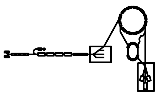

Kochbuch zur Portierung von USRs auf einen Gruppenmicro unter Linux

L. Hechler

GSI, Gesellschaft für Schwerionenforschung mbH
Planckstraße 1, D-64291 Darmstadt
Tel.: 06159/71-0

Issued: 21. März 2006
Key: G-GUP-02
Printed: 30. Juli 2008

Die USRs sind die gerätespezifische Software auf der GµP-Ebene. In ihnen steckt mittlerweile recht viel Grips, nicht wenige wurden über die Jahre sehr verfeinert, und nicht zuletzt ergeben sie zusammen genommen ziemlich viel Code. Aus diesen Gründen ist eine Neuentwicklung dieser Software für die PPC-GµP-Ebene in akzeptabler Zeit ausgeschlossen.

Bestehende USRs sollen deshalb weitgehend „unangetastet“ in die PPC-GµP-Umgebung integriert werden. Ihre Semantik, also ihre Funktionalität, kann dadurch erhalten werden. Nur die Syntax muss an die neue, objektorientierte Umgebung angepasst werden.

Dieses Kochbuch beschreibt Schritt für Schritt die notwendigen Anpassungen der USRs. Beschrieben werden außerdem die Entwicklungsumgebung wie Directories und Dateien, die USR-Bibliothek, die Klasse DevInfo sowie Werkzeuge zum Erstellen, Ablegen und Versionieren der USRs. Nicht zuletzt gibt es einige Tipps zum Thema (remote) Debugging.

Änderungsprotokoll		
Datum	Name	Kommentar
30. 11. 2004	LH	kreiert als usrs2ppc.tex
15. 08. 2005	LH	mal wieder; nach dem Umzug auf das asl-Cluster
3. 01. 2006	LH	Neuer Anlauf; Dateien in Subversion
28. 02. 2006	LH	Zusammenfassung, Kapitel 4 bis 7 und Anhang vorläufig fertig
21. 03. 2006	LH	Einige Ergänzungen und Korrekturen nach Umlauf
3. 01. 2007	LH	Komplettüberarbeitung für Release 4 begonnen
28. 02. 2007	LH	Klaudias Hinweise eingearbeitet
22. 03. 2007	LH	Erst mal wieder fertig
16. 04. 2007	LH	EqModGm innerhalb des Namespaces DeviceAccess
24. 07. 2008	LH	Hinweis Gerätekonstanten im DPR
30. 07. 2008	LH	Hinweis zu omniORB und TIMEOUT-Macro

Inhaltsverzeichnis

1. Eclipse **5**

2. Directories **5**

3. Dateien **6**
3.1. Header-Dateien 6
3.2. Incode-Dokumentation 7

4. Device Definitions **10**
4.1. `gm-dev-def.h` 10
4.2. `gm-dev-def.hh` 13

5. Devices **14**
5.1. `gm-device.hh` 14
5.2. `gm-device.cc` 15

6. USRs **19**
6.1. `gm-structtypes.hh` 19
6.2. `gm-usrs.hh` 21
6.3. `gm-usrs.cc` 24
6.4. Mapping von AccData und SISDataTypes 31
6.5. Error-Handling 31

7. EQMs **32**

8. Tools **34**
8.1. Allgemeine Tools 34
8.2. USRs generieren, testen, freigeben 35
8.3. EQMs generieren, testen, freigeben, laden 36

9. SE-Umstellung von V08 auf V09 **37**

10. Debugging **37**
10.1. Remote Debugging 37

A. Beispiel-Code Gerätemodell MX **38**
A.1. `mx-dev-def.hh` 38
A.2. `mx-device.hh` 41
A.3. `mx-device.cc` 42
A.4. `mx-usrs.hh` 46
A.5. `mx-usrs.cc` 49

Online Reference Manuals **54**

Literatur **54**

Index **55**

1. Eclipse

Geräte-Software wird weiterhin mit Hilfe von Eclipse entwickelt. Das verwendete Repository ist nun Subversion (Subclipse plug-in) und nicht mehr CVS, wie noch auf dem `lbg`-Cluster.

EQMs für M68k sowie USRs für M68k *und* PPC werden in *einem* Projekt behandelt. In der Regel sind EQMs für `V08` und `V09` übersetz- und linkbar, es gibt also nur *eine* Quelldatei für beide Systemversionen. Für USRs gibt es getrennte Quelldateien, wobei in der Regel M68k-USRs die Dateiendungen `*.h` und `*.c` haben (`V08`), und PPC-USRs die Endungen `*.hh` und `*.cc` (`V09`).

Alle Gerätemodelle sind im Repository unter `trunk/eq-models/<Gerätemodell>` zu finden. Hinweise zur Einrichtung bzw. Benutzung von Eclipse und Subversion sollte man auf der BEL-Website `http://www.acc.gsi.de/wiki/view/Dokumentation/DiverseAnleitungen` finden.[1]

2. Directories

Gerätemodell-Kürzel in Directory-Namen der Gerätesoftware werden hier allgemein mit `gm` bezeichnet, also etwa `~/workspace/gm`.

Reservierte[2] Gerätemodell-Projekte liegen üblicherweise unter dem Homedirectory des Benutzers in `~/workspace/gm`.

Bei Includes in Quelldateien werden nur noch die Dateinamen aber keine Pfade mehr angegeben, z. B. `#include <abc.h>`. Die entsprechenden Suchpfade werden dem Compiler als Option mitgegeben. Entsprechendes gilt für Bibliotheken. Möchte man trotzdem mal einen Blick werfen, dann gibt's die Umgebungsvariablen `$incasl`, `$libasl`, `$incvme`, `$libvme`, `$libppc` und `$incmsg`, die auf die entsprechenden Directories zeigen.

Alle Directories eines PPC-GμP[3] sind per NFS auf die Platten des asl-Clusters gemounted und somit auch von diesen Rechnern aus zugänglich. Davon machen Tools wie `relusrs` oder `releqms` Gebrauch. Wesentlich für die Gerätesoftware-Entwicklung sind fünf Directories. Drei davon sind mit den bekannten Tools `cdcpu`, `cdeqp` und `cdsys` erreichbar, die beiden anderen durch die Umgebungsvariablen `$libppc` und `$tsteqp`.

Hier eine Gegenüberstellung der Gerätesoftware-Directories, wie sie von PPC und ASL aus gesehen werden:

PPC-GμP	asl-Rechner
/opt/acc/cpu	/usr/local/acc/eldk/<PPC-GuP>/opt/acc/cpu
/opt/acc/eqp	/usr/local/acc/eldk/ppc_6xx/opt/acc/eqp
/opt/acc/sys	/usr/local/acc/eldk/ppc_6xx/opt/acc/sys
/usr/local/lib	/usr/local/acc/eldk/ppc_6xx/usr/local/lib
/usr/local/scratch/eqp	/usr/local/acc/eldk/ppc_6xx/usr/local/scratch/eqp

Zu beachten ist, dass `/opt/acc/cpu` CPU-spezifisch ist, jeder PPC-GμP also sein *eigenes* Directory hat. Alle anderen angegebenen Directories sind CPU-übergreifend.

Weiterhin sollte man beachten, dass (Soft-) Links in Directories eines PPC-GμP nur für einen PPC-GμP gelten. Schaut man von einem asl-Rechner aus in ein solches Directory, dann werden die Links als ungültig dargestellt.

[1] wobei noch nicht alle Dokus erstellt sind
[2] In Subversion bzw. Eclipse heißt das „checkout“. Mir fällt keine passende Übersetzung ein.
[3] also die, die man sieht, wenn man sich auf einem GuP eingeloggt hat

3. Dateien

Gerätemodell-Kürzel in Directory- und Dateinamen der Gerätesoftware werden hier allgemein mit `gm` bezeichnet, also etwa `~/workspace/gm` oder `gm-usrs.cc`.

Ein Gerätemodell-Projekt enthält in der Regel Dateien für M68k und PPC. Für EQMs ist das kein Problem, diese gibt es nur für M68k. USRs wird es, zumindest für eine Übergangszeit, für M68k- und PPC-GμP geben. Damit es hier nicht zu Namenskonflikten kommt, gelten folgende Regeln:

1. Dateien für EQMs heißen wie bisher.
2. Dateien für M68k-USRs heißen wie bisher.
3. Dateien für PPC-USRs haben die Endungen `*.hh`, `*.cc` und heißen ansonsten wie bisher.
4. Dateien für EQMs *und* USRs (egal ob M68k- oder PPC-USRs), bzw. Dateien für M68k-USRs *und* PPC-USRs haben die Endungen `*.h` und `*.c`.

Bei der Implementierung von Gerätesoftware der Version 09 sind folgende Dateien relevant:

1. Komlett neu zu erstellende Dateien:

   ```
   gm-structtypes.hh
   gm-dev-def.hh
   gm-device.hh
   gm-device.cc
   ```

2. Neu zu erstellende Dateien, aber auf Basis der M68k-USRs:

   ```
   gm-usrs.hh
   gm-usrs.cc
   ```

3. Anzupassende Dateien:

   ```
   gm-dev-def.h
   gm-<variant>.h
   gm-eqms.c
   ```

Bestehende Dateien eines Gerätemodells der Version 08 werden, bevor es losgeht mit den Anpassungen, mit Eclipse reserviert (checkout). Eclipse legt dabei ein Directory `~/workspace/gm` an, in das die Gerätesoftware kopiert wird. Diese Dateien werden nun, soweit notwendig, bearbeitet. Und in diesem Directory werden auch die neuen Dateien angelegt.

3.1. Header-Dateien

1. Damit Deklarationen nur genau einmal durchgeführt werden, müssen alle Header-Dateien (`*.h`, `*.hh`) mit

   ```
   #ifndef __<filename>__
   #define __<filename>__

   ...

   #endif
   ```

versehen werden, wobei - und . in Dateinamen durch _ ersetzt werden und alles komplett groß geschrieben wird. Beispiel gm-dev-def.hh:

```
#ifndef __GM_DEV_DEF_HH__
#define __GM_DEV_DEF_HH__

...

#endif
```

3.2. Incode-Dokumentation

1. Mindestens alle Header-Dateien *.hh sollen mit einer Incode-Doku auf Basis von Doxygen versehen werden. Entwickler von Anwendungs-Software sollten mit Hilfe dieser dann online zur Verfügung stehenden Doku in der Lage sein, Gerätezugriffe zu implementieren, ohne (in der Regel) auf weitere Dokumentationen zurückgreifen zu müssen.

2. In gm-dev-def.hh mit \mainpage beginnen. Daraus wird die allgemeine Überschrift und eine mehr oder weniger kurze Einleitung für das Gerätemodell generiert.

```
/** \mainpage MX - Equipment Model for Pulsed Magnets

   The UNILAC and all beam transport lines are equipped with
   magnets that are operated in pulsed mode.

   Pulsed mode means, that...
   ...

   \author Ludwig Hechler
   \date 21. Feb. 2006
   \version 9.12
 */
```

3. Im Kopf jeder Datei, auch in gm-dev-def.hh, mit einer kurzen, datei-spezifischen Einleitung beginnen. Hier als Beispiel die USRs des Gerätemodells MX.

```
 /** \file mx-usrs.hh
     \brief USR classes for equipment model pulsed magnets.

     All USRs in the control system have the same API
     defined by the base class Usr. USR classes for
     equipment model MX (pulsed magnets) are declared here.

     \author Ludwig Hechler
     \date 22. Aug. 2005
     \version 14.Dez.04, 09.00.00, LH, Created \n
              22.Aug.05, 09.00.01, LH, CORBA-independend AccData etc. \n
 */
```

Es ist sinnvoll, *alle* *.hh- und *.cc-Dateien zumindest mit einer solchen Einleitung zu versehen.

Nicht vergessen, jede \version-Zeile mit einem \n abzuschließen.

4. Dann wird jede USR-Klasse dokumentiert, hier wieder am Beispiel MX...

```
/**
   \brief Read field actual value.
   \property
     <table border="0" cellspacing="0">
     <tr><td>Name:</td><td>FIELDI</td></tr>
     <tr><td>Mode:</td><td>Read</td></tr>
     <tr><td>Therapy lock:</td><td>None</td></tr>
     <tr><td>Category:</td><td>Slave</td></tr>
     </table>
*/
class ReadFieldI : public Usr
```

5. ...sowie die verwendete Methode der Klasse, die ja `read()`, `write()` oder `call()` sein kann.

```
  /**
     \param vrtAcc The virtual accelerator for which the actual value
            should be read.
     \param para ULong; a single value with \n
            1 = read actual field calculated via DCCT value, \n
            2 = read raw hall probe value.
     \param data Float32; field actual value in Tm.
     \param stamp Time and event stamp of least measured actual value.
     \param eficd EFICD information of least measured actual value.
  */
  virtual AccDevRetStatus read(SLong vrtAcc,
                               const AccData& rcvPara,
                               AccData& sndData,
                               AccStamp& stamp,
                               AccEFICD& eficd);
```

6. Zur Generierung der Incode-Doku mit Doxygen muss auf `workspace/gm/doc` die Datei `Doxyfile` existieren. Diese steuert, was die Inputs sind und wie die Outputs aussehen. Man kann die Datei von einem bereits bestehenden Gerätemodell kopieren oder mit '`doxygen -g`' generieren.

