The description of the GPMM general purpose device and driver support for the memory mapped devices.

The GPMM driver was foreseen to support VME memory mapped cards. It allows easy integration of new cards to EPICS. It offers basic read and writes capability. Its functionality could be easily extended by the user define functions which are easy to write and does not require in depth knowledge of EPICS drivers. Recently the GPMM driver was extended for Linux based systems including embedded ones. The GPMM driver Source code could be found under the link: http://epics.web.psi.ch/software/GPMM

1. The setup of the startup script

Load the GPMM device and driver support in the *startup.script* by means of the require entry:

```
require "gpmm" /* sls way of loading drivers */
```

an alternative is:

```
dbLoadDatabase("GPMM.dbd")
ld < slsDevLib.o
ld < drvGPMM.o
ld < devGPMM.o
```

There are three types of functions, which should be used (in the *startup.script*) for the hardware card configuration:

- GPMMConfigure
    - addGPMMRegister
    - addGPMMInterrupt

The GPMMConfigure deals with the card itself. The addGPMMRegister refers to the card memory locations - called registers. The addGPMMInterrupt configures the interrupt to be used by EPICS channels.

The GPMMConfigure function invocation requires the following arguments:

```
GPMMConfigure(cardNr, baseAddr, cardName, addrAccessMode, dataAccessMode)
```

where:

cardNr	- the consecutive card number.
baseAddr	- VME base address of the given card.
cardName	- up to 15 character long string (card name).
addrAccessMode	- address access mode. 3 character long string ("A16" or "A24" or "A32").
dataAccessMode	- data access mode. 2 or 3 character long string ("D8" or "D16" or "D32").

example:

```
GPMMConfigure(0,0x01600000,"ADC","A32","D32")
```

The addGPMMRegister function invocation requires the following arguments:

```
addGPMMRegister(cardNr, refNr, addrOffs, bytesNr, readWrite, regName, userFunc)
```

where:

cardNr	- the consecutive card number. It makes the associations between hardware card and the register.
refNr	- the consecutive register number for the given card.
addrOffs	- offset with the respect to the base address for the hardware card.
bytesNr	- number of bytes requested for the given register (if 0 then no read verification , on the driver support level, is done).
readWrite	- test the register for reading or writing. The argument could be 'R' for reading, 'W' for writing, 'X' do not test the register for reading or writing.
regName	- name of the register (up to 20 characters long string).
userFunc	- user function which could be optionally called to process the data. The value 0 should be used if no additional function is required. More about this function will be written later in chapter 3.

The addGPMMInterrupt function invocation requires the following arguments:

```
addGPMMInterrupt(cardNr, IntrLevel, IntrVector, userISR)
```

where:

cardNr	- the consecutive card number. It makes the associations between hardware card and the interrupt service routine (ISR).
IntrLevel	- Interrupt Level.
IntrVector	- Interrupt Vector.
userISR	- Interrupt Service Routine is called when interrupt occurs. This routine cause that EPICS channel is processed. In most cases the default ISR i.e. defaultGPMMISR could be used which does what is needed to process the EPICS channel when interrupt occurs.

example:

```
addGPMMInterrupt(0, 3, 254, defaultGPMMISR)
```

2. The setup of the EPICS database

General statement.
For the time being it was done the implementation for the following record types: *ai, ao, bi, bo, mbbi, mbbo, longin, longout, mbbiDirect, mbboDirect, stringin, waveform* .

To access the registers (hardware card memory locations) the following fields should be set:

```
 field   (DTYP, "GPMM")
```

```
field   (INP, "#Cx Sy @zzz[:i1:i2][,OUT,SEQ,B=kk,m=kk,s=kk,iRead=ll]")  or
field   (OUT, "#Cx Sy @zzz[…][…]")
```

where:

x	- refers to the hardware card number (cardNr argument of the GPMMConfigure and addGPMMRegister functions.
y	- refers to the offset from the register address (zzz). The offset is expressed in terms of bytes regardless the record type.
zzz	- register name. It is the same string as regName argument in the addGPMMRegister function.

There are two types of optional arguments. One type is ":" separable, another one "," separable. The ":" separable arguments are the input arguments that could be used by the user defined function. There could be used maximally two arguments of that type i.e. i1 and i2 (which are integer numbers).

