EVR Usage Guide

Michael Davidsaver <mdavidsaver@gmail.com>

August. 2017, Rev. 8

Contents

1 What is Available?

More infomation on the Micro Research hardware can be found on their website http://www.mrf.fi/.

The software discussed below can be found on the EPICS application project on SourceForge http://sourceforge.net/projects/epics/.

The latest developments can be found in the 'mrfioc2' Git VCS repository.

https://github.com/epics-modules/mrfioc2

1.1 Prerequisites

Build system required modules

EPICS Base >= **3.14.10** EPICS Core
http://www.aps.anl.gov/epics/base/R3-14/index.php

MSI Macro expansion tool (Base <3.15.0 only)
http://www.aps.anl.gov/epics/extensions/msi/index.php

devLib2 >= **2.9** PCI/VME64x Hardware access library
https://github.com/epics-modules/devlib2/

Build system optional modules. Not required, but highly recommended.

autosave Automatic save and restore on boot
http://www.aps.anl.gov/bcda/synApps/autosave/autosave.html

iocstats Runtime IOC statistics (CPU load, . . .)
http://www.slac.stanford.edu/comp/unix/package/epics/site/devIocStats/
http://sourceforge.net/projects/epics/files/devIocStats/

Target operating system requirements

RTEMS >= 4.9.x

vxWorks >=6.7

Linux >= 3.2.1 (earlier versions may work)

1.2 Source

VCS Checkout

```
$ git clone https://github.com/epics-modules/mrfioc2.git
```

Edit 'configure/CONFIG_SITE' and 'configure/RELEASE' then run "make".

The following is a brief tour of the important locations in the source tree relating to the EVR.

1.3 Supported Hardware

The following devices are supported.

Name	# FP[a]	# FP UNIV[b]	# FP Inputs[c]	RTM[d]
VME-EVR-230[e]	4	4	2	Yes
VME-EVR-230RF	7[f]	2	2	Yes
PMC-EVR-230	3	0	1	No
CPCI-EVR-230	0	4	2	Yes[g]
cPCI-EVRTG-300	2[h]	2	1[i]	No
cPCI-EVR-300	0	12	2	0
PCIe-EVR-300DC	0	0	0	16
mTCA-EVR-300[j]	4	4/0	2	0/16

[a] Front panel outputs (TTL)
[b] Front panel universal output sockets
[c] Front panel inputs
[d] Supports Rear Transition Module
[e] This device has not been tested
[f] Outputs 4,5,6 are CML
[g] Supports PCI side-by-side module
[h] GTX outputs
[i] Special GTX interlock
[j] Two hardware flavors exist, one with 2x UNIV I/O sockets, the other with an IFB-300 connector,

2 System Overview

The purpose of this document is to act as a guide and reference when using the 'mrfioc2' EPICS support module for the Micro Research Finland (MRF) timing system[1]. It describes software for using the Event Generator (EVG) and Event Receiver (EVR).

The MRF Event Timing System can be deployed in two configurations (Fig. 1). The first is a unidirectional broadcast from a single source (EVG) to multiple

[1] List of supported hardware given in section 1.3.

destinations (EVRs). The Repeater devices simply retransmit its single input to all outputs (one to many). In the second configuration a return path from many EVRs back up to single central (master) EVR is added.

An EVR will act in one of two roles: either Leaf or Master. The Master EVR is necessary because, while the generator (EVG) is capable of receiving an event stream, it does not impliment the features of the receiver (EVR).

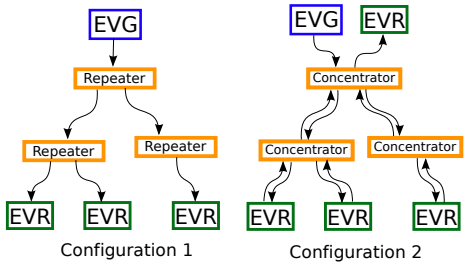

Figure 1: Two system configurations for the MRF Timing System

What is transmitted over the event link is a combination of 8-bit event codes and data. Data can take the form of a single 8-bit byte which is simply copied from sender to receiver (the Distributed Bus or DBus), and optionally a variable length byte array (Data Buffer).

These two types of data can be combined in two ways (Fig. 2) depending on whether or not the Data Buffer feature is used. In configuration A every 16-bit frame is split between an 8-bit event and the 8-bit Distributed Bus. In configuration B every frame carries an 8-bit event with the Distributed Bus or a Data Buffer byte sent in alternating frames.

In addition to data, the use of 8b10b encoding on the event link allows the local oscillator of each EVR to be phase locked to a reference sent by the EVG. The EVG itself is typically driven from an external oscillator.

When discussing the MRF timing system there are three clocks. The external reference clock for the EVG, the bit clock for transceivers, and the Event Clock. The relation between the reference and the Event clocks is determined by a programmable divider in the EVG and is usually a small integer number (eg. 4). The Event clock must be in the range between 50MHz and 125MHz. The

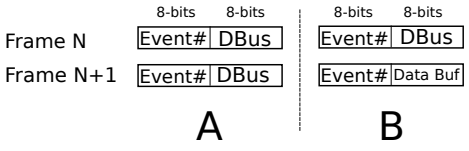

Figure 2: Two supported link allocation schemes

relation between the Event clock and the bit clock is a fixed factor of 20 which is determined by the frame size described above.

$$F_{bit}/20 = F_{Event} = F_{Ext}/N_{Divide}$$

2.1 Event Link Data

Data which is transferred over the event link is interpreted in four ways: Event Codes, DBus bits, Data Buffers, and Clock Phase. Each carries a different meaning, and is used in different ways.

2.1.1 Event Codes

An event is momentary. Typically an event causes something to happen (a trigger). The 255 usable event codes available in the MRF system can be thought of as 255 seperate physical wires. On every tick of the Event Clock a pulse is sent on one (and only one) of the "wires". Zero is the "idle" event which is sent when no other event is queued.

Event Codes will most often be used as triggers for external delay channels. However, there are a number of event codes which have special meaning in the MRF system. The meaning of all other codes is left to the system operator.

Code	Meaning
0x00	Idle, or null, event. Send when nothing happens.
0x70	Shift 0 into EVR timestamp shift register
0x71	Shift 1 into EVR timestamp shift register
0x7A	Reset EVR heartbeat timeout counter
0x7B	Reset all EVR dividers. Synchronize global phase
0x7C	Increment EVR timestamp counter (depending on mode)
0x7D	Reset timestamp counter
0x7F	End of sequence (not transmitted). Use in other contexts is discouraged.

Table 1: Special Event codes

2.1.2 Distributed Bus (DBus) bits

The Distributed Bus (DBus) consists of 8 bits of data which are stored on every EVR. This data is initialized to zero when the EVR starts, and overwritten whenever the EVR receives an event frame with DBus data. Depending on configuration this is either every frame, or every second frame (See fig. 2).

The DBus can thus be used to distribute either periodic, or non-periodic, signals with bandwidth up to $\frac{1}{2}$ (or $\frac{1}{4}$) of the Event clock.

The bits of the DBus can be routed to physical output. A special feature of DBus bit 4 allows its rising edge to increment the timestamp counter (depending on mode).

2.1.3 Data Buffers

When enabled, a protocol is used to broadcast arbitrary byte arrays from the EVG to all EVRs. Bytes are sent one at a time in the data part of every second frame. Special 8b10b codes are used to mark the beginning and end for each transfer. A simple checksum is also sent. The 230 series hardware allows buffers up to 2047 bytes in length.

In keeping with the convention of the original MRF EPICS Support package the first byte of a buffer is used as a header (Protocol ID) to identify it. No restrictions are placed on the body of buffer.

2.1.4 Event Clock Phase

The use of 8b10b encoding allows each EVR's local oscillator to lock to the EVG's reference clock. This allows operation at speeds higher then the event clock rate. This is used by the CML outputs described in section 3.5.

2.2 Global Time Distribution

The model of time implemented by the MRF hardware is two 32-bit unsigned integers: counter, and "seconds". The counter is maintained by each EVR and incremented quickly. The value of the "seconds" is sent periodically from the EVG at a lower rate.

During each "second" 33 special codes (see sec. 1) must be sent. The first 32 are the shift 0/1 codes which contain the value of the next "second". The last is the timestamp reset event. When received this code transfers the new "second" value out of the shift register, and resets the counter to zero. These actions start the next "second".

Note that while it is referred to as "seconds" this value is an arbitrary integer which can have other meanings. Currently only one time model is implemented, but implementing others is possible.

