files: `cvtTable.sdr` and `choiceCvt.sdr`.

INPUT:

```
cvtTable.ascii
```

OUTPUT:

```
SDR_CVTTABLE   cvtTable.sdr
SDR_CHOICECVT   choiceCvt.sdr
```

bldGblChoice

This program reads file `choiceGbl.ascii` (after it is processed by `cpp`) and generates file `choiceGbl.sdr`.

INPUT:

```
choiceGbl.ascii (after processing by cpp)
```

OUTPUT:

```
SDR_CHOICEGBL   choiceGbl.sdr
```

bldRecChoice

INPUT:

```
choiceRec.ascii (after processing by cpp)
SDR_DB_RECTYPE dbRecType.sdr
```

OUTPUT:

```
SDR_CHOICEREC  choiceRec.sdr
```

bldDevSup

INPUT:

```
devSup.ascii
SDR_DB_RECTYPE   dbRecType.sdr
```

OUTPUT:

```
SDR_DEVSUP   devSup.sdr
SDR_CHOICEDEV   choiceDev.sdr
```

bldRecDef

INPUTS:

```
SDR_DB_RECTYPE   dbRecType.sdr
dbCommon.ascii   All .ascii after cpp. dbCommon must be first
<ai, etc>.ascii
```

OUTPUTS:

```
SDR_DB_RECDES   dbRecDes.sdr
SDR_DB_DCTRECDES   dbDctRecDes.sdr
<aiRecord, etc>.h
```

bldDbRecType

INPUT:

```
dbRectype.ascii
```

OUTPUT:

```
SDR_DB_RECTYPE dbRecType.s
```

bldDrvSup

INPUT:

```
drvSup.ascii   From stdin
```

OUTPUT:

```
SDR_DRVSUP   devSup.sdr
```

7. DCT - Database Configuration Tool

INCLUDES:

```
choice.h
dbDctRecDes.h
dbDefs.f
dbFldTypes.h
dbRecords.h
dbRecType.h
```

INPUTS: All inputs concatenated into file `<appl>.dctsdr`

```
SDR_DB_RECTYPE
SDR_DB_RECDES
SDR_CHOICEGBL
SDR_CHOICECVT
SDR_CHOICEREC
SDR_CHOICEDEV
SDR_DEVSUP
```

IN/OUT: The records are all stored in `SDR_DB_RECORDS` format. The ”`.database`” file contains the `dctsdt` records plus all database record.

Chapter 11: IOC Initialization

1. Overview

After vxWorks is loaded at IOC boot time, the following commands are issued to load and initialize the control system software:

```
ld  <  targetmv167/iocCore
ld  <  targetmv167/drvSup
ld  <  targetmv167/recSup
ld  <  targetmv167/devSup

ld  <  initHooks.o

# iocLogDisable = 1
# TSconfigure(0)    #See below for options

dbLoad(”default.dctsdr”)
dbLoadRecords("<.db file>")
    and/or
dbLoadTemplates("<.db file>,"<template_def>")
    . . .
iocInit (”<resource file>”)
```

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

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

The `dbLoad` loads the information obtained from the ASCII definition files. One or more `dbLoadRecords`, and `dbLoadTemplate` commands load database files with the GDCT `.db` format. It is also possible to use `dbLoad` to load databases saved in the DCT `.database` format. In this case it is possible to have multiple `dbLoad` comnmands as long as all the databases must have identical SDR information.

`iocInit` performs the following functions:

```
coreRelease
```

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

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.

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` as follows:

```
#define  INITHOOKatBeginning   0
#define  INITHOOKafterSetEnvParams   1
#define  INITHOOKafterGetResources   2
#define  INITHOOKafterLogInit   3
#define  INITHOOKafterCallbackInit   4
#define  INITHOOKafterCaLinkInit1   5
#define  INITHOOKafterInitDrvSup   6
#define  INITHOOKafterInitRecSup   7
#define  INITHOOKafterInitDevSup   8
#define  INITHOOKafterTS_init   9
#define  INITHOOKafterInitDatabase   10
#define  INITHOOKafterCaLinkInit2   11
#define  INITHOOKafterFinishDevSup   12
#define  INITHOOKafterScanInit   13
#define  INITHOOKafterInterruptAccept   14
#define  INITHOOKafterInitialProcess   15
#define  INITHOOKatEnd   16
```

The following is the `default initHooks.c` file. It merely declares the IOC as the master timing IOC.