   Anschließend müssen (bzw. können) folgende Tags angepasst werden.

```
PROJECT_NAME           = "GM-USRs"
PROJECT_NUMBER         = 9.03
OUTPUT_DIRECTORY       = .
ALWAYS_DETAILED_SEC    = YES
ALIASES                = "property=\par Property:\n"  \
                         "call=\par Call:\n"
EXTRACT_STATIC         = YES
INPUT                  = ../gm-dev-def.hh ../gm-structtypes.hh \
                         ../gm-additional-file.hh ../one-more-file.hh \
                         ../gm-device.hh ../gm-usrs.hh
INCLUDE_PATH           = /usr/local/acc/develop/include/asl/v01 \
                         /usr/local/acc/develop/include/vme/v09
EXAMPLE_PATH           = .
EXAMPLE_PATTERNS       = *.hh *.cc
COLS_IN_ALPHA_INDEX    = 2
```

```
QUIET                  = YES
WARN_IF_UNDOCUMENTED   = NO
GENERATE_LATEX         = NO
```

Zu einigen Tags hier ein paar Hinweise:

`PROJECT_NUMBER` sollte die Versionsnummer der Geräte-Software sein (Eclipse tag), auf die sich die Dokumentation bezieht.

`ALIAS` definiert die neuen Tags `\property` und `\call`, die in der Incode-Doku benutzt werden.

`INPUT`. Hier müssen *alle* Quellen angegeben werden, die Incode-Doku enthalten.

`EXAMPLE_PATH` und `EXAMPLE_PATTERNS` ist sinnig, wenn man in der Incode-Doku Beispielcode mit `\example beispiel.cc` inkludiert.

Alle Tags ab `QUIET` sind eher Geschmackssache.

4. Device Definitions

4.1. gm-dev-def.h

1. Dies ist im Grunde die Originaldatei der M68k-Implementation. Es sind einige kleinere Änderungen notwendig, so dass sie im Großen und Ganzen erhalten bleiben kann, wie sie ist.

   Wesentlich ist, dass die umgestellte Datei in der Regel übersetzbar sein muss für EQMs und USRs der Versionen 08.dd.pp und 09.dd.pp, da die Software für die meisten Gerätemodelle während einer Übergangszeit für V08 *und* V09 geplegt werden muss. USRs V09 laufen auf einem PPC-GuP, die drei anderen auf einer M68k-CPU.

2. Noch bestehende Unterscheidungen in `CADUL` und `__GNUC__` sollen entfernt werden.

   Quellcode 1: Include unter M68k

   ```
   #ifdef __GNUC__
     #include "gendatatype.h"
   #endif
   #ifdef CADUL
     #include "lib68:gendatatype.h"
   #endif
   ```

   Quellcode 2: Include unter PPC

   ```
   #include <gendatatype.h>
   ```

3. Bei einigen Gerätemodellen werden die Gerätekonstanten ins Dualport-RAM der SE kopiert. Man hat das aus zwei Gründen gemacht.

   a) Weil mindestens eine der Konstanten tatsächlich auf der SE-Seite benötigt wird.

   b) Aus Performance-Gründen. Der Zugriff auf die lokale DB hat früher sehr lange gedauert.

   Gerätekonstanten sollten in Zukunft nur noch ins Dualport-RAM kopiert werden, wenn sie auch auf der SE-Seite benötigt werden! In allen anderen Fällen sollten sie komplett aus dem Dualport-RAM entfernt werden.

   Um im Dualport-RAM zu kennzeichnen, dass die Konstanten gültig sind, wurde ein weiteres DPR-Element, z. B. `constActual`, eingeführt.

   Quellcode 3: Gerätekonstanten im DPR unter M68k

   ```
   typedef struct MDataType {
     ...
     /*--------------------------------------*/
     /* Device constants from local database */
     /*--------------------------------------*/
     ULong constActual;
     DevConstType devConst;
     ...
   ```

   Dieses Element wird jetzt Teil des Konstanten-Records, weil es auch von den Konstanten, die beim Geräteobjekt liegen, benötigt wird. Da `gm-dev-def.h` sowohl für V08 als auch für V09 verwendbar sein soll, sieht die entsprechende Stelle im `MDataType` nun so aus:

Quellcode 4: Gerätekonstanten im DPR für M68k und PPC

```
typedef struct MDataType {
  ...
  /*--------------------------------------*/
  /* Device constants from local database */
  /*--------------------------------------*/
#ifdef SYSV08
  ULong constActual;
  DevConstType devConst;
#else
  DevConstDesc devConst;
#endif
  ...
```

Mit `#ifdef` unterscheidet man, welche Struktur für V08 und welche für V09 übersetzt werden soll. Die Macros `SYSV08` bzw. `SYSV09` werden mit Hilfe von `vmeconfig` und abhängig von der Voreinstellung (`sys68v08`, `sysvmev09`) als Option für den Compiler-Aufruf entsprechend generiert.

Der neue Typ `DevConstDesc` muss natürlich vorher entsprechend deklariert werden.

```
typedef struct DevConstType {
  ...  // bisherige Konstanten-Deklaration
}

typedef struct DevConstDesc {
  Boolean valid;
  DevConstType data;
} DevConstDesc;
```

Dabei sollten die Elemente, wenn möglich, `valid` und `data` heißen. Weitere Elemente, wie z. B. bei MX, sind möglich.

Diese Änderung muss auch in den EQMs berücksichtigt werden! Siehe dazu Kapitel 3, Punkt 7 auf Seite 32.

Achtung: Da `DevConstDesc` im DPR liegt, muss der Datentyp `Boolean` benutzt werden, der eine definierte Größe von 4 Bytes hat. Der Typ `bool` ist *nicht* erlaubt! !

4. Deklarationen für Default- und Therapie-USRs sind nun in den entsprechenden Dateien `default-dev-def.h` und `therapy-dev-def.h` untergebracht und sind für V09 nicht mehr nötig in `gm-dev-def.h`. Gemeint sind die property-spezifischen Deklarationen für `readCommand()` und `writeCommand()`, also für die Kommunikation der USRs mit EQMs über eine Command-Communication-Area. Hier ein Beispiel für die Property `POWER`:

```
/*--------------------*/
/* Property: Power, W */
/*--------------------*/
#define POWER_W__P_COUNT  0
#define POWER_W__D_COUNT  1
#define POWER_W_TIMEOUT  15   /* Seconds */

typedef UWord PowerWWorkparaType;
typedef UWord PowerWWorkdataType;
```

Da die Deklarationen aber für V08 noch benötigt werden, müssen selbige in `#ifdef SYSV08` eingeschlossen werden.

```
#ifdef SYSV08

/*--------------------*/
/* Property: Init, N  */
/*--------------------*/
#define INIT_N__P_COUNT  0
#define INIT_N__D_COUNT  0
#define INIT_N_TIMEOUT  15   /* Seconds */

...

//---------------------
// Property: Power, W
//---------------------
#define POWER_W__P_COUNT    1
#define POWER_W__D_COUNT    1
#define POWER_W_TIMEOUT    15   /* Seconds */
#endif
```

Das betrifft folgende Properties:

Default-USRs				Therapie-USRs			
INIT	N	RESET	N	OPERMODE	W	MAXEFI	W
VERSION	R	EQMERROR	R	MEDIINFO	R	DATAID	R, W
STATUS	R	ACTIVE	R, W	SAVESET	W	SAVEMST	W
INFOSTAT	R	POWER	R, W	CHECKSET	R	SECURITY	W
				MEDMEMBER	W		

Nur der `STORESET_W_TIMEOUT` muss nach wie vor für V08 *und* V09 deklariert sein.

5. Die Command-Communication-Timeouts der Default- und Therapie-USRs sind für Geräte deklariert, die am Timing teilnehmen, also auf Events hören. Für DC-Geräte sind diese eigentlich zu lang. Man sollte sie also in `gm-dev-def.h` redeklarieren. Hier wieder am Beispiel der Property `RESET`:

```
/*---------------------*/
/* Property : Reset, N */
/*---------------------*/
#ifdef SYSV08
#define RESET_N__P_COUNT  0
#define RESET_N__D_COUNT  0
#else
#undef RESET_N_TIMEOUT
#endif
#define RESET_N_TIMEOUT   4   /* Seconds */
```

Dabei sollte man darauf achten, dass `gm-dev-def.h` *nach* den Dateien `default-dev-def.h` und `therapy-dev-def.h` und *vor* den Default- und Therapy-USRs selbst inkludiert wird.

Ein vergessenes `#undef` führt zu einer Compiler-Meldung, die etwa so aussieht:

```
gm-dev-def.h:533:1: warning: "RESET_W_TIMEOUT" redefined
```

Die `typedef`s dürfen natürlich nicht redeklariert werden.

4.2. gm-dev-def.hh

1. Im Anhang A.1 auf Seite 38 ist eine Implementationsdatei am Beispiel von MX dargestellt. Diese kann als Vorlage zum Erstellen einer eigenen Datei `gm-dev-def.hh` verwendet werden. Alle Strings `MX`, `Mx` usw. (bzw. `GM`, `Gm`, ...) müssen auch hier wieder durch den Namen des Gerätemodells ersetzt werden.

2. Diese Header-Datei wird *nur* von den PPC-USRs includiert.

3. Das für USRs und EQMs gültige `gm-dev-def.h` wird innerhalb des Namespaces `EqModGm` inkludiert.

```
namespace DeviceAcess
{
  namespace EqModGm
  {
    ...
    //-----------------------------------------
    // Include gm-dev-def.h here so that its
    // definitions are part of namespace EqModGm
    //-----------------------------------------
    #include <gm-dev-def.h>
    ...
```

4. Die einzig weitere, obligatorische Deklaration ist die des Device-Data-Types. Der `DevData-Type`wurde unter M68k in `dpram-type.h` deklariert, er sieht also für alle Gerätemodelle gleich aus.

```
typedef struct DevDataType {
  MDataType mData;
  SDataType sData[MAX_VRT_ACC - MIN_VRT_ACC + 1];
} DevDataType;
```

   Die Header-Datei `dpram-type.h` gibt es auf der PPC-GμP-Seite nicht mehr.

5. Natürlich können weitere USR-spezifische Deklarationen hier implementiert werden, z. B. solche, die bisher in `gm-dev-def.h` vorgenommen wurden und dort nicht hingehören.

5. Devices

Geräte werden in der neuen PPC/Linux-Umgebung als Objekte repräsentiert. Für jedes Gerät an einem VME-Rahmen, genauer gesagt, für jedes Gerät, das in der lokalen Datenbasis enthalten ist, existiert ein Objekt auf dem PPC-GμP.

Da alle Geräte das gleiche API[4] haben, ist der größte Teil der Funktionalität einer Geräteklasse in der Mutterklasse `VmeDevice` bzw. deren Mutterklasse `AccDevice` implementiert. Nur die wenigen gerätemodell-spezifischen Eigenschaften müssen daher in einer Klasse `GmDevice` implementiert werden.

Gerätemodell-spezifisch sind

- die Gerätedaten im Dualport-RAM der SE und der Pointer darauf,
- die Behandlung des Support-Status (online, offline, usw.),
- die Gerätekonstanten und der Pointer darauf
- und die gerätemodell-spezifischen USRs.

Deklaration und Implemetierung einer Geräteklasse `GmDevice` geschieht in den beiden Dateien `gm-device.hh` und `gm-device.cc`, die für jedes Gerätemodell angelegt werden müssen.

5.1. `gm-device.hh`

1. Im Anhang A.2 auf Seite 41 ist eine Header-Datei am Beispiel von MX dargestellt. Diese kann als Vorlage zum Erstellen einer eigenen Datei `gm-device.hh` verwendet werden. Alle Strings `MX`, `Mx` usw. (bzw. `GM`, `Gm`, ...) müssen durch den Namen des Gerätemodells ersetzt werden.

2. Benötigte Symbole aus Namespaces, hier aus `AccAlarm`, werden explizit festgelegt.

   ```
   using AccAlarm::Alarm;
   using AccAlarm::AlarmHandler;
   using AccAlarm::IpAlarmTransceiver;
   ```

   Es soll tunlichst vermieden werden, komplette Namespaces zu importieren, also z. B. Formulierungen wie `using namespace DeviceAccess;` zu benutzen.

3. Jede Gerätemodell-Implementierung hat ihren eigenen Namespace `EqModGm` innerhalb des Namespaces `DeviceAccess`, wobei der Präfix `EqMod` bei allen gleich ist. An diesen wird der Gerätemodellname angehängt.

   ```
   namespace DeviceAccess
   {
     namespace EqModGm
     {
     ...
     }
   }
   ```

4. Eine Geräteklasse, die von `VmeDevice` erbt, muss im einfachsten Fall nur den Konstruktor und den Destruktor implementieren.

[4] Application Program Interface

```
explicit GmDevice(const string& name);
~GmDevice() {};
```

5. Im Normalfall wird es aber Gerätedaten geben, etwa die des Dualport-RAMs, so dass üblicherweise der Pointer auf diese Daten sowie eine Methode implementiert werden muss, die ihn, z. B. an die USRs, zurück liefert.

```
private:
  DevDataType* _devDataP;

public:
  ...
  DevDataType* devDataP() throw(AccDevException);
```

6. Falls die Geräte dieses Modells Gerätekonstanten haben, muss der Speicher für die Konstanten und die Methoden zum Setzen und Lesen derselben implementiert werden.

```
private:
  ...
  DevConstDesc _devConst;

public:
  ...
  void setDevConstants(const AccData& dbc) throw();
  DevConstDesc* devConstP() throw() { return &_devConst; };
```

7. Die gerätemodell-spezifischen Typen `DevDataType` und `DevConstDesc`werden in den Dateien `gm-dev-def.h` bzw. `gm-dev-def.hh` deklariert.

