Introduction to Device Support

Eric Norum

Writing Device Support – Scope

- An overview of the concepts associated with writing EPICS Device Support routines.
- Examples show the “stone knives and bearskins” approach.
- The ASYN package provides a framework which makes writing device support much easier.
  - The concepts presented here still apply.

Writing Device Support – Outline

- What is ‘Device Support’?
- The .dbd file entry
- The driver DSET
- Device addresses
- Support routines
- Using interrupts
- Asynchronous input/output
- Callbacks

What is ‘Device Support’?

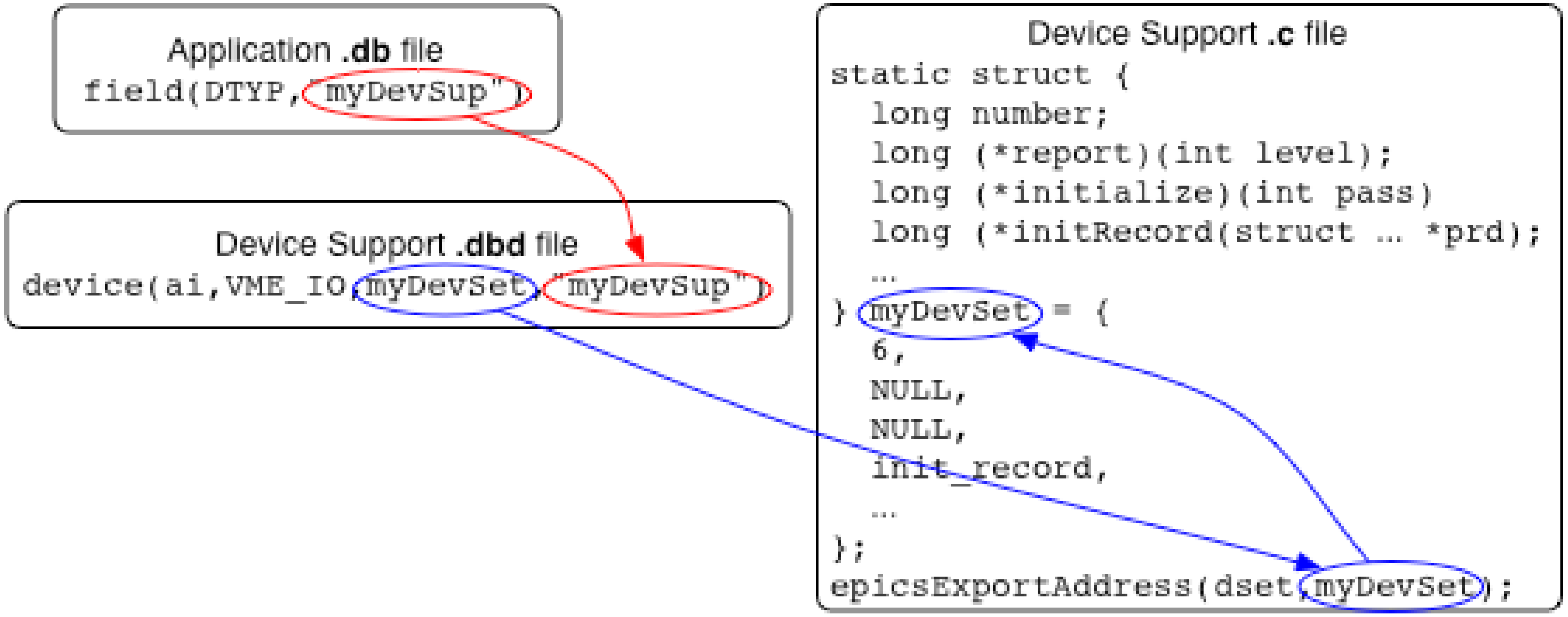
- Interface between record and hardware
- A set of routines for record support to call
  - The record type determines the required set of routines
  - These routines have full read/write access to any record field
- Determines synchronous/asynchronous nature of record
- Performs record I/O
  - Provides interrupt handling mechanism

Why use device support?

