Basic EPICS Device Support

COLLABORATORS

	TITLE : Basic EPICS Device Support	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Michael Davidsaver	August 2017	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1	April 2009	Initial revision	MAD
2	Febuary 2013	Add I/O Intr scanning example	MAD
3	November 2015	Update I/O Intr examples to include new APIs from Base 3.15 and 3.16 series.	MAD
4	August 2017	Signal alarms after init_record() has failed.	MAD

Contents

1 Introduction

Device support is the means of providing functionality specific to access hardware. This is done by providing several functions which the record support layer will call when appropriate. The functions which must be provided depend on the record type being supported. The *Record Reference Manual*[RecRef] provides a list of record types with descriptions and lists of device support functions. To determine the exact form of a record's device support functions the source found in *$EPICS_BASE/src/rec* is invaluable.

The latest version of this page can be found at: http://mdavidsaver.github.io/epics-doc/epics-devsup.html

Complete example code may be found in the *code-listings* directory at: https://github.com/mdavidsaver/epics-doc

2 Prepare IOC environment

It is assumed that EPICS Base is already built, that EPICS_BASE is set, and that the EPICS executables are in the system search PATH. If not, then first see Getting Started.

All paths given in this example are assumed to be relative to the *devsumexample* directory.

```
$ mkdir devsupexample
$ cd devsupexample
```

3 Periodic Example

Let us begin with an example. The Analog input record is intended to represent a value read from hardware and interpreted as a floating point number. This does not imply that the underlying hardware representation is a floating point number. The AI record support provides a facility for conversion between a raw integer value and a floating point number.

In this example the *hardware* device to be read is the system pseudo random number generator. Whenever the record is processed a new number is read into the process variable (PV) database.

```
makeBaseApp.pl -t ioc prng
```

This creates the makefiles needed to compile the code. The files we are about to create will be placed in *prngApp/src*. Later when *.db* files are created we will place them in *prngApp/Db*.

Take a moment to examine the files in *prngApp/src*. The file *prngMain.cpp* will be the point of entry for our IOC when run on a non-embedded platform. It is not very interesting through as it serves only to invoke the IOC shell.

The *prngApp/src/Makefile* does contain several interesting entries. Removing comments and blank lines leaves the following.

```
TOP=../..
include $(TOP)/configure/CONFIG
PROD_IOC = prng
DBD += prng.dbd
prng_DBD += base.dbd
prng_SRCS += prng_registerRecordDeviceDriver.cpp
prng_SRCS_DEFAULT += prngMain.cpp
prng_SRCS_vxWorks += -nil-
prng_LIBS += $(EPICS_BASE_IOC_LIBS)
include $(TOP)/configure/RULES
```

It is important to note that the EPICS build system attaches special significance to file names, not just extensions.

This Makefile will build the *prng* IOC (executable) from the two given C++ files and the database definition. Of these three only *prngMain.cpp* exists currently. The file *prng_registerRecordDeviceDriver.cpp* is automatically generated from the database definition. The database definition file *prng.dbd* is also generated by concatenating *base.dbd* with other *.dbd* files which we will

add later. At this point is effectively just a copy of *base.dbd*, which is part of the EPICS Base package and specifies, among other things, the basic record types (see *$EPICS_BASE/dbd/base.dbd*).

At this point, the *prng* IOC can now be compiled, and the resulting executable can be run. However, it will not be capable of doing anything more than the *softIoc* executable. In fact, they are functionally identical.

3.1 Writing Support

Now create the file *prngApp/src/devprng.c* with the following sections.

```
#include <stdlib.h>
#include <dbAccess.h>
#include <devSup.h>
#include <recGbl.h>
#include <alarm.h>

#include <aiRecord.h>

#include <epicsExport.h>

struct prngState {
  unsigned int seed;
};
```

Our device support code will be contained in the *init_record* and *read_ai* functions. Custom state information will be held in an instance of the *prngState* structure.

It should be noted that AI record support requires *read_ai* to be specified, but *init_record* is optional.