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

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

void initHooks  (callNumber)
```

```
int   callNumber;
{
    switch (callNumber) {
    case INITHOOKatBeginning :
        break;
    case INITHOOKafterSetEnvParams :
        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;
}
```

dbLoad

This command is used to load the ASCII configuration converted to `default.dctsdr` format. In addition it can also load DCT `.database` files which contain the ASCII information as welll as record instances. Multiple `dbLoad` commands can be issued to load multiple database files. Each must have identical SDR records except, of course, the `SDR_DB_RECORDS`. The IOC needs the following self defining records:

- **SDR_DB_RECTYPE**: Record Types
- **SDR_DB_RECORDS**: Database records.
- **SDR_DB_RECDES**: The record and field descriptions.
- **SDR_CHOICEGBL**: The global choices.
- **SDR_CHOICECVT**: The conversion choices.
- **SDR_CHOICEREC**: The record type specific choices.
- **SDR_CHOICEDEV**: The device support choices.
- **SDR_DEVSUP**: The device support description structures.
- **SDR_CVTTABLE**: The conversion tables
- **SDR_DRVSUP**: The driver support structures.

- **SDR_RECSUP**: The record support structures

It is highly recommanded that `dbLoad` load only the `default.dctsdr` information and use `dbLoadRecords` and/or `dbLoadTemplate` to load record instances.

dbLoadRecords and dbLoadTemplates

These commands are used to load record instance definition that are in the GDCT `.db` format. The simplest usage is to use `dbLoadRecords` with a single argument, which is the name of the `.db` file. `dbLoadRecords` also provides a macro substitution capability. `dbLoadTemplate` accepts a `.db` file that is actually a template that can be instantiated multiple times. The full usage of these commands is described in man pages.

Print a message specifying the EPICS release.

Set Environment Variables

At one time a number of EPICS related environment variables were defined. Many of the values associated with the variables were also needed by the IOCs. Although another mechanism, not using environment variables, is now used on Unix, the values are still needed on the IOCs. These values are defined by a routine `epicsEnvParams`, which is stored in the `epics/share/site` directory. Note that the values defined by `epicsEnvParams` can be overridden by the resource definition file described in the next section.

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 you MUST set type so that it matches the actual type of the global variable because there is no way for `GetResources` to know the actual type.

One use of `GetResources` is to override the default values for the EPICS realted environment variables described in the previous section. Please note that all of these environment variables are of type `DBF_STIRING`.

The variables that are used in the IOC are:

- `EPICS_TS_MIN_WEST`: The number of minutes west of GMT. For example in U.S. Central time zone this is 360.
- `EPICS_IOC_LOG_INET`: The inet address of the log server.
- `EPICS_IOC_LOG_PORT`: The port used by the log server.

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`.

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.

Start Callback Tasks

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

Initialize Channel Access Links - Pass 1

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

Initialize Driver Support

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

Initialize Record Support

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

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.

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.

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

Initialize Channel Access Links - Pass 2

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

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.

Initialize Database Scanners

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

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.

Perform Initial Processing

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

Start Channel Access Server

The Channel Access server is started.

Chapter 12: 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.

The database attributes defined in this chapter are fixed, i.e. they are common to all IOC databases. They are defined via C include files. Any changes to these include files can affect many IOC software components, which will have to be modified and recompiled. A serious reader of this chapter should obtain a listing of all the files in `epics/share/epicsH`.

In the IOC a single global variable (`pdbBase`) contains the address of the `dbBase` structures that defines the run time database. The various structures mentioned in `dbBase` are described in this chapter. Any IOC source module using the macros and other routines mentioned in this chapter must include a definition:

```
extern dbBase *pdbBase;
```

Then a particular routine accessing a database structure can either reference it via `pdbBase` or create a local copy which must be initialized via `pdbBase`.

2. Macros for Accessing Database Structures

This section describes macros that make it easier to access the database structures.

Defined in dbRecType.h

GET_PRECNAME

```
GET_PRECNAME (prectype, rec_type)
```

Typical Usage:

```
char *pstr;
if(!(pstr=GET_PRECNAME(precType,type)) {/*action if not found*/}
```

This macro returns a pointer to the record name.

Defined in dbRecords.h

GET_PRECLOC

```
GET_PRECLOC (precheader, rec_type)
```

Typical Usage:

```
struct recLoc *precLoc;
if(!(precLoc=GET_PRECLOC(precHeader,type)) {/*action if not found*/}
```

This macro returns a pointer to the record location structure.

Defined in dbRecDes.h

GET_PFLDDES