8. In besonderen Fällen kann die in `VmeDevice` implementierte Methode zur Ermittlung des Support-Status (online, offline, usw.) eines Gerätes nicht ausreichen. Dann muss die Methode durch eine gerätemodell-spezifische überladen werden.

```
virtual SLong checkSupStatus()
  throw(AccDevException);
```

5.2. `gm-device.cc`

1. Im Anhang A.3 auf Seite 42 ist eine Implementationsdatei am Beispiel von MX dargestellt. Diese kann als Vorlage zum Erstellen einer eigenen Datei `gm-device.cc` verwendet werden. Alle Strings `MX`, `Mx` usw. (bzw. `GM`, `Gm`, ...) müssen auch hier durch den Namen des Gerätemodells ersetzt werden.

2. Benötigte Symbole aus Namespaces, hier aus `AccAlarm`, werden explizit festgelegt.

```
using AccAlarm::Alarm;
using AccAlarm::AlarmHandler;
```

3. Der folgende Code befindet sich komplett in Namespace des Gerätemodells.

```
namespace DeviceAccess
{
  namespace EqModGm
  {
```

```
  ...
  }
}
```

4. Um Geräteobjekte eines Gerätemodells zu instanziieren, benötigt der Device-Manager `DevMan` eine Funktion `createGm`, die das für ihn tut, da er selbst von Gerätemodellspezifika nichts weiß. Diese stellt die Gerätesoftware als C-Funktion zur Verfügung.

```
extern "C" {
  AccDevice* createGm(const char* nomen) {
    return new AccDevice(new EqModGm::GmDevice(nomen));
  }
}
```

5. Ebenso benötigt der Device-Manager eine Funktion zum Löschen eines Geräteobjekts.

```
extern "C" {
  void destroyGm(AccDevice* p) {
    delete p;
  }
```

6. Im Konstruktor werden die (gerätemodell-spezifischen) Attribute der Klasse `VmeDevice` und `GmDevice` initialisiert und die USRs angemeldet.

```
GmDevice::GmDevice(const string& nomen) :
  VmeDevice(nomen, EQ_MODEL_NAME, EQ_MODEL_NR)
{
  // Init attributes
  //-------------------
  _devDataP = NULL;
  _devConst.valid = false;

  // Add GM-USRs
  //---------------
  addUsrs(_usrs, this);
}
```

   Die Konstanten `EQ_MODEL_NAME` und `EQ_MODEL_NR` sind in `gm-dev-def.h` definiert. `addUsrs()` ist eine Funktion, die in den USRs (`gm-usrs.cc`) implementiert werden muss.

7. Ebenso im Konstruktor werden Gerätekonstanten aus der lokalen DB gelesen und als Attribut des Geräteobjektes gespeichert. Die Methode `getDevConstants()` liefert die Daten in einem `AccData`-Element, die gerätemodell-spezifisch zu implementierende Methode `setDevConstants()` (siehe Punkt 9 weiter unten) bewerkstelligt die gerätemodell-gerechte Konvertierung. Da eventuell auftretende Fehler während der Instanziierung nicht an einen Benutzer gemeldet werden können, werden sie ins Logfile geschrieben.

```
  ...

  // Get/set device constants
  //-------------------------
  try {
    DeviceConstants::Dbs* db = DeviceConstants::Dbs::createDbs();
    setDevConstants(db->getDevConstants(nomen));
```

```
  }
  catch (DeviceConstants::Dbs::Error& e) {
    logmsg(LOG_ERR, "%s: GmDevice::GmDevice(): %s\n",
           nomen.c_str(), e.what());
  }

  ...
```

Gerätemodelle ohne DB-Konstanten benötigen diesen Code natürlich nicht.

8. Mit `setDevDataP()` wird der Pointer auf die gerätespezifischen Daten im Dualport-RAM der SE gesetzt. Die Methode wird von `checkSupStatus()` der Klasse `VmeDevice` aufgerufen, wenn das Gerät `online` oder `offline` wird.

```
void  GmDevice::setDevDataP() throw(AccDevException)
{
  _devDataP = static_cast<DevDataType*>(_devInfoP->devDataP());
}
```

Zu `DevDataType` siehe Kapitel 5.1, Punkt 7 auf Seite 15.

9. Gerätespezifische Konstanten werden unter PPC (im Moment[5]) direkt aus der `*.dbs`-Datei gelesen und mit `setDevConstants()` im Geräteobjekt gesetzt. Die Konstanten werden von `DevMan` zur Verfügung gestellt, der auch diese Methode aufruft. Quelle ist die entsprechend benannte Datei `<gup>.dbs` auf dem Root-Directory des GμP.

   Der Entwickler muss entscheiden, ob die Daten typsicher mit `assign<Typ>()` gelesen werden sollen, oder ob sie mit `convert<Typ>()` in den Zieltyp konvertiert werden dürfen. In der Beispieldatei im Anhang werden *alle* Daten typsicher gelesen und `dbc[3]`, was ein `SLong` ist, anschließend explizit in den Zieltyp `ULong` gewandelt.

   Im Normalfall wird man die Methode `setDevConstants()` so implementieren, dass sie bei einem Fehler keine Exception wirft (`throw()`), sondern nur das `valid`-Flag zu `false` setzt. Aufgetretende Fehler müssen in diesem Fall geloggt werden.

   Sind die Daten fehlerfrei gelesen worden, muss das `valid`-Flag `true` gesetzt werden.

```
void GmDevice::setDevConstants(const AccData& dbc)
{
  _devConst.valid = false;    // preset constants invalid

  try {
    ...                       // read constants here
    _devConst.valid = true;   // finally set constants valid
  }

  catch(AccDevException& e) {
    logmsg(LOG_ERR, "%s: Exception in setDevConstants()\n",
           nomen().c_str());
  }

  catch (runtime_error& r) {
    logmsg(LOG_ERR, "%s: Catched exception from AccData: %s\n",
           nomen().c_str(), r.what());
```

[5] Die Quelle der Konstanten wird in Zukunft eine andere sein. Das sollte sich aber nicht auf das Interface (AccData) auswirken.

```
  }
}
```

Eine `AccDevException` muss nur dann gefangen werden, wenn im `try`-Zweig eine explizit geworfen wird.

Ein `runtime_error` soll immer gefangen werden. `AccData` wirft solche Exceptions.

`logmsg()` sollte man natürlich mit aussagekräftigen Fehlertexten versehen.

10. Allgemein gilt für Methoden, die nicht von einer Anwendung aufgerufen werden, sondern z. B. vom Device Manager, dass auftretende Fehler in das Logfile zu schreiben sind. Der Code sähe z. B. so aus:

```
void GmDevice::aMethod()
{
  ...
  if (db.size() != GM_CONST_COUNT) {
    logmsg(LOG_ERR, "%s: Invalid constant count, value = %i\n",
           nomen().c_str(), db.size());
  }
  ...
}
```

Und das Ergebnis im Logfile `/var/log/equLog/messages` wäre:

```
Feb 14 12:37:42 belcg04 equLog[646]: gm-device.cc::setDevConstants:37:
  TK1MU1: Invalid constant count, value = 42
```

Mehr zum Thema Logging bitte der passenden Doku entnehmen oder Peter fragen.

11. Die Methode `devDataP()` liefert den Pointer auf die Gerätedaten im DPR. Ist der Pointer `NULL`, muss die Methode eine Exception werfen.

```
DevDataType* GmDevice::devDataP() throw(AccDevException)
{
  if (_devDataP)
    return _devDataP;
  else
    throw AccDevException(ODA_NULLPOINTER, ODA_OK,
                          "In GmDevice::devDataP(): Pointer is Null");
}
```

6. USRs

Ein wesentliches Ziel beim Umstieg auf einen PPC-GμP war es, dass USRs so einfach wie möglich aus der prozeduralen Umgebung übernommen werden können. Das bezieht sich auf die USRs selbst, also die Datei `gm-usrs.c` als auch auf die entsprechenden Header-Dateien, etwa `gm-dev-def.h`.

Der für USRs *und* EQMs zuständige „Device Definition"-Teil (`gm-dev-def.h`) bleibt weitgehend erhalten. Einige USR-spezifische Sachen können rausfliegen oder verlagert werden. Auch einige EQM-spezifische Anpassungen sind notwendig, womit erreicht wird, dass mit den selben Sources EQMs sowohl der Version 08 als auch der Version 09 generiert werden können.

Ausschließlich für USRs gibt es nun die neue Header-Datei `gm-dev-def.hh` mit einigen USR-spezifischen Deklarationen und Definitionen und dem Include von `gm-dev-def.h`.

Neu ist auch `gm-structtypes.hh`. Dort sind die Indizes für Properties mit dem Datentyp `Structure` drin, damit man einfacher und sicherer auf die einzelnen Elemente zugreifen kann. Auch für manche Arrays machen entsprechende Definitionen Sinn. Die (optionale) Datei soll sowohl auf Server- als auch auf Client-Seite benutzt werden[6].

Bisher war eine USR eine Funktion, nun ist sie eine Klasse, die von der Oberklasse `Usr` erbt. Da man Klassen-Deklaration und -Implementation in der Regel in zwei Dateien trennt, gibt's nun auch `gm-usrs.hh`.

Die drei wesentlichen Methoden der Klasse `Usr` sind `read()`, `write()` und `call()`, von denen die gerätespezifischen Klasse *genau eine* dieser Methoden überschreibt. Eine Lese-Property (z. B. `RCurrentI`) überschreibt `read()`, eine Schreib-Property (z. B. `WPositS`) überschreibt `write()` und eine Funktions-Property (z. B. `Reset`) überschreibt `call()`). Die beiden anderen, nicht überschriebenen Methoden werfen eine Ausnahme, wenn sie aufgerufen werden.

Last but not least, in `gm-usrs.cc` sind die Methoden der USRs-Klassen implementiert. Sie werden von `gm-usrs.c` übernommen und entsprechend angepasst.

Die Neuerstellung bzw. Anpassung der Dateien wird in den folgenden Kapiteln im Einzelnen erklärt.

Als Beispiel kann man die Dateien des Gerätemodells MX im Eclipse-Repository zu Hilfe nehmen (`trunk/eq-models/mx`), oder in die Beispiele im Anhang gucken.

6.1. `gm-structtypes.hh`

Dies ist ein vorläufiges Kapitel! Bessere Methoden zum Zugriff auf die Elemente einer Variablen vom Typ `AccData` werden zur Zeit (März 06) diskutiert. Vorgeschlagen wurden sogenannte „selbsterklärende Daten". Bis zu einem endgültigen Beschluss spricht nichts dagegen, die folgende Methode anzuwenden.

1. Bei Properties mit strukturierten Parametern oder Daten konnte man bisher die Empfangs- oder Sendepakete auf Server- und Client-Seite einfach mappen. D. h., man hat einen Pointer des entsprechenden Typs auf das Paket zeigen lassen und konnte so direkt auf die Elemente zugreifen. Ganz grob etwa so:

```
MagnInfoType* sndDataP;
...
sndDataP = (MagnInfoType*)dataAdd;
sndDataP->sVoltS = 47.11f;
```

[6] Diese Vorgehensweise wird wohl noch mal geändert werden. In der Diskussion sind u. a. sogenannte selbsterklärende Daten.

Das geht natürlich mit der `AccData`-Klasse nicht mehr. Stattdessen sehen die Zugriffe etwa so aus:

```
sndData[7].assign<Float32>(47.11f);
```

Hier muss man wissen, hinter welchem Index (von `sndData`) sich das entsprechende Element verbirgt.

Um auch in Zukunft ähnlich elegant auf die Daten zugreifen zu können, werden für jede Property, bei der es notwendig erscheint, drei Definitionen vorgenommen:

a) Eine Strukturdeklaration, wie bisher auch,

b) eine Aufzählung, deren Elemente die Indizes für den Zugriff auf die Elemente der Struktur definieren und

c) eine Konstante, die die Anzahl der Strukturelemente definiert.

Damit sollte man alles haben, was man braucht für den Datenzugriff.

Angenommen, eine Property heißt `MYINFO` und hat 3 Datenelemente zum Lesen des Status sowie eines Soll- und eines Istwertes. Dann sähen die Definitionen so aus:

```
namespace MyInfo {
  typedef struct {
    ULong status;
    Float32 setValue;
    Float32 actValue;
  } DataType;

  enum DataIndex {
    iStatus,
    iSetValue,
    iActValue
  };

  static const int DataElements = 3;
}
```

Der property-spezifische Namespace ist nötig, damit es bei den Elementen der Aufzählungstypen nicht zu Namenskonflikten kommt. Mit anderen Worten, für *jede* Property, die hier Definitionen hat, gibt es einen *eigenen* Namespace.

Damit kann man wieder „benamte“ Zugriffe machen und obiges Beispiel sähe so aus:

```
sndData[MyInfo::iVoltS].assign<Float32>(47.11f);
```

2. Bei der Namensgebung sollte man sich an ein paar Regel halten, um die Wiedererkennung zu erleichtern.

   - Alle Definitionen für eine Property befinden sich in einem eigenen Namespace, der heißt wie die Property.
   - Muss man gleichnamige Lese- und Schreib-Properties unterscheiden, dann wird dem Property-Namen ein zusätzliches `R` bzw. `W` vorangestellt, z. B. `namespace RCurrentS` und `namespace WCurrentS`.
   - Datendeklarationen haben den Präfix `Data`.
   - Parameterdeklarationen haben den Präfix `Para`.