3.1.1 init_record

```
static long init_record(aiRecord *prec)
{
  struct prngState* priv;
  unsigned long start;

  priv=malloc(sizeof(struct prngState));
  if(!priv){
    recGblRecordError(S_db_noMemory, (void*)prec,
      "devAoTimebase failed to allocate private struct");
    return S_db_noMemory;
  }

  recGblInitConstantLink(&prec->inp,DBF_ULONG,&start);

  priv->seed=start;
  prec->dpvt=priv;

  return 0;
}
```

This *init_record* function is called once for each record in the IOC database which uses DTYP *Random* (ie *devAiPrng* device support). The association between DTYP and our code is described below.

Our *init_record* allocates space for the structure used to keep internal state, then parses the Constant input link string to get the initial seed value.

The input link can only by a CONSTANT link so there is no need to verify this.

3.1.2 read_ai

```
static long read_ai(aiRecord *prec)
{
  struct prngState* priv=prec->dpvt;
  if(!priv) {
    (void)recGblSetSevr(prec, COMM_ALARM, INVALID_ALARM);
    return 0;
  }

  prec->rval=rand_r(&priv->seed);

  return 0;
}
```

Whenever a record using this device support is processed *read_ai* is invoked. It simply invokes the thread-safe version of the *rand* function to supply a raw (integer) value.

It is important to note that *read_ai* function will be invoked even if *init_record* returns an error. This means that DPVT may be NULL. As this is a symptom of an earlier error, it is not advisable to print a message. However, signaling an INVALID alarm condition will indicate to any users that VAL does not reflect an actual measurement.

3.1.3 Exporting

Now associate the name *devAiPrng* with our device support functions. This mechanism is a way of providing a set of functions which the record support will use in to perform certain functions.

```
struct {
  long num;
  DEVSUPFUN  report;
  DEVSUPFUN  init;
  DEVSUPFUN  init_record;
  DEVSUPFUN  get_ioint_info;
  DEVSUPFUN  read_ai;
  DEVSUPFUN  special_linconv;
} devAiPrng = {
  6, /* space for 6 functions */
  NULL,
  NULL,
  init_record,
  NULL,
  read_ai,
  NULL
};
epicsExportAddress(dset,devAiPrng);
```

The number of functions record support will look for and the meaning of these functions is determined by record support. For information on a specific record types see the *Record Reference Manual*[RecRef] and the source (ie *$EPICS_BASE/src/rec/aiRecord.c*).

3.2 Device Definition

Our final task is to make an addition to the IOC database for our prng device. Create the file *prngApp/src/prngdev.dbd*:

```
device(ai,CONSTANT,devAiPrng,"Random")
```

This defines the device *devAiPrng* as support for an AI record with a CONSTANT input link named "Random". The name *devAiPrng* must be unique in the IOC. The combination of record type and name string must also be unique. It is convention that device support names should take the form *devXxYyyy* where *Xx* is the record type and *Yyyy* identifies the driver providing the code. It is also wise to include the driver name in the DTYP string to avoid name conflicts.

3.3 Building

Now modify *prngApp/src/Makefile* to include the prng database and support code. Then go to the *devsupexample* directory and run *make*

```
TOP=../..
include $(TOP)/configure/CONFIG
PROD_IOC = prng
DBD += prng.dbd
prng_DBD += base.dbd
prng_DBD += prngdev.dbd # <- added
prng_SRCS += prng_registerRecordDeviceDriver.cpp
prng_SRCS += devprng.c  # <- added
prng_SRCS_DEFAULT += prngMain.cpp
prng_SRCS_vxWorks += -nil-
prng_LIBS += $(EPICS_BASE_IOC_LIBS)
include $(TOP)/configure/RULES
```

3.4 Database Configuration

The next task is to create an IOC database which uses the *Random* device support. Place the following in *prngApp/Db/prng.db* and add it to the makefile *prngApp/Db/Makefile*.

```
record(ai,"$(P)"){
  field(DTYP,"$(D)")
  field(DESC,"Random numbers")
  field(SCAN,"$(SCAN=1 second)")
  field(INP,"$(S)")
  field(LINR,"LINEAR")
  field(ESLO,1e-9)
  field(EOFF,-1)
}
```

This will allow us to create several PVs generating random numbers. The combination of record type *ai* and the *DTYP* field is used to identify the correct device support. When instantiated *$(P)*, *$(D)*, and *$(S)* will be replaced with the PV name, device support type (*Random*), and the initial seed value. These will be specified later.