- Could instead make a different record type for each hardware interface, with fields to allow full control over the provided facilities.
- A separate device support level provides several advantages:
  - Users need not learn a new record type for each type of device
  - Increases modularity
    - I/O hardware changes are less disruptive
    - Device support is simpler than record support
    - Hardware interface code is isolated from record API
- Custom records are available if really needed.
  - By which I mean "really, really, really needed!"
  - **Existing record types are sufficient for most applications.**

How does a record find its device support?

Through .dbd ‘device’ statements:



```
field(DTYP,"myDevSup")
```

```
device(ai,VME_IO,myDevSet,"myDevSup")
```

```
static struct {
   long number;
   long (*report)(int level);
   long (*initialize)(int pass)
   long (*initRecord(struct … *prd);
   …
} myDevSet = {
   6,
   NULL,
   NULL,
   init_record,
   …
};
epicsExportAddress(dset,myDevSet);
```

The .dbd file entry

- The IOC discovers device support from entries in .dbd files

  `device(`*`recType`*`,`*`addrType`*`,`*`dsetName`*`,"dtypeName")`

- *`addrType`* is one of

AB_IO	BITBUS_IO	CAMAC_IO	GPIB_IO
INST_IO	RF_IO	VME_IO	VXI_IO

- *dsetName* is the name of the C Device Support Entry Table (DSET)
  - By convention name indicates record and hardware type:

```
device(ai, GPIB_IO, devAidg535, "dg535")
device(bi, VME_IO, devBiXy240, "XYCOM-240")
```

The DSET

- A C structure containing pointers to functions
- Content dependent upon record type
- Each device support layer defines a DSET with pointers to its own functions
- A DSET structure declaration looks like:

```
struct dset {
  long number;
  long (*report)(int level);
  long (*initialize)(int pass);
  long (*initRecord)(struct … *precord);
  long (*getIoIntInfo)(…);
  … read/write and other routines as required
};
```

- **`number`** specifies number of pointers (often 5 or 6)
- A NULL is given when an optional routine is not implemented
- DSET structures and functions are usually declared **`static`**

The DSET – initialize

`long initialize(int pass);`

- Initializes the device support layer
- Optional routine, not always needed
- Used for one-time startup operations:
  - Start background tasks
  - Create shared tables
- Called twice by iocInit()
  - `pass=0` – Before any record initialization
    Doesn't usually access hardware since device address information is not yet known
  - `pass=1` – After all record initialization
    Can be used as a final startup step. All device address information is now known

The DSET – initRecord

`long initRecord(struct … *precord);`

- Called by iocInit() once for each record with matching DTYP
- Optional routine, but usually supplied
- Routines often
  - Validate the INP or OUTP field
  - Verify that addressed hardware is present
  - Allocate device-specific storage for the record
    - Each record contains a `void *dpvt` pointer for this purpose
  - Program device registers
  - Set record-specific fields needed for conversion to/from engineering units

The DSET – read/write

```
long read(struct … *precord);
long write(struct … *precord);
```

- Called when record is processed
- Perform (or initiate) the I/O operation:
  - Synchronous input
    - Copy value from hardware into `precord->rval`
    - Return 0 (to indicate success)
  - Synchronous output
    - Copy value from `precord->rval` to hardware
    - Return 0 (to indicate success)

The DSET – initRecord – Device Addresses

- Device support .dbd entry was

  `device(`*`recType`*`,`*`addrType`*`,`*`dset`*`,"`*`name`*`")`
- *addrType* specifies the type to use for the address link, e.g.

