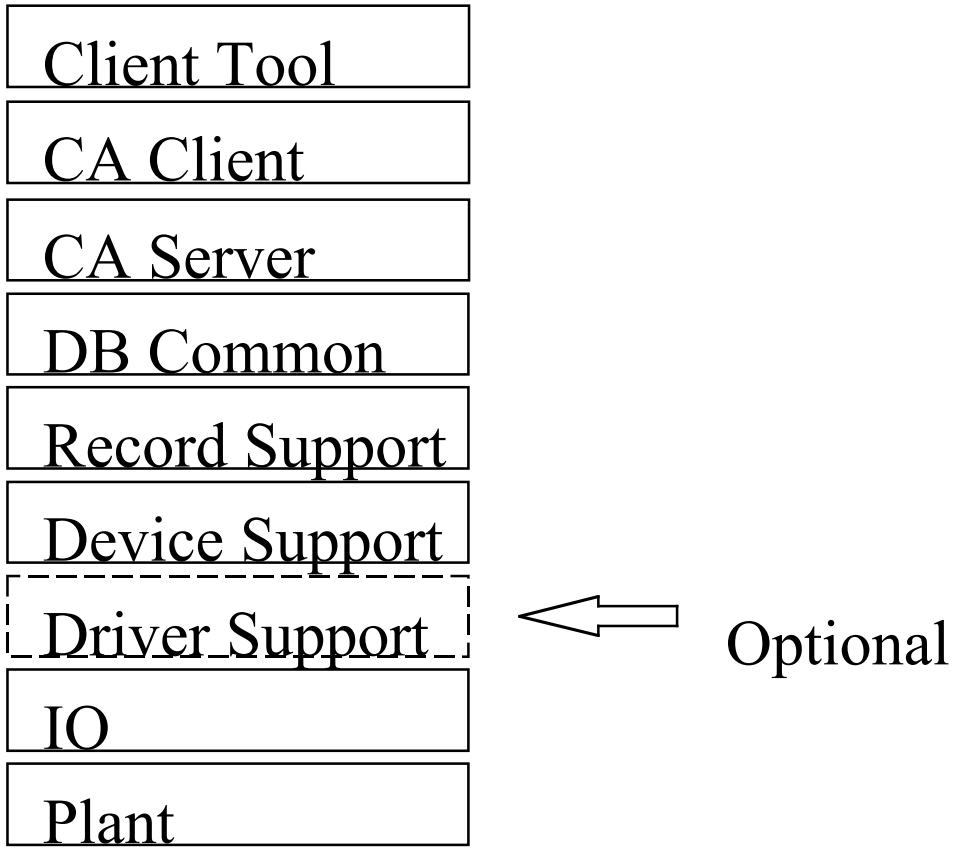
EPICS Record/Device/Driver Support Interfaces

Jeff Hill

Outline

- EPICS Software Architecture Review
- record support interface
- device support interface
- driver support interface
- EPICS status codes
- EPICS IO address formats
- installation

EPICS Software Architecture



Record Support

- provides implementation for a record type
- plugs into “database common” environment
- must prepare a record definition file which defines the data structure - xxx.dbd
- must supply a C source file which provides the execution semantics
- many examples in $(EPICS)/base/src/rec

Record Description File - xxx.dbd

```
recordtype(xxx) {
   include "dbCommon.dbd"
   field(PREC, DBF_DOUBLE) {
        prompt("Display Precision")
        promtGroup(GUI_DISPLAY)
   }
   .
   .
   .
}
```

Record Support Routines

provide execution semantics

- initialize record instance
- record processing
- special record processing
- convert database address
- initialize record type
- get array info
- set array info
- get units
- get precision
- get string for field's enumerated value
- set enumerated field's value using string
- graphic/control/alarm limits

Initialize Record Support Type

```
typedef long recInit_t ();
```

- record type specific initialization
- called once for each record type in the system

Initialize Record Instance

```
typedef long recInitInstance_t
  (void *precord, int pass);
```

- two pass record instance initialization
- initializations private to this record performed when pass==0
- initializations referring to other records performed when pass==1

Record Processing

```
typedef long recProcess_t
                    (void *precord);
```

- record personality implemented here
- see the application developers guide for the details

Before Record Processing

- decision to process a record
- check for record active (pact TRUE)
- verify that the record isn't disabled

Record Processing Routine

- 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

Asynchronous Record Processing

block scan tasks for slow devices?

not!