- Die Strukturdeklaration hat den Suffix `Type`.
- Die Aufzählung hat den Suffix `Index`.
- Die Konstante für die Anzahl der Elemente hat den Suffix `Elements`.
- Um die Strukturelemente deutlicher von den Aufzählungselementen zu unterscheiden, kann ihnen ein `s` (für Structure) vorangestellt werden, also z. B. `sStatus` oder `sActValue`.
- Die Elemente der Aufzählung heißen wie die Elemente der Struktur mit einem vorangestellten `i` (für Index). Haben die Strukturelemente ein vorangestelltes `s`, wird dieses durch das `i` ersetzt.

3. Das gleiche macht natürlich auch für Daten-Arrays Sinn, wenn jedes Array-Element eine eigene Bedeutung hat, die man „benamsen“ möchte.

4. Da `AccData` auf Client- und Server-Seite benutzt wird, kann und soll man diese Definitionen natürlich auch auf der Anwenderseite benutzen.

   Wo Anwendungen diese Datei finden, um sie zu inkludieren, ist noch nicht vollstängig geklärt. ?

5. `gm-structtypes` muss natürlich nicht implementiert werden, wenn es nicht notwendig erscheint.

6.2. gm-usrs.hh

1. Im Anhang A.4 auf Seite 46 ist ein Auszug aus einer Header-Datei dargestellt. Diese kann als Vorlage zum Erstellen einer eigenen Datei `gm-dev-def.hh` verwendet werden. Alle Strings `GM`, `Gm` usw. müssen auch hier wieder durch den Namen des Gerätemodells ersetzt werden.

2. Damit nur einmalig deklariert wird:

```
#ifndef __GM_USRS_HH__
#define __GM_USRS_HH__
```

3. Dies sind die üblichen Includes im Kopf der USRs.

```
#include <string>
#include <global-types.h>
#include <usr.hh>
#include <usrset.hh>
#include <gm-device.hh>
```

4. Benötigte Symbole aus Namespaces, hier aus `AccAlarm`, werden explizit festgelegt.

```
using AccAlarm::Alarm;
using AccAlarm::AlarmHandler;
```

5. Der gesamte folgende Code befindet sich im gerätemodell-spezifischen Namespace.

```
namespace DeviceAccess
{
  namespace EqModGm
  {
```

6. Für jede Property, genauer gesagt, für jede Kombination aus Property und Property-Mode (read, write, call), wird eine Klasse definiert. Eine Property `VOLTS`, mit der man eine Wert lesen *und* schreiben kann, wird also durch *zwei* Klassen repräsentiert, etwa `ReadVoltS` und `WriteVoltS`.

7. Jede Klasse soll (mit Doxygen) beschrieben werden. Ein Beispiel kann man sich im genannten Anhang A.4 auf Seite 46 ansehen.

8. Klassennamen werden groß geschrieben, im Gegensatz zu den USRs unter M68k, die ja Funktionen waren und deshalb klein geschrieben wurden. Dabei gibt es zwei mögliche Schreibweisen:

   a) Schreibweise ähnlich dem bisherigen Standard. Mit `R` oder `W` als Präfix für Lese- bzw. Schreib-Properties, mit `N` für Call-Properties.

   b) Schreibweise mit ausgeschriebenem Präfix `Read`, `Write` oder `Call`.

   Hier eine beispielhafte Gegenüberstellung der alten und möglichen neuen Schreibweise.

Schreibweise	Read	Write	Call
alt	`rVoltI`	`wVoltS`	`nInit`
neu 1	`RVoltI`	`WVoltS`	`NInit`
neu 2	`ReadVoltI`	`WriteVoltS`	`CallInit`

   Zu Schreibweisen allgemein siehe den Style-Guide [3].

9. Jede gerätespezifische USR muss die allgemeine Klasse `Usr` erben.

```
class WriteVoltS : public Usr
```

10. Der Konstruktor legt fest

    a) den Namen der Property,

    b) den Property-Mode, der `PROP_MODE_READ`, `PROP_MODE_WRITE` oder `PROP_MODE_CALL` sein kann,

    c) den Therapie-Lock-Modus, der `MEDLOCK_NONE`, `MEDLOCK_ALL` oder `MEDLOCK_VRTACC` sein kann,

    d) und ob es eine `MASTER_PROPERTY` oder eine `SLAVE_PROPERTY` ist.

    Außerdem wird der Pointer auf das Gerät initialisiert, damit die Property auf dessen Dualport-RAM zugreifen kann[7].

```
WriteVoltS(GmDevice* dev) :
  Usr("VOLTS", PROP_MODE_WRITE, MEDLOCK_VRTACC, SLAVE_PROPERTY),
  _dev(dev) {};
```

    Property-Mode, Therapie-Lock-Modus und Master/Slave kann man direkt der Funktion `createMap()` der M68k-USRs (V08) entnehmen:

    Erster Buchstabe des fünften Parameters (`property_class`):
    `R` ≙ `PROP_MODE_READ`, `W` ≙ `PROP_MODE_WRITE`, `N` ≙ `PROP_MODE_CALL`.

    Erster Buchstabe des zehnten Parameters (`fct_type`):
    `M` ≙ `MASTER_PROPERTY`, `S` ≙ `SLAVE_PROPERTY`.

[7] Hier wird es in Zukunft möglicherweise noch weitere Parameter geben. Im Großen und Ganzen könnte es das sein, was heute durch `createMap()` versorgt wird.

Zweiter Buchstabe des zehnten Parameters (`fct_type`):
`N` ≙ `MEDLOCK_NONE`, `L` ≙ `MEDLOCK_ALL`, `V` ≙ `MEDLOCK_VRTACC`.
Ist hier kein Buchstabe angegeben, gilt der Default `L`.

11. In seltenen Fällen möchte man eine bestehende USR unter einem weiteren Namen anmelden. Dazu implementiert man einfach einen zweiten Konstruktor, der als zusätzlichen Parameter den Property-Namen erhält.

```
WriteVoltS(GmDevice* dev, const string& property) :
  Usr(property, PROP_MODE_WRITE, MEDLOCK_VRTACC, SLAVE_PROPERTY),
  _dev(dev) {};
```

In `addUsrs()` (siehe Kapitel 6.3, Punkt 22 auf Seite 30) erzeugt man dann ein weiteres Objekt der Klasse `WriteVoltS` mit dem alternativen Property-Namen.

```
ua.addUsr(new WriteVoltS(vd));              // with default name
ua.addUsr(new WriteVoltS(vd, "MAGNSVCS")); // with auxilliary name
```

12. Der Destructor hat im Normalfall keine spezielle Funktionalität.

```
~WriteVoltS() {};
```

13. Der Funktionskopf einer USR war unter M68k immer gleich.

```
static void wVoltS(void* recvHeadAdd, void* sendHeadAdd,
                   void* recvParmAdd, void* dataAdd)
```

Unter PPC gibt es drei mögliche Methoden `read()`, `write` und `call()` für Lese-, Schreib- bzw. Funktions-Properties, die sich in ihrer Parameterliste unterscheiden.

```
virtual AccDevRetStatus read(SLong vrtAcc,
                             const AccData& rcvPara,
                             AccData& sndData,
                             AccStamp& stamp,
                             AccEFICD& eficd)
    throw(AccDevException);

virtual AccDevRetStatus write(SLong vrtAcc,
                              const AccData& rcvPara,
                              const AccData& rcvData)
    throw(AccDevException);

virtual AccDevRetStatus call(SLong vrtAcc,
                             const AccData& rcvPara)
    throw(AccDevException);
```

Dabei implementiert eine gerätespezifische Klasse immer nur *genau eine* Methode. Es gab Überlegungen, die Methoden `read()` und `write()`, z. B. für die Property `FIELDI`, in *einer* Klasse zusammenzufassen. Das geht aber nicht so einfach, weil jede USR u. a. mit den Eigenschaften `PROP_MODE_READ/WRITE/CALL` und `MEDLOCK_NONE/VRTACC/ALL` instanziiert wird. Lese- und Schreib-USR unterscheiden sich aber in diesen Eigenschaften.

14. Als privates Attribut muss noch der Zeiger auf das Gerät deklariert werden.

```
GmDevice* _dev;
```

15. Hier noch mal die komplette Deklaration einer Lese-Property am Beispiel MX. Alle spezifischen Elemente, die bei Deklarationen weiterer Properties angepasst werden müssen, sind unterstrichen[8].

```
class ReadFieldI : public Usr
{
public:
  ReadFieldI(MxDevice* dev) :
    Usr("FIELDI", PROP_MODE_READ, MEDLOCK_NONE, SLAVE_PROPERTY),
    _dev(dev) {};
  ~ReadFieldI() {};

  virtual AccDevRetStatus read(SLong vrtAcc,
                               const AccData& rcvPara,
                               AccData& sndData,
                               AccStamp& stamp,
                               AccEFICD& eficd)
     throw(AccDevException);

private:
  MxDevice* _dev;
};
```

Die Deklaration einer Schreib- oder einer Funktions-Property funktioniert entsprechend.

16. Am Ende von `gm-usrs.hh` muss noch die Funktion zum Anmelden der USRs deklariert werden. Sie sieht immer so aus:

```
void addUsrs(UsrSet& ua, GmDevice* vd);
```

6.3. gm-usrs.cc

1. Im Anhang A.5 auf Seite 49 ist wieder ein Auszug aus einer Implementationsdatei dargestellt, die als Vorlage dienen kann.

2. Implementationsdateien (`*.cc`) haben üblicherweise keine Incode-Doku. Man sollte aber wenigstens den Kopf in Doxygen dokumentieren. Dadurch kann man den kompletten Quellcode leicht in die Doku mit aufnehmen, indem man einfach die Datei in `Doxygen` mit aufnimmt. Beispiel `MX`:

```
/** \file mx-usrs.cc
    \brief USR class implementations for equipment model pulsed magnets.
    \author Ludwig Hechler
    \date 22. Aug. 2005
    \version 14.Dez.04, 09.00.00, LH, Created \n
             22.Aug.05, 09.00.02, LH, CORBA-independend AccData etc. \n
*/
```

Siehe dazu auch Kapitel 3.2 auf Seite 7.

3. USRs für M68k sind so implementiert, dass auch vom C++-Compiler C-Code generiert wird.

[8] Der Property-Name "`FIELDI`" müsste auch unterstrichen sein. Das geht aber in dieser LaTeX-Umgebung nicht.

```
#ifdef __cplusplus
extern "C" {
#endif

...

#ifdef __cplusplus
}
#endif
```

Das darf für die PPC-USRs natürlich *nicht* übernommen werden.

4. Dies sind die üblichen Includes im Kopf der USRs.

```
#include <devinfo.hh>
#include <usr-support.hh>
#include <default-dev-def.h>
#include <therapy-dev-def.h>
#include <std-msg.h>
#include <gm-msg.h>
#include <gm-usrs-version.hh>
#include <gm-structtypes.hh>
#include <gm-dev-def.hh>
#include <gm-usrs.hh>
```

Wobei gilt, dass `therapy-dev-def.h` nur für Gerätemodelle mit Therapieerweiterung notwendig ist, dass `std-msg.h` nur benötigt wird, wenn man Standard-Messages benutzt, dass `gm-structtypes.hh` optional ist (siehe Kapitel 6.1 auf Seite 19) und dass textttgm-usrs-version.hh von `vmeconfig` generiert wird.

5. Definitionen von USR-Timeouts und USR-Nummern sind nicht mehr nötig. Sie werden nicht übernommen.

```
/*----------------------------*/
/* USR timeout = 5s (4s + 1s) */
/*----------------------------*/
#define TIMEOUT 4

/*-------------*/
/* USR Numbers */
/*-------------*/
#define W_COPYSET_NR    40
#define W_CURRENTS_NR    1
#define R_CURRENTS_NR    2
#define R_CURRENTI_NR    3
...
```

Zudem kollidiert das Macro `TIMEOUT` mit einem gleichnamigen Macro aus `omniORB`. Bleibt es definiert in der Gerätesoftware , führt das zu einer Latte von Compile-Fehlern in `omniORB`-Dateien.

6. Benötigte Symbole aus Namespaces, hier aus `AccAlarm`, werden explizit festgelegt.

```
using AccAlarm::Alarm;
using AccAlarm::AlarmHandler;
```

7. Der gesamte folgende Code befindet sich im gerätemodell-spezifischen Namespace.

```
namespace DeviceAccess
{
  namespace EqModGm
  {
```

8. Lokale Funktionen, die nicht nach außen bekannt sein sollen, werden mit `static` deklariert. Nicht selten ist z. B. eine Funktion zum Kopieren der Gerätekonstanten ins Dualport-RAM der SE nötig:

```
static ULong copyDevConst(DevConstDesc* from, DevConstDesc* to)
{
  if (from->valid) {
    if (!to->valid) {
      to->data = from->data;
      to->valid = true;
    }
    return GM_OK;
  }
  else {
    return GM_NODEVCONST;
  }
}
```

   An dieser Stelle sei darauf hingewiesen, dass im Falle von Gerätekonstanten im Dualport-RAM *zwei* `valid`-Flags korrekt zu behandeln sind! Konstanten und deren `valid`-Flag liegen dann beim Geräteobjekt *und* eben im Dualport-RAM. Besonders für das `valid`-Flag der Konstanten im Dualport-RAM muss man sich überlegen, wo dieses initialisiert (gleich `false` gesetzt) werden soll. Möglichkeiten sind die Init- und Reset-EQMs auf SE-Seite oder im Geräte-Konstruktor auf GuP-Seite.

