Created:August 26, 1994
Modified:November 13, 1997

Gemini Controls Group Interface Control Document

ICD 10 - EPICS Synchro Bus Driver

A.N.Johnson

ICD-10/02

This document describes the EPICS and vxWorks interfaces to the Gemini Synchro Bus.

1.0 Introduction

1.1 Purpose

This document describes the device driver software which interfaces a Gemini Standard Controller to the VMIVME-5588 Reflective Memory card. This software forms part of the Gemini Standard Instrument Controller workpackage, described in the Gemini document SPE-C-G0023[1].

Intended Readership:

- Gemini Software Developers
- Gemini Software and Controls Group

1.2 Scope

This document describes the EPICS Driver Support, EPICS Device Support and vxWorks software available to use with a VMIVME-5588 card.

The software provides communication between IOCs for the EPICS record types:

- Long records
- Analogue records
- String records
- Array records

This allows a set of suitably equipped IOCs to pass integers, floating-point numbers, strings and arrays via the Reflective Memory Bus.

Also provided is a subroutine-callable software interface, which allows applications to communicate over the bus without having to use EPICS records and thus permits much faster data transfers.

1.3 References

[1] SPE-C-G0023, *SIC WPD*, Gemini 8m Telescopes Project

[2] SPE-C-G0014, *Gemini Software Requirements Spec*, Gemini 8m Telescopes Project

[3] GSCG.grp.017, *Glossary*, Gemini 8m Telescopes Project

[4] 500-005588-000 *VMIVME-5588 Reflective Memory Board Product Manual*, VME Microsystems International Corporation

1.4 Definitions, Acronyms and Abbreviations

See document [3] *Glossary* for a complete list of acronyms and terms used in the Gemini ICDs.

1.5 Revisions

1. August 26, 1994; Andrew N. Johnson, Original release.
2. November 12, 1997; Bret Goodrich, Updated to VMIVME-5588.

2.0 General Description

This section gives an overview of the device driver software, putting it into context within the Standard Instrument Controller system as a whole.

2.1 Product Perspective

The Synchro Bus driver forms part of the Gemini Standard Instrument Controller work-package, and implements a fast, deterministic interface between several IOCs. The other new software components of the Standard Instrument Controller development are:

- Time Bus Driver
- Sample Control System

Of these, only the Sample Control System has any direct interaction with the Synchro Bus driver — it provides a demonstration and test facility for the other components of the SIC, and contains an EPICS database. Other drivers should not interact directly with the Synchro Bus driver in any way for the following reasons:

- All such interaction should be controlled via the EPICS database
- Some IOCs will not have any Synchro Bus hardware installed
- It is desirable to be able to create an EPICS system which does not load unnecessary drivers

2.2 Product Functions

The aim of the Synchro Bus driver development is to provide a means for passing information between several IOCs in a fast and semi-deterministic manner. This is implemented using the Reflective Memory system which is provided by the VMIC cardset. The record types to be supported by the EPICS driver and device layer software are as follows:

- Long records
- Analogue records
- String records
- Array records

The Array records provide a means of transferring several values across the bus in a synchronous fashion, and should be used in preference to several Analogue or Long records where data transfer must be synchronous. Alternatively synchronization can be achieved by using the page structure of the memory with I/O interrupt scanning on the input records.

Each Synchro Bus record provides a one-to-many broadcast over the bus, in that there can only be one source IOC for each record, but any number of IOCs can read the current value of the record. A protection mechanism is included within the driver to ensure that the data which a receiving IOC reads is self-consistent.

2.3 User Characteristics

Users of the driver will fall into one of the following categories:

- Work Package Developer
- Gemini Operations Maintenance staff

The characteristics of these users are described below

2.3.1 Work Package Developer

Gemini Applications Developer as described in [2] *SRS* section 4.3.4.1.

2.3.2 Gemini Operations Maintenance staff

As described in [2] *SRS* section 2.5.2.3.

2.4 General Constraints

The software development work which this specification describes shall conform to the general constraints and requirements set out in the Work Package Description for the Standard Instrument Controller. This in turn requires conformance to the Gemini Software Requirements Specification.