- initiate the I/O operation and set pact TRUE
- return immediately without completing record processing
- when I/O operation completes call record processing again
- set pact FALSE inside record processing

Special Record Processing

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

- the record description tags fields with these attributes
- the file “special.h” defines special field attributes
- special attribute >= 100 then this routine called before/after modifying the field

Convert Database Address

```
typedef long recCvtDbaddr_t
  (struct dbAddr *paddr);
```

- supply this routine only if the field is not stored within the record data structure
- Example: fields that are arrays

Get Array Info

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

- arrays are variable sized up to a maximum specified by “recCvtDbAddr()”
- this routine identifies the current length
- offset!=0 is only used for ring buffers

Put Array Info

```
typedef long recPutArrayInfo_t(
  struct dbAddr *paddr,
  long nNew);
```

- called when the array length is modified by dbCommon
- Ex: client (or another record) writes array

Get Units

```
typedef long recGetUnits_t (
  struct dbAddr *paddr,
  char units[8]);
```

Get Precision

```
typedef long recGetPrecision (
  struct dbAddr *paddr,
  long *precision);
```

- suggested number of digits used to display the field's analog value

Get String Corresponding to Field's Enumerated Value

```
typedef long recGetEnumStr_t (
  struct dbAddr *paddr,
  char *pbuffer);
```

- special string conversion
- used to associated a string with a binary state

Get String Table for Field With Enumerated Value

```
typedef long recGetEnumStrs_t (
  struct dbAddr *paddr,
  struct dbr_enumStrs *p);
```

- used to obtain a table of strings for all states

Set Enumerated Value With String

```
typedef long recPutEnumStr_t (
  struct dbAddr *paddr,
  const char *pbuffer);
```

- sets the current state using a string

Graphic/Control/Alarm Limits

```
typedef long recGetGraphicDouble_t (
   struct dbAddr *paddr,
   struct dbr_grDouble *p);
typedef long recGetControlDouble_t (
   struct dbAddr *paddr,
   struct dbr_ctrlDouble *p);
typedef long recGetAlarmDouble_t (
   struct dbAddr *paddr,
struct dbr_alDouble *p);
```

- called to obtain the analog limits

Record Support Entry Table

```
struct rset aiRSET={
  RSETNUMBER, report,
  initialize, initInstance,
  process, special,
  getValue,cvtDBAaddr,
  getArrayInfo, putArrayInfo,
  getUnits, getPrecision,
  getEnumStr, getEnumStrTbl,
  putEnumStr, getGraphicDouble,
  getControlDouble, getAlarmDouble};
```

- set entry to NULL if default action OK

Device Support

- interface between device driver and record
- intimate knowledge of record(s)

Device Support Routines (Analog Input Record)

- initialization
- report
- initialize instance
- read ai device value
- linear convert
- attach to device interrupt

Device Support Header Files

```
#include <aiRecord.h>
#include <devSup.h>
#include <dbScan.h>
```

AI Device Support Type Initialization

```
long aiDevInit (unsigned pass)
```

- device type specific initialization
- common to all record types
- called, pass = 0, prior to initializing each record during "iocInit()"
- called, pass = 1, after initializing each record during "iocInit()"

AI Device Report

```
long aiDevReport (struct
  aiRecord * pai, int level);
```

- common to all record types
- called once for every record instance when the user types "dbior <level>"
- device status to "stdout" from this routine
- detail increases with increasing "level"

AI Device Initialization for Record

```
long aiDevInitInstance(struct aiRecord
  *pai)
```

- called from within “iocInit()”
- once for each rec inst attached to device

```
pPriv = (struct XXXX *) calloc(1u, sizeof(struct
  XXXX);
pai->dpvt = (void *) pPriv;
```

- the device address (pai->inp) is normally verified here

Device Interrupt Causes Record to be Processed

- device supports “scan on interrupt”
- higher scan rate
- scan synchronized with device
- device scans IO - change of state results in record processing

Each Interrupt Source

- prior to enabling and attaching to interrupt
- before returning from aiDevGetIoIntInfo()

```
scanIoInit(&pXXXX->ioScanPvt);
```

- number of "`ioScanPvt`" allocated depends on the interrupt granularity of device
- one interrupt for all signals
- or one for each signal

Each Interrupt Occurrence

```
scanIoRequest(pXXXX->ioScanPvt);
```

- safe to call scanIoRequest() from ISR
- don't call scanIoRequest() until after database init ("iocInit()") completes

AI Device Get IO Interrupt Info

```
long aiDevGetIoIntInfo (
   int cmd,
   struct aiRecord *pai,
   IOSCANPVT *ppvt);
```

- associates interrupt source with record

```
*ppvt = pXXX->ioScanPvt;
```

- cmd==0 - insert into IO interrupt scan
- cmd==1 - remove from IO Interrupt scan