9. Die Quellen der Default- und Therapie-USRs werden nach wie vor *inkludiert* und nicht separat übersetzt. Die Dateien müssen *innerhalb* des gerätemodell-spezifischen Namespaces inkludiert werden (siehe Punkt 7), ansonsten gibt es Übersetzungsfehler.

```
#include <default-usrs.cc>
#include <default-usrs-vme.cc>
#include <therapy-usrs.cc>
#include <therapy-master-usrs.cc>
#include <therapy-slave-usrs.cc>
```

   Die Default-USRs, die nun getrennt sind in USRs für *alle* Plattformen (z. Zt. PPC und X86) und in VME-spezifische USRs, *müssen* inkludiert werden. Therapie-USRs sind je nach Gerätemodell optional.

10. Für jede Klasse, die in `gm-usrs.hh` definiert wurde, wird hier deren Methode, also `read()`, `write()` oder `call()`, implementiert. Als Beispiel die Methode `read()` für die Klasse `ReadFieldI`.

```
AccDevRetStatus ReadFieldI::read(SLong vrtAcc,
                                 const AccData& rcvPara,
                                 AccData& sndData,
                                 AccStamp& stamp,
                                 AccEFICD& eficd)
    throw(AccDevException)
```

11. Die sogenannten Init-USRs, in denen `createMap()` aufgerufen wurde, werden ersatzlos gestrichen.

```
static void wPowerI(void)
{
  createMap(EQ_MODEL_NR, W_POWER_NR, EQ_MODEL_NAME, "POWER   ", "W ", 0,
            Integer16, 1, BitSet16, "ML", POWER_W_TIMEOUT + 1, 1, 0);
}
```

12. Die Funktion `getStandardParameter()`

```
primStat = getStandardParameter(recvHeadAdd, &physDev, &logDev,
                                &actAcc, &dprP);
```

wird durch `checkAccessState()` ersetzt,

```
primStat = _dev->checkAccessState(this, vrtAcc);
```

wobei die beiden Parameter immer die gleichen sind, nämlich ein Pointer auf das aufrufende USR-Objekt (also auf sich selbst) und der virtuelle Beschleuniger.

13. Im M68k-System gibt es einen Pointer auf das *komplette* Dualport-RAM. Man hat im Prinzip also Zugriff auf *alle* Geräte der SE. Der Pointer (`dprP`) wird von `getStandardParameter()` gesetzt und mit `logDev` muss man das richtige Gerät auswählen.

```
DPRType* dprP;
MDataType* mdp = NULL;
...
mdp = &dprP->deviceData[logDev].mData;
```

Im PPC-System hat das Geräteobjekt einen Pointer, der *nur* auf die DPR-Daten *dieses* Gerätes zeigt. Es wird also kein `logDev` mehr benötigt. Den Pointer erhält man über die Methode `devDataP()` des Gerätes.

```
MDataType* mdp = NULL;
...
mdp = &_dev->devDataP()->mData;
```

Der `dprP` ist nicht mehr notwendig. Den `DPRType` gibt es nicht mehr. Ab `mData` bzw. `sData` ist der Zugriff dann wie bisher.

14. Auch die Gerätekonstanten sind Teil des Geräteobjekts (siehe Kapitel 5.1, Punkt 6 auf Seite 15). Sie werden durch die Methode `getDevConstants()` eingelesen (siehe Kapitel 5.2, Punkt 7 auf Seite 16. Mit der Methode `devConstP()` hat man Zugriff darauf.

```
iMax = _dev->devConstP()->data.maxCurrent;
```

Legt man sie zusätzlich im Dualport-RAM ab (siehe Kapitel 6.3, Punkt 8 auf Seite 26), ist der Zugriff im Prinzip wie bisher.

```
iMax = mdp->devConst.data.maxCurrent
```

Die Gerätekonstanten im Dualport-RAM abzulegen macht nur Sinn, wenn auch die EQMs Informationen daraus benötigen. Laufzeitprobleme durch langsame Datenbankzugriffe auf USR-Seite gibt es keine mehr.

Zur Behandlung des `valid`-Flags siehe Kapitel 6.3, Punkt 8 auf Seite 26.

15. Viele Funktionen der USR-Bibliothek sind nun als Methoden der Klasse `DevInfo` implementiert. Im Vergleich zu den oft benutzten Bibliotheksfunktionen unter M68k

```
readCommand(EQM_STATUS, dprP, physDev, STATUS_R__P_COUNT,
            &dcount, NULL, &workdata,
            STATUS_R_TIMEOUT, &primStat);

writeCommand(EQM_ACTIVE, dprP, physDev, ACTIVE_W__P_COUNT,
             ACTIVE_W__D_COUNT,&workpara, &workdata, 1,
             ACTIVE_W_TIMEOUT, &primStat);
```

fallen bei den DevInfo-Methoden die Parameter `dprP` und `physDev` weg.

```
_dev->devInfoP()->readCommand(EQM_STATUS, STATUS_R__P_COUNT,
                              &dCount, NULL, &workdata,
                              STATUS_R_TIMEOUT, &primStat);

_dev->devInfoP()->writeCommand(EQM_ACTIVE, ACTIVE_W__P_COUNT,
                               ACTIVE_W__D_COUNT, &workpara, &workdata,
                               COMMAND_ACKN, ACTIVE_W_TIMEOUT,
                               &primStat);
```

Siehe auch die Beschreibungen in [2].

16. Ein einzelner Wert (einer Lese-Property) wurde bisher so

```
*sndDataP = mdp->mSts.ulong;
```

oder auch so

```
sndDataP->status = mdp->mSts.ulong;
```

zurück geliefert. Das geht nun am besten so

```
sndData.assign(1, ULong(mdp->mSts.ulong));
```

wobei man, wenn möglich, vordefinierte Typen benutzen sollte.

```
sndData.assign(1, StatusRWorkdataType(mdp->mSts.ulong));
```

Alternativ kann man auch die Methode `push_pack()` verwenden.

```
sndData.push_back(mdp->mSts.ulong);
```

`push_back` vergrößert `sndData` um ein Element und hängt den neuen Wert hinten an. Da die initiale Größe von `sndData` gleich Null ist, hat man im Beispiel am Ende ein Element vom Typ `ULong` in `sndData`.

17. Möchte man mehrere Werte zurück liefern, dann geht das wie folgt.

```
sndData.resize(3);
sndData[0].assign<ULong>(0xFFFFFFFF);
sndData[1].assign<Float32>(47.11);
sndData[2].assign<Float32>(08.15);
```

Nach dem `resize()` ist der Datentyp der 3 Elemente unbestimmt. Erst mit `assign<Type>` wird der Typ des Elements, und gleichzeitig sein Wert, festgelegt.

Möchte man die Elemente mit einem Initialwert vorbesetzen, kann man auch

```
sndData.assign(3, Float32(0.0));
sndData[1] = 47.11f;
sndData[2] = 08.15f;
```

schreiben. Nach dem `assign(count, type)` ist der Datentyp aller Elemente festgelegt, im Beispiel auf `Float32`.

Statt der Methode `assign<>()` kann man auch den Operator `=` benutzen. Auch dieser legt den Datentyp fest. Man kann also auch jetzt

```
sndData[0] = ULong(0xFFFFFFFF);
```

schreiben. Das Element 0 bekommt damit den neuen Typ `ULong`.

Für Properties mit mehr als einem Wert sollte man allerdings die Hinweise in Kapitel 6.1 auf Seite 19) beachten.

18. Der Zugriff auf empfangene Parameter oder Daten (einer Schreib- bzw. Funktions-Property) geht im Prinzip genau so

    ```
    fields = rcvData[0].convert<Float32>();
    ```

    wobei der empfangene Wert in den angegebenen Typ, im Beispiel `Float32`, konvertiert wird. Möchte man keine Konvertierung, schreibt man

    ```
    fields = rcvData[0].value<Float32>();
    ```

    was zu einer Exception führt, wenn `rcvData[0]` nicht vom angegebenen Typ ist.

    Mit

    ```
    rcvPara.size();
    rcvData.size();
    ```

    bekommt man die Anzahl der Elemente in `rcvPara` bzw. `rcvData`.

    Der Operator `=` für Zuweisungen wie etwa

    ```
    fields = rcvData[0];
    ```

    ist nicht definiert.

19. Das Setzen von Time- und Eventstamp funktioniert im Prinzip wie bisher.

    ```
    setStamps(stamp, sdp->act[sync].stamps);
    setActTimeStamp(stamp);
    ```

    Die erste Funktion setzt die angegebenen Stamps, hier aus dem Dualport-RAM. Die zweite Funktion setzt die aktuelle Uhrzeit als Timestamp und die Eventstamp zu Null.

20. Das Setzen der EFICD-Parameter funktioniert auch wie bisher.

    ```
    primStat = setEFICD(eficd, sdp->act[sync].eficd);
    zeroEFICD(eficd);
    ```

    Die erste Funtion setzt die angegebenen Parameter, hier aus dem Dualport-RAM. Die zweite Funktion setzt alle Parameter zu Null.

21. USRs können Alarme verschicken. Dazu wird einfach ein Alarmobjekt angelegt, gefüllt und verschickt.

```
AccAlarm::Alarm alarm;
...
alarm.setActValDvtn(_dev->nomen().c_str(),      // nomenclature
                    GM_VALUE_UNDERFLOW,         // error message
                    sdp->rfc[syncRfc].fields,   // reference value
                    sdp->act[syncAct].fieldi);  // actual value
alarm.setTimestamp();                           // actual time
alarm.setEventstamp(0);                         // no event stamp
...
_dev->alarmHandler()->send(alarm);
```

Für jeden Alarmtyp gibt es eine Methode `set...` zum Füllen des Alarms. Ein direkter Zugriff auf die Alarmstruktur ist nicht mehr möglich.

22. Alle USR-Objekte müssen nach wie vor angemeldet werden. Die Default-USRs sind obligatorisch. Die Therapie-USRs sind optional.

    Bereits angemeldete USRs können, wenn nötig, ersetzt werden. Dazu entfernt man zunächst die schon angemeldete USR (`removeUsr`) und meldet anschließend eine andere (gleichen Namens) an.

    USRs können unter einem zweiten Property-Namen nochmals angemeldet werden. Dazu muss der alternative Property-Name explizit angegeben werden, hier im Beispiel die USR `WriteVoltS` (Property `VOLTS) als \verb`MAGNSVCS|.

```
void addUsrs(UsrSet& ua, MxDevice* vd)
{
  // Add default USRs
  //-----------------------------------------
  #include <add-default-usrs.cc>
  #include <add-default-usrs-vme.cc>

  // Add therapy USRs
  //-----------------------------------------
  #include <add-therapy-usrs.cc>
  #include <add-therapy-master-usrs.cc>
  #include <add-therapy-slave-usrs.cc>

  // Replace default USRs with GM-USRs
  //-----------------------------------------
  ua.removeUsr("ACTIV", Usr::PROP_MODE_WRITE);
  ua.addUsr(new WriteActive(vd));

  ua.removeUsr("POWER", Usr::PROP_MODE_WRITE);
  ua.addUsr(new WritePower(vd));

  // GM-USRs
  //---------------------------------
  ua.addUsr(new WriteFieldS(vd));
  ua.addUsr(new ReadFieldS(vd));
  ua.addUsr(new ReadFieldI(vd));
  ua.addUsr(new ReadMagnInfo(vd));
  ua.addUsr(new ReadCalc(vd));
  ...
  ua.addUsr(new WriteVoltS(vd));
```

```
    ua.addUsr(new WriteVoltS(vd, "MAGNSVCS"));
    ...
  }
```

6.4. Mapping von AccData und SISDataTypes

Ein `AccData`-Container kann leere Elemente enthalten, die dann vom Typ `void` sind. Userface kann diesen Typ nicht in einen gültigen Typ konvertieren, wenn die Anwendung Daten vom `SISDataType Structure` erwartet. Deshalb dürfen die USRs keine strukturierten Daten (Daten mit Elementen verschiedenen Typs) schicken, die `void`-Elemente enthalten. Tun sie es doch, erhält die Anwendung einen entsprechenden Userface-Fehler.

In USRs, die strukturierte Daten schicken, sollte man also die `AccData`-Methode `resize()` vermeiden,

```
sndData.resize(37);
```

da diese alle Elemente mit dem Typ `void` initialisiert.

Statt dessen nutzt man die Methode `assign()`, mit der man den initialen Typ und Wert aller Elemente explizit angeben kann.

```
sndData.assign(37, ULong(0));
```

Lokale Objekte gehen mit Hilfe des entsprechenden Konstruktors im Prinzip genau so.

```
AccData x(37, ULong(0));
```

6.5. Error-Handling

USRs bis Version 08 haben alle Ok- und Fehlermeldungen über Rückgabeparameter von `exitUsr()` zurückgegeben.

USRs ab Version 09 nutzen statt dessen die Methode `setCompl()` (siehe deren Beschreibung in [2]). Diese realisiert die Definitionen zur Fehlerbehandlung von USRs, welche lauten:

1. Ist die Schwere einer Meldung ‘Success’ oder ‘Information’, wird diese über den Funktionswert der USR (`AccDevRetStatus`) zurück geliefert.
2. Ist die Schwere einer Meldung ‘Warning’, ‘Error’ oder ‘Fatal Error’, wirft die USR eine Exception (`AccDevException`).

7. EQMs

EQMs sollen so implementiert sein, dass sie sowohl für V08 als auch für V09 übersetz- und linkbar sind.

Die reine DPR-Anpassung für V08 ist mittlerweile für alle Gerätemodelle abgeschlossen. An dieser Stelle wird nur noch auf Änderungen und Ergänzungen eingegangen, die für V09 notwendig sind. Entsprechende Anpassungen im Device-Definition-Teil wurden bereits im Kapitel 4.1 beschrieben.