The fields *ESLO* and *EOFF* serve to define a linear scale to use when converting (integer) raw values to (floating point) engineering units.

Note: when changing *prngApp/Db/prng.db* remember to run *make* to update *db/prng.db*.

3.5 Running

In *devsupexample* create the IOC boot infrastructure to run the first example (prng1).

```
makeBaseApp.pl -a linux-x86 -i -t ioc -p prng prng1
```

In *iocBoot/iocprng1/st.cmd*:

```
< envPaths
cd ${TOP}
dbLoadDatabase "dbd/prng.dbd"
prng_registerRecordDeviceDriver pdbbase
# V Add this line V
dbLoadRecords("db/prng.db","P=test:prng,D=Random,S=324235")
cd ${TOP}/iocBoot/${IOC}
iocInit
```

Now run the IOC.

```
make
cd iocBoost/iocprng1
../../bin/linux-x86/prng st.cmd
```

Then watch the value of the PV *test:prng*

```
$ camonitor test:prng
test:prng                      2009-02-21 15:29:15.364549 0.155918
test:prng                      2009-02-21 15:29:16.364611 -0.681225
 ...
```

4 Asynchronous Example

The preceding example assumes that calls to *read_ai* will return quickly. This is true of *rand_r* which does only a simple computation, but not true of many operations which access hardware. In these cases it is desirable to start an operation, spend time doing other things, and only update the database when the result becomes available.

Support for this mode of operation is provided via the *PACT* flag. The following example creates another device support which demonstrates asynchronous processing.

Add the following line to *prngApp/src/prngdev.dbd*

```
device(ai,CONSTANT,devAiPrngAsync,"Random Async")
```

4.1 Writing Support

Now create the file *prngApp/src/devprngasync.c* and add the following sections.

```
#include <stdlib.h>
#include <dbAccess.h>
#include <devSup.h>
#include <recSup.h>
#include <recGbl.h>
#include <alarm.h>
#include <callback.h>

#include <aiRecord.h>

#include <epicsExport.h>

struct prngState {
  unsigned int seed;
  CALLBACK cb; /* New */
};

static void prng_cb(CALLBACK* cb);
```

Note the addition to the *prngState* struct of a CALLBACK.

4.1.1 init_record

```
static long init_record(aiRecord *prec)
{
  struct prngState* priv;
  unsigned long start;
```

```
  priv=malloc(sizeof(struct prngState));
  if(!priv){
    recGblRecordError(S_db_noMemory, (void*)prec,
      "devAoTimebase failed to allocate private struct");
    return S_db_noMemory;
  }

  /* New */
  callbackSetCallback(prng_cb,&priv->cb);
  callbackSetPriority(priorityLow,&priv->cb);
  callbackSetUser(prec,&priv->cb);
  priv->cb.timer=NULL;

  recGblInitConstantLink(&prec->inp,DBF_ULONG,&start);

  priv->seed=start;
  prec->dpvt=priv;

  return 0;
}
```

The callback function and priority are set.

4.1.2 read_ai

```
static long read_ai(aiRecord *prec)
{
  struct prngState* priv=prec->dpvt;
  if(!priv) {
    (void)recGblSetSevr(prec, COMM_ALARM, INVALID_ALARM);
    return 0;
  }

  if( ! prec->pact ){
    /* start async operation */
    prec->pact=TRUE;
    callbackSetUser(prec,&priv->cb);
    callbackRequestDelayed(&priv->cb,0.1);
    return 0;
  }else{
    /* complete operation */
    prec->pact=FALSE;
    return 0;
  }
}
```

The operation of *read_ai* changes substantially. When the record is processed control passes into *read_ai* with the *PACT* field set to false. To start an asynchronous operation this field is set to TRUE, and a delayed action is scheduled. While *PACT* is set the IOC will not try to process this record again.

When the callback has completed it must manually process the record which will complete the operation and allow *PACT* to be cleared.

