MCOR Device-Support Design (DRAFT-0.3)

Till Straumann

7/03/2012

1 Introduction

This document describes the design of the support software for a new MCOR-controller hardware implementation. The new MCOR-controller (henceforth simply: "MCOR" or "controller") is implemented using a FPGA which communicates with an embedded computer ("COM-X" form-factor) via a PCI-express link.

The software ("MCOR device support") shall integrate the controller with the EPICS control system which is achieved by running EPICS-IOC software on the embedded computer.

This document addresses the "glue" software which interfaces the controller to the low-levels of the EPICS database. This interface is known as "device-support" in the EPICS jargon.

In addition, the software must provide a low-level API for direct access to the MCOR-setpoint DACs to be used by *fast-feedback* which operates without going through the EPICS database layers.

Throughout this document we assume that the functionality of the controller is accurately described by [JJOv03].

2 Scope

The scope of this design shall cover the "power-supply controller" aspects of the MCOR. It does not include the embedded EVR ("event receiver") nor the associated sections "Fiber Optic Transceiver Data" and "EEPROM Serial ID Memory Contents" in [JJOv03].

There is a short paragraph titled "Beam Synchronous Acquisition (BSA)" but it is unclear what functionality is actually provided and whether it falls inside our scope.

3 Requirements

The MCOR device support shall meet the following requirements:

- Make the registers listed under "BAR0 Memory Map" ([JJOv03], pp. 4) accessible to basic EPICS records (*ai*, *ao*, *longin*, *longout*, *bi*, *bo*, *mbbi*, *mbbo*).

  E.g., it shall be possible to modify a bit-field embedded in a 32-bit register using a *mbbo* record and at the same time to monitor a *different* bit in the same register using a *bi* record.

  It is up to the EPICS-database designer to pick appropriate record types from the supported set listed above.
- Make the "ADC-memory buffer" accessible to *waveform* records.
- Make the "MCOR System Information" registers accessible to *stringin* records.
- For each available (i.e., supported by hardware) interrupt a *IO-Interrupt scan-list* shall be allocated and a scan request shall be posted upon occurrence of each interrupt.

  As an exception the *waveform* device-support implementation may handle waveform-digitizer interrupts internally, without providing IO-Interrupt scanning.
- Provide direct access to the MCOR "Set Point Requested" register(s) via an API "inspired" by the `mgntsetDAC()` function:

  ```
  typedef int (*MgntSetDAC)(void *card_p, int dac_channel, epicsInt32 dac_value);
  ```

  We propose to modify the respective function prototype in the "Magnet" software and use a function pointer so that our version can be installed when appropriate.

  For non-feedback operation, the system designer may choose to use this same routine from proprietary (i.e., *not* included in this design but implemented elsewhere) device-support or alternatively to use our device-support to access this register in the same generic way as any other register (see section 6).

4 Assumptions

- Design and implementation of the EPICS database are outside of our scope.
- Only a single instance of MCOR-controller needs to be supported.
- Proper logic is implemented elsewhere (e.g., by the EPICS database) so that conflicting access (concurrent access by EPICS records and feedback) to the setpoint DACs is avoided.
- Proper logic is implemented elsewhere (e.g., by the EPICS database) to ensure the MCOR is operating in a mode suitable for feedback operation (e.g., no "ramping") while being accessed by feedback via the low-level API.

5 OS/Platform

The MCOR device support software shall run under the linux operating system. However, OSI techniques shall be used as much as possible. The software shall not assume a particular CPU-

"endian-ness" or word size. Under linux, UIO techniques shall be used and in-kernel code shall be avoided or at least minimized.

The MCOR device support shall be compatible with the 3.14.x release series of EPICS.

6 MCOR Device Support

We propose to use the "*devBusMapped*" package to implement MCOR device support. Using this package makes it possible to minimize C-code and define most details directly in the EPICS database, more specifically in the *INP* or *OUT* link definitions, respectively.

devBusMapped associates a name (in its own name-space) with a particular instance of device. The EPICS database then refers to the device by this name which represents the "base-address" of the memory-mapped device and to individual registers by numerical offsets.