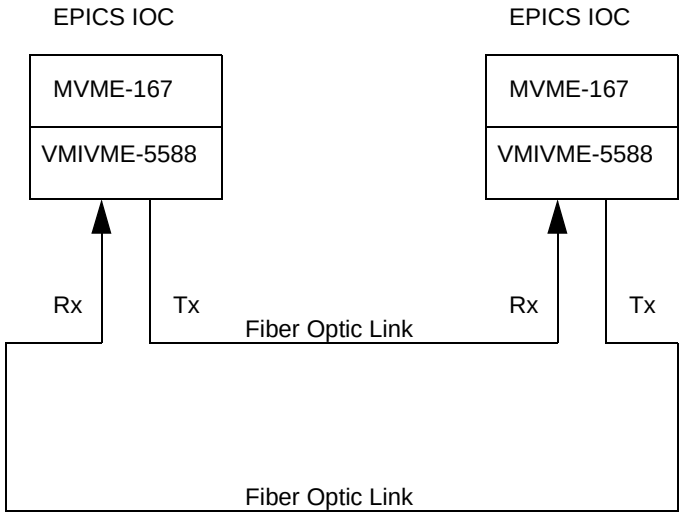
2.5 Assumptions and Dependencies

This specification has been based on the assumptions listed below. If any of these prove to be invalid, the relevant parts of this document can no longer be relied upon to accurately describe the requirements of the driver software.

2.5.1 Byte Order

It is assumed that all processors which will communicate over the Reflective Memory bus using this driver software will be based on the Motorola 680x0 family, or will at least all use the same byte-order addressing, and thus endian problems will be ignored.

FIGURE 1. Hardware Architecture



2.6 Communication Architecture

Synchro Bus IOCs communicate using EPICS database records or application-specific shared memory which interface through the VMIC VMIVME-5588 Reflective Memory system. This fibre-optic based distributed shared memory is referred to as the “Synchro Bus” in other Gemini documents.

3.0 Implementation: EPICS Interface

This section describes how the Synchro Bus facilities are implemented using EPICS database records and vxWorks procedure calls.

3.1 Reflective Memory Overview

The Synchro Bus is implemented using the facilities of the Reflective Memory provided by the VMIC card. This is constructed as a distributed block of memory such that each node keeps a copy of the whole memory contents. Reads from this memory can be satisfied immediately from the local copy. Writing to the memory updates the local copy and also causes the new data and its address to be sent out over the fibre-optic network and hence copied to all of the other nodes as well. FIFO buffers on the input and output fibre data allow local CPU accesses to take place without affecting the update information coming in from other nodes.

3.1.1 Memory Structure

The shared memory provided by the VMIC is divided by the EPICS driver software into 1 Kilobyte logical pages. This page structure allows the available storage space to be partitioned between several applications so that a change in one should not require modification to the others (for example if a new record is added). It also allows for convenient groupings of records for use with the I/O interrupt scanning as described below (Section 3.1.3).

3.1.2 Symbol Naming

All the Synchro Bus input and output records use the EPICS Instrument I/O address type to select the shared memory location which they use. This address consists of a single ASCII string, which is looked up in the Reflective Memory name database. This database is constructed by the IOC at boot time from a file which contains details of all the Synchro Bus records accessed by this IOC. The file is required to enable the IOC to convert any such name into an address in the shared memory space. The use of names in this way is much preferable to having “magic numbers” giving a page number and offset in the database.

3.1.3 Interrupts

In addition to passing write data around the fibre ring, the VMIC cards provide the ability for a node to cause an interrupt to occur in another node, or to be broadcast to all nodes. A total of three independent interrupt generators are supported by the hardware, one of which is reserved by the EPICS driver interface to support I/O Interrupt scanning of input records. The remaining two interrupts are available for use by applications via the C subroutine interface.

The I/O Interrupt scanning mechanism is closely related to the page structure of the memory, in that an interrupt is only generated when the first (“trigger”) output record on a page is processed. This interrupt causes processing to occur for all input records on that page in other IOCs which have their SCAN field set to I/O Interrupt. Thus by grouping related records on the same page, the processing of the “trigger” output record can cause a number of input records to process and thus read updated values from the shared memory. This architecture may not be desired in all situations however because of the database locksets in the receiving IOC — it may be preferable in some cases to

have a single input record set to I/O Interrupt and for that to FLNK to a processing chain containing the remaining input records. This is an application design decision.

3.1.4 Record Behaviour

The Reflective Memory architecture has the effect of permitting simultaneous read and write accesses to the shared memory contents at the speed of the local bus. However it also makes it possible for two CPUs to update the same location at the same time, with the result that the memory contents at this location can get out of synchronization — some nodes will end up with one value, others with the other, depending on their relative locations on the fibre ring. To prevent this confusion from occurring there must be a means of restricting when and/or where in the shared memory space a particular CPU is permitted to write data values.