2.2.1 Light Source Time Model

In this model the "seconds" value is an actual 1Hz counter. The software default is the POSIX time of seconds since 1 Jan. 1970 UTC. Each new second is started with a trigger from an external 1Hz oscillator, usually a GPS receiver. Most

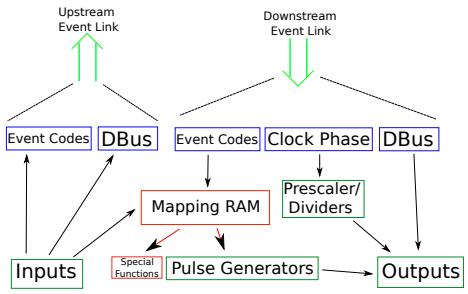

Figure 3: Logical connections inside an EVR

GPS receivers have a one pulse per second (PPS) output. Time is converted to the EPICS epoch (1 Jan. 1990) for use in the IOC.

Several methods of sending the seconds value to the EVG are possible:

External hardware has been created by Diamond light source which communicates with a GPS receiver over a serial (RS232) link to receive the timestamp and connects to two external inputs on the EVG. These inputs must be programmed to send the shift 0/1 codes.

Time from an NTP server can be used without special hardware. This requires only a 1Hz (PPS) signal coming from the same source as the NTP time. Several commerial vendors supply dedicated NTP servers with builtin GPS receivers and 1Hz outputs. A software function is provided on the EVG which is triggered by the 1Hz signal. At the start of each second it sends the next second (current+1), which will be latched after the following 1Hz tick.

3 Receiver Functions

Internally an EVR can be thought of as a number of logical sub-units (Fig. 3) connecting the upstream and downstream event links to the local inputs and outputs. These sub-units include: the Event Mapping Ram, Pulse Generators, Prescalers (clock dividers), and the logical controls for the physical inputs and outputs.

3.1 Pulse Generators

Each pulse generator has a an associated Delay, Width, Polarity (active low/high), and (sometimes) a Prescaler (clock divider). When triggered by the Mapping

Ram it will wait for the Delay time in its inactive state. Then it will transition to its active state, wait for the Width time before transitioning back to its inactive state.

Resolution of the delay and width is determined by the prescaler. A setting of 1 gives the best resolution.

In addition, the Mapping Ram can force a Pulse Generator into either state (Active/Inactive).

Note: Some Pulse Generators do not have a prescaler. In this case the prescaler property will always read 0 instead of >=1.

3.2 Event Mapping Ram

The Event Mapping Ram is a table used to define the actions to be taken by an EVR when it receives a particular event code number. The mapping it defines is a many-to-many relations. One event can cause several actions, and one action can be caused by several events.

The actions which can be taken can be grouped into two catagories: Special actions, and Pulse Generator actions. Special actions include those related to timestamp distribution, and the system heartbeat tick (see § 9.9.2 on page 41 for a complete list). Each Pulse Generater has three mapable actions: Set (force active), Reset (force inactive), and Trigger (start delay program). Most applications will use Trigger mappings.

3.3 Prescalers (Clock Divider)

Prescaler sub-units take the EVR's local oscillator and output a lower frequency clock which is phased locked to the local clock, which is in sync with the global master clock. The lower frequency must be an integer divisor of the Event clock.

To provide known phase relationships, all dividers can be synchronously reset when a mapped event code is received. This is the Reset PS action. See 9.9.2 on page 41.

3.4 Outputs (TTL)

This sub-unit represents a local physical output on the EVR. Each output may be connected to one source: a Distributed Bus bit, a Prescaler, or a Pulse Generator (see § 9.6.1 on page 36 for a complete list).

3.5 Outputs (CML and GTX)

Current Mode Logic outputs can send a bit pattern at the bit rate of the event link bit clock (20x the Event Clock). This pattern may be specified in one of three possible ways.

As four 20 bit sub-patterns (rising, high, falling, and low). As two periods (high and low). These specify a square wave with variable frequency and duty factor. As an arbitrary bit pattern (<= 40940 bits) which begins when the output goes [TODO: high or low?].

In the sub-pattern mode. The rising and falling patterns are transmitted when the output level changes, while the high and low patterns are repeated in between level changes.

The GTX outputs found only on the EVRTG (e^-gun) receiver function similarly to the CML outputs at twice the frequency. Thus for this device patterns are 40 bits.

3.6 Inputs

An EVR's local TTL input can cause several actions when triggered. It may be directly connected to one of the upstream Distributed Bus bits, it may cause an event to be sent on the upstream links, or applied to the local Mapping Ram.

The rising edge of a local input can be timestamped.

3.7 Global Timestamp Reception

Each EVR receives synchronous time broadcasts from an EVG. Software may query the current time at any point. The arrival time of certain event codes can be saved as well. This can be accomplished with the 'event' record device support.

Each EVR may be configured with a different method of incrementing the timestamp counter. See section 9.1.12.

In addition to being slaved to an EVG, those EVR models/firmware which provide a Software Event transmission function can send timestamps as well. This can be used to simulate timestamps in a standalone environment such as a test lab. see the TimeSrc property in 9.2 on page 33.

TimeSrc=0 The default, which disables EVR timestamp generation.

TimeSrc=1 In External mode the EVR will send a timestamp when event 125 is received. Reception of 125 can be either from an input, or for DC EVRs the sequencer.

TimeSrc=2 In Sys Clock mode, the EVR will generate a software 125 event based on the system clock. This is the simplest standalone mode.

3.8 Data Buffer Tx/Rx

A recipient can register callback functions for each Protocol ID. It will then be shown the body of every buffer arriving with this ID.

A default recipient is provided which stores data in a waveform record.

4 IOC Deployment

This section outlines a general strategy for adding an EVR to an IOC. First general information is presented, followed by a section describing the extra steps needed to use mrfioc2 under Linux.

An example IOC shell script is included as "iocBoot/iocevrmrm/st.cmd".

4.1 Device names

All EVGs and EVRs in an IOC are identified by an unique name. This is first given in the IOC shell functions described below, and repeated in the INP or OUT field of all database records which reference it. Both EVGs, and EVRs share the same namespace. This restriction is needed since some code is shared between these two devices.

4.2 VME64x Device Configuration

The VME bus based EVRs and EVGs are configured using one of the following IOC shell functions.

```
# Receiver
mrmEvrSetupVME("anEVR", 3, 0x30000000, 4, 0x28)
```

In this example EVR "anEVR" is defined to be the VME card in slot 3. It is given the A32 base address of 0x30000000 and configured to interrupt on level 4 with vector 0x28.

Note: VME64x allows for jumpless configuration of the card, but not automatically assignment of resources. Selection of an unused address range and IRQ level/vector is necessarily left to the user.

Note: Before setup is done the VME64 identifer fields are verified so that specifying an incorrect slot number is detected and setup will safely abort.

4.3 PCI Device Configuration

PCI bus cards are identified with the mrmEvrSetupPCI() IOC shell function.

Since PCI devices are automatically configured only the geographic address (bus:device.function) needs to be provided. This information can usually be found at boot time (RTEMS) or in /proc/bus/pci/devices (Linux).

The IOC shell function devPCIShow() is also provided to list PCI devices in the system.

```
# Receiver
mrmEvrSetupPCI("PMC", "1:2.0")
```

This example defines EVR "PMC" to be bus 1 device 2 function 0.

Support for using mTCA slot number is available on some targets (Linux only as of devlib2 2.9). This does any automatic lookup of PCI address from slot number. Be aware that PCIe "slot" numbers, while stable across reboots, may change with hardware configuration, firmware, or OS upgrades.

```
mrmEvrSetupPCI("PMC", "slot=5")
```

Note: Before setup is done the PCI identifer fields are verified so that specifying an incorrect location is detected and setup will safely abort.

4.4 PCI Setup in Linux

In order to use PCI EVRs in the Linux operating system a small kernel driver must be built and loaded. The source for this driver is found in 'mrmShared/linux/'. This directory contains a Makefile for use by the Linux kernel build system (not EPICS).

To build the driver you must have access to a configured copy of the kernel source used to build the target system's kernel. If the build and target systems use the same kernel, then the location will likely be '/lib/modules/`uname -r`/build'. In case of a cross-built kernel the location will be elsewhere.