Optional arguments "," separable refer to the way how the data is read/write to the memory location. The convention is as such that, each optional argument is referenced by a single letter (ex.: B,m,s). Capital letters refer to bytes and lower case letters refer to bits. The meaning of the optional arguments is as following:

B	- is the number of bytes to be read out or written to the card. The letter *B* stands for *Bytes*.
m	- number of bits to be read out written to the card. The letter *m* stands for the *mask.*
s	- number of bits of an offset with the respect to *S* parameter of the INP or OUT filed. This parameter could be used for bi, bo, mbbi, mbbo, mbbiDirect, mbboDirect records. The letter *s* stands for the *shift.*
kk	- is a decimal value.
iRead	- could be used by *ao, bo, mbboDirect, mbbo, longout, waveform* records to be initialized during boot up process with the data read out from the memory location they deal with. If the initRead is not specified the default action is not to read data during boot up.
ll	- could be ***Y*** or ***VAL*** . Option ***Y*** means initialize record from the hardware ***VAL*** means initialize the record from VAL filed (this is useful when using save and restore) at boot up.
OUT	- could be used for waveform record to set it up as an output one.
SEQ	- could be used for waveform record to read or write the waveform from/to the memory location in a sequential way (i.e. referring to the same address).

Remark:

For the mbbi, mbbo, mbbiDirect and mbboDirect the optional arguments could be used alternatively with the standard fields like nobt (number of bits) and shft (number of bits to shift).

example:

```
record(mbbiDirect,"GPMM-TEST:mbbiD01") {
            field(DTYP, "GPMM")
            field(INP, "#C3 S5 @Firmware,s=2,m=7")}
```

For the given example the channel GPMM-TEST:mbbiD01 reads 7 bits of data from the memory location:

baseAddr (of the card 3) +
addrOffs (of the Firmware register) +
5 (Bytes) + 2 (bits)

3. The usage of the user defined function

The aim of the user define function (last argument of the addGPMMRegister function) is to do some data manipulation required by the user. The user function (if defined) is called twice when the EPICS record is processed (ie. once before accessing the desired memory location and once after accessing the memory). This function is called from the level of driver support functions which reads/writes the data to desired memory locations.

Simplified example of the user function for input records like ai, longin, mbbi… looks like :

```
int myGPMMFunc(
    unsigned short card,
    unsigned short signal,
    struct recDesc  *parm,
    unsigned short   sRecordType,
    unsigned short   sRecordPass,
    char *chanName,
    void *pVal
){
unsigned long  lVal;
unsigned short sVal;
         short status=RETURN_OK;

/* for ai ao longin longout records (R)VAL is 4 bytes long */
lVal= *((unsigned long  *)pVal);
/* for mbbi mbbo bi bo RVAL is 2 bytes long*/
sVal= *((unsigned short *)pVal);

PROCESS_DATA_FROM_INPUT_RECORD_BEGIN

/* do your data processing here. Use lVal or sVal depending on
your record type */
if(sRecordType==REC_AI){
     printf("Record AI\n");
     printf("myGPMMFunc C%d S%d @%s %s 0x%.8X\n",card,signal,
                   parm->sDesc,chanName,(unsigned int) lVal);
}
status =  RETURN_FROM_INPUT_RECORD_OK;
/* if error than status = RETURN_FROM_INPUT_RECORD_ERROR  */

PROCESS_DATA_FROM_INPUT_RECORD_END

return (status);
}
```

For output records use the corresponding macros:

PROCESS_DATA_FROM_OUTPUT_RECORD_BEGIN
PROCESS_DATA_FROM_OUTPUT_RECORD_END

RETURN_FROM_OUTPUT_RECORD_OK
RETURN_FROM_OUTPUT_RECORD_ERROR

3.1 Advanced user defined function programming

The following example explains how the user define function is called when the mbboDirect Record is process on the device driver level.

```
writeMbboDirect(){

        /* invoke the user function on enter */
        status=userFunction();

        if( status == PROCESS_DATA){
         /* data is written to the desired memory location */
         }

        if (status == ENTER_FUNC){
        /* invoke the user function on exit */
         userFunction();
         }

} /* return from the WriteMbboDirect() */
```

The user function returns predefined status code. There are following predefined codes:

RETURN_OK	: This code is interpreted as PROCESS_DATA **|** ENTER_FUNC
RETURN_ERROR	: This suppresses further actions and forwards the error status to the device support level.
RETURN_WRITE_SKIP	: This code is interpreted as ENTER_FUNC
RETURN_READ_SKIP	: Similar like code above but for read type records like mbbiDirect
RETURN_USER_FUNC_SKIP	: This code is interpreted as PROCESS_DATA

There could be also used in the user function following combinations:
status= RETURN_WRITE_SKIP **|** RETURN_USER_FUNC_SKIP
status= RETURN_READ_SKIP **|** RETURN_USER_FUNC_SKIP

In general, user function refers to the rval field of the epics channel. The user function should be written with special percussion since the EPICS rval filed for the mbbi, mbbo, mbbiDirect and mbboDirect is declared as short (2 bytes) and for ai and ao is declared as long (4 bytes). The val filed for longin and longout records is declared as long (4 bytes).