However, *devBusMapped* does only support basic, scalar record types (as well as *stringin* and *stringout*) and we therefore propose to implement dedicated device-support for *waveform* records to read the ADC buffer memory.

Optionally, interrupts may be supported using so-called EPICS "IO-Interrupt scan-lists" (if implemented by the firmware).

6.1 Driver

A small *driver* shall scan the PCI configuration space for a MCOR-controller and register the name "`mcor0.0`" with *devBusMapped*. The driver also takes care of other initializations such as the IO-Interrupt scan-lists. These scan-lists are also referred-to by names from the EPICS database (see section 6.2.2). However, since the current version of [JJOv03] does not provide details about the interrupt hardware implementation yet, it is not possible at this point to define further details on our side.

If no valid controller can be found then no name(s) shall be registered.

Optionally, the driver may also implement the necessary "glue-code" to enable *devBusMapped* to support IO-Interrupt scanning (see section 6.2.7).

6.2 DevBusMapped

devBusMapped is documented in a README file in the package's `src/` subdirectory. Please consult this README for further details which are not given here. However, we provide an introduction to *devBusMapped* along with examples which are taylored to the MCOR case.

6.2.1 Dependencies

devBusMapped requires the LCLS-EPICS module "`miscUtils`" and the "`devLib2`" package. The MCOR device support module also requires the "`drvUioPciGen`" module which essentially is a "OSI" abstraction for PCI access via *devBusMapped*. The MCOR device support module itself is called "`drvPciMcor`" and must be added to to the final IOC application.

6.2.2 Link Field Syntax

To access a particular register (or a subset of bits in it) from a supported EPICS record type, the `DTYP` field of the record must be set to `BusAddress`.

The `INP` or `OUT` link field encodes the register access and uses the following syntax ("`VME_IO` style"):

```
#C<inst>S<shft>@<device>+<reg_offset>,<access_method>[,<iointr_scanlist>]
```

A register is addressed using a computed offset from the device "base-address". The base-address is represented by the symbolic name "`mcor0.0`" which is registered by the driver. The computed offset equals

$$(inst << shft) + reg\_offset$$

The numerical <`inst`> and <`shft`> parameters are useful to instantiate identical register blocks from templates (see below).

The <`access_method`> lets the database designer define how the register contents are read and written, respectively. It is defined using the syntax

```
<name> [ '(' <numerical_arg> { ',' <numerical_arg> } ')' ]
```

A set of predefined access methods exists (but it is possible to extend *devBusMapped* by adding user-defined access methods). The predefined methods implement 8-, 16- and 32-bit wide access to memory-mapped registers. There are different methods for "big-endian" (e.g., VMEbus) and "little-endian" (e.g., PCI) registers.

Since we are dealing with a PCIe device, the "little-endian" methods are appropriate here. They are called `be8` (there is only a "big-endian" method for 8-bit access since endian-ness is not an issue for single-byte access), `le16` and `le32`. In addition, there are `be8s` and `le16s` which indicate that the accessed entity is *signed*. This information may be necessary, e.g., when reading a (signed) 16-bit register into a *longin* or *ai* record so that the number is sign-extended correctly[1].

The (optional) <`numerical_arg`> parameters are interpreted by device support modules for individual records and/or the access methods. The only device-support currently using any arguments are *stringin* and *stringout* device-support. These *require* a single argument indicating the number of characters to be transferred.

The optional <`iointr_scanlist`> parameter is explained in section 6.2.7.

[1]E.g., reading a 8-bit register holding the value `0xff` into a *longin* with the `be8` method yields a reading of `255` whereas employing `be8s` would yield `-1`.