To build the module for use on the host system:

```
$ make -C /location/of/mrmShared/linux \
KERNELDIR=/lib/modules/`uname -r`/build modules_install
$ sudo depmod -a
$ sudo modprobe mrf
```

Building for a cross-target might look like:

```
$ make -C /location/of/mrmShared/linux \
KERNELDIR=/location/of/kernel/src \
ARCH=arm CROSS_COMPILE=/usr/local/bin/arm- \
INSTALL_MOD_PATH=/location/of/target/root \
modules_install
```

Once the module is installed on the running target the special device file associated with each EVR must be created. If your target system is running UDEV this will happen automatically. See mrmShared/linux/README for example UDEV config. If UDEV is not present, then you must do the following.

```
# grep mrf /proc/devices
254 mrf
# mknod -m 666 /dev/uio0 c 254 0
```

If may be necessary to change the file permission to allow the IOC process to open it. UDEV users may find one of the following commands useful for constructing a rules file.

```
# udevinfo -a -p $(udevinfo -q path -n /dev/uio0 )

# udevadm info -a -p $(udevadm info -q path -n /dev/uio0 )
```

Each additional device adds one to the number (uio1, uio2, ...).

Once the device file exists with the correct permissions the IOC will be able to location it based on the bus:device.function given an to mrmEvrSetupPCI().

Note: UIO numbers are not considered during setup since these may change after a reboot. To ensure repeatability only PCI immutable ID fields, PCIe "slot" numbers, the address triplet (bus:device.function) are used.

4.5 Example Databases

The MRFIOC2 module includes example database templates for all supported devices (see §1.3). While each is fully functional, it is expected that most sites will make modifications. It is suggested that the original be left unchanged and a copy be made with the institute name and other information as a suffix. (evr-pmc-230.substitutions becomes evr-pmc-230-nsls2.substitutions).

The authors would like to encourage users to send their customized databases back so that they may be included as examples in future releases of MRFIOC2.

The templates consist of a substitutions file for each model (PMC, cPCI, VME-RF). This template instanciates the correct number of records for the inputs/outputs found on each device. It also includes entries for event mappings and database events which will be frequent targets for customization.

Each substitutions file will be expanded during the build process with the MSI utility to create a database file with two undefined macros (P and C). 'SYS' and 'D' define a common prefix shared by all PVs and must be unique in the system. 'EVR' is a card name also given as the first argument of one of the mrmEvrSetup*() IOC shell functions (unique in each IOC).

Thus an IOC with two identical VME cards could use a configuration like:

```
mrmEvrSetupVME("evr1",5,0x20000000,3,0x26)
mrmEvrSetupVME("evr2",6,0x21000000,3,0x28)
dbLoadRecords("evr-vmerf-230.db", "SYS=test,D=evr:a,EVR=evr1")
dbLoadRecords("evr-vmerf-230.db", "SYS=test,D=evr:b,EVR=evr2")
```

4.5.1 autosave

All example database files include "info()" entries to generate autosave request files. The example IOC shell script "iocBoot/iocevrmrm/st.cmd" includes the following to configure autosave.

```
save_restoreDebug(2)
dbLoadRecords("db/save_restoreStatus.db", "P=mrftest:")
save_restoreSet_status_prefix("mrftest:")

set_savefile_path("${mnt}/as","/save")
set_requestfile_path("${mnt}/as","/req")
```

This enables some extra debug information which is useful for testing, and loads the autosave on-line status database. It also sets the locations where .sav and .req files will be searched for.

```
set_pass0_restoreFile("mrf_settings.sav")
set_pass0_restoreFile("mrf_values.sav")
set_pass1_restoreFile("mrf_values.sav")
set_pass1_restoreFile("mrf_waveforms.sav")
```

Sets three files which will be loaded. The "values" are loaded twices as is the autosave convention.

```
iocInit()

makeAutosaveFileFromDbInfo("as/req/mrf_settings.req", "autosaveFields_pass0")
makeAutosaveFileFromDbInfo("as/req/mrf_values.req", "autosaveFields")
makeAutosaveFileFromDbInfo("as/req/mrf_waveforms.req", "autosaveFields_pass1")
```

After the IOC has started the request files are generated. This is where the "info()" entries in the database files are used.

```
create_monitor_set("mrf_settings.req", 5 , "")
create_monitor_set("mrf_values.req", 5 , "")
create_monitor_set("mrf_waveforms.req", 30 , "")
```

Finally the request files are re-read and monitor sets are created.

5 Testing Procedures

This section presents several step by step procedures which may be useful when testing the function of hardware and software.

In the "documentation/demo/" directory several IOC shell script files with the commands given in this section as well as other examples.

5.1 EVG and EVR Checkout

This procedure requires both a generator, receiver, and a fiber jumper cable to connect them.

It is assumed that no cables are connected to the front panel of either EVG or EVR. The example "iocBoot/iocevrmrm/st.cmd" script is used with `SYS=TST` and `D=evr` for the receiver and `D=evg` for the generator. Verify this with the following commands at the IOC shell.

```
>dbgrep("*Link:Clk-SP")
TST{evr}Link:Clk-SP
>dbgrep("*FracSynFreq-SP")
TST{evg-EvtClk}FracSynFreq-SP
```

The following examples use the IOC shell commands `dbpr()` and `dbpf()`. Remote use of `caput` and `caget` is also possible.

```
>dbpf("TST{evg-EvtClk}Source-Sel","FracSyn")
>dbpf("TST{evg-EvtClk}FracSynFreq-SP","125.0")
>dbpf("TST{evr}Link:Clk-SP","125.0")
>dbpf("TST{evr}Ena-Sel","Enabled")
>dbpr("TST{evr}Link-Sts")
...
... VAL: 0
```

This sets the event link speed on both the EVR and EVG. The EVG is commanded to use its internal synthesizer instead of an external clock.

Now use the fiber jumper cable to connect the TX port of the generator to the RX port of the receiver. (The Tx port will have a faint red light coming from it).

Once connected the red link fail LED should go off and the link status PV should read OK (1).

```
>dbpr("TST{evr}Link-Sts")
...
... VAL: 1
```

At this point the receivier has locked to the generator signal, but no data is being sent. This includes the heartbeat event. Thus the heartbeat timeout counter should be increasing.

```
>dbpr("TST{evr}Cnt:LinkTimo-I")
...
... VAL: 45
>dbpr("TST{evr}Cnt:LinkTimo-I")
...
... VAL: 47
```

Now we will set up the generator to send a periodic event code.

```
>dbpf("TST{evg-Mxc:0}Prescaler-SP", "125000000")
>dbpr("TST{evg-Mxc:0}Frequency-RB",1)
...
EGU: Hz ...
... VAL: 1
>dbpf("TST{evg-TrigEvt:0}EvtCode-SP", "122")
>dbpf("TST{evg-TrigEvt:0}TrigSrc-Sel", "Mxc0")
>dbpf("TST{evg-TrigEvt:1}EvtCode-SP", "125")
>dbpf("TST{evg-TrigEvt:1}TrigSrc-Sel", "Mxc0")
>dbpf("TST{evr}Evt:Blink0-SP", "125")
```

This configures multiplexed counter 0 (Mxc #0) to trigger on the event clock frequency divided by 125000000. In this case this gives 1Hz. Trigger event #0 is then configured to send event code 122, and trigger event #1 to send code 125, when Mxc #0 triggers.

At this point both the EVG's amber EVENT OUT led and the EVR's EVENT IN led should flash at 1Hz.

For diagnostics the EVR's Blink0 mapping is configured to blink the EVR's EVENT OUT led when event code 125 is received. Setting to 0 will cause it to stop blinking.

Event code 122 is the heartbeat reset event. Since it is being sent the link timeout counter should no longer be increasing.

```
>dbpr("TST{evr}Cnt:LinkTimo-I")
...
... VAL: 120
>dbpr("TST{evr}Cnt:LinkTimo-I")
...
... VAL: 120
```

At this point, if the system is given an NTP server the EVG will get a correct (but unsynchronized) time and messages similar to the following will be printed.

```
Starting timestamping
epicsTime: Wed Jun 01 2011 17:54:53.000000000
TS becomes valid after fault 4de6b533
```

The first two lines come from the EVG and indicate that it is sending a timestamp. The third line comes from the EVR and indicates that it is receiving a correct timestamp.

The counter for the 1Hz event should now be increasing.

```
>dbpr("TST{evr}1hzCnt-I")
... VAL: 5
>dbpr("TST{evr}1hzCnt-I")
... VAL: 6
```

5.2 Timestamp Test

An external 1Hz pulse generator is required for this test. It should be connected to front panel input 0 on the EVG. This is LEMO connector expecting a TTL signal.

```
>dbpr("TST{evr}Link-Sts")
...
... VAL: 1
```

If the event link status is not OK then perform setup as described in the previous test.