In order to make the use of the Gemini Synchro Bus as flexible as possible this restriction is not enforced by the EPICS driver software at all. It is the responsibility of the applications designer to ensure that it is impossible for two IOCs to update the same Synchro Bus location simultaneously. Normally this will be achieved by only letting one IOC have an output record mapped to a particular location, but there are legitimate situations where it is desirable to have the same location mapped to output records in more than one IOC. In these cases the application must somehow determine when it is legal for a particular IOC to process its output record and update the Reflective Memory.

3.1.5 Atomic Record Update Protection

In any application where there are two asynchronous systems communicating, there must be a means for the two to be synchronised when they pass non-trivial amounts of information between themselves to prevent the receiving system from getting garbled data. For the Synchro Bus, this has been implemented by adding two protection fields to the shared memory structure for each record. While these fields contain identical values the contents of the record will not be modified.

When a new value is to be written by an output record, one of the protection fields is changed before the new value is inserted, and the second protection field is made identical to the first afterwards. To read a value from the shared memory, the second protection field is fetched first, the record value copied and then the first protection field is compared to the value fetched from the second field. If the two protection values were different, the record value which was read is suspect so the read is tried again from the beginning. This algorithm guarantees that the record value was not partly over-written in the period while it was being read.

3.1.6 Synchro Bus Status

In addition to the input and output record pairs which implement communications over the Synchro Bus, the driver contains binary input record support to allow various VMIC status bits to be accessed from the database. These provide information such as whether there is a signal on the input fibre, whether the fibre ring is broken, transfer errors and FIFO status.

3.1.7 Power-up Behaviour

Because the Synchro Bus is a highly distributed system such that each node keeps its own copy of the shared data, careful consideration must be made of the effects of turning an IOC off while the bus is in use. This has two main consequences:

- The fibre-optic ring will be broken while the IOC is turned off.
- When the IOC is turned back on again, its memory will be in a random state.

These effects must be known to both the applications designers and operations and maintenance staff as they place demands on both of these groups.

While the fibre ring is broken, data updates between the remaining IOCs will not be seen by any nodes beyond the break. If an IOC is to be switched off or removed for maintenance and continued operation of the rest of the system is desired, a bypass cable must be used to complete the connectivity of the ring. It is obviously desirable and may be essential that the applications software be told that a break is to be made in the fibre, to allow it to temporarily halt all use of the Synchro Bus ensure that no vital data are lost while the bypass is being fitted.

When an IOC is inserted into the ring and turned on, it will boot EPICS and initialise its Synchro Bus records, however its memory contents will be purely random at this point. Only later when the output records in the other nodes are processed and values written into the shared memory will the contents of the new node start to resemble those of the other nodes. Even then only the records which were processed since the new IOC was added will have the correct memory contents. For this reason the output records update all the shared information associated with the record whenever they are processed. If for example the type information was only updated when an output record was initialised then those IOCs switched on at a later moment would never see the initialisation so the type data would remain random. It is worth noting that the original purpose of the Synchro Bus was to handle fast, non-static data, and for this type of application the behaviour of the system is not restrictive.

It may be desirable for Synchro Bus output records to be processed at initialisation, because this causes their shared memory to be initialised, but care needs to be taken with trigger records — it may be undesirable to start off the processing chains in some remote systems. This issue must be considered by the database application designer.

3.2 Record Type Summary

Facility	Record Type	Direction
Array Input (any type)	aai	From Bus
Array Output (any type)	aao	To Bus
64-Bit Floating-Point Input	ai	From Bus
64-Bit Floating-Point Output	ao	To Bus
Bus Status Flags	bi	From VMIC
32-Bit Integer Input	longin	From Bus
32-Bit Integer Output	longout	To Bus
40 Character String Input	stringin	From Bus
40 Character String Output	stringout	To Bus

3.3 Name Database

All memory allocation within the Synchro Bus shared memory space must be done using the facilities provided in the driver. This uses the data from one or more symbol files to statically allocate memory for each of the requested data areas. After IOC initialisation the address of any symbol can be found with a single call to a driver routine.