Let us now discuss an example: we would like to read the "Bulk Supply Voltage" into an *ai* record. The Bulk-Supply register block is at offset 0x400 and the desired register at offset 0x08 from there, i.e., at 0x408 from the board base-address. Since the register is 32-bit wide, we use `le32` access:

```
record(ai, "$(prefix):BLK_VOLT") {
    field(DTYP, "BusAddress")
    field(INP,  "#C0S0@mcor0.0+0x408,le32")
    field(SCAN, "1 second")
    ...
}
```

We could also have used the `<inst>` and `<shft>` parameters to generate the 0x400, e.g., setting `<inst>=1` and `<shft>=10` since $1 << 10 = 0x400$:

```
    field(INP,  "#C1S10@mcor0.0+0x8,le32")
```

The `reg_offset` has to be only 0x8 in this case.

The usefulness of the instance and shift parameters can be seen when we address the multiple MCOR Channel Register blocks from a template. E.g., the "Samples per Average" register (*unsigned* 16-bit register at offset 0x28 in the channel register block) can be addressed as

```
record(longout, "$(prefix):$(channel):SPAVG_RAW") {
    field(DTYP, "BusAddress")
    field(OUT,  "#C$(channel)S6@mcor0.0+0x28,le16")
    field(DRVL, "0")
    field(DRVH, "31")
    ...
}
```

Since the channel register blocks are spaced by $0x40 = 1 << 6$ we set `<shft>` to 6 and can then expand the template for channels 0..15. The `DRVL` and `DRVH` fields limit the range of numbers to what the hardware supports. Note that the semantics of this particular register define the actual "Samples per Average" to be *one more* than the register contents. To make this transparent to the user, it may be desirable to add a *calc* record upstream of this longout which subtracts one from the user input.

6.2.3 Accessing Individual Bits with *bi, bo, mbbi, mbbo*

When reading/writing individual bits with *bi/bo* records then the `MASK` field identifies the bit (or bits) to be read or written. The algorithms used are:

bi *devBusMapped* sets the `RVAL` field from the register contents using the `MASK` field:

$$RVAL = register\_contents \;\&\; MASK.$$

However, if `MASK` is zero then $RVAL = register\_contents$.

`VAL` is computed from `RVAL` using the *bi* record's conversion algorithm: $VAL = RVAL\ ?\ 1 : 0$.

bo If MASK is nonzero then it is used to merge bits into the current register contents:

$$register\_contents = (register\_contents \,\&\, \sim MASK) \mid (VAL \;?\; MASK : 0)$$

If MASK is not set (i.e., zero) then VAL is copied into the register:

$$register\_contents = VAL$$

Hence, MASK may be used to selectively modify a single bit in a register leaving the other ones alone.

Multiple (adjacent) bits in a single register may be accessed with *mbbi*/*mbbo* records under control of the NOBT and SHFT fields which are used to define the (immutable[2]) MASK field:

$$MASK = NOBT == 0 \;?\; 0xffffffff : ((1 << NOBT) - 1) << SHFT$$

The algorithms to set VAL are as follows:

mbbi *devBusMapped* sets the RVAL field from the register contents using

$$RVAL = register\_contents \,\&\, MASK$$

and then converts this to VAL using the *mbbi* record's standard conversion algorithm: look ($RVAL >> SHFT$) up in the state values table if state values have been set (ZRVL etc.):

$$VAL = table[RVAL >> SHFT]$$

otherwise set $VAL = RVAL >> SHFT$.

mbbo Use the record's conversion algorithm to compute RVAL from VAL: If state values have been set (ZRVL etc.) then look VAL up and set $RVAL = table[VAL] << SHFT$. If state values are not set then $RVAL = VAL << SHFT$.

devBusMapped merges RVAL under control of MASK into the register contents. The old register contents are read into RBV:

$$\begin{aligned} RBV &= register\_contents \\ register\ contents &= (register\_contents \,\&\, \sim MASK) \mid (RVAL \,\&\, MASK) \end{aligned}$$

6.2.4 Reading Character Strings with *stringin*

devBusMapped now supports reading character strings into *stringin* records (writing into *stringout* is also supported). Since strings stored in device memory are not always "NULL-terminated" the number of characters to be transferred may be specified by adding an optional "length" parameter to the access method. If no such length is given then the transfer is limited by the size of the *stringin* record's buffer. In any case, the transfer is stopped when a "NULL" character is read. A terminating "NULL" is written into the record.

