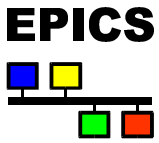
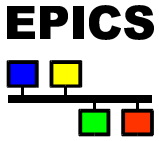


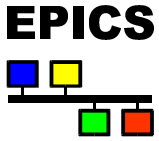
Device and Driver Support
(needs updating to R3.14.1)

Andrew Johnson
APS



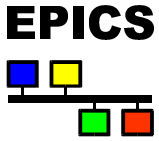
Outline

- What is device support?
- The .dbd file entry
- The DSET
- Device addresses
- Facilities available from
  - vxWorks
  - EPICS
- Using Interrupts
- Asynchronous I/O
- Callbacks and Watchdogs
- Driver Support
- The DrvET



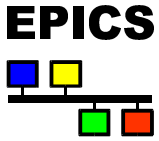
What is device support

- Interface between record and hardware
- Provides an API for record support to call
  - Record type determines routines needed
- Intimate knowledge of the record type
  - Full read/write access to any record field
- Performs record I/O on request
- Determines whether a record is to be synchronous or asynchronous
- Provides I/O Interrupt support
  - Determines whether I/O Interrupt allowed



Why use it

- Why not make record types for each hardware interface, with fields to allow full control over the facilities it provides?
  - Users don't have to learn about a new record type for each I/O board they want to use
  - Changing the I/O hardware is much easier
  - Record types provided are sufficient for most hardware interfacing tasks
  - It is still possible to create new record types for special purposes if necessary
  - Device support is simpler than record support
  - Device support isolates the hardware interface code from changes to the record API
  - Bug-fixes or enhancements to record support only need to happen in one place
  - Modularity (good software engineering practice)



The .dbd file entry

- The IOC discovers what device supports are present from entries in the .dbd file

```
device(recType,addrType,dset,"name")
```

- *addrType* is one of

AB_IO	BITBUS_IO	BBGPIB_IO
CAMAC_IO	GPIB_IO	INST_IO
RF_IO	VME_IO	VXI_IO

- *dset* is the 'C' symbol name for the Device Support Entry Table (DSET)
  - By convention the *dset* name indicates the record type and hardware interface
  - This is how record support and the database code call device support routines
- For example

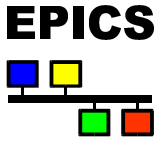
```
device(ai,INST_IO,devAiSymb,"vxWorks variable")
device(bo,VME_IO,devBoXy240,"Xycom XY240")
```

The DSET

- … is a 'C' `struct` containing function pointers, the content of which can vary by record type
- Each device support layer defines a named DSET with pointers to its own routines
- All DSET structure declarations start

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

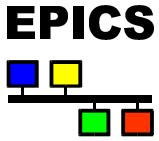
- *number* gives the number of function pointers in the DSET, usually 5 or 6 for standard types
- A NULL pointer is given when an optional routine is not implemented
- DSET routines are usually declared `static` (local) as their symbol names are not needed at runtime



DSET: initialize

```
long initialize(int pass);
```

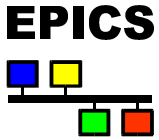
- Initializes the device support layer
- Optional routine, not always needed
- Used for one-time startup operations, e.g.
  - Start background tasks
  - Create shared tables
- Routine called twice by iocInit:
  - `pass=0` — Before record initialization
    - Doesn't usually access hardware as it doesn't know what I/O is to be used by this database
  - `pass=1` — After all record initialization
    - Can be used as a final startup step, all devices addressed by database are known by this point



DSET: initRecord

```
long initRecord(struct … *precord);
```

- Called at iocInit once for each record, to tell device support about the record
- Optional routine, but usually supplied
- Device support code should
  - Check the INP or OUT field address
  - Check if addressed hardware is present
  - Allocate any private storage required
    - Every record type has a `void *dpvt` field for device support to use as it wishes
  - Program device registers etc. as needed
  - Set some record-specific fields needed for I/O conversion to/from engineering units



DSET: read/write

- Most record types need a DSET routine

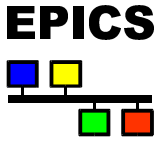
```
long read(struct … *precord);
```

or

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

- This implements the I/O operation which occurs when the record is processed.
- Precise action depends on the record type and whether the device is synchronous or asynchronous.
- Generally, synchronous input support
  - Reads hardware value into `precord->rval`
  - Returns 0 (meaning OK)
- and synchronous output support
  - Copies value in `precord->rval` to hardware
  - Returns 0 (OK again)

Except in practice there's usually a bit more than this to do…



Device addresses

- Device support .dbd entry was

  `device(`*`recType`*`,`*`addrType`*`,`*`dset`*`,"`*`name`*`")`

- *addrType* tells database software what type to use for the address link, e.g.

