# Dokumentation
# IgH-Echtzeit-Bibliothek

erstellt im Auftrag

der Firma

IgH

durch die

Ingenieurgemeinschaft IgH

Essen

im Herbst 2004

Dieser Text[1] ist durch die *IgH* GmbH auf Veranlassung des Auftraggebers erstellt worden. Der Text ist zum internen Gebrauch bestimmt, die Ausführungen sind urheberrechtlich geschützt.

Ingenieurgemeinschaft IgH
Heinz-Bäcker-Str. 34
D-45356 Essen
Tel.: +49-201-36014-0
Fax.: +49-201-36014-14
E-mail: igh@igh-essen.com

[1] `m-igh_rt_api`

# Inhaltsverzeichnis


0.1 Aufbau der Steuerungssoftware . . . . . . . . . . . . . . . . . . . . . . . 2

0.1.1 Allgemeines . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

0.1.2 Netzwerkfähigkeit . . . . . . . . . . . . . . . . . . . . . . . . . 2

0.1.3 Echtzeitfähigkeit . . . . . . . . . . . . . . . . . . . . . . . . . 3

0.1.4 Kernelmodul . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

0.1.5 Floating Point Operationen im Kernel . . . . . . . . . . . . . 6

0.1.6 Parameter . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

0.1.7 Kanäle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

0.1.8 Zustandsmaschine . . . . . . . . . . . . . . . . . . . . . . . . 14

0.1.9 Fehlerbehandlung durch die Software . . . . . . . . . . . . . . 20

0.1.10 Regler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

0.1.11 Funktionsgenerator . . . . . . . . . . . . . . . . . . . . . . . . 27

0.1.12 Hardwaretreiber . . . . . . . . . . . . . . . . . . . . . . . . . 28

0.1.13 Beckhoff CX1100-Treiber . . . . . . . . . . . . . . . . . . . . . 28

0.1.14 Sonstige Funktionalitäten . . . . . . . . . . . . . . . . . . . . 32

0.1.15 Kommunikation zwischen Client und Server . . . . . . . . . . 32

0.1.15.1 Schnittstelle zum Kernelmodul . . . . . . . . . . . . 32

0.1.16 Organisation der Dateien . . . . . . . . . . . . . . . . . . . . . 38

0.1.17 Netzwerk . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

## 0.1 Aufbau der Steuerungssoftware

### 0.1.1 Allgemeines

In der Steuertechnik existiert generell ein Trend hin zum PC. Prozessoren, Bussysteme und entsprechende I/O-Karten sind flexibel und leistungsfähig. Die PC-Architektur erlaubt außerdem den Einsatz gängiger, verbreiteter und durch massenhaften Einsatz gut getesteter Software.

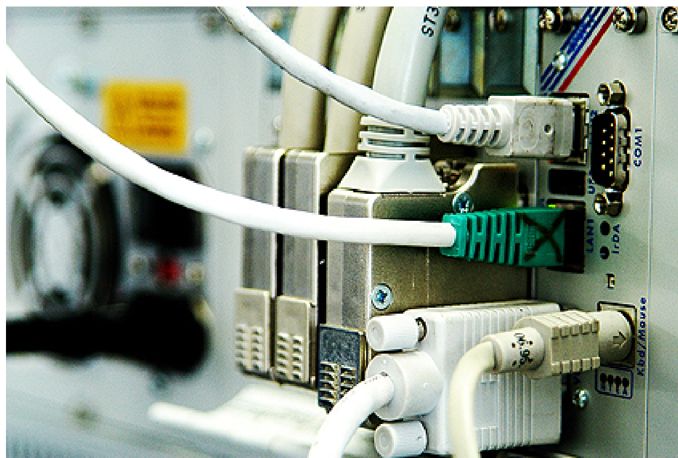


Abbildung 1: MSR mit Industrie-PC

Obwohl Linux eigentlich für Zwecke der Datenverarbeitung konzipiert wurde, besitzt es auch für Steuerungs- und Regelungsaufgaben viele Vorteile: Der verfügbare Quellcode ermöglicht Anpassungen an spezielle Hardware, Skalierbarkeit, gezielte Optimierungen, um etwa Latenzzeiten zu verringern sowie tiefreichende Diagnose. Kernel-Module lassen sich unkompliziert einbinden, Linux erlaubt zudem umfangreiche Einblicke in die Kernelstrukturen zur Laufzeit. Gnu-Tools und aller anderen unix-typischen Entwicklungswerkzeuge stehen für Analyse, Test und Optimierung bereit.