  `device(bo,VME_IO,devBoXy240,"Xycom XY240")`

  sets `pbo->out`:
  - `pbo->out.type = VME_IO`
  - Device support uses `pbo->out.value.vmeio` which is a

```
struct vmeio {
    short card;
    short signal;
    char *parm;
};
```

- IOC Application Developer’s Guide describes all types

A simple example (vxWorks or RTEMS)

```
#include <recGbl.h>
#include <devSup.h>
#include <devLib.h>
#include <biRecord.h>
#include <epicsExport.h>
static long initRecord(struct biRecord *prec){
   char *pbyte, dummy;
   if ((prec->inp.type != VME_IO) ||
       (prec->inp.value.vmeio.signal < 0) || (prec->inp.value.vmeio.signal > 7)) {
          recGblRecordError(S_dev_badInpType, (void *)prec, "devBiFirst: Bad INP");
          return -1;
   }
   if (devRegisterAddress("devBiFirst", atVMEA16, prec->inp.value.vmeio.card, 0x1,
                                                                           &pbyte) != 0) {
          recGblRecordError(S_dev_badCard, (void *)prec, "devBiFirst: Bad VME
   address");
          return -1;
   }
   if (devReadProbe(1, pbyte, &dummy) < 0) {
          recGblRecordError(S_dev_badCard, (void *)prec, "devBiFirst: Nothing there!");
          return -1;
   }
   prec->dpvt = pbyte;
   prec->mask = 1 << prec->inp.value.vmeio.signal;
   return 0;
}
```

A simple example (vxWorks or RTEMS)

```
static long read(struct biRecord *prec)
{
    volatile char *pbyte = (volatile char *)prec->dpvt;

    prec->rval = *pbyte;
    return 0;
}

static struct {
    long number;
    long (*report)(int);
    long (*initialize)(int);
    long (*initRecord)(struct biRecord *);
    long (*getIoIntInfo)();
    long (*read)(struct biRecord *);
} devBiFirst = {
    5, NULL, NULL, initRecord, NULL, read
};
epicsExportAddress(dset,devBiFirst);
```

The DSET – report

```
long report(int level);
```

- Called by **`dbior`** shell command
- Prints information about current state, hardware status, I/O statistics, etc.
- Amount of output is controlled by the level argument
  - `level=0` – list hardware connected, one device per line
  - `level>0` – provide different type or more detailed information

A simple example – device support .dbd file

The .dbd file for the device support routines shown on the preceding pages might be

```
device(bi, VME_IO, devBiFirst, “simpleInput”)
```

A simple example – application .db file

An application .db file using the device support routines shown on the preceding pages might contain

```
record(bi, "$(P):statusBit")
{
    field(DESC, "Simple example binary input")
    field(DTYP, "simpleInput")
    field(INP, "#C$(C) S$(S)")
}
```

A simple example – application startup script

An application startup script (st.cmd) using the device support routines shown on the preceding pages might contain

```
dbLoadRecords("db/example.db","P=test,C=0x1E0,S=0")
```

which would expand the .db file into

```
record(bi, "test:statusBit")
{
    field(DESC, "Simple example binary input")
    field(DTYP, "simpleInput")
    field(INP, "#C0x1E0 S0")
}
```

Useful facilities

- ANSI C routines (EPICS headers fill in vendor holes)
  - epicsStdio.h – printf, sscanf, epicsSnprintf
  - epicsString.h – strcpy, memcpy, epicsStrDup
  - epicsStdlib.h – getenv, abs, epicsScanDouble
- OS-independent hardware access (devLib.h)
  - Bus address → Local address conversion
  - Interrupt control
  - Bus probing
- EPICS routines
  - epicsEvent.h – process synchronization semaphore
  - epicsMutex.h – mutual-exclusion semaphore
  - epicsThread.h – multithreading support
  - recGbl.h – record error and alarm reporting

Device interrupts

- vxWorks/RTEMS interrupt handlers can be written in C
- VME interrupts have two parameters
  - Interrupt level (1-7, but don't use level 7 on M68k) – often set with on-board jumpers or DIP switches
  - Interrupt vector (0-255, <64 reserved on MC680x0) – often set by writing to an on-board register
- OS initialization takes two calls
  1. Connect interrupt handler to vector
     ```
     devConnectInterruptVME(unsigned vectorNumber,
                  void (*pFunction)(void *),void *parameter);
     ```
  2. Enable interrupt from VME to CPU
     ```
     devEnableInterruptLevelVME (unsigned level);
     ```