3.3.1 Symbol File Syntax

One or more symbol files can be used to allocate memory for particular named symbols. The symbol file is created using a text editor, and should have the following format:

- Blank lines are ignored
- Comment lines must have a '**#**' in the first column
- Each symbol definition appears on a line of its own, and starts with one of the keywords (in lower case) **`page`**, **`analogue`**, **`long`**, **`string`**, **`array`** or **`user`**
- The keyword is followed by up to two parameters on the same line, separated by white space (spaces or tab characters)
- The first parameter is the symbol name, and is optional for the fixed-length keywords **`page`**, **`analogue`**, **`long`** and **`string`**
- Symbol names may contain any printable ASCII character excluding white space, although some may be illegal in record I/O address specifications.
- The second parameter is a short integer, and has different meanings for the different keywords (see below)
- The second parameter can only be present if a symbol name is provided

3.3.2 Page Definition

The **`page`** keyword starts a new 1 Kilobyte logical page (see Section 3.1.1 on page 5 for a discussion of the page structure). Pages can be named, although the name is not used at all by the driver software so it is optional. A name is required though if it is desired to declare a specific page number for this page, in which case the page number is given as the second parameter to the **`page`** keyword. Normally pages are numbered in sequence starting from page zero, but this can be overridden and pages declared out of order if

desired. Extreme care should be taken in these circumstances as there are no checks for duplicate use of page numbers with different or overlapping definitions. The first record declared after each **`page`** line will have a zero page offset and thus be used as the trigger record for I/O interrupts on that page.

If a record overlaps a page boundary, a warning message will be displayed during the IOC start-up script. It may be possible to operate even with this warning, although the page numbers allocated for later pages will be offset from those expected (unless explicit page numbers are used, in which case some records may be declared as overlapping others), and input records which appear after the overlap but on the same page will not be triggered by I/O Interrupt scanning.

```
page [name [npage]]
```

3.3.3 Fixed Length Records

The keywords **`analogue`**, **`long`** and **`string`** are used to declare storage for these EPICS record types. Analogue types are double-precision floating point, while longs are 32-bit integers. Strings have a fixed maximum length of 40 characters to include the terminating '\0'. A symbol name is optional with these keywords, in which case the storage space is allocated but no symbol defined. This facility allows a more flexible organisation of the shared memory, but it is probably wise to avoid the using it unless it proves to be essential for a particular application.

```
analogue [name]
long [name]
string [name]
```

3.3.4 Array Records

Array records provides the Synchro Bus with the ability to transfer a series of values of any simple scalar type (this includes char, short, long, float and double as well as 40-character strings). In order to make this as flexible as possible, the array type does not have to be indicated in the symbol file, although the storage space required for the array data must be given as the second parameter. A symbol name is mandatory with the **`array`** keyword. To calculate the second parameter, multiply the number of elements in the array by the size of each element as given in the following table. An additional record overhead is added automatically to the space allocated when the symbol is defined.

Value stored	FTVL field	Element size
8-bit signed integer	CHAR	1
8-bit unsigned integer	UCHAR	1
16-bit signed integer	SHORT	2
16-bit unsigned integer	USHORT	2
32-bit signed integer	LONG	4
32-bit unsigned integer	ULONG	4
32-bit floating point	FLOAT	4
64-bit floating point	DOUBLE	8

Value stored	FTVL field	Element size
40 character string	STRING	40
16-bit enumerated type	ENUM	2

```
array name nbytes
```

3.3.5 User Defined Storage

The keyword **`user`** helps provide the means by which user-defined subroutines can co-exist in the shared memory space with Synchro Bus records. It is used to declare storage space which can be used safely by C subroutines to pass information between IOCs without involving or affecting the operation of the record interface. The keyword syntax requires that both a symbol name and the number of bytes of storage required be provided as parameters to the **`user`** keyword. The use of the storage space thus allocated is completely free to the user-defined software. A pointer to this named block of shared memory can be obtained by passing the name string to the driver routine **`rmAddr`** with a symbol type of **`RM_TYPE_USER`**. For more details of the **`rmAddr`** routine and a further description of the C interface to the driver, see Section 4.0 on page 15 below.

```
user name nbytes
```

3.3.6 Loading the Name Database

The symbol file must be loaded into the IOC during the system start-up, and before the call to iocInit which initialises all the EPICS processes. The file is loaded by passing it as the standard input stream to the driver routine **`rmLoadSymbols`** as follows:

```
rmLoadSymbols < src/synchro.rms
```

In the above example, the symbol data is contained in a file called synchro.rms which is kept in the application's src subdirectory. It is possible to load more than one symbol file by adding extra calls to **`rmLoadSymbols`** in a similar manner. This allows unrelated applications to keep different symbol files, and for IOCs to load only those files they require. If this is done, it is recommended that each application be allocated a series of pages, and that each file starts with a page keyword giving the explicit page number to be used for these records.