### 0.1.2 Netzwerkfähigkeit

Die Konzeption der Steuerungssoftware ermöglicht einen durchgängig netzwerkbasierten Zugriff auf die Parameter und Daten des Steuerungsprozesses. Dies ermöglicht Steuerung, Datenerfassung, Visualisierung, Ablage, Datenbank usw. auf verschiedenen Rechnern mit unterschiedlichen Betriebssystemen zu verteilen, soweit dies notwendig und sinnvoll ist. Abbildung 2 zeigt eine mögliche Konfiguration.

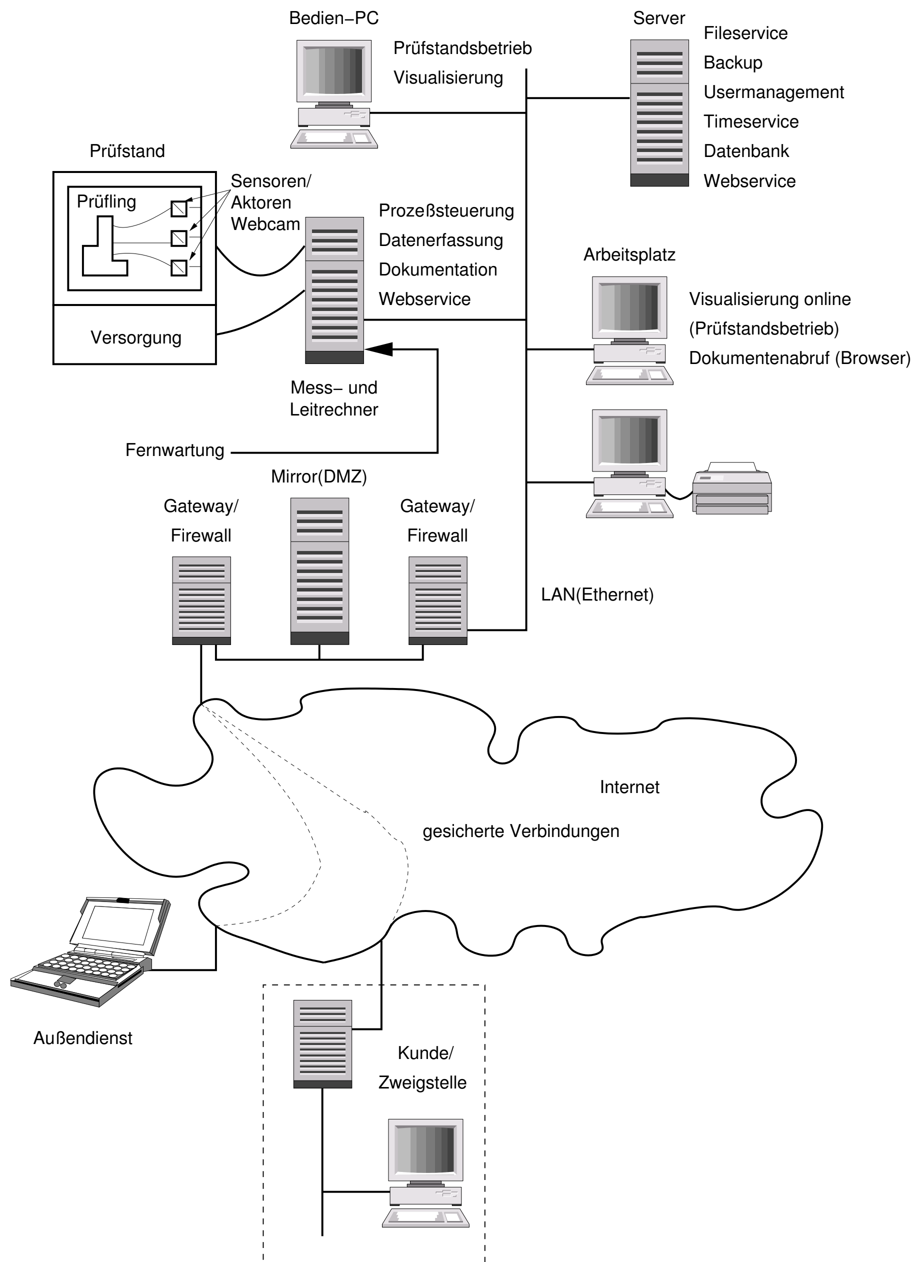


Abbildung 2: Netzwerkfähigkeit

### 0.1.3 Echtzeitfähigkeit

Ein System arbeitet echtzeitfähig, wenn es garantiert innerhalb einer definierten maximalen Antwortzeit auf ein Ereignis reagiert. Die Dauer dieser Antwortzeit ist naturgemäß vom Prozess abhängig, der kontrolliert werden soll. Echtzeitfähigkeit ist nicht notwendig gleichzusetzen mit Schnelligkeit! Zwischen einem Ereignis und der Antwort des Echtzeitsystems verstreicht grundsätzlich eine gewisse Zeit, die sogenannte Latenzzeit. Die Streuung der Latenzzeit bei wiederkehrenden Ereignissen wird Jitter genannt.

Echtzeitverhalten ist eine notwendige Eigenschaft eines Betriebssystems für Steuerungsaufgaben. Für Linux existieren eine Reihe von Erweiterungen, die eine tatsächliche Echtzeitfähigkeit verbessern oder gewährleisten, zum Beispiel RT-Linux [2] oder RTAI [3].

Aufgrund der patentrechtlich schwierigen Lage bezüglich RT-Linux, wird bei den IgH-Systemen entweder die RTAI-Erweiterung genutzt oder keine Echtzeiterweiterung verwendet, wobei der Jitter des Standardkernels von max. 1 ms akzeptiert wird.