[2]The MASK field has the SPC_NOMOD property (see e.g., mbbiRecord.dbd), i.e., it can not be modified directly by the user.

If the string is a normal character string in device memory then the ordinary “`be8`” method offers itself. Note that you would use e.g., a 32-bit access if individual characters are aligned on 32-bit boundaries in device memory (characters spaced by 4 bytes). In more exotic cases you may have to provide your own access method (which is repeatedly called by device-support to satisfy the requested string length or until NULL is read).

E.g., to read a four character string from device memory into a *stringin* a database snipped could look like this:

```
record(stringin, "$(prefix):VERSION") {
    field(DTYP, "BusAddress")
    field(INP,  "#C0S0@mcor0.0+0x5c8,be8(4)")
    field(PINI, "YES")
    ...
}
```

In the previous example we had set the `PINI` field to `YES` so that the record is only processed once (at initialization) during its lifetime because we assume that we read *static* string data from the device.

6.2.5 Thread Safety

devBusMapped allocates one mutex per registered base address and uses it to serialize access to the device’s registers. However, the mutex is only taken around “read-modify-write” (as implemented by the *bo* and *mbbo* algorithms) and “write” (*ao*, *longout*) as well as “string” (as implemented by the *stringin* and *stringout* algorithms) operations. *devBusMapped* assumes that a single (8-, 16- or 32-bit wide) register access is atomic and does not require locking the mutex. Hence, simple “read” access does not take the mutex[3].

The mutex guarantees that e.g., multiple *bo* records may safely manipulate different bits in the same register.

Even though the granularity of most of *devBusMapped*’s operations are individual registers (and thus each register could be protected by a *different* mutex) a single mutex is shared for all registers of a device[4].

However, if individual mutexes e.g., for the individual channel register blocks were desired then this could easily be achieved, simply by registering different names for the individual blocks (the driver would have to do this and the database link fields would need to refer to these names where appropriate).

[3]Since we assume that reading and writing a register is atomic, a “read” by one record while another record modifies the same register yields – depending on timing – either the old or the new contents. The result would not be different if “read” would lock the mutex.

[4]Of course, *devBusMapped* has no knowledge of what registers belong to a single “device” since all this information is encoded in the database’s link fields. All resources (including the mutex) which are associated with a “device” are actually associated with the symbolic name representing the registered “base-address” – in our case “`mcor0.0`”. Hence, all records which refer to the same symbolic name from their `INP` or `OUT` link field are accessing the same “device” and thus use the same mutex.

6.2.6 Initialization

During initialization of the EPICS database *devBusMapped* implements the following semantics for *output*-style records (*ao*, *longout*, *bo* and *mbbo*):

- If `PINI` is set (`YES`) then it is assumed that the database designer wants to enforce writing an initial value out to the device. Nothing special is done by *devBusMapped* which relies on the initial record processing to write the initial value.
- If `PINI` is *not* set then *devBusMapped* reads the current register settings from the hardware back and propagates them into the record. The record's `UDF` field as well as any alarms are reset. This lets the record reflect the hardware state at the time of initialization. However, note that the EPICS record support code does e.g., not check for alarm conditions during record initialization so that while the correct value makes it into the database any violation of alarm limits by this value may go unnoticed.

No special action is taken in the case of *input*-style records (*ai*, *longin*, *bi* and *mbbi*).

6.2.7 Interrupt Support

devBusMapped supports the "IO-Interrupt" scanning method. However, some device-specific C-code must be written in order to make this support work. The necessary code is part of the "driver" (see section 6.1).