`rmLoadSymbols` will report any errors it finds in the file as it reaches them, but will continue processing to the end of the file. It returns a zero status if no errors were found, otherwise it gives the EPICS error number relating to the last error it reported. The possible errors are:

Error	Message	Reason
S_dev_noMemory	Can't create RM symbol table	vxWorks memory full?
S_dev_badSignal-Number	Bad RM page number:	Page number is <0 or >255
S_dev_multDevice	Duplicate RM page symbol:	2 page symbols have been defined with the same name
S_dev_multDevice	Duplicate RM analogue symbol:	2 analogue symbols have the same name
S_dev_multDevice	Duplicate RM long symbol:	2 long symbols have the same name

Error	Message	Reason
S_dev_multDevice	Duplicate RM string symbol:	2 string symbols have the same name
S_dev_badRequest	Missing parameter(s) to RM: user	Name and/or nbytes parameters were missing for user symbol
S_dev_multDevice	Duplicate RM user symbol:	2 user symbols have the same name
S_dev_badRequest	Missing parameter(s) to RM: array	Name and/or nbytes parameters were missing for array symbol
S_dev_multDevice	Duplicate RM array symbol:	2 array symbols have the same name
S_dev_badRequest	Unrecognised RM type:	Symbol type keyword does not correspond to a valid type
S_dev_noMemory	RM Page overflow, page n	Too many symbols defined to fit in a single page

3.3.7 Displaying the Name Database

The name database can be displayed at any time using the routine **`rmPrintSymbols`** from the vxWorks shell. This lists each name defined along with its address and symbol type. Before iocInit the address will be shown only as an offset into the shared memory area. The type data is given as a number, as defined in the vmi5588.h header file.

3.3.8 Example

The following is an example symbol file, and is a portion of the symbol file which forms the acceptance tests for the Synchro Bus driver software:

```
# Pages 10 and 11 are for the Subroutine Record Tests

page Page_10 10
analogue SYM_ALOG
long     SYM_LONG
string   SYM_STRG
array    SYM_ARRY 8
user     SYM_USER1 1
user     SYM_USER2 2
user     SYM_USER3 3
user     SYM_USER4 4
user     vmi5588testStruct 40

page Page_11
user SYM_USER_BIG 1024
```

When just these symbols are loaded using rmLoadSymbols, the output from rmPrint-Symbols is

```
Name              Addr     Type
SYM_USER4        F0202C70  user
Page_11          F0203000  page
SYM_USER1        F0202C64  user
```

```
SYM_ARRY          F0202C4C   array
SYM_USER2         F0202C68   user
SYM_ALOG          F0202C00   analogue
SYM_LONG          F0202C10   long
SYM_USER3         F0202C6C   user
Page_10           F0202C00   page
SYM_USER_BIG      F0203000   user
SYM_STRG          F0202C1C   string
vmi5588testStru   F0202C74   user
```

3.4 Output Records

3.4.1 Purpose

A database programmer can create Synchro Bus analogue, long, string or array output records and set the output address specification (OUT field) to a Reflective Memory symbol name. When an output record of this type is processed, the output value (from the VAL field or via a DOL link) is written to the addressed shared memory location, and all reflective memory input records with the same location in connected IOCs will subsequently be able to read the new value. If the record is the trigger record for its logical page, an interrupt is also broadcast onto the Synchro Bus to notify the other IOCs of the page update and trigger any I/O Interrupt input records on the page.

The output record type must match the type declared for the symbol in the name database. This data type information is also maintained within the shared memory for each record, and is rewritten every time the record is processed.

3.4.2 Record Details

Record Type	DTYP field	OUT field	Comments
aao (array out)	VMI5588 Synchro Bus	*@name*	Arrays of any data type
ao (analog out)	VMI5588 Synchro Bus	*@name*	Double precision floating-point
longout	VMI5588 Synchro Bus	*@name*	32-bit integer
stringout	VMI5588 Synchro Bus	*@name*	String of up to 40 characters

3.4.3 Usage

The PINI field for most output records should probably be set to YES. This ensures that the shared memory contents including the data type are properly initialised on power-up. For some records however this may not be desirable, and special consideration is needed for trigger records — the effect of triggering interrupts on the remote IOCs needs to be assessed (see also the discussion in Section 3.1.7 on page 7).

On the subject of ensuring that all IOCs have valid copies of the Synchro Bus data, the applications engineer must also ensure that relatively static data is refreshed after a ring break if it might be needed by a new or repaired IOC. This is less likely to be a problem as slow data is unlikely to be transferred over the Synchro Bus, but there may be some circumstances where it is, so it may be worth having a processing chain which processes these records when the ring status goes to Good (see Section 3.6 on page 14). There is a

maximum data rate over the optical fibre which cannot be exceeded, so it is undesirable to refresh data periodically if it does not need doing.