1. Noch bestehende Unterscheidungen in `CADUL` und `__GNUC__` sollen entfernt werden. Aus

   ```
   #ifdef __GNUC__
     #include "gendatatype.h"
   #endif
   #ifdef CADUL
     #include "lib68:gendatatype.h"
   #endif
   ```

   wird einfach

   ```
   #include <gendatatype.h>
   ```

2. Default-Properties in V09 haben ein eigenes Device-Definition-File. Dieses muss im Kopf der EQMs entsprechend includiert werden.

   ```
   #ifdef SYSV09
     #include <default-dev-def.h>
   #endif
   #include <gm-dev-def.h>
   ...
   ```

3. Wie in Kapitel 4.1, Punkt 3 auf Seite 10 beschrieben, hat sich die Kennzeichnung für die Gültigkeit von Konstanten (im Dualport-RAM) geändert. Um im Code weiterhin kurz und prägnant darauf zugreifen zu können, behilft man sich mit einem ‘dirty trick’.

   Im Kopf der EQMs definiert man ein versionsabhängiges Macro namens `CONST_VALID`, das dem `constActual`- bzw. dem `valid`-Element der Konstanten entpricht.

   ```
   #ifdef SYSV08
     #define CONST_VALID constActual
   #else
     #define CONST_VALID devConst.valid
   #endif
   ```

   Code, der unter V08 z. B. so aussah,

   ```
   mdp = &dpr.deviceData[logDev].mData;
   if (mdp->constActual) {
   ```

   muss dann einfach durch

   ```
   mdp = &dpr.deviceData[logDev].mData;
   if (mdp->CONST_VALID) {
   ```

   ersetzt werden, womit er für V08 und V09 übersetzbar wird. Enthalten die Konstanten weitere versionsabhängige Elemente, kann man das für diese natürlich genau so machen (siehe z. B. MX).

   Wie gesagt: ‘dirty’ – aber ‘quick’.

4. Die eigentlichen Konstanten vom `DevConstType` liegen unter V09 nun im `DevConstDesc`-Element `data`. Auch hier werden also unterschiedliche Zugriffe für V08 und V09 benötigt. Das geht am einfachsten mit einem eigenen Pointer auf die Konstanten.

```
DevConstType *dcp;

#ifdef SYSV08
  dcp = &mdp->devConst;
#else
  dcp = &mdp->devConst.data;
#endif

Imin = dcp->minCurrent;
Imax = dcp->maxCurrent;
...
```

   Oder man behilft sich wieder mit einem passenden Macro, wie im vorigen Punkt beschrieben.

5. Für obligatorische EQMs sind die Namen für die `Workpara`- und `Workdata`-Typen vereinheitlicht worden. Zum Beispiel:

```
#ifdef SYSV08
  ActiveWWorkparaType *paraPoi;
  ActiveWWorkdataType *dataPoi;
#else
  DefActiveWWorkparaType *paraPoi;
  DefActiveWWorkdataType *dataPoi;
#endif
```

   Diese Definitionen finden sich für V09 in `$incasl/default-dev-def.h`.

6. Da EQMs nun für V08 und V09 generierbar sind, müssen sie nach einer Änderung und dem Commit *doppelt* getaggt werden, einmal für V08, einmal für V09. Wenn irgend möglich, sollten die Versionsnummern hinter der Systemversion die gleiche sein, also z. B. `08.01.37` und `09.01.37`.

8. Tools

Hier wird kurz dargestellt, wie man USRs und EQMs erstellen, freigeben und laden kann mit Hilfe aus V08 bekannter und zum Teil neuer Werkzeuge.

8.1. Allgemeine Tools

- Die Voreinstellungen für die Entwicklung von Geräte-Software für PPC-GμPs und SEs unter V09 werden mit `sysvmev09` vorgenommen. Unter V08 hieß das `sys68v08`.

  Die Voreinstellungen für Entwicklungen auf der asl-Ebene, einschließlich USRs für X86-Plattformen (asl-Rechner), werden mit `sysaslv01` vorgenommen. Unter VMS gab es keine Entsprechung dazu.

  Die aktuellen Voreinstellungen für *beide* oben genannten Plattformen kann man mit `sysact` vornehmen.

  Sind Entwicklungen in übergreifenden Projekten, wie z. B. `incasl`, notwendig, kann man die entsprechenden Umgebungsvariablen mit `syslocal` 'umbiegen', so dass sie auf lokale Directories im Workspace des Users zeigen.

- `cdeqp` und `cdcpu` sind aus V08 bekannt. Sie wechseln in V09 natürlich auf andere Directories und funktionieren *nur* auf asl-Rechnern, nicht auf PPC-GμPs!

  Bei `cdcpu` sollte man beachten, auf welche Directories gewechselt wird.

  ```
  cdcpu           ->  /usr/local/acc/eldk
  cdcpu belcg04   ->  /usr/local/acc/eldk/belcg04/opt/acc/cpu
  cdcpu belcs042  ->  /usr/local/acc/eldk/belcg04/opt/acc/cpu/ec2
  ```

  Mit `cdcpu` ohne Angabe von GμP oder SE landet man viel 'weiter oben' im Directory-Baum als unter V08 gewohnt.

- Mit

  ```
  genmsg -i gm-msg.msg
  ```

  generiert man unter Linux die Message-Files. Dabei werden C- und Fortran-Headerfiles in `$incasl` erzeugt, die Messages mit Code und Text in die Message- Bibliothek integriert und, bei Angabe der Option `-i`, die Datei `gm-interpret.c` generiert.[9]

  Mit

  ```
  getmsg 204324602
  ```

  kann man sich den Text für einen Message-Code ausgeben lassen. Dabei sind auch oktale (`07654321`) und hexadezimale (`0x6789ABCD`) Zahlenangaben möglich. Eventuell ist vorher noch ein

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$libasl
  ```

  notwendig, damit `getmsg` die Bibliothek findet.

- Zum Generieren der Gerätemodell-Dokumentation kann man

  ```
  gendocu gm-gm.tex
  ```

  verwenden.

[9] Zur Zeit (März 07) fehlt noch das automatische Kopieren nach und Generieren auf VMS.

- Um einen VME-Computer in verschiedenen Modes (run, boot, ram, debug) zu booten benutzt man

  ```
  vmeboot
  ```

  Zur Zeit können nur SEs gebootet werden. Die INIT-Property für PPC-GμPs ist noch nicht implementiert.

8.2. USRs generieren, testen, freigeben

Dies erfolgt grob in 4 Schritten.

1. Einstellungen mit

   ```
   vmeconfig
   ```

   vornehmen wie bisher und mit

   ```
   make [clean] [all]
   ```

   generieren. Treten keine Fehler beim Generieren mehr auf, sollten

2. die USRs getestet werden. Dazu ruft man z. B.

   ```
   copyusrs gm 09.08.15 belcg04
   ```

   auf, um die Datei `gm-usrs.so.09.08.15` nach `$tsteqp` zu kopieren und die notwendigen Links darauf für den GμP `belcg04` zu setzen. Anschließend muss der DevMan neu gestartet werden. Das geht auf einem PPC-GμP mit

   ```
   service devman stop
   service devman start
   ```

   In Zukunft kann man das (womöglich) auch mit der Property `INIT` des DevMan machen.

   Für USRs, die auf einem asl-Rechner laufen sollen, muss man explizit die gewünschte Plattform als Option angeben.

   ```
   copyusrs -cpu=x86 kgb 09.08.15 asl711
   ```

   Sind die USRs ausgetestet, werden

3. die Quellen ins Repository zurück geschrieben (Eclipse: Team→Commit) und mit einer Versionsnummer versehen (Eclipse: Team→Branch/Tag). Bitte jeweils die Kommentare nicht vergessen, damit man später in der History nachsehen kann, was wann warum passiert ist.

   Wenn auch Änderungen in den EQMs notwendig sind, ist es sinnvoll, zunächt diese vorzunehmen, bevor eine neue Versionsnummer mit Team→Branch/Tag vergeben wird. Versionsnummern gelten für das gesamte Projekt, also für USRs *und* EQMs.

4. Nun können die USRs zunächst allgemein

   ```
   relusrs gm 09.08.15
   ```

   und anschließend für spezifische GμPs

   ```
   relusrs -gup=ktrcg07 gm 09.08.15
   ```

freigegeben (released) werden.

Anschließend muss der DevMan auf dem entsprechenden GμP wieder neu gestartet werden (siehe oben), damit dieser die neuen USRs lädt.

Ein `genusrs` ist unter V09 nicht mehr vorhanden. Dieses ist durch `make` und `copyusrs` ersetzt worden.

Ebenso gibt es kein `download` mehr. Nach `copyusrs` bzw. `relusrs` und anschließendem Restart des DevMan laufen die neuen USRs.

8.3. EQMs generieren, testen, freigeben, laden

Dies erfolgt grob in 5 Schritten.

1. Einstellungen mit

   ```
   vmeconfig
   ```

   vornehmen wie bisher und mit

   ```
   make [clean] [all]
   ```

   generieren. Treten keine grundsätzlichen Fehler beim Generieren mehr auf, kann man sich mit

   ```
   geneqms var1 var2 var3 ...
   ```

   alle Varianten für ein Board (`F` oder `G`) generieren lassen.

2. Nun sollte zumindest eine Variante ausführlich getestet werden. Dazu benötigt man ein *echtes* Download (Option `-d` bzw. `--download`) der EQMs in die (Test-) SE.

   ```
   ecload -d f-gm-var1-eqms090815-ecm0903 belcs041
   ```

   Sind die EQMs ausgetestet, werden

3. die Quellen ins Repository zurück geschrieben (Eclipse: Team→Commit) und mit einer Versionsnummer versehen (Eclipse: Team→Branch/Tag). Auch hier die Kommentare nicht vergessen!

   Wenn noch Änderungen in den USRs ausstehen, zunächt diese vornehmen, bevor eine neue Versionsnummer mit Team→Branch/Tag vergeben wird.

4. Nun können die EQMs zunächst allgemein

   ```
   releqms -var=var1 f gm 09.08.15
   ```

   und anschließend für spezifische SEs

   ```
   releqms -ec=ktrcs074 -var=var1 f gm 09.08.15
   ```

   freigegeben (released) werden.

5. Das Laden der EQMs in die gewünschte (Produktions-) SE mit

   ```
   ecload f-gm-var1-eqms090815-ecm0903 ktrcs074
   ```

   ist der letzte Schritt. Dabei wird üblicherweise nur noch der Datei*name* an den PPC-GμP geschickt, nicht die komplette S-Record-Datei. Der GμP erwartet die Datei auf der Directory der angegebenen SE (wo sie hoffentlich mit `releqms` hinkopiert wurde).

9. SE-Umstellung von V08 auf V09

Die Software-Umstellung einer SE von V08 auf V09 ist leider etwas umständlich. Folgende Reihenfolge muss eingehalten werden.

1. SE in einen VME-Rahmen mit M68k-GuP (z. B. `uktcs3e7`) stecken.
2. Download Lauf-ECM V09 mit Boot-ECM V08
   a) `sys68v08`
   b) `vmeboot -b uktcs3e7`
   c) `download -verb -noboot -noinit -sys=$gecmv09 uktcs3e7`
3. SE in einen VME-Rahmen mit PPC-GuP (z. B. belcs043) stecken.
4. Download Boot-ECM V09 mit Lauf-ECM V09
   a) `sysvmev09`
   b) `ecload --noboot -v $gbecmv09 belcs043`

10. Debugging

10.1. Remote Debugging

Auf dem PPC-GμP (z. B. belcg03):

```
belcg03> gdbserver 1:23456 my-program
```

Dabei ist `1` egal, `23456` ist die Portnummer[10], über die kommuniziert wird und `my-program` das zu debuggende Programm.

Auf dem Entwicklungsrechner (z. B. asl711):

```
asl711> ddd --debugger ppc-linux-gdb my-program &
```

oder einfacher

```
asl711> dddppc my-program &
```

Im Debugger:

```
gdb> target remote belcg03:23456
```

[10] Portnummern kleiner als 16 000 sollen nicht benutzt werden, weil sie möglicherweise für andere Anwendungen reserviert sind

A. Beispiel-Code Gerätemodell MX

Zum Teil sind nur Auszüge dargstellt.

A.1. mx-dev-def.hh

```
/** \mainpage MX - Equipment Model for Pulsed Magnets

    The UNILAC and all beam transport lines are equipped with
    magnets that are operated in pulsed mode.

    Pulsed mode means, that...
    ...

    \author Ludwig Hechler
    \date 21. Feb. 2006
    \version 9.12
 */

/** \file mx-dev-def.hh
    \brief USR device definitions for equipment model pulsed magnets.

    Define the device data structure for equipment model MX,
    pulsed magnets.