- An `IOSCANPVT` object just be allocated and initialized.
- The `IOSCANPVT` object must be registered with the *devBusMapped* framework under a *name*. The specific scan-list may then be referenced by the database designer by providing the same name for the `<iointr_scanlist>` parameter in the database `INP` or `OUT` field(s).

  Since [JJOv03] does not provide details about the implementation of interrupts we do not define the number of scan-lists or their names yet.
- An interrupt handler must be implemented and installed during initialization. The interrupt handler must issue a `scanIoRequest()` which triggers scanning of the list associated with the handled interrupt.

In order for IO-Interrupt scanning to be enabled for a particular record the following conditions must be met:

- The `INP` or `OUT` link field must contain the `<iointr_scanlist>` parameter and supply the name of the correct scan-list (as defined by the driver, see section 6.1).
- The `SCAN` field must be set to "`I/O Intr`".

Let us conclude this section with an example. We assume that the MCOR hardware raises an interrupt when a fault is detected. The driver handles such an interrupt and requests IO-Interrupt scanning of a scan-list which is identified by the name "`fltintr`"[5].

[5]Note that `fltintr` is *not* an "official" name; it was just introduced for this example

We want a *longin* record to be scanned as a result of the interrupt and read the latched fault status register (unsigned 16-bit register at offset 0x442). The relevant database snipped could look like this:

```
record(longin, "$(prefix):LFAULTS") {
    field(DTYP, "BusAddress")
    field(INP,  "#C0S0@mcor0.0+0x442,le16,fltintr")
    field(SCAN, "I/O Intr")
    ...
}
```

We could also define multiple *bi* records along with meaningful names for the fault stati and have them all on the same scan-list. However, [JJOv03] does not specify the meaning of individual bits at the time of this writing.

6.3 System Information Block

The registers of the system information block contain valuable, static information about the hard- and firmware which is coded as (short) ASCII strings. The EPICS record of choice for reading these strings would be the *stringin* record.

6.4 ADC Waveforms

In this section we layout the design of the digitizer waveform support. Because *devBusMapped* does not support *waveform* records (or any other non-scalar record), a dedicated device-support module, henceforth “MWDS” for “MCOR *waveform* Device-Support”, has to be implemented.

Before we discuss the proposed design we shall recapture the currently implemented hardware features since [JJOv03] only provides scarce and outdated information.

6.4.1 Summary of Hardware Features

According to [JJOeml1], the waveform digitizer can be operated in different modes which affect the sample rate and the conditions which constitute a valid trigger event etc.

However, the fundamental data acquisition functionality is invariant and can be summarized as follows:

- There is a single bit available in a control register which “arms” the data acquisition.
- Data acquisition starts with the first valid trigger event after the digitizer is “armed” (in “soft-trigger” AKA “immediate” mode setting the “arm” bit *is* the trigger event.
- The “arm” bit self-clears when waveform capture is complete. An interrupt is raised at the same time.

- All triggers occurring after the one which initiated capture are ignored until the digitizer is "re-armed", i.e., the "arm" bit has undergone a new zero-to-one transition.
- There is no DMA hardware.

6.4.2 Proposed Device-Support Implementation

The trigger- and other mode settings can be controlled with *devBusMapped* just like other aspects of the MCOR. The MWDS only implements the "arming" of the digitizer, synchronization with the end of data capture and data transfer.

Consequently, the "arm" bit shall *not* be controlled by any other record via *devBusMapped* but shall be "owned" by MWDS. The database designer *must not* add any record to the database which is able to modify the "arm" bit. It is OK, however, to just read/monitor this bit.

Note that if the "arm" bit is located in a register which holds other bits which *are* controlled by *devBusMapped* then the MWDS must lock *devBusMapped*'s mutex while modifying the "arm" bit ([JJOeml1] does not elaborate on the exact register layout).

We propose to use the *waveform* record's `RARM` field to control (and reflect) the state of the "arm" bit.

Due to the indefinite time which elapses between "arming" the digitizer and termination of data capture the classical "asynchronous record-processing" method offers itself here.