Digitale Steuerungen und Regelungen verfügen über einen Taktgeber, der die entsprechenden Berechnungsroutinen in äquidistanten Zeitschritten aufruft. Dabei liegen die Abtastraten für die hier betrachteten Systeme zwischen 100 Hz und 10 kHz. Die weitaus überwiegende Zahl der industriellen digitalen Regelungen arbeiten mit Abtastraten in diesem Bereich.

### 0.1.4 Kernelmodul

Regelung und Ablaufsteuerung sind in einem dynamisch ladbaren Kernelmodul [4] untergebracht. Abbildung 3 zeigt den prinzipiellen Aufbau des Moduls.

Das Kernelmodul verfügt über folgende wichtige Komponenten:

- Initialisierung

  Die Initialisierung des Moduls erfolgt direkt nach dem Laden.

  - Initialisieren und Parametrieren der I/O-Klemmen
  - Registrieren der Interruptroutinen beim Betriebssystem
  - Registrieren der Kommunikationsschnittstelle (char-device) beim Betriebssystem
  - Registrierung von Parametern und Kanälen an der Kommunikationsschnittstelle
  - Start des Steuer- und Regelteils

- Steuer- und Regelung

  Die gesamte Steuerung und Regelung wird in der Interruptroutine ausgeführt. Hierzu gehören in folgender Reihenfolge:

  - Auslesen der I/O-Klemmen (digital und analog)

[2] FSM-Labs RT-Linux Projekt: www.fsmlabs.com
[3] www.aero.polimi.it/ rtai/index.html
[4] Gültigkeit dieser Doku für `rt_lib-2.8`