Check the current time source status

```
>generalTimeReport(2)
Backwards time errors prevented 0 times.

Current Time Providers:      "EVR", priority = 50
        Current Time not available
    "NTP", priority = 100
```

```
        Current Time is 2011-06-02 10:23:26.058125.
    "OS Clock", priority = 999
        Current Time is 2011-06-02 10:23:26.057101.

Event Time Providers:
    "EVR", priority = 50
```

This shows that the NTP time source is functioning. This is required for this test.

```
>dbpf("TST{evg-TrigEvt:1}EvtCode-SP", "125")
>dbpf("TST{evg-TrigEvt:1}TrigSrc-Sel", "Front0")
>dbpf("TST{evr}Evt:Blink0-SP", "125")
```

Sends event code 125 on the rising edge for front panel input 0. For diagnostics sets the blink mapping. If the led is not blinking then check the 1Hz pulse generator.

```
dbpr("TST{evr}Time:Valid-Sts")
...
... VAL: 1
```

Indicates that the EVR has received a valid time

```
>generalTimeReport(2)
Backwards time errors prevented 0 times.

Current Time Providers:     "EVR", priority = 50
        Current Time is 2011-06-02 10:26:50.683808.
    "NTP", priority = 100
        Current Time is 2011-06-02 10:26:50.681220.
    "OS Clock", priority = 999
        Current Time is 2011-06-02 10:26:50.683854.

Event Time Providers:
    "EVR", priority = 50
```

Shows that a valid time is now being reported.

```
$ camonitor TST{evr:3}Time-I
TST{evr:3}Time-I               2011-06-02 10:32:11.999993 Thu, 02 Jun 2011 10:32:12 -0400
TST{evr:3}Time-I               2011-06-02 10:32:12.999993 Thu, 02 Jun 2011 10:32:13 -0400
TST{evr:3}Time-I               2011-06-02 10:32:13.999993 Thu, 02 Jun 2011 10:32:14 -0400
TST{evr:3}Time-I               2011-06-02 10:32:14.999993 Thu, 02 Jun 2011 10:32:15 -0400
```

The timestamp indicator record takes its record timestamp from the arrival of the 125 event code. As can be seen, this time is stored immediately before the sub-seconds is zeroed. This can be verified by switching this.

```
$ caget TST{evr:3}Time-I.TSE
TST{evr:3}Time-I.TSE            125
$ caput TST{evr:3}Time-I.TSE 0
Old : TST{evr:3}Time-I.TSE          125
New : TST{evr:3}Time-I.TSE          0
$ camonitor TST{evr:3}Time-I
TST{evr:3}Time-I                2011-06-02 10:35:31.005655 Thu, 02 Jun 2011 10:35:31 -0400
TST{evr:3}Time-I                2011-06-02 10:35:32.005655 Thu, 02 Jun 2011 10:35:32 -0400
TST{evr:3}Time-I                2011-06-02 10:35:33.005655 Thu, 02 Jun 2011 10:35:33 -0400
TST{evr:3}Time-I                2011-06-02 10:35:34.005655 Thu, 02 Jun 2011 10:35:34 -0400
```

Now a time latched by software when this record is processed. For real-time system this time should be stable.

6 Firmware Update

6.1 PCIe-EVR-300DC, mTCA-EVR-300

These devices support upgrade of firmware through PCIe register access. As such, a failed upgrade will result in an unusable device.

To test if a card may be upgrade with this mechanism, run *flashinfo* and *flashread* command. The following shows a device which can be upgraded.

```
epics> mrmEvrSetupPCI("EVR1", "03:00.0")
  ...
epics> flashinfo("EVR1:FLASH")
Vendor: 20 (Micron)
Device: ba
ID: 18
Capacity: 0x1000000
Sector: 0x10000
Page: 0x100
S/N:  23 51 61 31 16 00 14 00 31 26 05 15 ee 45
epics> flashread("EVR1:FLASH", 0, 64)
00090ff0 0ff00ff0 0ff00000 0161001f
70636965 65767233 30306463 3b557365
7249443d 30584646 46464646 46460062
000c376b 37307466 62673637 36006300
epics>
```

Before upgrading, it is suggested to backup the existing firmware. If the size of the existing firmware is known, then this size can be used. Otherwise, use the capacity reported by *flashinfo*. All Xilinx bit files for a particular device typically have the same size.

In this example of a PCIe-EVR-300DC with the 207.0 firmware, the exact size is 3011417 bytes, which is arbitrarily rounded up to 3MB.

```
epics> flashread("EVR1:FLASH", 0, 0x300000, "PCIe-EVR-300DC.207.0.backup.bit")
| 3080192
  ...
```

Now write the new firmware file.

```
epics> flashwrite("EVR1:FLASH", 0, "PCIe-EVR-300DC.207.6.bit")
```

If the update process is interrupted, do not power cycle! Re-run the update process to completion.

After the write completes successfully, power cycle the card to load the new bit file.

6.2 VME EVRs and EVGs

Update for VME cards is accomplished through the ethernet jack label "10 BaseT". The procedure covered in the MRF manual.

6.3 cPCI-EVRTG-300

Undocumented.

6.4 PMC-EVR-230

Firmware update for the PMC module EVR is accomplished through a JTAG interface as with the cPCI-EVRTG-300. For reasons of physical space the JTAG wires are not brought to a connector, but connected to 4 I/O pins of the PLX 9030 PCI bridge chip. In order to control these pins and update the firmware some additional software is needed. Software update may be performed by using either the parallel port support or through JTAG pins. The running Kernel must be built with the CONFIG_GENERIC_GPIO and CONFIG_GPIO_SYSFS options if the latter approach is to be used.

If the parallel port support is available, a message is printed to the kernel log when the Linux kernel module provided with mrfioc2 (mrmShared/linux) is loaded.

```
Emulating cable: Minimal
```

The kernel module also exposes the 4 I/O pins via the Linux GPIO API. The 4 pins are numbered in the order: TCK, TMS, TDO, and TDI. The number of the first pin is printed to the kernel log when the MRF kernel module is loaded.

```
GPIO setup ok, JTAG available at bit 252
```

In this example the 4 pins would be TCK=252, TMS=253, TDO=254, and TDI=255.

6.4.1 Creating an SVF file from a BIT file

The firmware file will likely be supplied in one of two formats having the extensions .bit or .svf. If the provided file has the extension .svf then proceed to section 6.4.2.

To convert a .bit file to a .svf file it is necessary to get the iMPACT programming tool from Xilinx. The easiest way to do this is with the "Lab Tools" bundle.

http://www.xilinx.com/support/download/index.htm

The following instructions are for iMPACT version 14.2.

1. Install and run the iMPACT program.
2. When prompted to create a project click cancel
3. On the left side of the main window is a pane titled "iMPACT FLows". Double click on "Create PROM File"
4. Select "Xilinx Flash/PROM" and click the first green arrow.
5. Select "Platform Flash" and "xcf08p" and click "Add Storage Device" then the second green arrow.
6. Select an output file name and path. Ensure that the file format is MCS. Click OK
7. Several small dialogs will appear. When prompted to "Add device" select the .bit file provided by MRF.
8. When prompted to add another device click No.
9. On the left side of the main window is a pane titled "iMPACT Processes". Double click on "Generate File".
10. The .mcs file should now be written.
11. Exit and restart iMPACT.

See http://www.xilinx.com/support/documentation/user_guides/ug161.pdf starting on page 67 for more detailed instructions.