```
GET_PFLDDES (prectypdes, ind_fld)
```

Typical Usage:

```
struct fldDes *pfldDes;
if(!(pfldDes=GET_PFLDDES(precTypDes,ind)) {/*action if not found*/}
```

This macro returns a pointer to the field description structure for a particular field of the record type defined by precTypDes.

GET_PRECTYPDES

```
GET_PRECTYPDES (precdes, ind_rec)
```

Typical Usage:

```
struct recTypDes *precTypDes;
if(!(precTypDes=GET_PRECTYPDES(precDes,ind)) {/*action if not found*/}
```

This macro returns a pointer to the record type description structure.

Defined in choice.h

GET_CHOICE

```
GET_CHOICE (pchoice_set, ind_choice)
```

Typical Usage:

```
char *pchoice;
if(!(pchoice=GET_CHOICE(pchoiceSet,ind)) {/*action if not found*/}
```

This macro returns a pointer to the string defining a particular choice.

GET_PCHOICE_SET

```
GET_PCHOICE_SET(parr_choice_set, ind_arr)
```

Typical Usage:

```
struct choiceSet *pchoiceSet;
if(!(pchoiceSet=GET_PCHOICE_SET(parrChoiceSet,ind)) {/*action if not found*/}
```

This macro returns a pointer to the structure defining a particular choice set.

GET_PARR_CHOICE_SET

```
GET_PARR_CHOICE_SET (pchoice_rec, ind_arr)
```

Typical Usage:

```
struct arrChoiceSet *parrChoiceSet;
if(!(parrChoiceSet=GET_PARR_CHOICE_SET(pChoiceRec,ind)) {/*action*/}
```

This macro returns a pointer to the structure defining an array of choice sets.

GET_DEV_CHOICE

```
GET_DEV_CHOICE (pdev_choice_set, ind_choice)
```

Typical Usage:

```
struct devChoice *pdevChoice;
if(!(pdevChoice=GET_DEV_CHOICE(pdevChoice,ind)) {/*action*/}
```

This macro returns a pointer to the structure defining a device choice.

GET_PDEV_CHOICE_SET

```
GET_PDEV_CHOICE_SET (pchoice_dev, ind_rec)
```

Typical Usage:

```
struct devChoiceSet *pdevChoiceSet;
if(!(pdevChoiceSet=GET_PDEV_CHOICE_SET(pchoiceDev,ind)) {/*action*/}
```

This macro returns a pointer to the structure defining the device choices for a particular record.

Defined in recSup.h

GET_PRSET

```
GET_PRSET (precsup, rec_type)
```

Typical Usage:

```
struct rset *prset;
if(!(prset=GET_PRSET(precSup,type)) {/*action if not found*/}
```

This macro returns a pointer to a record support entry table.

Defined in devSup.h

GET_PDSET

```
GET_PDSET (pdevsup,dtype)
```

Typical Usage:

```
struct dset *pdset;
```

```
if(!(pdset=GET_PDSET(pdevSup,type)) {/*action if not found*/}
```

This macro returns a pointer to a device support entry table.

GET_PDEVSUP

```
GET_PDEVSUP (precdevsup, rec_type)
```

Typical Usage:

```
struct devSup *pdevSup;
if(!(pdevSup=GET_PDEVSET(precDevSup,rectype)) {/*action if not found*/}
```

This macro returns a pointer to a structure defining the ndevice support entry tables for a particular record type.

Defined in drvSup.h

GET_PDRVET

```
GET_PDRVET (pdrvsup,type)
```

Typical Usage:

```
struct drvet *pdrvet;
if(!(pdrvet=GET_PDRVET(pdrvSup,type)) {/*action if not found*/}
```

This macro returns a pointer to a driver entry table.

GET_PDRVNAME

```
GET_PDRVNAME (pdrvsup,type)
```

Typical Usage:

```
char *pdrvName;
if(!(pdrvName=GET_PDRVNAME(pdrvSup,type)) {/*action if not found*/}
```

This macro returns a pointer to the driver name.

3. Database Structures

The following is a partial description of various database related structures. The associated include files are located in `epics/share/epicsH`. The include files themselves should be consulted for a complete description. Each file describing database structures contains the following:

- Structure definitions for the associated information.
- A brief description of the memory layout.
- A set of macros for accessing the structures.

For the purposes of understanding this document it is sufficient to show the structure declarations and the associated memory layout. If you are going to study program listings you should first study the complete include files. In particular become familiar with the macros which access the structures. The actual IOC code almost always uses macros to access the structures.

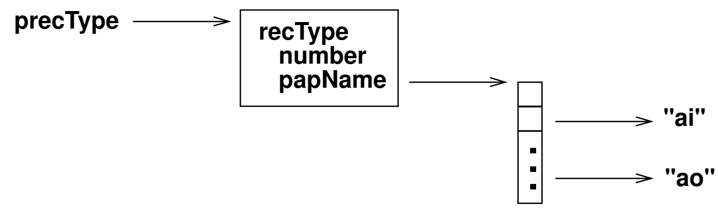


Figure 12-1: Record Types

dbRecType.h - Record Types

This file describes the possible record types. All include files which define structures containing components for multiple record types assume that the record type order is that specified by structue `recType`.