Synchro Bus Output records do not support I/O Interrupt scanning — this is not a sensible option. An output record which has a zero address offset from its logical page (i.e. the first record on a page) will trigger I/O Interrupt scanning on all input records in other IOCs in the same page. Input records in the same IOC are not triggered (this would be a simple modification to the driver if it is required).

3.5 Input Records

3.5.1 Purpose

A database programmer can create Synchro Bus analogue, long, string or array input records and set the input address specification (INP field) to a Reflective Memory symbol name. Whenever the input record is processed, its VAL field will be updated with the current contents of the shared reflective memory. The record type must match the type declared for the symbol in the name database, and must also match the contents of the shared memory whenever the record is processed.

3.5.2 Record Details

Record Type	DTYP field	INP field	Comments
aai (array in)	VMI5588 Synchro Bus	*@name*	Arrays of any data type
ai (analog in)	VMI5588 Synchro Bus	*@name*	Double precision floating-point
longin	VMI5588 Synchro Bus	*@name*	32-bit integer
stringin	VMI5588 Synchro Bus	*@name*	String of up to 40 characters

3.5.3 Usage

It is generally not a good idea to have the PINI field of input records set to YES because it is unlikely that the contents of the local shared memory will have been initialised in the short time between turning the IOC on and it initialising. No harm will be done if PINI is set however, because the memory contents are checked before the record value is read — the input record value will be set to Undefined if the shared memory contents do not match the record type.

The input record value will also be set to undefined if another record overlaps it for any reason and the data type information in the shared memory does not match the input record type. In fact if there are record overlaps then other errors may also occur, the most likely being a device timeout with the error message “update protection count.” This is an indication that even though the shared record type field matches the record type correctly, the two protection fields did not become equal in ten attempts to read the record value. See Section 3.1.5 on page 6 for details of this.

If the Reflective Memory Fibre Ring is broken for some reason, this will not affect any of the Synchro Bus input or output records. They will continue to function, although the data will not be able to circulate correctly around all the nodes. An application should monitor the status of the Fibre Ring using a Status Input bi record as described below to halt data transfers and flag alarms as desired.

Synchro Bus Input records support I/O Interrupt Scanning, using the logical page as a means of having multiple independent interrupt sources. See Section 3.1.3 on page 5 for a more detailed discussion on I/O Interrupt scanning.

3.6 Status Input

3.6.1 Purpose

The VMIC Reflective Memory hardware provides a means to test the integrity of the fibre-optic cable under software control. In order to access this status information from the database, support is provided in the Synchro Bus driver to allow a number of binary input records to get this information. Each bi record monitors a single status bit from the reflective memory card. The status of the card is read every time one of these records is processed, and the relevant status bit returned. In all cases a status of zero is good. The status bit to be tested is selected by the string given in the record's INP field.

3.6.2 Record Details

Record Type	DTYP field	INP field	Comments
bi	VMI5588 Synchro Bus	@RM_NOSIG	No Input Fibre Signal
bi	VMI5588 Synchro Bus	@RM_NOSYNC	No PLL Synchronization
bi	VMI5588 Synchro Bus	@RM_RESYNC	Input Sync lost recently
bi	VMI5588 Synchro Bus	@RM_NORING	Fibre Ring broken
bi	VMI5588 Synchro Bus	@RM_BADXFR	Bad CRC on input packet
bi	VMI5588 Synchro Bus	@RM_TXHALF	Transmit FIFO half full
bi	VMI5588 Synchro Bus	@RM_RXHALF	Receive FIFO half full
bi	VMI5588 Synchro Bus	@RM_IRQ1	Device Support Int. pending
bi	VMI5588 Synchro Bus	@RM_IRQ2	User Interrupt #1 pending
bi	VMI5588 Synchro Bus	@RM_IRQ3	User Interrupt #2 pending

3.6.3 Usage

Each different Status Input record address returns the value of the named bit as defined for the **`rmStatus`** user subroutine. If an application contains both subroutine calls to **`rmStatus`** and Status records, the developer must be aware that the RM_RESYNC and RM_BADXFR status bits will be cleared by a Status record which addresses them, and so these status values may be missed by the subroutine status check. Similarly if the subroutine clears these bits, they will not be accurately reported by the appropriate Status record.

A "bad status" value from the VMIC card will only raise an alarm if the bi record has been set up to do so by setting the OSV field to the required alarm severity.