1. Create a new iMPACT project. Select "Prepare a Boundary-Scan File" and the SVF format.

2. When prompted, select a name for the resulting .svf file
3. When prompted to "Assign New Configuration File" select the .mcs file just created.
4. When prompted to select a PROM type choose "xcf08p"
5. An icon representing the PROM should now appear as the only entry in the JTAG chain.
6. Right click on this icon and select Program.
7. In the dialog which appears check Verify and click OK.
8. The .svf file should now be written.
9. Exit iMPACT

6.4.2 Programming with UrJTAG

http://urjtag.org/

As of August 2012 support to the Linux GPIO "cable" was not included in any UrJTAG release. It is necessary to checkout and build the development version (commit id b6945fc65 from 9 Aug. 2012 works). This requires the Git version control tool. To build and use UrJTAG on target system, there may be a need to install certain packages in the system.

```
$ sudo apt-get install pciutils make autoconf autopoint libtool
pkg-config bison libusb-1.0-0-dev libusb-dev flex python-dev
```

With all necessary tools available, configure and build UrJTAG.

```
$ git clone git://urjtag.git.sourceforge.net/gitroot/urjtag/urjtag
$ cd urjtag/urjtags
$ ./autogen.sh --disable-nls --disable-python --prefix=$PWD/usr
$ make && make install
```

Firmware update may be performed using the parallel port support if available, e.g. when loading the kernel driver:

```
$ sudo modprobe uio
$ sudo modprobe parport
$ sudo insmod mrf.ko
$ dmesg
...
[    69.046938] mrf-pci 0000:08:0d.0: MRF Setup complete
[    69.047007] mrf-pci 0000:09:0e.0: PCI IRQ 72 -> rerouted to legacy IRQ 16
```

```
[    69.047589] mrf-pci 0000:09:0e.0: GPIOC 00249412
[    69.047626] mrf-pci 0000:09:0e.0: GPIO setup ok, JTAG available at bit 252
[    69.144196] mrf-pci 0000:09:0e.0: Emulating cable: Minimal
[    69.144239] mrf-pci 0000:09:0e.0: MRF Setup complete
...
```

The "Emulating cable: Minimal" message indicates that Minimal JTAG cable type can be used to communicate with a device. A ppdev device should be available for usage with UrJTAG:

```
$ sudo modprobe ppdev
$ dmesg
...
[    69.028268] ppdev: user-space parallel port driver
...
$ ls /dev | grep parport
parport0
```

On the target system run UrJTAG as root:

```
# ./usr/bin/jtag
jtag> cable Minimal ppdev /dev/parport0
Initializing ppdev port /dev/parport0
jtag> detect
IR length: 26
Chain length: 2
Device Id: 00100001001000011111000010010011 (0x2123E093)
  Manufacturer: Xilinx (0x093)
  Part(0):      xc2vp4 (0x123E)
  Stepping:     2
  Filename:     /epics/urjtag/share/urjtag/xilinx/xc2vp4/xc2vp4
Device Id: 11100101000001010111000010010011 (0xE5057093)
  Manufacturer: Xilinx (0x093)
  Part(1):      xcf08p (0x5057)
  Stepping:     14
  Filename:     /epics/urjtag/share/urjtag/xilinx/xcf08p/xcf08p
jtag> part 1
jtag> svf /location/of/pmc-prom.svf stop progress
```

Alternatively, a GPIO cable may be utilized if the kernel was built with options required (CONFIG_GENERIC_GPIO and CONFIG_GPIO_SYSFS), on the target system run UrJTAG as root (or a user which can export and use GPIO pins).

```
# ./usr/bin/jtag
jtag> cable gpio tck=252 tms=253 tdo=254 tdi=255
jtag> detect
IR length: 26
Chain length: 2
Device Id: 00100001001000011111000010010011 (0x2123E093)
```

```
  Manufacturer: Xilinx (0x093)
  Part(0):      xc2vp4 (0x123E)
  Stepping:     2
  Filename:     /epics/urjtag/share/urjtag/xilinx/xc2vp4/xc2vp4
Device Id: 11100101000001010111000010010011 (0xE5057093)
  Manufacturer: Xilinx (0x093)
  Part(1):      xcf08p (0x5057)
  Stepping:     14
  Filename:     /epics/urjtag/share/urjtag/xilinx/xcf08p/xcf08p
jtag> part 1
jtag> svf /location/of/pmc-prom.svf stop progress
```

Note that the device IDs may not be correctly recognized. This will not effect the programming process.

If no errors are printed then the update process was successful. The new firmware will not be loaded until the PMC module is reset (power cycle system).

7 NTPD Time Source

It is possible to use an EVR as a time source for the system NTP daemon on Linux. This is implemented using the shared memory clock driver (#28).

http://www.eecis.udel.edu/~mills/ntp/html/drivers/driver28.html

An IOC is configured to write data to a shared memory segment by adding a line to its start script.

```
time2ntp("evrname", N)
```

Here "evrname" is the same name given when configuring the EVR (see 4.1). The memory segment ID number N must be between 0 and 4 inclusive. The NTP daemon enforces that segments 0 and 1 require root permissions to use. Segments 2, 3, and 4 can be accessed by an unprivileged user.

It is suggested to use an unprivileged segment to avoid running the IOC as root. However, this would allow any user on the system to effectively control NTPD. So it is not recommended for systems with untrusted users.

The NTP daemon is configured from the file */etc/ntp.conf*. On Debian Linux systems using DHCP it will be necessary to modify */etc/dhcp/dhclient-exit-hooks.d/ntp* instead.

```
server 127.127.28.N minpoll 1 maxpoll 2 prefer
fudge 127.127.28.N refid EVR
```

This will configure NTPD to read time from segment N. Here N must match what was specified for *time2ntp()*.

When functioning correctly NTPD status should look like:

```
$ ntpq -p
remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
+time.cs.nsls2.l .GPS.      1 u   29   64  377    2.684   -0.001   0.089
*SHM(3)          .EVR.      0 l    7    8  377    0.000    0.000   0.001
```

The shared memory interface can only be used to provide time with microsecond precision. So this measurement, taken from a production NSLS2 server, showing a jitter of ± 1 microsecond is the best which can be obtained.

If the propogation time from the time source to the EVR is known, then the offset can be given by adding "time1 0.XXX" to the 'fudge' line in *ntp.conf*.

8 Implementation Details

Details of some parts of the driver which may be useful in understanding (and trouble shooting) the behavior of the driver.

8.1 Event code FIFO Buffer

Each EVR implements a hardware First In First Out buffer for event codes. When certain "interesting" event code numbers are received the code and arrival time are placed in this buffer. Two interrupt condition are generated by the FIFO: not empty, and full. The first is asserted when the first event added, and cleared when the last event is removed. The second occurs when last free entry in the buffer is consumed. Further event occurrences are lost.

When the not empty interrupt occurs the fifo drain task (named EVRFIFO in epicsThreadShowAll()) is woken up by a message queue. This task runs at scan high priority (90). Once awakened it will remove at most 512 event codes from the buffer before sleeping again. The number 512 is an arbitrary number chosen to prevent the starvation of lower priority tasks if a high frequency event code is accidentally mapped into the FIFO. A minimum sleep time is enforced by the **mrmEvrFIFOPeriod** variable. This governs the maximum rate that events can be reported through the FIFO. Setting to 0 will disable it.

Each of the event codes 1-255 has an IOSCANPVT and a list of callback functions (type EVR::eventCallback) which will be invoked when the event occurs.

An invocation of an IOSCANPVT list may place an arbitrary number of CALLBACKs into the message queue of the three EPICS callback scan tasks (High,

Medium, and Low). If these message queues are overflowed then CALLBACK in other drivers my be lost. The scanIoRequest() function does not report this error prior to Base 3.15.0.2.

To avoid this disastrous occurrence the EVR driver will not re-run the scan list for an event, until all actions **at all priorities** from the previous run have finished. This is implemented by placing a special sentinel CALLBACK in all three queues. An event will not be re-run until all three of the CALLBACK have run.

The FIFO servicing code can indicate two error conditions. Occurrences of these errors are recorded in the `FIFO Overflow Count` and `FIFO Over rate` counters.

The `FIFO Overflow Count` gives the number of times the hardware FIFO buffer has overflowed. This is a serious error since arbitrary event code (including the timestamping codes) will be lost.

The `FIFO Over rate` counter counts the number of times any event reoccurred before the actions of the last occurrence were finished processing. This is less serious since other event codes are not effected.

8.2 Data Buffer reception

Each EVR can receive a single data buffer. Once a data message has been received, the reception engine is disabled to allow time to download the buffer. Then the engine can be re-enabled in preparation for the next message. An interrupt is generated when the message has been fully received, and the engine disabled.

Instead of a separate thread, buffer reception is implemented as a two stage callback run by the High (first) and Medium (second) priority scan tasks. The first callback copies the buffer into memory and immediately re-enables buffer reception, it then passes the data to the second callback. This callback passes the buffer to a list of user callback functions which have registered interest in the Protocol ID found in the message header.

8.3 Timestamp validation

It is impossible to verify a time without a second trusted reference. Since such a reference is not generally available, the driver can only make some checks against corruption.