```
struct recType {
    long   number;    /* number of types   */
    char   **papName;   /* ptr to arr of ptr to name   */
};
```

Figure 12-1 shows the memory layout of the record type structures.

This is among the simpler of the IOC structures thus let's discuss a few details. The external variable `dbRecType` points to structure `recType`. This structure contains two elements: `number` and `papName`. `number` specifies the number of record types. `papName` is a pointer to an array of pointers to record names. Notice in the figure the unnamed array of pointers. It is permissible for any pointer in the array to be `NULL`.This type of structure will be seen many times in the following subsections. Thus whenever a variable starts with "`pap`" it means "pointer to array of pointers".

dbRecords.h - Record Locations

These structures describe the location of the actual database records.

```
typedef struct{
    ELLNODE   next;
    void   *precord;
} RECNODE;

struct recLoc{   /* record location */
    long   rec_size;   /* record size in bytes */
    long   record_type;   /* record type  */
    ELLLIST    *preclist   /* LIST head of sorted RECNODEs  */
};
struct recHeader{ /*record header*/
    long   number;   /*number of record types*/
    struct recLoc  **papRecLoc; /*ptr to arr of ptr to recLoc*/
};
```

Figure 12-2 shows the memory layout of the database records.

dbRecDes.h - Record Description

These structures describe each record type and each field of each record type.

```
/* conversion types*/
#define CT_DECIMAL 0
#define CT_HEX   1
/* lowfl, highfl */
#define CON   0
#define VAR   1
```

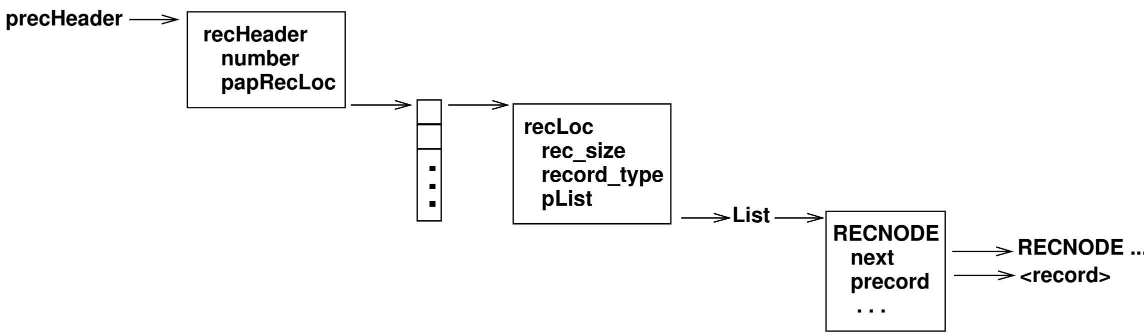


Figure 12-2: Database Records