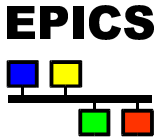
  `device(bo,VME_IO,devBoXy240,"Xycom XY240")`

  applies to the `pbo->out` field:

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

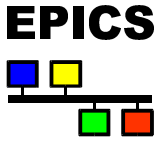
```
struct vmeio {
    short card;
    short signal;
    char *parm;
};
```

- See `<epics>/include/link.h` for other structs, and my Database lecture or IOC Application Developers Guide for the format of addresses.



A first example …

```
#include <sysLib.h>
#include <devSup.h>
#include <recSup.h>
#include <biRecord.h>

long initRecord(struct biRecord *prec){
     char *pbyte, dummy;
     prec->pact = 1;
     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 address");
          return S_dev_badInpType;
     }
     if (sysBusToLocalAdrs(VME_AM_SUP_SHORT_IO,
                    (char *)prec->inp.value.vmeio.card,
                    &pbyte) == ERROR) {
          recGblRecordError(S_dev_badCard, (void *)prec,
                    "devBiFirst: Can't convert VME address");
          return S_dev_badCard;
     }
     if (vxMemProbe(pbyte,READ,1,&dummy) < 0) {
          recGblRecordError(S_dev_badCard, (void *)prec,
                    "devBiFirst: Nothing there!");
          return S_dev_badCard;
     }
     prec->dpvt = pbyte;
     prec->mask = 1 << prec->inp.value.vmeio.signal;
     prec->pact = 0;
     return OK;
}
```

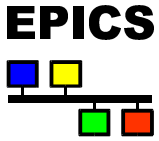


… continued

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

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

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

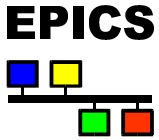
vxWorks Facilities

- vxWorks Operating System provides
  - ANSI C libraries
    - stdio.h — fopen, printf, sscanf, etc.
    - string.h — memcpy, strcat, strcmp etc.
    - math.h — sin, log, fabs etc.
    - stdlib.h — strtol, calloc, cfree etc.
  - Board Support Package: sysLib
    - Bus address Local address conversions
    - Bus interrupt control (enable, disable)
  - Other useful libraries
    - wdLib — watchdog timers
    - intLib, intArchLib — interrupt connection
    - taskLib — creating and controlling tasks
    - semLib, semBLib, senCLib, semMLib — binary, counting and mutual exclusion semaphores
    - rngLib — ring buffer handling
    - msgQLib — inter-task message queues
    - logLib — error messages from ISR
- Use `man intLib` to see the list of routines or `man memcpy` for individual details



EPICS facilities

- EPICS core software provides
  - recGbl.h — error and alarm reporting for records
  - callback.h — task level callbacks
  - taskwd.h — callback if background task dies
  - devLib.h — OS-neutral h/w interface library
    - Good idea but not widely used, API not brilliant
  - ellLib.h — general-purpose linked-list support
  - epicsAssert.h — preferred to ANSI assert.h
  - errlog.h — error message logging (printf etc.)
- See IOC Application Developers Guide for descriptions of most routines available



Using Interrupts

- vxWorks ISRs can be written in 'C'
- On VME two parameters are needed:
  - Interrupt level — prioritization, 1-7
    - Often set with I/O board DIP switches or jumpers
    - Level 7 is non-maskable and can crash vxWorks
  - Interrupt vector — unique within this CPU
    - Vector numbers < 64 are reserved by MC680x0
    - Use `veclist` on IOC to see vectors in use
    - Written to a register on the card
- OS Initialization requires two actions:
  - Connect ISR to relevent interrupt vector

```
intLib.h:
    long intConnect(VOIDFUNCPTR *ivec,
            void (*fp)(int param), int param);
iv.h:
    #define INUM_TO_IVEC(num) …
```

  - Enable VME interrupt level onto CPU board

```
sysLib.h:
    long sysIntEnable(int level);
```