4.1.3 callback

```
static void prng_cb(CALLBACK* cb)
{
  aiRecord* prec;
```

```
  struct prngState* priv;
  rset* prset;
  epicsInt32 raw;

  callbackGetUser(prec,cb);
  prset=(rset*)prec->rset;
  priv=prec->dpvt;

  raw=rand_r(&priv->seed);

  dbScanLock((dbCommon*)prec);
  prec->rval=raw;
  (*prset->process)(prec);
  dbScanUnlock((dbCommon*)prec);
}
```

This generic callback is taken from the *EPICS Application Developer's Guide*[AppDev]. It manually invokes record processing. It this way *read_ai* is called while *PACT* is set.

Note: Due to the way database processing and the *PACT* flag are handled no additional locking is required. More complicated scenarios involving multiple PVs might require, for example, a mutex to guard *priv→seed*.

Remember to add *devprngasync.c* to *prngApp/src/Makefile*.

4.1.4 Exporting

```
struct {
  long num;
  DEVSUPFUN  report;
  DEVSUPFUN  init;
  DEVSUPFUN  init_record;
  DEVSUPFUN  get_ioint_info;
  DEVSUPFUN  read_ai;
  DEVSUPFUN  special_linconv;
} devAiPrngAsync = {
  6, /* space for 6 functions */
  NULL,
  NULL,
  init_record,
  NULL,
  read_ai,
  NULL
};
epicsExportAddress(dset,devAiPrngAsync); /* change name */
```

4.2 Running

The database file can be reused so only one change is necessary.

In *iocBoot/iocprng1/st.cmd* add a line:

```
< envPaths
cd ${TOP}
dbLoadDatabase "dbd/prng.dbd"
prng_registerRecordDeviceDriver pdbbase
dbLoadRecords("db/prng.db","P=test:prng,D=Random,S=324235")
# V Add this line V
dbLoadRecords("db/prng.db","P=test:prngasync,D=Random Async,S=324235")
cd ${TOP}/iocBoot/${IOC}
iocInit
```

5 Interrupt Driven Example

This section demonstrates a variation on the Periodic example which allows the periodic SCAN (eg. "1 second") to be switch with a SCAN rate driven from a device specific source ("I/O Intr"). In this example, an arbitrary thread is used.

Add the following line to *prngApp/src/prngdev.dbd*

```
device(ai,CONSTANT,devAiPrngIntr,"Random Intr")
```

5.1 Writing Support

Now create *prngApp/src/devprngintr.c* with the following sections.

```
#include <stdlib.h>
#include <epicsExport.h>
#include <dbAccess.h>
#include <devSup.h>
#include <recGbl.h>
#include <alarm.h>
#include <dbScan.h>
#include <dbDefs.h>
#include <ellLib.h>
#include <cantProceed.h>
#include <epicsThread.h>
#include <epicsMutex.h>
#include <initHooks.h>

#include <aiRecord.h>

static ELLLIST allprngs = ELLLIST_INIT;

struct prngState {
  ELLNODE node;
  unsigned int seed;
  unsigned int lastnum;
  epicsMutexId lock;
  IOSCANPVT scan;
  epicsThreadId generator;
};
```

As before we define a private structure. Several addition members are added. The most significant is *IOCSCANPVT scan;* as will be seen in the following sections.

5.1.1 init

```
static void start_workers(initHookState state);

static long init(int phase)
{
  if(phase==0)
    initHookRegister(&start_workers);
  return 0;
}
```

Use of *I/O Intr* scanning is not permitted before a certain point in the IOC startup sequence (*iocInit()*). Here a callback function is added which will receive notification of all future updates.

Note that a device support *long init(int)* function is called exactly twice (phase 0 and 1) during the initialization sequence. By waiting until this point, the callback function will not be called for some of the hook states (they have already happened). To capture all states the hook must be registered before *iocInit()* is called. This can be accomplished by using a *registrar()* function.

5.1.2 init_record

```
static long init_record(aiRecord *prec)
{
  struct prngState* priv;
  unsigned long start;

  priv=callocMustSucceed(1,sizeof(*priv),"prngintr");

  recGblInitConstantLink(&prec->inp,DBF_ULONG,&start);

  priv->seed=start;
  scanIoInit(&priv->scan);
  priv->lock = epicsMutexMustCreate();
  priv->generator = NULL;
  ellAdd(&allprngs, &priv->node);
  prec->dpvt=priv;

  return 0;
}
```

Each record is initialized with its own private structure and (later) worker thread.