```
#define PROMPT_SZ   24
union fld_types{
    char   char_value;
    unsigned   char   uchar_value;
    short   short_value;
    unsigned   short   ushort_value;
    long   long_value;
    unsigned long   ulong_value;
    float   float_value;
    double   double_value;
    unsigned short   enum_value;
};
struct range {
    long   fldnum;
    unionfld_typesvalue;
};
struct  fldDes{  /* field description */
    char   prompt[PROMPT_SZ]; /*Prompt string for DCT */
    char   fldname[FLDNAME_SZ];/*field name */
    short  offset;   /* Offset in bytes from beginning of record */
    short  size;   /* length in bytes of a field element */
    short  special;   /* Special processing requirements */
    short  field_type;   /* Field type as defined in dbFldTypes.h */
    short  process_passive;   /*should dbPutField process passive records*/
    short  choice_set;   /* index of choiceSet GBLCHOICE & RECCHOICE*/
    short  cvt_type;   /* Conversion type for DCT */
    short  promptflag;   /* Does DCT display this field */
    short  lowfl;   /* Is range1 CON or VAR */
    short  highfl;   /* Is range2 CON or VAR */
    short  interest;   /* interest level for reporting */
    union  fld_types  initial;   /* initial value */
    struct range  range1; /* Low value for field (Used by DCT) */
    struct range  range2; /* High value for field (Used by DCT) */
};
struct recTypDes{  /* record type description */
    short   rec_size;  /* size of the record  */
    short   no_fields;  /* number of fields defined */
    short   no_prompt;  /* number of fields to configure */
    short   no_links;  /* number of links */
    short   *link_ind;  /* addr of array of ind in apFldDes */
    unsigned long  *sortFldName;  /* addr of array of sorted fldname */
    short   *sortFldInd;  /* addr of array of ind in apFldDes */
    struct fldDes  **papFldDes;  /* ptr to array of ptr to fldDes */
};
struct   recDes{  /* record description */
    long  number;   /*number of recTypDes*/
    struct recTypDes **papRecTypDes;/*ptr to arr of ptr to recTypDes*/
};
extern struct recDes *dbRecDes;
```

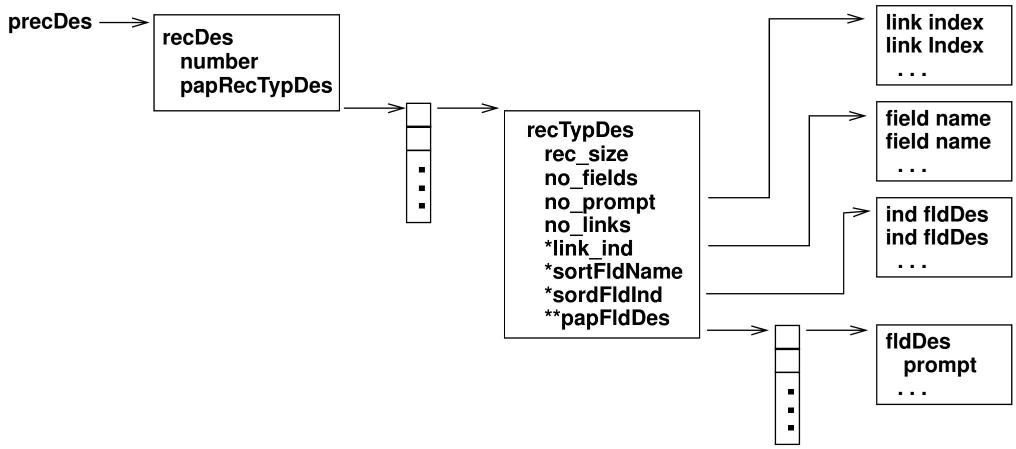


Figure 12-3: Record Descriptions

Figure 12-3 shows the memory layout of the record descriptions. Lets first discuss structure `recTypDes` and then `fldDes`.

Structure recTypDes describes the information specific to each record type. It contains the following information:

- **rec_size**: The size of a record in bytes.
- **no_fields**: The number of fields in a record.
- **no_prompt**: The number of prompt fields, i.e. fields that are configured via DCT.
- **no_links**: The number of link fields in a record.
- **link_in**: Address of an array of field indices for the link fields.
- **sortFldName**: Address of an array of sorted field names. Note that the array is an array of long words. Thus the field names must be a maximum of 4 characters in length.
- **sortFldInd**: Addrsss of an array of indexes into `apFldDes` for the sorted field names. Thus when a sorted field name is located the field description can be located via this array.
- **papFldDes**: Pointer to an array of pointers to field descriptions.

Structure `fldDes` contains a complete description of a field. The database access routines and various utilities such as DCT use these definitions to access the database (the fields common to all record types are also used). Thus, with the exception of record and device support routines, the software has no knowledge of particular record types. This makes it possible to add new record and device support and/or modify existing support without affecting most of the IOC and utility software. Lets discuss each field attribute:

- **prompt**: This is the prompt string used by DCT.
- **fldname**: The field name.
- **offset**: Offset in bytes from the beginning of the record.
- **size**: Size of the field in bytes.
- **special**: Is special processing required when field value is changed. This is discussed in detail below,
- **field_type**: The field type, i.e. `DBF_xxx`.
- **process_passive**: This field determines in `dbPutField` requests to this field will cause passive records to be processed.

- **choice_set**: This is used by `DBF_GBLCHOICE` and `DBF_RECCHOICE` field types to specify the associated choice set.
- **cvt_type**: This is used by DCT to display and decode field values. It applys only to the field types `DBF_UCHAR` through `DBF_ULONG`. It must have the value `CT_DECIMAL` or `CT_HEX`.
- **promptflag**: Is this a field the user can configure via DCT?
- **lowfl**: Specifies if `range1` is constant or a variable. Must have the value `CON` or `VAR`.
- **highfl**: Similar to `lowfl` except for `range2`.
- **interest**: Interest level for this field. This is used for reporting purposes. For example `dbpr` honors this field. The lower the value the higher the interest level.
- **initial**: Initial value for field.
- **range1**: Low value for field. If `lowfl =VAR` then this must specify another field in the same record.
- **range2**: High value for field. Similar to `range1`. Note that the field value must lie in the range “`range1`<=value<=`range2`”.

choice.h - Choice Definitions