NB not intEnable

Interrupt Scanning

- Records are processed when hardware signal occurs (e.g. input bit changed)
- Granularity depends on hardware and device support software
  - Interrupt per signal … Interrupt per card
  - Can be simulated using a background task
- **`#include <dbScan.h>`**
- Call `scanIoInit` once for each interrupt 'source' to initialize a pointer

```
void scanIoInit(IOSCANPVT *ppvt);
```

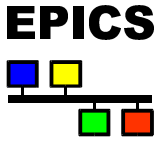
  - The same interrupt 'source' can be used for any number of records and record types
- DSET must have a `getIoIntInfo` routine which indicates which 'source' to use
- When interrupt occurs, call `scanIoRequest` passing back the pointer for this source

```
void scanIoRequest(IOSCANPVT pvt);
```

DSET: getIoIntInfo

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

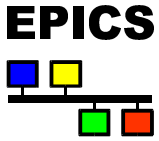
- Device support copies the `IOSCANPVT` value for this record into `*ppvt`
  - You may call `scanIoInit` from here if you haven't done so yet for this interrupt source
- Return 0 if Ok, else non-zero when scan will be reset to `Passive`
- Can usually ignore the `cmd` value
  - Whenever a record is set to `scan=I/O Intr` the routine is called with `cmd=0`
  - If scan field is later to be changed to something else, routine will be called again with `cmd=1`



DSET: report

```
long report(int type);
```

- Optional, called by `dbior` shell command
- Omitted if device support calls driver support, as this will give the I/O status
- Should print out information about current state, hardware connected etc.
- If `type=0`, just give a list of the hardware connected one card per line
- Higher values of `type` provide increasing amounts of data, or different pages
- Additional Suggestions:
  - Collect I/O statistics (count successful reads, write, interrupts, errors etc.) and use one page to display this data
  - Have a page to list the card's registers, which will help you to debug your device support



Other routines

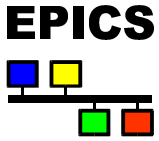
- Analogue Input and Output record DSETs include a sixth DSET routine:

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

- Called just before (`after=0`) and just after (`after=1`) the database changes the value of the linr, egul or eguf fields of the record
- "Before" usually does nothing.
- "After" call must recalculate eslo from egul, eguf and the ADC/DAC range
- If `linr=Linear`, ai will convert rval using

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

- The ao conversion is similar but in reverse

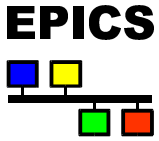


Asynchronous I/O

- Device support must not wait for slow I/O
- If hardware read/ write operations take "a long time," use asynchronous record processing
  - If the device doesn't provide a suitable completion interrupt, a background task can poll it periodically
- An asynchronous read/write routine
  - Looks at `precord->pact`, and if false (idle) it:
    - Starts the I/O operation
    - Sets `precord->pact` true, then returns 0
  - When the operation completes, device support must run the following code **at task level**, not in the ISR:

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

  - The record's process routine will call the device support read/write routine again. Now `pact` is true
    - Completes the I/O, sets the `rval` field etc.
    - Returns 0



Using Callbacks

- An ISR cannot call the record's process routine directly
  - Very few operations are allowed inside an ISR
- Use the callback system to do this
  - It has ring buffers to queue callback requests, which can be made from interrupt level
  - One of three (prioritized) callback tasks executes the registered callback function
- Header file `callback.h` contains

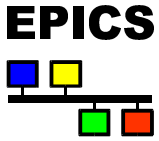
```
typedef struct … CALLBACK;
```

These are defined as macros:

```
callbackSetCallback(void (*pfunc)(CALLBACK *cb),
                    CALLBACK *cb);
callbackSetPriority(int prio, CALLBACK *cb);
callbackSetUser(void *user, CALLBACK *cb);
callbackGetUser(void *user, CALLBACK *cb);
```

This is a procedure:

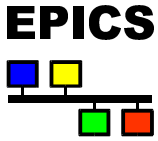
```
void callbackRequest(CALLBACK *cb);
```



Watchdog Timers

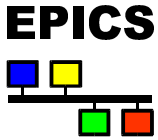
- A common requirement is for a timeout if the interrupt might never occur
- Use vxWorks watchdog library wdLib
  - Create watchdog timer, returns a wdId
  - Start timer giving timeout period, wdId, callback routine, and an integer context value
  - Cancel timer, else callback executed **in an ISR** if not cancelled by end of timeout period
- Header file `wdLib.h` defines:

```
typedef … WDOG_ID;
WDOG_ID  wdCreate (void);
long wdStart(WDOG_ID wdId, int delay,
             void (*pfunc)(int param), int param);
long wdCancel(WDOG_ID wdId);
```