1. User sets `RARM` field. As a result of this, the *waveform* record is processed.
2. "Phase-1 processing" sets the "arm" bit and returns.
3. The interrupt handler which is dispatched when data capture is complete requests "Phase-2 processing" of the record.

   If interrupts are unavailable then a dedicated task must poll the "arm" bit for completion of the data capture. "Phase-2 processing" could happen from the same task context.
4. During "Phase-2 processing" the `RARM` field is reset and data are transferred from digitizer memory to the *waveform* record's buffer in main memory.

   Data transfer *must* take care of byte-swapping when necessary and *may* optionally convert the samples to the format indicated by the *waveform*'s `FTVL` field.

   If format-conversion is not implemented then the MWDS *must* check `FTVL` and issue a fatal error if `FTVL` does not match the supported number format.

   Note that no DMA is available - data transfer has to use PIO.

Note that this MWDS design leaves the record *passive*. It is never directly scanned but processes (asynchronously) as a result of writing a nonzero value to `RARM`.

7 Error Handling

7.1 Errors During Database Initialization

Failure to initialize a database record due to missing hardware or syntax errors in a `INP` or `OUT` link field result in a error message printed to the console and `PACT` being set in order to prevent any record processing from happening.

7.2 Errors Reported by *devBusMapped*

devBusMapped sets the `STAT/SEVR` fields of a record to `READ_ALARM/INVALID_ALARM` or `WRITE_ALARM-/INVALID_ALARM` if a read or write error, respectively, is reported by the underlying bus-access method.

The predefined standard access methods (`be8`, `be8s`, `le16`, `le16s`, `le32`) *always* succeed. Therefore, the `STAT/SEVR` fields are never set when these methods are used.

7.3 Errors Reported by *waveform* Device-Support

No run-time errors can occur. MWDS does not modify `STAT/SEVR`.

8 Timing System Interface

MCOR device support implements an API which allows the user (not the "physicist/end-user" but the system engineer who uses the MCOR device support module from an IOC-application) to install a callback which provides the caller (MCOR device support) with a time-stamp. If the `TSE` field of the processing record is set to -2 then MCOR device support will execute this callback and propagate the supplied time-stamp into the record. *devBusMapped* may need to be enhanced in order to provide such a feature.

The function prototype of such a callback is

```
int (*DevBusMappedTSGet)(epicsTimeStamp *p_ts);
```

The return value of the callback shall be ignored. By default, `epicsTimeGetCurrent` is used.

A different callback may be installed by calling

```
DevBusMappedTSGet devBusMappedTSGetInstall(DevBusMappedTSGet newGetCallback);
```

e.g., from initialization code.

9 Feedback Interface

For sake of convenience we restate the function pointer prototype of the proposed "set-DAC" low-level API here:

```
typedef int (*MgntSetDAC)(void *card_p, int dac_channel, epicsInt32 dac_value);
```

The `card_p` is an opaque pointer which identifies a particular instance of DAC controller. In the case of MCOR, all DACs are part of the MCOR hardware. Since only a single instance of MCOR is supported by the software (see section 4) the DAC controller may be identified implicitly. The `card_p` argument is therefore unused. The caller must provide a `NULL` pointer (to ensure backwards compatibility if the semantics of this argument are ever changed).

The `dac_channel` parameter identifies one of the 16 channels controlled by a given MCOR. The valid range for this parameter shall span 0..15.

The `dac_value` parameter is written without further transformations to the "Set Point Requested" register.

The function shall return zero upon success and a nonzero status when an error occurs (e.g., invalid argument).

It shall be safe to call the function at any time but it may return an error when executed prior to driver initialization.

The function does not take the *devBusMapped* mutex for `mcor0.0`. It relies on exclusive access (and in any case a 32-bit write operation being atomic).

References

[JJOv03] J. Olsen, *MCOR, Version 0.4*, 6/25/1012.

[JJOeml1] J. Olsen, *private email*, 4/10/1012.