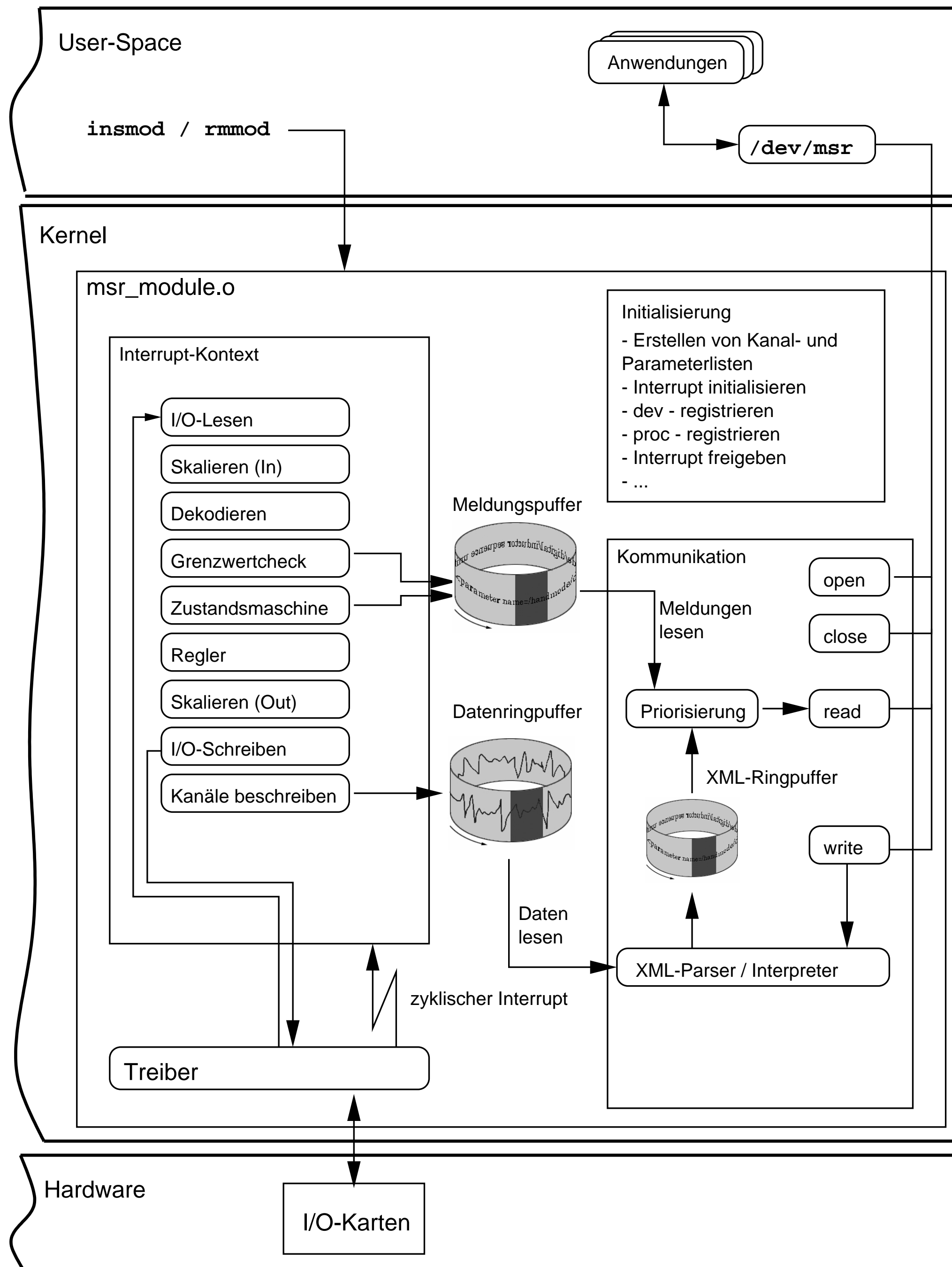


Abbildung 3: Module

- Skalieren der Rohdaten
- Dekodieren der digitalen Werte
- Grenzwertüberwachung der Daten
- Berechnung von Zwischengrößen
- Ausführung der Zustandsmaschinen