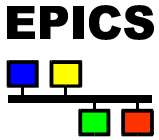
Asynchronous example

```
/*************************************************************************\
* Copyright (c) 2002 The University of Chicago, as Operator of Argonne
*     National Laboratory.
* Copyright (c) 2002 The Regents of the University of California, as
*     Operator of Los Alamos National Laboratory.
* EPICS BASE Versions 3.13.7
* and higher are distributed subject to a Software License Agreement found
* in file LICENSE that is included with this distribution.
\*************************************************************************/

/* devAiTestAsyn.c - Device Support to test async processing */

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "alarm.h"
#include "callback.h"
#include "cvtTable.h"
#include "dbDefs.h"
#include "dbAccess.h"
#include "recGbl.h"
#include "recSup.h"
#include "devSup.h"
#include "link.h"
#include "dbCommon.h"
#include "aiRecord.h"
```



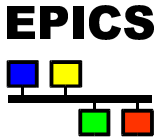
… continued

```
/* Create the dset for devAiTestAsyn */
static long init_record();
static 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};
```



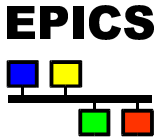
… *continued*

```
static long init_record(pai)
    struct aiRecord     *pai;
{
    CALLBACK *pcallback;
    /* ai.inp must be a CONSTANT*/
    switch (pai->inp.type) {
    case (CONSTANT) :
        pcallback = (CALLBACK *)(calloc(1,sizeof(CALLBACK)));
        pai->dpvt = (void *)pcallback;
        if(recGblInitConstantLink(&pai->inp,DBF_DOUBLE,&pai->val))
            pai->udf = FALSE;
        break;
    default :
        recGblRecordError(S_db_badField,(void *)pai,
                "devAiTestAsyn (init_record) Illegal INP field");
        return(S_db_badField);
    }
    return(0);
}
```



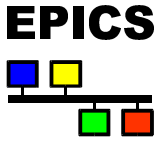
… continued

```
static long read_ai(struct aiRecord *pai)
{
    CALLBACK *pcallback = (CALLBACK *)pai->dpvt;
    /* ai.inp must be a CONSTANT */
    switch (pai->inp.type) {
    case (CONSTANT) :
        if(pai->pact) {
                printf("Completed asynchronous processing: %s\n",pai->name);
                return(2); /* don`t convert */
        } else {
                if(pai->disv<=0) return(2);
                printf("Starting asynchronous processing: %s\n",pai->name);
                pai->pact=TRUE;
                callbackRequestProcessCallbackDelayed(
                    pcallback,pai->prio,pai,(double)pai->disv);
                return(0);
        }
    default :
        if(recGblSetSevr(pai,SOFT_ALARM,INVALID_ALARM)){
                if(pai->stat!=SOFT_ALARM) {
                        recGblRecordError(S_db_badField,(void *)pai,
                           "devAiTestAsyn (read_ai) Illegal INP field");
                }
        }
    }
    return(0);
}
```



Driver Support

- Optional layer below device support
- No formal behavioral requirements
  - Does not use the open/close/read/write interface, which doesn't map well into EPICS
- Usually used to:
  - Share code between similar device support layers for different record types
    - Most digital I/O interfaces support record types bi, bo, mbbi, mbbo, mbbiDirect and mbboDirect
  - Initialize a software module before individual device layers using it are initialized
    - The order in which device support initialization is called is not guaranteed
  - Provide a common interface to an I/O bus such as GPIB, Bitbus, Allen-Bradley etc.
    - I/O devices placed on this bus need their own device support, but share the interface hardware
  - Provide a wrapper around commercial device drivers or other software libraries



The DrvET

- … is a struct containing function pointers
- Each driver defines a named DrvET with pointers to its own routines
- Drivers have an entry in the .dbd file:

```
driver(DrvETname)
```

- All DrvET structure declarations start

```
struct drvet {
    long number;
    long (*report)(int type);
    long (*initialize)(int pass);
};
```

- A driver may extend the DrvET, adding routines for device support to call
  - In practice very few do this, usually providing named library routines to call instead