```
struct choiceSet { /* This defines one set of choices*/
    long   number;   /*number of choices */
    char   **papChoice;   /*ptr to arr of ptr to choice string   */
    };
struct arrChoiceSet{ /*An array of choice sets for particular record type*/
    long   number;   /*number of choice sets */
    struct choiceSet **papChoiceSet;  /*ptr to arr of ptr to choiceSet*/
    };
struct choiceRec{ /*define choices for each record type*/
    long   number;   /*number of arrChoiceSet */
    struct arrChoiceSet **papArrChoiceSet;  /*ptr to arr of ptr to arrChoiceSet*/
    };

/* device choices */
struct devChoice{
    long   link_type;  /*link type for this device*/
    char   *pchoice;   /*ptr to choice string */
    };
struct devChoiceSet { /* This defines one set of device choices*/
    long   number;  /*number of choices */
    struct devChoice **papDevChoice; /*ptr to arr of ptr to devChoice */
    char   **papChoice;   /*ptr to arr of ptr to choice string */
    };
struct devChoiceRec{ /*define device choices for each record type*/
    long  number;   /*number of devChoiceSet  */
    struct devChoiceSet **papDevChoiceSet; /*ptr to arr of ptr to devChoiceSet*/
    };
```

Figure 12-4 shows the memory layout of the choice definitions. In the database a choice field is stored as an unsigned short value. The meaning is determined via the associated choice structures. Four types of structures are referenced via the following pointers (stored in `struct dbBase`):

- **pchoiceCvt**: Specifies a conversion choice.
- **pchoiceGbl**: Specifies a member of a set of global choices. These are choices that are common to all record types.
- **pchoiceRec**: Specifies a member of a set of record choices. These are choices that are special to the particular record type.
- **pchoiceDev**: Specifies a device choice, i.e. the set of device support routines for this record. Each record type has its own sets of device support routines.

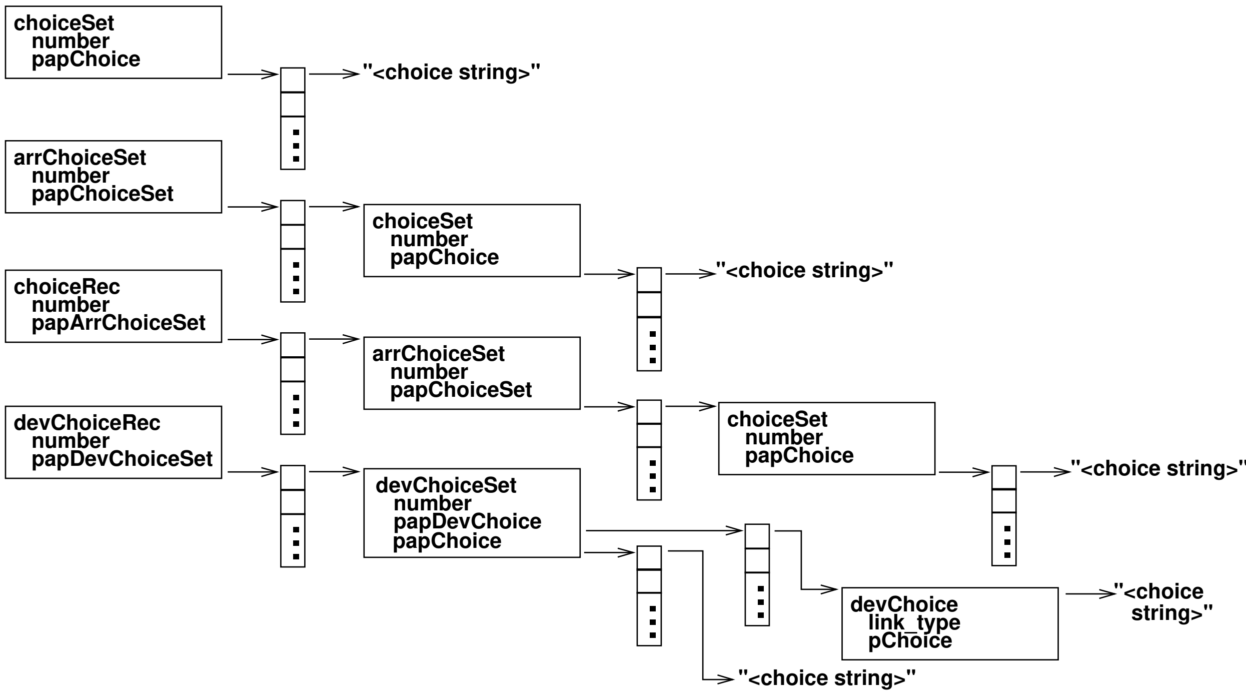


Figure 12-4: Choice Definitions

cvtTable.h