The seconds part of the timestamp should only change when the 1Hz reset event (125) is received from the EVG. Therefore a callback is attached to that event code. When a new seconds value arrives it is compared to the previous stored value. If it is exactly 1 greater then it is taken to be the new seconds value. If it is not then the EVR time is declared invalid.

When the time is invalid, it can only become valid after five sequential seconds values are received. Any out of sequence value resets the count.

9 EVR Device Support Reference

The EPICS support module for MRF devices consists of a number of supports which are generally tied to a specific logical sub-unit. Each sub-unit may be thought of as an object having a number of properties. For example, each Delay Generator has properties 'Delay' and 'Width'. These properties can be read or modified in several ways. A delay can specified as an integer number of ticks of its reference clock (hardware view), or in seconds as a floating point number (user view).

In this example the properties 'Delay' and 'Width' should be settable in exact integer as well as the more useful, but imprecise, floating point units (eg. seconds). This needs to be accomplished by two different device supports (longout, and ao). Of course it is also useful to have some confirmation that settings have been applied so read-backs are desireable (longin, ai).

Some of the device supports defined are as follows. The full list is given in `mrfCommon/src/mrfCommon.dbd`.

```
device(longin  , INST_IO, devLIFromUINT32, "Obj Prop uint32")
device(longin  , INST_IO, devLIFromUINT16, "Obj Prop uint16")
device(longin  , INST_IO, devLIFromBool,   "Obj Prop bool")

device(ai  , INST_IO, devAOFromDouble, "Obj Prop double")
device(ai  , INST_IO, devAOFromUINT32, "Obj Prop uint32")
device(ai  , INST_IO, devAOFromUINT16, "Obj Prop uint16")
```

Unless otherwise noted, all device support use **INST_IO** input/output links with the format:

```
@OBJ=$(OBJECTNAME), PROP=Property Name
```

Since the Pulser sub-unit has the property 'Delay' which supports both integer and float settings, the following database can be constructed.

```
record(ao, "$(PN)Delay-SP")
{
  field(DTYP, "Obj Prop double")
  field(OUT , "@OBJ=$(OBJ), PROP=Delay")
  field(PINI, "YES")
  field(DESC, "Pulse Generator $(PID)")
  field(FLNK, "$(PN)Delay-RB")
}
```

```
record(ai, "$(PN)Delay-RB")
{
  field(DTYP, "Obj Prop double")
  field(INP , "@OBJ=$(OBJ), PROP=Delay")
  field(FLNK, "$(PN)Delay:Raw-RB")
}
record(longin, "$(PN)Delay:Raw-RB")
{
  field(DTYP, "Obj Prop uint32")
  field(INP , "@OBJ=$(OBJ), PROP=Delay")
}
```

This provides setting in engineering units and readbacks in both EGU and raw for the delay property.

Note: In is inadvisible to have to more then one output record pointing to the same property of the same device. However, it is allowed since there are cases where this is desireable.

The following sections list the properties for all sub-units with functional descriptions.

9.1 Global

Properties in this section apply to the EVR as a whole. The object Name is given as the first argument of **mrmEvrSetupPCI()** or **mrmEvrSetupVME()**. This name will be refered to afterwards as $(OBJ).

See: evrApp/Db/evrbase.db

Property Name	Type(s)	Writeable	I/O Intr	Notes
Enable	bool	Yes		
PLL Lock Status	bool	No		
Link Status	bool	No	X	
Timestamp Valid	bool	No	X	
Model	uint32	No		
Version	uint32	No		
Sw Version	string	No		
FIFO Overflow Count	uint32	No		
FIFO Over rate	uint32	No		
HB Timeout Count	uint32	No	X	
Clock	double	Yes		
Timestamp Source	uint32	Yes		
Timestamp Clock	double	Yes		
Timestamp Prescaler	uint32	No		
Timestamp		No		
Event Clock TS Div	uint32	No		
Receive Error Count	uint32	No	X	

For example, the **bool**ean property **Enable** could be written by the following record.

```
record(bo, "$(P)ena") {
  field(DTYP, "Obj Prop bool")
  field(OUT , "@OBJ=$(OBJ), PROP=Enable")
  ...
}
```

9.1.1 Enable

Type(s): bool

Master enable for the EVR. If not set then very little will happen.

9.1.2 PLL Lock Status

Type(s): bool

This indicates whether the phase locked loop which synchronizes an EVR's local oscilator with the phase of the EVG's oscilator. Outputs will not be stable unless the PLL is locked.

Except for immediately ($\ll 1sec$) after a change to the fractional synthesizer this property should always read as true (locked). Reading false for longer then one second is likely an indication that the fractional synthesize is misconfigured, or that a hardware fault has occured.

9.1.3 Link Status

Type(s): bool

Indicates when the event link is active. This means that the receiver sees light, and that valid data is being decoded.

A reading of false may be caused by a number of conditions including: EVG down, fiber unplugged or broken, and/or incorrent fractional synthesizer frequency.

9.1.4 Timestamp Valid

Type(s): bool

Indicates if the EVR has a current, valid timestamp. Condition under which the timestamp is declared invalid include:

- TS counter reset event received, but "seconds" value not updated.
- Found timestamp with previous invalid value. Catches old timestamp in buffers.
- TS counter exceeded limit (eg. missed reset event)
- New "seconds" value is less then the last valid values, or more then two greater then the last valid value. (Light Source time model only). This will reject single "bad" values sent by the EVG.
- Event Link error (Status is error).

The timestamp will become valid when a new "seconds" value is received from the EVG.

9.1.5 Model

Type(s): uint32

The hardware model number.

9.1.6 Version

Type(s): uint32

The firmware version number.

9.1.7 Sw Version

Types(s): string

A string describing the version of the driver software. This is captured when the driver is compiled.

9.1.8 FIFO Overflow Count

Type(s): uint32

Counter the number of hardware event buffer overflows. There is a single hardware buffer for all event codes. When it overflows arbitrary events will fail to be delivered to software. This can cause the timestamp to falsely be invalidated, and can disrupt database processing which depends on event reception.

This is a serious error which should be corrected.

Note: An overflow does **not** effect physical outputs.

9.1.9 FIFO Over rate

Type(s): uint32

Counts overflows in all of the per event software buffers.

This indicates that the period between successive events is shorter then the runtime of the code (callbacks, and database processing) that is causes. Extra events are being dropped and cause no action.

Actions of other event codes are not effected.

9.1.10 HB Timeout Count

Type(s): uint32

The number of times the hardware heartbeat timer has expired. This indicates that the EVG is not sending event code 122 which may mean that it is misconfigured or hung.

9.1.11 Clock

Type(s): double

Frequency of an EVR's local oscilator. This must be close enough to the EVG master oscilator to allow the phase locked loop in the EVR to lock.

The native analog units are Hertz (Hz). This can be changed with the LINR and ESLO fields. Use ESLO of 1e-6 to allow user setting/reading in MHz.

9.1.12 Timestamp Source

Type(s): uint32

Determines what causes the timestamp event counter to tick. There are three possible values.

Event clock Use an integer divisor of the EVR's local oscilator.

Mapped code(s) Increments the counter whenever certain events arrive. These codes can be defined using special mapping records.

DBus 4 Increments on the 0->1 transition of DBus bit #4.

9.1.13 Timestamp Clock

Type(s): double

Specifies the rate at which the timestamp event counter will be incremented. This determines the resolution of all timestamps.

This setting is used in conjunction with the 'Timestamp Source'.

When the timestamp source is set to "Event clock" this property is used to compute an integer divider from the EVR's local oscilator frequency to the given frequency. Since this may not be exact it is recommended to read back the actual divider setting via the "Timestamp Prescaler" property.

In all modes this value is stored in memory and used to convert the timestamp event counter values from ticks to seconds.

By default the analog units are Hertz (Hz). This can be changed with the LINR and ESLO fields. Use ESLO of 1e-6 to allow user setting/reading in MHz.

9.1.14 Timestamp Prescaler

Type(s): uint32

When using the "Event clock" timestamp source this will return the actual divisor used. In other modes it reads 0.

9.1.15 Timestamp

Special device support.

When processed creates a human readable string with either the current event link time, or the event link time when code # was last received. If Code is omitted or 0 then the current wall clock time is used. Code may also have any valid event number 1-255. Then it will print the time of the last received event.

```
record(stringin, "$(P)Time-I")
{
  field(DTYP, "EVR Timestamp")
  field(INP , "@OBJ=$(OBJ), Code=0")
  field(SCAN, "Event")
  field(EVNT, "$(EVNT1HZ)")
}
```

9.1.16 Event Clock TS Div

Type(s): uint32

This is an approximate divider from the event link frequency down to 1MHz. It is used to determine the heartbeat timeout.