Read Analog Device Value

```
long aiDevRead_(struct aiRecord * pai) {
  long rval;
  if (device OK) then
      rval=pDevMemoryMap->
            aiRegister[pai->dpvt->signal];
      pai->rval = rval;
  else
      recGblSetSevr(pai,
            READ_ALARM, INVALID_ALARM);
  endif
}
```

AI Device Linear Convert

```
long aiDevLinearConv (
  struct aiRecord *pai, int after);
```

- Setup the slope (and any offset) for the conversion to engineering units

```
if (!after) then
  return S_XXXX_OK;
endif
/* A 12 bit DAC is assumed here */
pai->eslo = (pai->eguf - pai->egul)/0xfff;
pai->roff = ????;
```

From `convert()` in aiRecord.c

```
double val;
val = pai->rval + pai->roff;
/*
 *  adjust with slope/offset
 *  if linear convert is used
 */
if(pai->aslo!=0.0) val*=aslo;
if(pai->aoff!=0.0) val+=aoff;
val = (val * pai->eslo)
  + pai->egul;
```

Asynchronous Devices

- read/write routine sets “PACT” true and returns zero for success.
- asynchronous IO completion call-back completes record processing
- don’t process a record from within an ISR

Example Asynchronous Read

```
long devXxxRead (struct aiRecord *pai) {
   if (pai->pact) [
        return S_devXxx_OK;     /* zero */
   }
   pai->pact = TRUE
   devXxxBeginAsyncIO(pai->dpvt);
   return S_devXxx_OK;
}
```

Example Asynchronous Read Completion

```
void devXxxAsyncIOCompletion(struct aiRecord *pai, long ioStatus)
{
   struct rset *prset = (struct rset *) pai->rset;

   dbScanLock(pai);
   if (ioStatus != S_devXxx_OK) {
        recGblSetSevr(pai, READ_ALARM, INVALID_ALARM);
   }
   (*prset->process)(pai);
   dbScanUnlock(pai);
}
```

Device Support Entry Table

```
struct ai_dev_sup
{
   long                 number; /* number of items in table */
   aiDevReport_t        *report;
   devInit_t            *init;
   aiDevInitRec_t       *initRec;
   aiDevGetIoIntInfo_t  *getIoIntInfo;
   aiDevRead_t          *read;
   aiDevLinearConv_t    *specialLinearConv;
};
LOCAL devInit_t                 devInitXXXX;
LOCAL aiDevInitRec_t            aiDevInitRecXXXX;
LOCAL aiDevGetIoIntInfo_t       aiDevGetIoIntInfoXXXX;
LOCAL aiDevRead_t               aiDevReadXXXX;
LOCAL aiDevLinearConv_t         aiDevLinearConvertXXXX;
struct ai_dev_sup devAiXXXX = { 6L, /* external scope */
   aiDevReportXXXX,
   aiDevInitXXXX,
   aiDevInitRecXXXX,
   aiDevGetIoIntInfoXXXX,
   aiDevReadXXXX,
   aiDevLinearConvertXXXX
};
```

Driver Support (Optional)

- communication hardware initializes prior to its being used by device support
- ex: VXI resource manager, MXI, GPIB, bit bus, AB DF1, CAMAC, CAN, bus repeaters
- when you want to limit a device driver's knowledge of EPICS

Driver Support Routines

- initialize
- report

Driver Initialize

```
long    drvInitFunc();
```

- device driver specific initialization here
- this routine is called once by iocInit() prior to database initialization

Driver Report

```
long    drvReportFunc(int level);
```

- called when the user types "dbior <level>"
- provides device status to stdout
- increasing details with increasing "level"
- raw addresses, raw signal values, and device status
- used by persons maintaining the hardware

Driver Support Entry Table

```
#include <drvSup.h>
typedef long    drvInitFunc_t (void);
typedef long    drvReportFunc_t (int level);
LOCAL drvReportFunc_t drvXXXXReport;
LOCAL drvInitFunc_t drvXXXXInit;
struct drvSupEntryTable {
   long              number;
   drvReportFunc_t   *report;
   drvInitFunc_t     *init;
}drvXXXX =
{  2L,         /* the number of functions */
   drvReportXXXX,
    drvInitXXXX
};
```

EPICS “Built-In” IO address Formats

GPIB	link	station	parameter			
BITBUS	link	node	port	signal	parameter	
VME	card	signal	parameter			
CAMAC	branch	crate	slot	addr	func	parameter
VXI	flag	frame	slot	la	signal	parameter
INST	parameter					