    \author Ludwig Hechler
    \date 22. Aug. 2005
    \version 14.Dez.04, LH, Created \n
             22.Aug.05, LH, CORBA-independend AccData etc. \n
              2.Feb.06, LH, std messages included \n
*/

//-------------------------------------
// Use this header on GuP side only!
//-------------------------------------

#ifndef __MX_DEV_DEF_HH__
#define __MX_DEV_DEF_HH__

#include <std-msg.h>
#include <mx-msg.h>


namespace DeviceAccess
{
  namespace EqModMx
  {

    //-----------------------------------------
    // Include mx-dev-def.h here so that its
    // definitions are part of namespace EqModMx
```

```
//-----------------------------------------
#include <mx-dev-def.h>

//-------------------------------
// FieldI, CurrentI, VoltI modes
//-------------------------------
static const ULong FIELDI_VIADCCT            = 0x00000001;
static const ULong FIELDI_VIAHALL            = 0x00000002;
static const ULong FIELDI_VIADUMMYPOLYNOM    = 0x00000003;
static const ULong FIELDI_VIAMASK            = 0x000000FF;
static const ULong CURRENTI_VIADCCT          = 0x00000100;
static const ULong CURRENTI_VIAHALL          = 0x00000200;
static const ULong CURRENTI_VIADUMMYPOLYNOM  = 0x00000300;
static const ULong CURRENTI_VIAMASK          = 0x0000FF00;
static const ULong VOLTI_VIADCCT             = 0x00010000;
static const ULong VOLTI_VIAHALL             = 0x00020000;
static const ULong VOLTI_VIAMASK             = 0x00FF0000;


//------------------------
// Messages used in x2y.c
//------------------------
static const ULong USR_OK         = MX_OK;
static const ULong BL2I_RANGE     = MX_WFIELDS_RANGE;
static const ULong I2U_RANGE      = MX_WCURRS_RANGE;
static const ULong U2DAC_RANGE    = MX_WVOLTS_RANGE;
static const ULong U2I_RANGE      = STD_VALUE_RANGE;
static const ULong I2BL_RANGE     = STD_VALUE_RANGE;
static const ULong INVPOS_UNKNOWN = MX_INVPOSUNKNOWN;
static const ULong NOFIELD        = MX_NOFIELD;
static const ULong NOCURRENT      = MX_NOCURRENT;
static const ULong ACTVALHALL     = MX_ACTVALHALL;
static const ULong ACTVALDCCT     = MX_ACTVALDCCT;
static const ULong POLYNOM_OK     = MX_POLYNOM_OK;
static const ULong NOPOLYNOM      = MX_NOPOLYNOM;
static const ULong POLYNOM_INTER  = MX_POLYNOM_INTER;
static const ULong POLYNOM_RANGE  = MX_POLYNOM_RANGE;

//-----------------------------------------
// Check/Nocheck for x2y functions.
// No check should be done for actual values.
//-----------------------------------------
static const bool CHECK_RANGE = true;
static const bool NOCHECK_RANGE = false;

//--------------------------------------
// To easily create two mapping entries
//--------------------------------------
static const char emodName[][9] = {EQ_MODEL_NAME, EQ_MODEL_DEG};

//------------------
// Device Data Type
```

```
    //------------------
    typedef struct DevDataType {
      MDataType mData;
      SDataType sData[MAX_VRT_ACC - MIN_VRT_ACC + 1];
    } DevDataType;
  }
}

#endif
```

A.2. mx-device.hh

```
/** \file mx-device.hh
    \brief Device class for equipment model pulsed magnets.

    All devices in the control system have the same API.
    So most of the functionality of a device is implemented
    in the base class VmeDevice. Only equipment model specific
    characteristics must be implemented in the pulsed magnets
    device class that inherits VmeDevice.

    \author Ludwig Hechler
    \date 20. Nov. 2006
    \version 14.Dez.04, 09.00.00, LH, Created \n
             22.Aug.05, 09.00.01, LH, CORBA-independend AccData etc. \n
             20.Nov.06, 09.04.00, LH, Adaptions for release 4 \n
              9.Jan.07, 09.12.06, LH, devDataP(): Null pointer exception \n
*/

#ifndef __MX_DEVICE_HH__
#define __MX_DEVICE_HH__

#include <string>
#include <global-types.h>
#include <accdevice.hh>
#include <vmedevice.hh>
#include <mx-dev-def.hh>


namespace DeviceAccess
{
  namespace EqModMx
  {

    class MxDevice : public VmeDevice
    {
    private:
      DevDataType* _devDataP;
      DevConstDesc _devConst;

    public:
      explicit MxDevice(const string& name);
      ~MxDevice() {};
      void setDevDataP() throw(AccDevException);
      void setDevConstants(const AccData& dbc);
      DevDataType* devDataP() throw(AccDevException);
      DevConstDesc* devConstP() throw() { return &_devConst; };
    };
  }
}

#endif
```

A.3. mx-device.cc

```
/** \file mx-device.cc
    \brief Device class implementaion for equipment model pulsed magnets.
    \author Ludwig Hechler
    \date 2. Jan. 2007
    \version 14.Dez.04, 09.00.00, LH, Created \n
             22.Aug.05, 09.00.01, LH, CORBA-independend AccData etc. \n
             21.Nov.06, 09.00.02, LH, Adaptions for release 4 \n
              2.Jan.07, 09.12.05, LH, Always use ADC 1 for field-controlled PS \n
              9.Jan.07, 09.12.06, LH, devDataP(): Null pointer exception \n
*/

#include <macros.h>
#include <dbs.hh>
#include <devinfo.hh>
#include <mx-device.hh>
#include <mx-usrs.hh>
#include <mx-helpers.hh>


namespace DeviceAccess
{
  namespace EqModMx
  {

    // Provide device creator for DevMan.
    // This must be a C, not a C++ function.
    //----------------------------------------
    extern "C" {
      AccDevice* createMx(const char* nomen) {
    return new AccDevice(new EqModMx::MxDevice(nomen));
      }
    }


    // Provide device destroyer for DevMan.
    // This must be a C, not a C++ function.
    //----------------------------------------
    extern "C" {
      void destroyMx(AccDevice* p) {
    delete p;
      }
    }


    //----- Constructor -----------------------------------

    MxDevice::MxDevice(const string& nomen) :
      VmeDevice(nomen, EQ_MODEL_NAME, EQ_MODEL_NR)
    {
      // Init attributes
```

```
  //-------------------
  _devDataP = NULL;
  _devConst.valid = false;

  // Get/set device constants
  //----------------------------
  try {
DeviceConstants::Dbs* db = DeviceConstants::Dbs::createDbs();
setDevConstants(db->getDevConstants(nomen));
  }
  catch (DeviceConstants::Dbs::Error& e) {
logmsg(LOG_ERR, "%s: MxDevice::MxDevice(): %s\n",
       nomen.c_str(), e.what());
  }

  // Add MX-USRs
  //---------------
  addUsrs(usrs(), this);
}


//----- setDevDataP ----------------------------------------------

void MxDevice::setDevDataP() throw(AccDevException)
{
  _devDataP = static_cast<DevDataType*>(_devInfoP->devDataP());
}


//----- setDevConstants ------------------------------------------

void MxDevice::setDevConstants(const AccData& dbc)
{
  int i;
  int j;
  int sets;
  Float32* p;


  // Set constants invalid before filling
  // [tbs] mit mutex drumrum???
  //---------------------------------------
  _devConst.valid = false;

  try {
switch (dbc.size()) {
case 4:  _devConst.polySets = 0; break;
case 40: _devConst.polySets = 2; break;
case 76: _devConst.polySets = 4; break;
default:
  logmsg(LOG_ERR, "%s: Invalid constant count, value = %i\n",
     nomen().c_str(), dbc.size());
```

```
  throw AccDevException();  // just to skip the rest
  break;
}

_devConst.data.minCurrent = dbc[0].value<Float32>();
_devConst.data.maxCurrent = dbc[1].value<Float32>();
_devConst.data.calibration = dbc[2].value<Float32>();
_devConst.data.devsubtype = dbc[3].value<SLong>();

p = &_devConst.data.minCurrent;
for (i = 4; i < dbc.size(); i++) {
  p[i] = dbc[i].value<Float32>();
}

//----------------------------------------
// Check for illegal subtype combinations
//----------------------------------------
if ((!hasHallProbe(_devConst.data.devsubtype)) &&
    ((_devConst.data.devsubtype & DST_CTRLMASK) != DST_CTRLCURR)) {
  //------------------------------------------------------
  // Illegal subtype combination: has *no* hall probe &&
  // is or may be field controlled
  //------------------------------------------------------
  logmsg(LOG_ERR, "%s: Illegal subtype combination #1\n",
     nomen().c_str());
}
else if (hasHallProbe(_devConst.data.devsubtype) &&
     ((_devConst.data.devsubtype & DST_ACTVALMASK) == DST_ACTVAL1) &&
     ((_devConst.data.devsubtype & DST_CTRLMASK) != DST_CTRLFIELD)) {
  //-------------------------------------------------
  // Illegal subtype combination: has hall probe &&
  // has 1 actual value && is *not* field controlled
  //-------------------------------------------------
  logmsg(LOG_ERR, "%s: Illegal subtype combination #2\n",
     nomen().c_str());
}
else if (hasInverter(_devConst.data.devsubtype) &&
     (_devConst.data.devsubtype & DST_CTRLMASK) != DST_CTRLCURR) {
  //-----------------------------------------------
  // Illegal subtype combination: has inverter &&
  // is or may be field controlled
  //-----------------------------------------------
  logmsg(LOG_ERR, "%s: Illegal subtype combination #3\n",
     nomen().c_str());
}
else {
  //---------------------------------------------------
  // If all went ok, set constants valid (again)...
  // [tbs] mit mutex drumrum???
  //---------------------------------------------------
  _devConst.valid = true;
}
```

```
      }

      catch(AccDevException& e) {
    // nothing to do, just exit
      }

      catch (runtime_error& r) {
    logmsg(LOG_ERR, "%s: MxDevice::setDevConstants(): %s\n",
          nomen().c_str(), r.what());
      }
    }


    //----- DevDataP -------------------------------------------------

    DevDataType* MxDevice::devDataP() throw(AccDevException)
    {
      if (_devDataP)
    return _devDataP;
      else
    throw AccDevException(ODA_NULLPOINTER, ODA_OK,
                "In MxDevice::devDataP(): Pointer is Null");
    }
  }
}
```

A.4. mx-usrs.hh

```
/** \file mx-usrs.hh
    \brief USR classes for equipment model pulsed magnets.

    All USRs in the control system have the same API
    defined by the base class Usr. USR classes for
    equipment model MX (pulsed magnets) are declared here.

    \author Ludwig Hechler
    \date 21. Mar. 2006
    \version 14.Dez.04, 09.00.00, LH, Created \n
             22.Aug.05, 09.00.01, LH, CORBA-independend AccData etc. \n
         21.Mar.06, 09.00.02, LH, Incode docu completed \n
*/

#ifndef __MX_USRS_HH__
#define __MX_USRS_HH__

#include <string>
#include <global-types.h>
#include <usr.hh>
#include <usrset.hh>
#include <mx-device.hh>


namespace DeviceAccess
{
  namespace EqModMx
  {

    //----- Write Power ---------------------------------------------------
    /**
       \brief Switch power on or off.
       \property
       <table border="0" cellspacing="0">
       <tr><td>Name:</td><td>POWER</td></tr>
       <tr><td>Mode:</td><td>Write</td></tr>
       <tr><td>Therapy lock:</td><td>All</td></tr>
       <tr><td>Category:</td><td>Master</td></tr>
       </table>
    */

    class WritePower : public Usr
    {
    public:
      WritePower(MxDevice* dev) :
    Usr("POWER", PROP_MODE_WRITE, MEDLOCK_ALL, MASTER_PROPERTY),
    _dev(dev) {};
      ~WritePower() {};

      /**
```

```
 \param vrtAcc Not used
 \param para Not used
 \param data UWord; a single value with \n
 0 = switch on magnet's power supply, \n
 1 = switch off magnet's power supply.
  */
  virtual AccDevRetStatus write(SLong vrtAcc,
                const AccData& para,
                const AccData& data);
private:
  MxDevice* _dev;
};



//----- Read CurrentS ----------------------------------------------------
/**
   \brief Read current set value.
   \property
   <table border="0" cellspacing="0">
   <tr><td>Name:</td><td>CURRENTS</td></tr>
   <tr><td>Mode:</td><td>Read</td></tr>
   <tr><td>Therapy lock:</td><td>None</td></tr>
   <tr><td>Category:</td><td>Slave</td></tr>
   </table>
*/

class ReadCurrentS : public Usr
{
public:
  ReadCurrentS(MxDevice* dev) :
Usr("CURRENTS", PROP_MODE_READ, MEDLOCK_NONE, SLAVE_PROPERTY),
_dev(dev) {};
  ~ReadCurrentS() {};

  /**
 \param vrtAcc The virtual accelerator for which the set value
 should be read.
 \param para Not used
 \param data Float32; current set value in A.
 \param stamp Not used
 \param eficd Not used
  */
  virtual AccDevRetStatus read(SLong vrtAcc,
               const AccData& para,
               AccData& data,
               AccStamp& stamp,
               AccEFICD& eficd);
private:
  MxDevice* _dev;
};
```

```
    //----- AddUsrs -------------------------------------------------

    void addUsrs(UsrSet& ua, MxDevice* vd);

  }
}

#endif
```

A.5. mx-usrs.cc

```
/** \file mx-usrs.cc
    \brief USR class implementations for equipment model pulsed magnets.
    \author Ludwig Hechler
    \date 2. Jan. 2007
    \version 24.Feb.06, 09.00.00, LH, Created \n
             22.Aug.05, 09.00.02, LH, CORBA-independend AccData etc. \n
             24.Feb.06, 09.00.03, LH, Docu with Doxygen \n
              2.Mar.06, 09.00.04, LH, newDprData in copyDevConst \n
         14.Aug.06, 09.00.05, LH, ReadMagnSvcS: Use DAC2U, not ADCdcct2U \n
          8.Dec.06, 09.00.06, LH, Call status EQM with writeCommand() \n
*/