```
struct brkInt{ /* breakpoint interval */
    long   raw;    /*raw value for beginning of interval */
    float  slope;   /*slope for interval */
    float  eng;   /*converted value for beginning of interval*/
    };
struct brkTable { /* breakpoint table */
    char   *name;   /*breakpoint table name */
    long   number;   /*number of brkInt in this table */
    long   rawLow;   /*lowest raw data value allowed */
    long   rawHigh;   /*highest raw data value allowed */
    struct brkInt  **papBrkInt;  /* ptr to array of ptr to brkInt */
    };
struct arrBrkTable { /* array of brkTable */
    long   number;   /*number of break tables */
    struct brkTable  **papBrkTable;/* ptr to array of ptr to brkTable */
    };
```

recSup.h - Record Support

```
typedef long (*RECSUPFUN) ();  /* ptr to record support function */
struct rset {  /* record support entry table */
    long   number;  /*number of support routines */
    RECSUPFUN   report;  /*print report */
    RECSUPFUN   init;  /*init support */
    RECSUPFUN   init_record;  /*init record */
    RECSUPFUN   process;  /*process record  */
    RECSUPFUN   special;  /*special processing */
    RECSUPFUN   get_value;  /*get value field */
    RECSUPFUN   cvt_dbaddr;  /*cvt  dbAddr */
    RECSUPFUN   get_array_info;
```

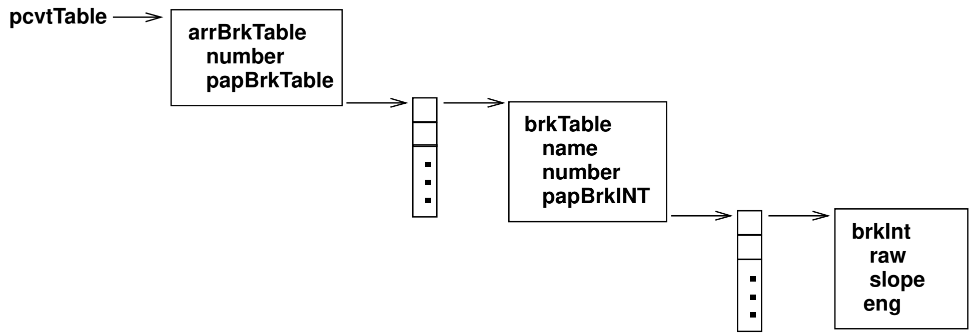


Figure 12-5: Conversion Tables

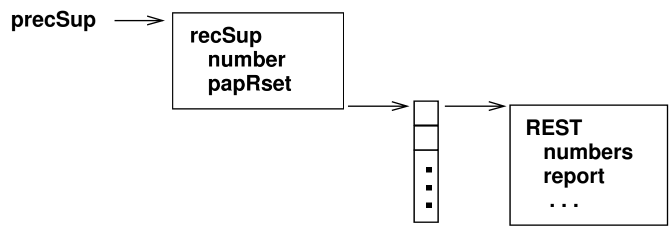


Figure 12-6: Record Support