The example skeleton for user function looks like:

```
int myGPMMFunc(
    unsigned short card,
    unsigned short signal,
    struct recDesc  *parm,
    unsigned short   sRecordType,
    unsigned short   sRecordPass,
    char *chanName,
    void *pVal
)
{
  unsigned long  lVal;
  unsigned short sVal;
  short                 status=RETURN_OK;

/* for ai ao record. lVal refers to the rval field */
lVal= *((unsigned long  *)pVal);
sVal= *((unsigned short *)pVal);   /* for mbbi mbbo bi bo */


  if(sRecordPass==REC_ENTER){
/*-------------------------------------------*/
/* code executed on function enter */
/*-------------------------------------------*/
     if(sRecordType==REC_AI){
              printf("Record AI\n");
              printf("myGPMMFunc C%d S%d @%s %s
0x%.8X\n",card,signal,
              parm->sDesc,chanName,(unsigned int) lVal);
     }
}
  else{
/*------------------------------------------*/
/* code executed on function exit  */
/*------------------------------------------*/
    printf("FUNCTION PASS on EXIT\n");
  }

  return (status);

}
```

4. The user Interrupt Service Routine

When the interrupts are used by means of the addGPMMInterrupt function the ISR routine should be specified. In most cases it is sufficient to use the default ISR defultGPMMISR which does all what is needed to process the EPICS channel. User however can define its own routine to perform some additional actions.

The skeleton of user ISR routine is presented below:

```
void myGPMMIsr(struct io_GPMM  *pPrivat){

/* Insert your code below */


/* end of your code */

  pPrivat->iIntrCounter++;
  scanIoRequest(pPrivat->paioscanpvt);
}
```

5. Example of the startup script and the database

The startup script example:

```
require "gpmm"

#GPMMConfigure(cardNr, baseAddr, cardName, addrAccessMode,
dataAccessMode)
#addGPMMRegister(cardNr, refNr, addrOffs, bytesNr, regName, userFunc)

GPMMConfigure(0,0x01600000,"DAC","A32","D32")
addGPMMInterrupt(0,3,254, defaultGPMMISR)

addGPMMRegister(0,0,0x20,4,’R’,"Firmware",debugGPMMFunc)
addGPMMRegister(0,1,0x40,4,’W’,"Control",0)

GPMMConfigure(1,0x02000000,"ADC","A24","D32")
addGPMMRegister(1,0,0x10,4,0,’R’,"Status")
```

In the given example, there are configured two cards i.e. "DAC" and "ADC". The DAC card contains two registers i.e. "Firmware" and "Control". The DAC card uses interrupt (Vector 254, Level 3). The GPMM-TEST:SOFT-TRIG channel is trigger when the interrupt occurs.

The EPICS data base example:

```
record(ai,"GPMM-TEST:ai01") {
            field(DESC, "DAC Firmware register")
            field(DTYP, "GPMM")
            field(INP, "#C0 S0 @Firmware")
            field(PINI, "YES")}

record(ao,"GPMM-TEST:ao01") {
            field(DESC, "DAC Control register")
            field(DTYP, "GPMM")
            field(OUT, "#C0 S0 @Control,iRead=Y")}

record(mbbiDirect,"GPMM-TEST:mbbiD01") {
            field(DESC, "ADC Status register")
            field(DTYP, "GPMM")
            field(NOBT, "16")
            field(INP, "#C1 S0 @Status")}

record(mbbiDirect,"GPMM-TEST:mbbiD02") {
            field(DESC, "ADC Status register")
            field(DTYP, "GPMM")
            field(NOBT, "16")
            field(INP, "#C1 S2 @Status")}

record(mbbiDirect,"GPMM-TEST:SOFT-TRIG") {
            field(DESC, "ADC Control register")
            field(SCAN, “I/O Intr”)
            field(DTYP, "GPMM")
            field(NOBT, "16")
            field(INP, "#C0 S2 @Control")}
```

6. The debug tools

There is a way to check if the hardware settings given in the startup script are correct. In order to perform such a check up first the ioc should be booted and then type:

```
dbior "drvGPMM",1
```

to get all configuration and status information.

```
dbior "drvGPMM",2
```

to get the readout for defined registers. If the readout is not correct that means that most probably the hardware settings were not correct.

7. Change Log

Version 3.0.4 :

1. It was added the support for waveform record in order to use it in addition as an output one. To do so add the parameter OUT in the INP field. See the chapter 2.

2. It was introduced the new feature for waveform record in order to use SEQuential memory read/write capability. To do so add the parameter SEQ in the INP field. See the chapter 2.

Version 3.0.3 :

1. It was added the device support for: MBBO, MBBI, LONGIN, LONGOUT records.

2. It was introduced the new feature for the **iRead** option. If **iRead** is set to **VAL** then the record is initialized from the VAL field. This feature could be used by auto save and restore mechanism.

3. It was added the functionality to deal with Linux OS.