The *allprngs* list is used to keep track of all private structures created. No locking is needed in this example because *init_record()* is called during the singled threaded phase of database initialization, and the list will not be modified afterward.

The function *scanIoInit()* is used to prepare this structure's *I/O Intr* scan list.

5.1.3 Init hook

```
static void start_workers(initHookState state)
{
  ELLNODE *cur;
  if(state!=initHookAfterInterruptAccept)
    return;
  for(cur=ellFirst(&allprngs); cur; cur=ellNext(cur)) {
    struct prngState *priv = CONTAINER(cur, struct prngState, node);
    priv->generator = epicsThreadMustCreate("prngworker",
                                            epicsThreadPriorityMedium,
                                            epicsThreadGetStackSize(epicsThreadStackSmall),
                                            &worker, priv);
  }
}
```

Here the hook function uses the state *initHookAfterInterruptAccept* to start a worker for each of the private structures.

5.1.4 Worker

With EPICS Base less than 3.16.0.

```
static void worker(void* raw)
{
  struct prngState* priv=raw;
  while(1) {
    epicsMutexMustLock(priv->lock);
    priv->lastnum = rand_r(&priv->seed);
    epicsMutexUnlock(priv->lock);

    scanIoRequest(priv->scan);
```

```
    epicsThreadSleep(1.0);
  }
}
```

The worker thread itself simply loops. Each iteration updates the private structure with a new random number. When the number is ready the function *scanIoRequest()* to queue a request to process all records associated with this scan list.

scanIoRequest() does not process the records directly, and can be called from interrupt context.

With EPICS Base greater than or equal to 3.16.0 (as of Nov. 2015 not released).

```
static void worker(void* raw)
{
  struct prngState* priv=raw;
  while(1) {
    epicsMutexMustLock(priv->lock);
    priv->lastnum = rand_r(&priv->seed);
    epicsMutexUnlock(priv->lock);

    scanIoImmediate(priv->scan, priorityHigh);
    scanIoImmediate(priv->scan, priorityMedium);
    scanIoImmediate(priv->scan, priorityLow);

    epicsThreadSleep(1.0);
  }
}
```

5.1.5 get_iointr_info

```
static long get_ioint_info(int dir,dbCommon* prec,IOSCANPVT* io)
{
  struct prngState* priv=prec->dpvt;

  if(priv) {
    *io = priv->scan;
  }
  return 0;
}
```

So far we have seen how to initialize a scan list with *scanIoInit()*, and how to request a scan with *scanIoRequest()*. Of course, this is only interesting if doing so will actually cause some records to be processed.

A record is added to a scan list by setting its SCAN field to *I/O Intr*. This can either be set in a .db files, or changed at runtime. In either case this results in a call to the *get_ioint_info()* function with *dir=0*. If this function is not provided, or if no scan list is given (**io = NULL*) then the record will revert to *Passive* SCAN.

Once a record is successfully added to a scan list, it will be processed once for each call to *scanIoRequest()*. The *I/O Intr* scanning mechanism is based on a fixed length queue shared by all scan lists. So care must be taken to avoid overflowing this queue.

Setting the SCAN field to a value other than *I/O Intr* will cause the *get_ioint_info()* function to be called a second time with *dir=1*. In this case, the record will be removed from the provided scan list. It is essential that the same scan list is provided for both calls. Doing otherwise will result in undefined behavior (typically a crash).

5.1.6 read_ai

```
static long read_ai(aiRecord *prec)
{
  struct prngState* priv=prec->dpvt;
  if(!priv) {
    (void)recGblSetSevr(prec, COMM_ALARM, INVALID_ALARM);
```

```
    return 0;
  }

  epicsMutexMustLock(priv->lock);
  prec->rval = priv->lastnum;
  epicsMutexUnlock(priv->lock);

  return 0;
}
```

The device support read function simply copies the latest random number.