9.1.17 Receive Error Count

Type(s): uint32

The number of event link errors which have occurred.

9.2 EVG functions (DC firmware)

The PCIe-EVR-300DC and mTCA-EVR-300 provide one EVG style Sequencer RAM, as well as sending software triggered events upstream.

Property Name	Type(s)	Writeable	Notes
DCEnable	bool	Yes	Apply delay compensation
DCTarget	uint32	Yes	Desired total delay
DCRx	uint32	No	Measured delay from root EVG/EVM
DCInt	uint32	No	Internal Delay
DCStatusRaw	uint32	No	DC status register (bitmask)
DCTOPID	uint32	No	Global topology ID
EvtCode	uint32	Yes	Send software event
TimeSrc	uint32	Yes	Timestamp Transmision mode (enum)
NextSecond	string	No	Next time which will be sent and TimeSrc!=0
Time Error	double	No	Difference between system time and Tx'd time
Sync TS	cmd	Yes	Sync Tx'd time to system time

See EVG documentation for details. For details of sequencer usage.

9.3 SFP

Information and status from the Small Form factor Plugable (SFP) transceiver module. Access to this feature requires EVR firmward version 5 (starting with 25 May 2012). It is automatically disabled at runtime if an unsupported version is detected.

Property Name	Type(s)	Writeable	Notes
Update	bool	Yes	Triggers read of the SFP EEPROM
Vendor	string	No	Module vendor name
Part	string	No	Vendor's part number
Rev	string	No	Part revision
Date	string	No	Date of manufacture
Serial	string	No	SFP module serial number
Temperature	uint32	No	Module temperature in C
Link speed	uint32	No	Bit rate
Power TX	uint32	No	Optical power of SFP transmitter
Power RX	double	No	Optical power seen by SFP receiver

9.4 Pulse Generator

Properties in this section apply to the Pulse Generator (Pulser) sub-unit named $(OBJ):Pul# where # is a number between 0 and 15.

See: evrApp/Db/evrpulser.db

Property Name	Type(s)	Writeable	I/O Intr	Notes
Enable	bool	Yes		
Polarity	bool	Yes		
Prescaler	bool	Yes		
Delay	double, uint32	Yes		
Width	double, uint32	Yes		

For example, the property **Delay** could be set by either of the following records.

```
record(ao, "$(D)ena") {
  field(DTYP, "Obj Prop double")
  field(OUT , "@OBJ=$(OBJ):Pul#, PROP=Enable")
  ...
}
record(longout, "$(D)ena") {
  field(DTYP, "Obj Prop uint32")
  field(OUT , "@OBJ=$(OBJ):Pul#, PROP=Enable")
  ...
}
```

9.4.1 Enable

Type(s): bool

When not set, the output of the Pulse Generator will remain in its inactive state (normally low). The generator must be enabled before mapped actions will have any effect.

9.4.2 Polarity

Type(s): bool

Reverses the output polarity. When set, changes the Pulse Generator's output from normally low to normally high.

9.4.3 Prescaler

Type(s): uint32

Decreases the resolution of both delay and width by an integer multiple. Determines the tick rate of the internal counters used for delay and width with respect to the EVR's local oscillator.

9.4.4 Delay

Type(s): double and uint32

Determines the time between when the Pulse Generator is triggered and when it changes state from inactive to active (normally low to high).

This can be given in integer ticks, or floating point seconds. This can be changed with the LINR and ESLO fields. Use ESLO of 1e6 to allow user setting/reading in microseconds.

9.4.5 Width

Type(s): double and uint32

Determines the time between when the Pulse Generator changes state from inactive to active (normally low to high), and when it changes back to inactive.

This can be given in integer ticks, or floating point seconds. This can be changed with the LINR and ESLO fields. Use ESLO of 1e6 to allow user setting/reading in microseconds.

9.5 Prescaler (Clock Divider)

Properties in this section apply to the Prescaler sub-unit. Prescaler objects are named $(OBJ):PS# where # is between 0 and 2.

See: evrApp/Db/evrscale.db

9.5.1 Divide

Type(s): uint32

Sets the integer divisor between the Event Clock and the sub-unit output.

By default the analog units are Hertz (Hz). This can be changed with the LINR and ESLO fields. Use ESLO of 1e-6 to allow user setting/reading in MHz.

9.6 Output (TTL and CML)

Properties in this section apply to the Output sub-unit. Output objects are named either $(OBJ):FrontOut#, $(OBJ):FrontOutUniv#, or $(OBJ):RearUniv# where the range of number # depends on the hardware model.

See: evrMrmApp/Db/mrmevrout.db

9.6.1 Map

Type(s): uint32

The meaning of this value is determined by the specific implimentation used.

For the MRM implimentation the following codes are valid.

#	Output Source	#	Output Source
63	Force High	15	Pulse generator 15
62	Force Low	14	Pulse generator 14
61	Tri-state	13	Pulse generator 13
42	Prescaler (Divider) 2	12	Pulse generator 12
41	Prescaler (Divider) 1	11	Pulse generator 11
40	Prescaler (Divider) 0	10	Pulse generator 10
39	Distributed Bus Bit 7	9	Pulse generator 9
38	Distributed Bus Bit 6	8	Pulse generator 8
37	Distributed Bus Bit 5	7	Pulse generator 7
36	Distributed Bus Bit 4	6	Pulse generator 6
35	Distributed Bus Bit 3	5	Pulse generator 5
34	Distributed Bus Bit 2	4	Pulse generator 4
33	Distributed Bus Bit 1	3	Pulse generator 3
32	Distributed Bus Bit 0	2	Pulse generator 2
		1	Pulse generator 1
		0	Pulse generator 0

9.6.2 Enable

Type(s): bool

When set to True the mapping set with the **Map** property is used. When False a mapping of Force Low is used.

9.7 Output (CML/GTX only)

Additional properties for Current Mode Logic (CML) and GTX outputs. Output objects are named either $(OBJ):FrontOut#, $(OBJ):FrontOutUniv#, or $(OBJ):RearUniv# where the range of number # depends on the hardware model.

See: evrApp/Db/evrcml.db

Property Name	Type(s)	Writeable	I/O Intr	Notes
Enable	bool	Yes		
Power	bool	Yes		
Reset	bool	Yes		
Mode	uint16	Yes		
Pat Rise	UCHAR waveform	Yes		
Pat High	UCHAR waveform	Yes		
Pat Fall	UCHAR waveform	Yes		
Pat Low	UCHAR waveform	Yes		
Waveform	UCHAR waveform	Yes		
Pat Recycle	bool	Yes		
Freq Trig Lvl	bool	Yes		
Counts Init	double, uint32	Yes		
Counts High	double, uint32	Yes		
Counts Low	double, uint32	Yes		
Freq Mult	uint32	No		

9.7.1 Enable

Type(s): bool

Trigger permit.

9.7.2 Power

Type(s): bool

Current driver on.

9.7.3 Reset

Type(s): bool

Pattern reset.

9.7.4 Mode

Type(s): uint16

Selects CML pattern mode. Possible values are: 4x Pattern (0), Frequency (1), Waveform (2).

4x Pattern Uses the Pat Rise, Pat High, Pat Fall, and Pat Low properties to store four 20 bit (0 -> 0xfff) sub-patterns.

Frequency Uses the Freq Trig Lvl, Counts High, and Counts Low properties

Waveform Uses the bit pattern stored by the Pattern Set property.

9.7.5 Pat Rise/Pat High/Pat Fall/Pat Low/Waveform

Type(s): UCHAR waveform

Each property stores a seperate bit waveform as an array of bytes.

The four patterns are 20 or 40 bit waveforms are sent once at either edge (rising/falling), and repeatidly when when at a stable level.

Rising and Falling patterns start as soon as the edge is detected and will interrupt the pattern currently being sent.

The High and Low patterns are sent after an edge pattern is sent and will repeat until the next edge.

The Waveform pattern is a variable length patten (max $40940 = 20 * 2047$ or $81880 = 40 * 2047$)

9.7.6 Pattern Recycle

Type(s): bool

In waveform mode a trigger cause the output to begin sending the pattern from its start. When the end of the pattern is reached the output will either go in active, or begin sending the pattern again, based on this property.