IO Type Isn't "Built-In"

```
/* from $EPICS/base/include */
#include <link.h>
```

- use "INST_IO" addressing
- parse parameter string in device support

Record Support Installation

- add to db description “tnInclude.dbd”

```
include “xxxrecord.dbd”
```

Device Support Installation

- add to the db description “xyzInclude.dbd”

```
device (xxx, INST_IO, devXxxYyy, “yyy device”)
```

- `xxx` - record type
- `INST_IO` - IO addressing type
- `devXxxYyy` - device support entry table name
- `“yyy device”` - configuration menu label for DTYP field
- load object modules prior to "iocInit()"
- optionally install error codes

Driver Support Installation

- add to db description “tnInclude.dbd”

```
driver (drvXxx)
```

- `drvXxx` - driver support entry table name
- load object modules prior to "iocInit()"
- optionally install your error codes

Record Instance

```
record (ai, ”myRecordName") {
  field(DTYP, ”yyy device")
  field(SCAN, "I/O Intr")
  field(INP, "@yyy 3")
  field(LINR, "LINEAR")
  field(EGUF, ”0")
  field(EGUL,"-10")
  field(HOPR, "10")
  field(LOPR, "0")
}
```

EPICS Status Codes (Optional)

- define a subsystem code and error constants in your subsystem's header file

```
#define M_xxxx (520 <<16) /*VXI Driver*/
#define S_XXX_bad (M_xxxx| 1) /* a really bad failure*/
#define S_XXX_worse (M_xxxx| 2) /* even worse */
```

- zero is always a success code

```
#define S_XXX_OK 0
```

- -1 is always a failure code

Install ERROR Codes (Optional)

- Define error codes in sub-systems header file
- Add your sub-system's header file to "ERR_S_FILES in config/CONFIG_SITE"

```
ERR_S_FILES += <path>/drvXXX.h
```

- Type "gmake" in "$EPICS/base"

More Information

- many examples in $(EPICS)/base/src/rec
- many examples in $(EPICS)/base/src/dev
- many examples in $(EPICS)/base/src/drv
- “Application developers Guide” by Marty Kraimer (on the WEB)
- “Record Reference Manual” by Marty Kraimer (on the WEB)

EPICS Device Support Installation Hands On

Jeff Hill

Outline

- Examine example device support
  - specific to analog input record
  - source code: xxxApp/src/devAiXxx.c
  - Emphasis on the EPICS interfaces
  - Instrument IO addressing (INST_IO)
    - format: “xxx <signal number>”
- Walk through installation steps
- Verify normal and off-normal operation

Build System Installation

- Install source file into build system
  - Modify xxxApp/src/Makefile.Vx
    - SRCS.c += ../ **devAiXxx.c**
      - create devAiXxx.o from devAiXxx.c
    - LIBOBJS += **devAiXxx.o**
      - add devAiXxx.o to xxxLib

Database Description Installation

- Modify xxxInclude.dbd data base description
  - device (**ai**, **INST_IO**, **devAiXxx**, "**device x**")
- uncomment record include in base.dbd as required (as required)
  - #include "aiRecord.dbd"

Build the application

- in xxxApp/src
- type “gmake”
- verify source code rebuild
- verify data base description rebuild

Create Record Instance With Sophisticated Tool - Text Editor

- Create file xxxApp/Db/myRecInstance.db

```
record (ai, ”myRecordName") {
      field(DTYP, ”device x")
      field(SCAN, "I/O Intr")
      field(INP, “@xxx 3")
      field(LINR, "LINEAR")
      field(EGUF, "10")
      field(EGUL,"-10")
      field(HOPR, "10")
      field(LOPR, "0")
}
```

Modify the startup script

- Load the database before “iocInit” is called
  - cd “xxxApp/Db/”
  - dbLoadRecords (“myRecInstance.db”)
- Load the object code - devAiXxx.o before “iocInit” is called
  - in application object library - xxxLib
    - ld < xxxLib

Verify normal and off-normal operation

- reboot IOC
- watch for messages occurring during iocInit
- verify that the record is in UDF alarm
  - dbpr “myRecordName”, nnn
  - look at fields “STAT”and “SEVR”

Verify normal and off-normal operation

- cause a scan on interrupt event
  - aiDevXxxUpdate (unsigned signalNo, boolean deviceOk, unsigned value)
  - post new ai device value
    - verify that the record isn't in alarm state
    - look at STAT, SEVR. VAL fields
  - place ai device in invalid state
    - verify that the record is in alarm state
    - look at STAT, SEVR fields