As random numbers are generated by the worker thread we can use *SCAN=I/O Intr* to update the record with minimum latency. It is also possible to set *SCAN=10 second* to throttle back the rate.

```
struct {
  long num;
  DEVSUPFUN  report;
  DEVSUPFUN  init;
  DEVSUPFUN  init_record;
  DEVSUPFUN  get_ioint_info;
  DEVSUPFUN  read_ai;
  DEVSUPFUN  special_linconv;
} devAiPrngIntr = {
  6, /* space for 6 functions */
  NULL,
  init,
  init_record,
  get_ioint_info,
  read_ai,
  NULL
};
epicsExportAddress(dset,devAiPrngIntr);
```

Define the device support table.

5.2 Rate Limiting (Base < 3.16.0)

The I/O Intr mechanism is implemented using the EPICS callback API. Internally *scanIoRequest()* calls *callbackRequest(CALLBACK*)* up to three times as each IOSCANPVT consists of three such callbacks, one for each of the available priorities. Each priority has a fixed length FIFO for requests, and one or more worker threads to run them.

A fixed length FIFO can, of course, become full if requests are queued too quickly. Because this FIFO is shared process wide, an overflow can be caused by any driver in the IOC, and affect all drivers using the queue. This manifests as the dreaded "callbackRequest: cbLow ring buffer full" log message.

The devprngintrrate.c example demonstrates how to avoid this for EPICS Base 3.14 and 3.15.

Beginning in EPICS Base 3.15.0.1 the "I/O Intr" handling API was extended to include *scanIoSetComplete()*. Prior to this, rate limiting could be accomplished by assuming that callbacks are executed in the order queued. With the introduction in 3.15 or parallel processing of callback queues, this no longer hold and the *scanIoSetComplete()* technique must be used.

5.3 Rate Limiting (Base >= 3.16.0)

This section describes unreleased functionality as of Nov. 2015.

```
void scanIoImmediate(IOSCANPVT scan, int priority);
```

Beginning with EPICS Base 3.16.0 an additional API call is available which does record processing in the calling thread (do **not** call from device support). This is demonstrated in the worker thread example shown above.

For situations where blocking is not tolerable (actual interrupt context) The *scanIoSetComplete()* must still be used.

6 When to Use

The Periodic example presented above suffers from two problems. Because all processing for a given SCAN rate is done on a single thread, any device support which blocks the thread will cause the entire IOC to become unresponsive. Also, the periodic SCANs are based on the system time of the computer running the IOC, which will rarely be tied to the processing cycle of the device being controlled.

The Asynchronous example solves the first problem by allowing work to be offloaded in another thread. The second problem is partly solved. While start of processing is not tied a device, the completion can be.

The I/O Intr examples allows all work to be done outside the normally scanning process, and the results pushed into records.

Now to consider some situations:

Simple periodic scanning is appropriate when the value to be read (or written) is immediately accessible, and the source of the value gives no notification of when a new value is available. An example of this is a temperature measurement from a memory mapped device (eg. local CPU temp.).

Asynchronous processing is appropriate in a number of cases. When the value is not immediately accessible, but rather arrives some time after it is requested. It is also useful when the underlying operation is a blocking system call. Also, asynchronous processing works with the Channel Access put with callback operation (*ca_put_callback()*).

I/O Intr processing is most appropriate when data is delivered without an explicit request (unsolicited), perhaps driven by an external trigger or clock.

However, there are a number of situations where I/O Intr processing can be used instead of Asynchronous processing. Instead of one Asynchronous record, use two records. The first initiates the request, while the second uses *SCAN=I/O Intr* to scan on completion. This may be helpful if the result(s) of the operation should be stored in several records.

This also allows flexibility if many requests can be queued before a response arrives. It may also avoid the need to provide an explicit cancel action.

7 Now What

These examples demonstrate writing device support for a single AI record. Support for other record types (BO, STRINGIN, EVENT, ...) differs only in which fields the *read_** function must update. Also interesting are some of the other fields which effect conversion of raw values in the analog record types, and the ability to bypass this conversion and specify the value in engineering units directly.

8 References

8.1 References

[1] [RecRef] The Epics Collaboration EPICS 3.14 Record Reference Manual Wiki http://www.aps.anl.gov/epics/-wiki/index.php/RRM_3-14

[2] [AppDev] Marty Kraimer et al. 'EPICS Application Developer's Guide '. http://www.aps.anl.gov/epics/base/-R3-15/2-docs/AppDevGuide/