I/O interrupt record processing

- Record is processed when hardware interrupt occurs
- Granularity depends on device support and hardware
  - Interrupt per-channel vs. interrupt per-card
- `#include <dbScan.h>` to get additional declarations
- Call **scanIoInit** once for each interrupt source to initialize a local value:

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

- DSET must provide a **getIoIntInfo** routine to specify the interrupt source associated with a record – a single interrupt source can be associated with more than one record
- Interrupt handler calls **scanIoRequest** with the '**ioscanpvt**' value for that source – this is one of the very few routines which may be called from an interrupt handler

The DSET – getIoIntInfo

```
long getIoIntInfo(int cmd, struct … *precord, IOSCANPVT
  *ppvt);
```

- Set ***ppvt** to the value of the **IOSCANPVT** variable for the interrupt source to be associated with this record
- You may call **scanIoInit** to initialize the **IOSCANPVT** variable if you haven’t done so already
- Return 0 to indicate success or non-zero to indicate failure – in which case the record **SCAN** field will be set to **Passive**
- Routine is called with
  - (`cmd=0`) when record is set to SCAN=I/O Intr
  - (`cmd=1`) when record SCAN field is set to any other value

The DSET – specialLinconv

```
long specialLinconv(struct … *precord, int after);
```

- Analog input (ai) and output (ao) record DSETs include this sixth routine
- Called just before (`after=0`) and just after (`after=1`) the value of the **LINR**, **EGUL** or **EGUF** fields changes
- "Before" usually does nothing
- "After" recalculates **ESLO** from **EGUL/EGUF** and the hardware range if LINR is LINEAR. Doesn't change ESLO if LINR is SLOPE.
- If **LINR** field is appropriate ai record processing will compute **val** as

```
val = ((rval + roff) * aslo + aoff) * eslo + eoff
```

Ao record processing is similar, but in reverse

Asynchronous I/O

- Device support must not wait for slow I/O
- Hardware read/write operations which take “a long time” to complete must use asynchronous record processing
  - $T_{I/O} > 50\ \mu s$ – definitely “a long time”
  - $T_{I/O} < 2\ \mu s$ – definitely “not a long time”
  - $2\ \mu s < T_{I/O} < 50\ \mu s$ – ???
- If device does not provide a completion interrupt a “worker” thread can be created to perform the I/O
  - this technique is used for message-based (GPIB, USB, Serial, Ethernet) devices

Asynchronous I/O – read/write operation

- Check value of `precord->pact` and if zero:
  - Set `precord->pact` to 1
  - Start the I/O operation
    - write hardware or send message to worker thread
  - Return 0
- When operation completes run the following code from a thread (i.e. NOT from an interrupt handler)

```
struct rset *prset = (struct rset *)precord->rset;
dbScanLock(precord);
(*prset->process)(precord);
dbScanUnlock(precord);
```

- The record's process routine will call the device support read/write routine – with `precord->pact=1`
  - Complete the I/O, set rval, etc.

Asynchronous I/O – callbacks

- An interrupt handler must not call a record's process routine directly
- Use the callback system (**callback.h**) to do this
- Declare a callback variable

```
CALLBACK myCallback;
```

- Issue the following from the interrupt handler

```
callbackRequestProcessCallback(&myCallBack,priorityLow,precord
  );
```

- This queues a request to a callback handler thread which will perform the lock/process/unlock operations shown on the previous page
- There are three callback handler threads
  - With priorities Low, Medium and High

The ASYN Support Module

- This should be your first consideration for new device support
- It provides a powerful, flexible framework for writing device support for
  - Message-based asynchronous devices
    - In many cases these can be supported with no C programming at all (ASYN+StreamDevice)
  - Register-based synchronous devices