9.7.7 Freq Trig Lvl

Type(s): bool

When in frequency mode and a trigger arrives the output is forced to this level.

9.7.8 Counts High/Low/Init

Type(s): uint32 or double

Stores a value which is the number of counts (uint32) or time (double) of the high or low part of a square wave.

The number of ticks must be $>$20 or 40, whichever is the time of one period of the event clock.

The Counts Init property holds the value which is loaded into the counter when a trigger arrives. This allows for a phase difference between the output and the trigger source.

9.7.9 Freq Mult

Type(s): uint32

This read only property gives the multiplier for the CML/GTX output clock. This will be either 20 (CML) or 40 (GTX).

9.8 Input

Properties in this section apply to the Input sub-unit. Input objects are named $(OBJ):FPIn# where the range of the number # depends on the hardware model.

See: evrApp/Db/evrin.db

Property Name	Type(s)	Writeable	I/O Intr	Notes
Active Level	bool	Yes		
Active Edge	bool	Yes		
External Mode	uint16	Yes		
External Code	uint32	Yes		
Backwards Mode	uint16	Yes		
Backwards Code	uint32	Yes		
DBus Mask	uint16	Yes		

9.8.1 Active Level

Type(s): bool

When operating in level triggered mode, determines if codes are sent when the input level is low, or high.

9.8.2 Active Edge

Type(s): bool

When operating in edge triggered mode, Determines if codes are sent on the falling or rising edge in the input signal.

9.8.3 External Mode

Type(s): uint16

Selects the condition, Level (1), Edge (2), or None (0), in which to inject event codes into the local mapping ram. These codes are treated as codes coming from the downstream event link.

9.8.4 External Code

Type(s): uint32

Sets the code which will be applied to the local mapping ram whenever the 'External Mode' condition is met.

9.8.5 Backwards Mode

Type(s): uint16

Selects the condition, Level (1), Edge (2), or None (0), in which to send on the upstream event link.

9.8.6 Backwards Code

Type(s): uint32

Sets the code which will be sent on the upstream event link whenever the 'Backwards Mode' condition is met.

9.8.7 DBus Mask

Type(s): uint16

Sets the upstream Distributed Bus bit mask which is driven by this input. DBus bits from multiple sources are condensed with a bit-wise OR.

9.9 Event Mapping

Properties in this section describe actions which should be taken when an event code is received.

9.9.1 Pulse Generator Mapping

Special device support acting on pulser generator objects.

See: evrApp/Db/evrpulsermap.db

Causes a received event to trigger a Pulse Generator (Pulser) sub-unit, or force it into an active (set) or inactive (reset) state.

These records will have DTYP set to "EVR Pulser Mapping".

Each record will cause one event to trigger, set, or reset one Pulse Generator. It is possible (and likely) that more then one record will interact with each event code or Pulse Generator. However, each pairing must be unique.

```
record(longout, "$(P)$(N)$(M)") {
    field(DTYP, "EVR Pulser Mapping")
    field(OUT , "@OBJ=$(OBJ):Pul0, Func=$(F=Trig)")
    field(PINI, "YES")
    field(DESC, "Mapping for Pulser $(PID)")
    field(VAL , "$(EVT)")
    field(LOPR, "0")
    field(HOPR, "255")
    field(DRVL, "0")
    field(DRVH, "255")
}
```

In this example the event '$(EVT)' specified in the 'VAL' field will cause function '$(F)' on Pulse Generator # '$(PID)'. Current functions are 'Trig', 'Reset', and 'Set'.

9.9.2 Special Function Mapping

Special device supportacting on global EVR objects.

See: evrApp/Db/evrmap.db

Allows a number of special actions to be mapped to certains events. These actions include:

Blink An LED on the EVRs front panel will blink when the code is received.

Forward The received code will be immediately retransmits on the upstream event link.

Stop Log Freeze the circular event log buffer. An CPU interrupt will be raised which will cause the buffer to be downloaded. This might be a useful action to map to a fault event.

Log Include this event code in the circular event log.

Heartbeat This event resets the heartbeat timeout timer.

Reset PS Resets the phase of all prescalers.

TS reset Transfers the seconds timestamp from the shift register and zeros the sub-seconds part.

TS tick When the timestamp source is 'Mapped code' then any event with this mapping will cause the sub-seconds part of the timestamp to increment.

Shift 1 Shifts the current value of the seconds timestamp shift register up by one bit and sets the low bit to 1.

Shift 0 Shifts the current value of the seconds timestamp shift register up by one bit and sets the low bit to 0.

FIFO Bypass the automatic allocation mechanism and always include this code in the event FIFO.

In the following example the front panel LED on the EVR will blink whenever event 14 is received.

```
record(longout, "$(P)map:blink") {
    field(DTYP, "EVR Mapping")
    field(OUT , "@OBJ=$(OBJ), Func=Blink")
    field(PINI, "YES")
    field(VAL , "14")
    field(LOPR, "0")
    field(HOPR, "255")
}
```

9.10 Database Events

Special device support acting on global EVR objects.

See: evrApp/Db/evrevent.db

A device support for the 'event' recordtype is provided which uses the Event FIFO to record the arrival of certain interesting events. When set to SCAN 'I/O Intr' the event record device support will process the record causing the requested DB event. Supports setting it timestamp from device support (set TSE to -2).

```
record(longout, "$(P)$(N)") {
    field(DTYP, "EVR")
    field(SCAN, "I/O Intr")
    field(INP , "@OBJ=$(OBJ), Code=$(CODE)")
    field(VAL , "$(EVNT)")
    field(TSE , "-2") # from device support
    field(FLNK, "$(P)$(N):count")
}
record(calc, "$(P)$(N):count") {
    field(SCAN, "Event")
    field(EVNT, "$(EVNT)")
    field(CALC, "A+1")
    field(INPA, "$(P)$(N):count NPP")
    field(TSEL, "$(P)$(N).TIME")
}
```

In this example the hardware event code '$(CODE)' will cause the database event '$(EVNT)'.

Note: that while both '$(CODE)' and '$(EVNT)' are numbers, they need not be the same. Hardware code 21 can cause DB event 40.

9.11 Data Buffer Rx

Records associated with receiving variable length data messages.

9.11.1 Enable

See: evrApp/Db/evrbase.db

Object name $(OBJ):BUFRX

Type(s): bool

Selects Event link data mode. This chooses between DBus only (1) , and DBus+Buffer (0) modes. In DBus only mode Data Buffer reception is not possible.

9.11.2 Data Rx

See: evrMrmApp/Db/mrmevrbufrx.db

Implemented for: waveform

When a buffer with the given Protocol ID is received a copy is placed in this record. It is possible to have many records receiving the same Protocol ID. Data is received as a byte array and interpreted according to FTVL. For multi-byte types the transmission byte order is assumed to be big endian. Data is truncated to a multiple of the element size.

Many record (or other listeners) may register for the same Protocol ID.

The special Protocol ID 0xff00 may be used to cause a listener to receive messages destined for any ID.

Note: In order to avoid extra copy overhead this record bypasses the normal scanning process. It function like "I/O Intr", however the SCAN field should be left as "Passive".

```
record(waveform, "$(P)dbus:recv:u32")
{
  field(DESC, "Recv Buffer")
  field(DTYP, "MRM EVR Buf Rx")
  field(INP , "@OBJ=$(OBJ), Proto=$(PROTO), P=Data Rx")
  field(FTVL, "ULONG")
  field(NELM, "2046")
}
```

9.12 Data Buffer Tx

Records associated with sending variable length data messages.

This section is shared between the EVR and EVG.

9.12.1 Outgoing Event Data Mode

See: mrmShared/Db/databuftxCtrl.db

Object name $(OBJ):BUFTX

Type(s): bool

Selects Event link data mode. This chooses between DBus only (1), and DBus+Buffer (0) modes. In DBus only mode Data Buffer transmission is not possible.

9.12.2 Data Tx

Special device support “MRF Data Buf Tx”.

See: mrmShared/Db/databuftx.db

This records sends a block of data with the given Protocol ID. If FTVL specifies a multi-byte type then data will be converted to big endian byte order for transmission.

```
record(waveform, "$(P)dbus:send:u32")
{
  field(DESC, "Send Buffer")
  field(DTYP, "MRF Data Buf Tx")
  field(INP , "@C=$(C), Proto=$(PROTO), P=Data Tx")
  field(FTVL, "ULONG")
  field(NELM, "2046")
}
```