I/O Interrupt scanning is not supported for Status Input records — use periodic processing at a fast scan rate to obtain the maximum status information. In general however a 1 second scan should normally be sufficient for most applications.

4.0 Implementation: C Subroutine Interface

4.1 Purpose

A subroutine interface is provided which can be used simultaneously with the database records and provides the ability to avoid the EPICS record processing overheads and so obtain the best performance from the medium. The interface subroutines available are described below.

When using this low-level interface, all responsibility for handling memory initialisation, ring break recovery and synchronization between CPUs must lie with the applications developer — the discussion of these aspects in Section 3.1 should be studied to gain appreciation into the behaviour of the Reflective Memory. The effects of accessing memory outside of the areas reserved for user-defined shared storage are not guaranteed.

4.2 Header File

The header file associated with the Reflective Memory C subroutine interface is distributed as part of the Gemini EPICS distribution in the base/include directory and is named vmi5588.h. This declares all the routines listed below, and the data type and status bit preprocessor symbols required. Note that the file also contains some data structures and routines which are intended for private use by the EPICS driver software — only the interfaces described in this document should be used by user-level code.

4.3 Name Database

The first two routines here have already been described in Section 3.3 on page 8. These load the Name Database from a symbol file and print the current database respectively. The third routine is called to obtain a pointer to a named shared memory user area.

4.3.1 rmLoadSymbols

- Definition

  ```
  long rmLoadSymbols(void);
  ```

- Inputs

  Standard Input (symbol file)

- Outputs

  Return status, 0=good, or S_dev_* error number

Reads a symbol file from the standard input file descriptor and creates a Name Database (or appends to the existing one) from its contents. See Section 3.3 on page 8 for a description of the symbol file syntax.

4.3.2 rmPrintSymbols

- Definition

  ```
  void rmPrintSymbols(void);
  ```

- Inputs

None

- Outputs

  Standard Output (symbols)

Prints the contents of the Name Database. For each symbol, the symbol name and its address and type are displayed. If the IOC has already been initialised the address given is the local machine address, otherwise the value given is an offset into the shared memory structure (note this is not just an offset from the base address of the vmi5588 card).

4.3.3 rmAddr

- Definition

```
void *rmAddr(char *pname, int rmType);
```

- Inputs

  Symbol name

  Symbol type (RM_TYPE_USER)

- Outputs

  Returns pointer to shared memory location, or NULL

Looks up a symbol name in the Name Database and returns its associated physical address within the shared Reflective Memory card. The symbol type argument rmType must match the type definition in the database — for the purposes of the low-level user interface routines this argument should always be RM_TYPE_USER, a preprocessor macro defined in the header file which returns the symbol type for all user symbols.

4.4 Hardware Status

Two routines provide information about the vmi5588 card and its settings. The information provided by the status call can also be obtained via the database. The node ID provides the local node address, which is a means of uniquely identifying each vmi5588 card on the ring and is required for directed interrupts. These routines will return an error status if called before iocInit or if there is no card present in the system.

4.4.1 rmStatus

- Definition

```
long rmStatus(long reset);
```

- Inputs

  Status bit-pattern to clear

- Outputs

  Returns vmi5588 card status bit-pattern, or S_dev_* error number

Reads vmi5588 card status and returns a bit-pattern comprising 10 status bits. The values for these status bits are defined as preprocessor macros in the header file with the following names and meaning:

Macro Name	Meaning when bit set
RM_NOSIG	No Signal from Input Fibre
RM_NOSYNC	Input PLL not synchronised
RM_RESYNC	Input PLL Synchronisation lost recently (latched)
RM_NORING	Fibre Ring broken
RM_BADXFR	Bad CRC on incoming packet, data lost (latched)
RM_TXHALF	Transmit FIFO half full
RM_RXHALF	Receive FIFO half full
RM_IRQ1	EPICS Device Support Interrupt pending
RM_IRQ2	User Interrupt #1 pending
RM_IRQ3	User Interrupt #2 pending

The input parameter can be used to clear one or both of the latched status bits by ORing together the relevant status bit values. For example,

```
stat = rmStatus(RM_RESYNC);
```

reads the card status into stat and clears the RM_RESYNC bit if set, but does not clear the RM_BADXFR bit. If the value of the status returned is outside the range 0 to 65535 inclusive, the value is an S_dev_* error number, implying the card cannot be accessed.

4.4.2 rmNodeID

- Definition

  ```
  long rmNodeID(void);
  ```

- Inputs

  None

- Outputs

  Returns Node number of this VMI5588 card, or S_dev_*

Returns the node address of the vmi5588 card, a number in the range 0 to 255 inclusive.