#include <devinfo.hh>
#include <usr-support.hh>
#include <default-dev-def.h>
#include <therapy-dev-def.h>
#include <std-msg.h>
#include <mx-msg.h>
#include <mx-usrs-version.hh>
#include <mx-structtypes.hh>
#include <mx-dev-def.hh>
#include <mx-helpers.hh>
#include <mx-usrs.hh>
#include <x2y.hh>


namespace DeviceAccess
{
  namespace EqModMx
  {

    //=========================================================================
    // Helpers
    //=========================================================================


    // Copy device constants (into dpr)
    //-------------------------------------------------------------
    static ULong copyDevConst(DevConstDesc* from, DevConstDesc* to,
                  SDataType sdata[])
    {
      int i;

      if (from->valid) {
    if (!to->valid) {
      to->data = from->data;
      to->polySets = from->polySets;
      to->valid = true;
      for (i = MIN_VRT_ACC; i <= MAX_VRT_ACC; i++) {
        sdata[i].rfc[sdata[i].syncRfc].newDprData = true;
```

```
  }
}
return MX_OK;
  }
  else {
return STD_INVCONST;
  }
}



//=========================================================================
// Include Default- and Therapy-USRs
//=========================================================================
#include <default-usrs.cc>
#include <default-usrs-vme.cc>
#include <therapy-usrs.cc>
#include <therapy-master-usrs.cc>
#include <therapy-slave-usrs.cc>



//=========================================================================
// MX-USRs
//=========================================================================


//----- WritePower -------------------------------------------------

AccDevRetStatus WritePower::write(SLong vrtAcc,
                  const AccData& rcvPara,
                  const AccData& rcvData)
{
  ULong primStat = MX_OK;
  ULong secStat = MX_OK;
  MDataType* mdp = NULL;
  SLong dataCount;
  PowerWWorkdataType workdata;


  //---------------------------
  // check if device is online
  //---------------------------
  primStat = _dev->checkAccessState(this, vrtAcc);

  if (primStat & 1) {
mdp = &_dev->devDataP()->mData;
primStat = copyDevConst(_dev->devConstP(), &mdp->devConst,
            _dev->devDataP()->sData);
  }

  if (primStat & 1) {
```

```
if (isDegaussingMagnet(_dev->devInfoP()->eqmVariantNum())) {
  //-------------------------------------------
  // Degaussing-Magnet oder Spillabbruchkicker:
  // Nur Istwertlesen möglich
  //-------------------------------------------
  primStat = MX_DEGONLYACT;
}
  }

  if (primStat & 1) {
if (rcvData.size() == 1) {
  workdata = rcvData[0].convert<UWord>();
  if (workdata == 0 || workdata == 1) {
    _dev->devInfoP()->writeCommand(EQM_POWER, POWER_W__P_COUNT,
                   POWER_W__D_COUNT, NULL, &workdata,
                   COMMAND_ACKN, CONNECT_W_TIMEOUT,
                   &primStat);
  }
  else {
    primStat = MX_POWER_UNDEF;
  }
}
else {
  primStat = STD_ILL_DATACOUNT;
}
  }

  if (primStat == MX_CHKCON_FAIL &&
  (mdp->devConst.data.devsubtype & DST_PSMASK) == DST_MSHARE) {
//---------------------------------
// Magnet with shared power supply
// that's switchable only manually
//---------------------------------
primStat = MX_MANLOAD;
  }

  return setCompl(primStat, secStat, "Write POWER");
};



//----- ReadCurrentS --------------------------------------------------

AccDevRetStatus ReadCurrentS::read(SLong vrtAcc,
                   const AccData& rcvPara,
                   AccData& sndData,
                   AccStamp& stamp,
                   AccEFICD& eficd)
{
  ULong primStat = MX_OK;
  ULong secStat = MX_OK;
  DevDataType* ddp = NULL;
```

```
  MDataType* mdp = NULL;
  SDataType* sdp = NULL;
  SRfc* sRfcP = NULL;


  //---------------------------
  // check if device is online
  //---------------------------
  primStat = _dev->checkAccessState(this, vrtAcc);

  if (primStat & 1) {
ddp = _dev->devDataP();
mdp = &ddp->mData;
sdp = &ddp->sData[vrtAcc];
sRfcP = &sdp->rfc[sdp->syncRfc];
primStat = copyDevConst(_dev->devConstP(), &mdp->devConst, sdp);
  }

  if (primStat & 1) {
if (isDegaussMagOrTH3MK1(_dev->devInfoP()->eqmVariantNum())) {
  //-------------------------------------------
  // Degaussing-Magnet oder Spillabbruchkicker:
  // Nur Istwertlesen möglich
  //-------------------------------------------
  primStat = MX_DEGONLYACT;
}
  }

  if (primStat & 1) {
if (sRfcP->newDprData) {
  //-----------------------------------------------
  // If neccessary, re-caculate the original values
  //-----------------------------------------------
  primStat = dac2AllValues(sRfcP->data.dac, &mdp->devConst.data,
              mdp->inverter, mdp->mSts.ulong, CHECK_RANGE,
              &sRfcP->origVoltS, &sRfcP->origCurrentS,
              &sRfcP->origFieldS, &sRfcP->newDprData);
}
  }

  if (primStat & 1) {
//---------------------------------------
// Simply get original current set value
//---------------------------------------
sndData.push_back(static_cast<Float32>(sRfcP->origCurrentS));
  }

  //----------------
  // Set completion
  //----------------
  return setCompl(primStat, secStat, "Read CURRENTS");
};
```

```
    //----- addUsrs -----------------------------------------------

    #define ACT_DEV_IMPL MxDevice

    void addUsrs(UsrSet& ua, MxDevice* vd)
    {
      // Add Default and Therapy USRs
      //-----------------------------------------
      #include <add-default-usrs.cc>
      #include <add-default-usrs-vme.cc>
      #include <add-therapy-usrs.cc>
      #include <add-therapy-master-usrs.cc>
      #include <add-therapy-slave-usrs.cc>

      // Replace default USRs with MX-USRs
      //-----------------------------------------
      ua.removeUsr("POWER", Usr::PROP_MODE_WRITE);
      ua.addUsr(new WritePower(vd));

      // MX-USRs
      //-----------------------------------------
      ...
      ua.addUsr(new ReadCurrentS(vd));
      ua.addUsr(new ReadCurrentI(vd));
      ua.addUsr(new WriteVoltS(vd));
      ...
    }
  }
}
```

Online Reference Manuals

AccData:	`www.acc.gsi.de/data/documentation/accdata`
AccDevice:	`www.acc.gsi.de/data/documentation/accdevice`
Alarm handling:	`www.acc.gsi.de/data/documentation/alarm`
Device Manager:	`www.acc.gsi.de/data/documentation/devman`
DevInfo:	`www.acc.gsi.de/data/documentation/devinfo`
VmeDevice, USRs & USR-Bibliothek:	`www.acc.gsi.de/data/documentation/vmedevice`
Gerätemodelle	`www.acc.gsi.de/data/documentation/eq-models/`

Der Zugriff ist (zur Zeit) beschränkt auf Mitglieder der Gruppe BEL. Zum Einloggen benutzt man seinen Wiki-Namen und das entsprechende Passwort.

Literatur

[1] Ludwig Hechler. Der Gruppenmicro unter Linux. Accelerator Controls Documentation B-GUP-01, Gesellschaft für Schwerionenforschung, Darmstadt, August 2004. (Source: $ddoc/linuxgup.tex).

[2] Peter Kainberger. The USR Support Methods. Accelerator Controls Documentation, Gesellschaft für Schwerionenforschung, Darmstadt, March 2007. Available from World Wide Web: `http://www.acc.gsi.de/some/dir/myfile.html`.

[3] Udo Krause. C/C++ Style Guide. Accelerator Controls Documentation O-SIS-10, Gesellschaft für Schwerionenforschung, Darmstadt, Dezember 2000. (Source: cstyle.tex).

Index

— **A** —

AccData
- assign(n, v) . . . 28
- assign<T>(v) . . . 28
- convert() . . . 29
- operator=() . . . 28
- push_back() . . . 28
- resize() . . . 28
- size() . . . 29
- und SISDataType . . . 31
- value() . . . 29

addUsrs() . . . 30
Alarme . . . 29

— **C** —

cdcpu . . . 34
cdeqp . . . 34
checkAccessState() . . . 27
checkSupStatus() . . . 17
- in GmDevice . . . 15

copyusrs . . . 35

— **D** —

Dateien . . . 6
- gm-dev-def.h . . . 10
- gm-dev-def.hh . . . 13
- gm-device.cc . . . 15
- gm-device.hh . . . 14
- gm-structtypes.hh . . . 19
- gm-usrs.cc . . . 24
- gm-usrs.hh . . . 21
- Header-∼ . . . 6

Debugging . . . 37
- Remote ∼ . . . 37

Default USRs . . . 26
DevConstDesc . . . *siehe* Gerätekonstanten
devConstP() . . . 27
DevConstType . . . *siehe* Gerätekonstanten
devDataP() . . . 27
DevDataType . . . 13, 15
Device Definitions . . . 10
- `TIMEOUT`-Macro . . . 25
- `default-dev-def.h` . . . 11, 25, 32, 33
- `therapy-dev-def.h` . . . 11, 25

Devices . . . 14
Directories . . . 5
Dokumentation . . . *siehe* Incode-Doku
Download . . . *siehe* ECLoad
Doxyfile . . . 8
Doxygen . . . 8
Dualport-RAM
- devDataP() . . . 27
- dprP . . . 27
- Gerätekonstanten im ∼ . . . 10, 26, 27

— **E** —

Eclipse . . . 5
- Checkout . . . 6
- Commit EQMs . . . 36
- Commit USRs . . . 35
- Tag EQMs . . . 33, 36
- Tag USRs . . . 35

ECLoad
- ecload . . . 36
- SE-Umstellung V08 auf V09 . . . 37

ecload . . . 36
EQMs . . . 32
- doppeltes Tagging . . . 33

Error-Handling . . . 31
Exceptions . . . 31

— **G** —

gendocu . . . 34
geneqms . . . 36
genmsg . . . 34
Gerätekonstanten
- constActual . . . 32
- DevConstDesc . . . 10, 15, 33
- devConstP() . . . 27
- DevConstType . . . 10, 33
- getDevConstants() . . . 16, 27
- im Dualport-RAM . . . 10, 26, 27
- im Geräteobjekt . . . 15, 16, 27
- setDevConstants() . . . 15–17
- valid-Flag . . . 10, 17, 26, 32

getDevConstants() . . . 16, 27
getmsg . . . 34
getStandardParameter() . . . 27
gm-dev-def.h . . . 10
gm-dev-def.hh . . . 13
gm-device.cc . . . 15
gm-device.hh . . . 14
gm-structtypes.hh . . . 19

gm-usrs.cc 24
gm-usrs.hh 21

— H —

Header-Dateien 6

— I —

$incasl 5
Includes
• add*-usrs.cc 30
• default-dev-def.h 25
• default-usrs*.cc 26
• therapy*-usrs.cc 26
• therapy-dev-def.h 25
$incmsg 5
Incode-Doku 7
• Doxyfile 8
• Doxygen 8
• in Implementationsdateien 24
$incvme 5

— K —

Klassen
• AccData 20, 28
• AccDevice 14
• GmDevice 14–16
• ReadFieldI 26
• Schreibweise 22
• Usr 19, 22
• USR-∼ 22
• VmeDevice 14–16
– checkSupStatus() 15
Konstanten . . . . . *siehe* Gerätekonstanten

— L —

$libasl 5
$libppc 5
$libvme 5
Logging 18
logmsg() 18

— M —

make 35, 36

— O —

omniORB
• TIMEOUT-Macro 25

— P —

Property 19, 22, 24
• alternativer Name 23
• Default-∼ 11
• Therapie-∼ 11

— R —

readCommand() 28
releqms 36
relusrs 35
Remote Debugging 37
Return Status 31

— S —

SE-Umstellung V08 auf V09 37
service devman 35
setActTimeStamp() 29
setDevConstants() 15–17
setEFICD() 29
setStamps() 29
SISDataType
• und AccData 31
Structure
• gm-structtypes.hh 19
Subversion 5
sys68v08 34
sysact 34
sysaslv01 34
syslocal 34
sysvmev09 34

— T —

Therapie USRs 26
Tools 34
• cdcpu 5, 34
• cdeqp 5, 34
• cdsys 5
• copyusrs 35
• dddppc 37
• ecload 36
• EQMs erstellen 36
• gendocu 34
• geneqms 36

- genmsg . . . 34
- getmsg . . . 34
- make . . . 35, 36
- releqms . . . 36
- relusrs . . . 35
- service devman . . . 35
- sys68v08 . . . 34
- sysact . . . 34
- sysaslv01 . . . 34
- syslocal . . . 34
- sysvmev09 . . . 34
- USRs erstellen . . . 35
- vmeboot . . . 35
- vmeconfig . . . 35, 36

$tsteqp . . . 5

— **U** —

Umgebungsvariablen
- $incasl . . . 5
- $incmsg . . . 5
- $incvme . . . 5
- $libasl . . . 5
- $libppc . . . 5
- $libvme . . . 5
- $tsteqp . . . 5

USRs . . . 19
- `TIMEOUT`-Macro . . . 25
- addUsrs() . . . 30
- Alarme . . . 29
- alternativer Property-Name . . . 23
- Default-∼ . . . 26
- Error-Handling . . . 31
- Exceptions . . . 31
- Return Status . . . 31
- Therapie-∼ . . . 26

— **V** —

vmeboot . . . 35
vmeconfig . . . 35, 36

— **W** —

writeCommand() . . . 28

— **Z** —

zeroEFICD() . . . 29