```
    RECSUPFUN   get_enum_str;   /*get string from enum item */
    RECSUPFUN   get_enum_strs;   /*get all enum strings */
    RECSUPFUN   put_enum_str;   /*put string to enum item */
    RECSUPFUN   get_graphic_double;
    RECSUPFUN   get_control_double;
    RECSUPFUN   get_alarm_double;
    };
struct recSup {
    long   number;  /*number of record types */
    struct rset  **papRset;  /*ptr to arr of ptr to rset */
    };
#define RSETNUMBER ( (sizeof(struct rset) - sizeof(long))/sizeof(RECSUPFUN) )
```

Figure 12-6 shows the memory layout of the record support definitions. Each record type must have an associated set of record support routines. Note that only the record and device support routines use the record structure declarations while accessing a record. The record support routines are intended to isolate the rest of the IOC software from details of record access and processing.

devSup.h - Device Support

```
typedef long (*DEVSUPFUN) ();  /* ptr to device support function*/
struct dset {  /* device support entry table */
    long   number;   /*number of support routines*/
    DEVSUPFUN  report;  /*print report*/
    DEVSUPFUN  init;  /*init support*/
    DEVSUPFUN  init_record;  /*init support for particular record*/
    DEVSUPFUN  get_ioint_info;  /*get I/O interrupt information*/
    /*other functions are record dependent*/
    };
struct devSup {
    long   number;  /*number of dset */
```

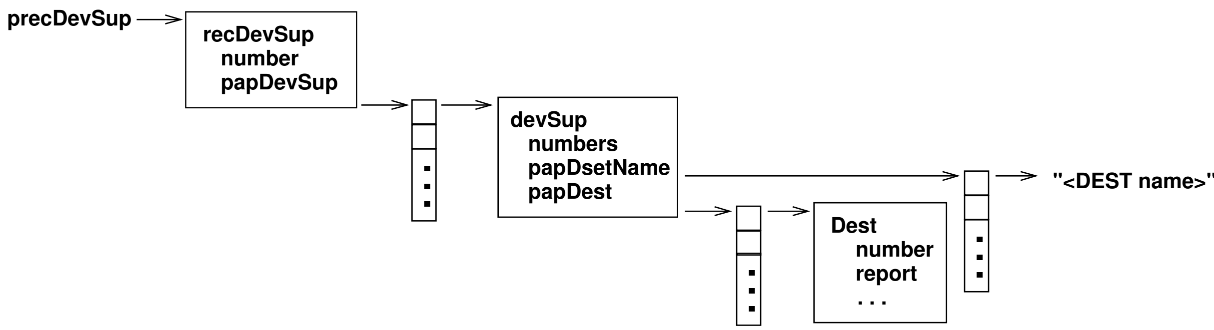


Figure 12-7: Device Support

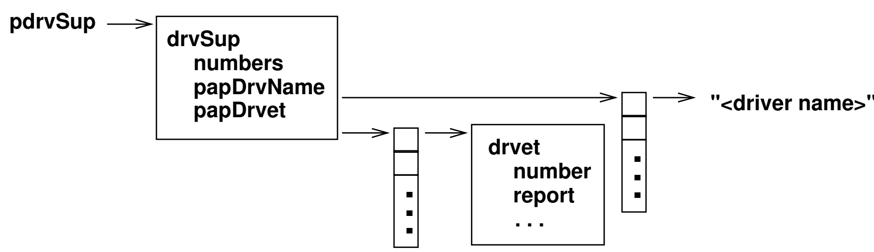


Figure 12-8: Driver Support

```
    char   **papDsetName;  /*ptr of arr of ptr to name */
    struct dset **papDset;  /*ptr to arr of ptr to dset */
    };
struct recDevSup {
    long   number;  /*number of record types */
    struct devSup **papDevSup; /*ptr to arr of ptr to devSup */
    };
extern struct recDevSup *devSup;
```

Figure 12-7 shows the memory layout of the device support definitions. The device support routines are intended to isolate the record processing routines from device specific details.

drvSup.h - Driver Support

```
typedef int (*DRVSUPFUN) ();/* ptr to driver support function*/
struct drvet {   /* driver entry table */
    long  number;   /*number of support routines*/
    DRVSUPFUN   report;   /*print report*/
    DRVSUPFUN   init;   /*init support*/
    DERSUPFUN   reboot;  /*reboot entry*/
    /* other functions are device dependent*/
    };
struct drvSup {
    long   number;   /*number of dset */
    char   **papDrvName;   /*ptr to arr of ptr to drvetName*/
    struct drvet  **papDrvet;   /*ptr to arr ptr to drvet */
    };
#define DRVETNUMBER ( (sizeof(struct drvet) -sizeof(long))/sizeof(DRVSUPFUN) )
```

Figure 12-8 shows the memory layout of the driver support definitions.

Index of Functions and Commands

E

G

I

L

M

O

P

R

S

T

V

W