4.5 Interrupts

The vmi5588 card provides 4 interrupt channels numbered 0 to 3, of which channel 1 is used by the EPICS driver software to implement page triggering. Channels 2 and 3 are available for user software to pass information between nodes. Channel 0 does not provide communication between IOCs but generates interrupts in the event of data transfer errors or the input or output FIFOs becoming full.

Three routines provide access to the vmi5588 interrupt system, allowing user-level subroutines to send and receive notification of events over the bus. Both broadcast and node-specific interrupts are supported. These routines can only be used after iocInit; an error status will be returned if called before the driver has been initialised or if there is no vmi5588 card present in the system.

4.5.1 rmIntSend

- Definition

```
long rmIntSend(int irqNumber, int nodeId);
```

- Inputs

  Interrupt Number (2 or 3)

  Node number to receive interrupt, or -1 for broadcast

- Outputs

  Return status, 0=good, or S_dev_* error number

Sends an Interrupt to the vmi5588 node given by the nodeId. If a broadcast interrupt is desired, the value -1 should be used for the node number. Two independent interrupts are available which can be used by different applications, or for two different meanings in a single application.

4.5.2 rmIntConnect

- Definition

```
long rmIntConnect(int irqNumber, VOIDFUNCPTR proutine);
```

- Inputs

  Interrupt Number (0, 2 or 3)

  Pointer to user Interrupt Service Routine

- Outputs

  Return status, 0=good, or S_dev_* error number

Connects incoming vmi5588 interrupts on the channel given by irqNumber to a user-defined subroutine which must be declared as returning void and taking a single parameter which contains the node number of the interrupt source:

```
void proutine(int srcNode)
```

This routine must follow the vxWorks rules for interrupt service routines, which restrict the OS calls which can be made and prohibit busy-waiting for example. EPICS permits some of its routines to be used within ISRs — see the EPICS IOC Application Developer's Guide for details. The user routine is actually called by a wrapper routine within the driver which reinitializes the interrupt hardware after the user routine has returned.

4.5.3 rmIntDisconnect

- Definition

```
long rmIntDisconnect(int irqNumber);
```

- Inputs

  Interrupt Number (0, 2 or 3)

- Outputs

  Return status, 0=good, or S_dev_* error number

Disconnects the user Interrupt Service Routine on the given interrupt channel number.

5.0 Performance and Usage

As part of the Acceptance Testing for the Synchro bus software, several performance measurements were made and statistically averaged over a large number of readings. The tests are described in more detail in the Synchro Bus Software Test Specification, and the data obtained give

- Output record to Input record latency between 2 SNL programs
- C subroutine interrupt latency
- C subroutine data transfer rate

The results measured are summarised in the following tables:

TABLE 1. Latency, SNL to Trigger output record to I/O Interrupt input record to SNL

Measurement (μs)	100 cycles	500 cycles
Min. latency, A to B		
Min. latency, B to A		
Max. latency, A to B		
Max. latency, B to A		
Mean latency, A to B		
Mean latency, B to A		
RMS, A to B		
RMS, B to A		

TABLE 2. Latency, C subroutine rmIntSend to C subroutine user ISR

Measurement (μs)	100 cycles	500 cycles
Min. latency, A to B		
Min. latency, B to A		
Max. latency, A to B		
Max. latency, B to A		
Mean latency, A to B		
Mean latency, B to A		
RMS, A to B		
RMS, B to A		

TABLE 3. Time for C subroutine to memcpy 1 Kbyte of data into Reflective Memory

Measurement (μs)	100 cycles	500 cycles
Min. time		
Max. time		
Mean time		
RMS time		
Data Rate (Mb/s)		

A comparison between Tables 1 and 2 above shows the penalty involved in using EPICS records, the State Notation Language and Channel-Access to transfer data via the Synchro bus. The time delays involved here are such that the Synchro Bus gives a significant advantage to using Channel-Access over an Ethernet, but that for the highest performance applications it is essential to utilise the C subroutine interface. The original performance requirement placed on the Synchro Bus driver software was:

> A series of 10 32-bit floating point numbers representing Zernike polynomial coefficients shall be able to be transmitted between two IOCs at an update rate of 200Hz, with a maximum latency of 200 microseconds from source to destination records. In addition to this, the goal for the timing jitter of this transmission should be 20 microseconds.

It is clear from the above results that this requirement can be reached easily by using the C subroutine interface, but not at all from the EPICS record support.