- Berechnung der Regler
- Berechnung der Ausgabewerte (analog und digital)
- Beschreiben der Ausgangskanäle
- Ablegen der Meßdaten in einem Kanalringpuffer

Variablen auf dem Modul können wahlweise als Parameter oder Kanäle registriert werden. Siehe hierzu Kapitel 0.1.6 und 0.1.7 Dies geschieht bereits im Quellcode, da die Bedeutung einer Variablen bereits bei Programmerstellung zugewiesen wird. Parameter sind überlicherweise Einstellwerte von Reglern, Kanäle sind in der Regel von den I/O-Klemmen erfasste Meßwerte. Während Parameter sowohl gelesen als auch geschrieben werden können, können Kanäle nur gelesen werden. Da die Daten in den Kanälen im Takt der Abtastrate anfallen, verfügt jeder Kanal über einen eigenen Ringpuffer, in dem alle aufgenommenen Werte abgelegt werden. Per Kommando vom Clientrecher kann das Senden dieser Werte initiiert werden. Der Ringpuffer sorgt dann dafür, daß keine Werte verloren gehen.

Alle Kanäle und Parameter werden während der Initialisierung des Moduls von den Registierungsfunktionen in entsprechend verkettete Listen eingetragen, die dann der Kommunikationsschnittstelle zur Verfügung stehen.

### 0.1.5 Floating Point Operationen im Kernel

Im Standard-Linuxkernel und auch in den Modulen ist keine einzige Floating-Point Berechnung zu finden. Dies ist in der Regel auch nicht notwendig, da keine Betriebssystemfunktionalität zwingend auf Fließkommaberechnungen angewiesen ist. In einigen wenigen Fällen, wie zum Beispiel bei der Berechnung von Zeiten wird Festkomma-Berechnung auf Integerarithmetik abgebildet. Aus diesem Grund ist die Zeit in einer Struktur kodiert:

```
struct timeval {
     long tv_sec;        /* seconds */
     long tv_usec;  /* microseconds */
  };
```

Für Steuerungsaufgaben, bei denen komplexe Regelungen und Koordinatentransformationen durchgeführt werden, sind jedoch Fließkommatberechnungen unverzichtbar. Allein die räumlichen Berechnungen erfordern massenhaft trigonometrische Kalkulationen. Da die gesamte Steuerungssoftware als Kernelmodul realisiert ist, müssen nun auch im Kernelspace Fließkommaberechnungen durchgeführt werden. Dabei sind zwei Punkte zu beachten. Die mathematischen Routinen müssen verfügbar sein, um auch im Kernel etwa trigonometrische Funktionen nützen zu können. Die Math-Bibliothek ist also statisch zum Kernelmodul zu linken. Dazu dient die Linker-Option

```
LDFLAGS =  -L/usr/lib -lm
```

Optional bietet RTAI eine eigene Mathlibrary `rtai_libm`, in der alle mathematischen Funktionsaufrufe für die Verwendung im Kernel kompiliert sind. Für die Verwendung dieser Funktionen muß `rtai_libm.o` als Modul geladen werden.

Der zweite Punkt betrifft das Retten der Floating-Point Register. Aus Geschwindigkeitsgründen werden bei Interruptaufrufen des Kernels die Floating-Pointregister nicht gesichert, weil sie normalerweise unverändert bleiben.

Alle echtzeitrelevanten Programmteile, also Regler, Koordinatentransformation und Zustandsmaschine werden in der Interruptroutine ausgeführt. Da diese zyklisch die Userspace-Applikationen, also Anwenderprogramme und Daemonen unterbricht, in denen Fließkommaberechnungen durchgeführt werden, müssen in der Interruptroutine alle Floating-Point-Register gesichert und am Ende der Routine wiederhergestellt werden. Nachfolgend der Rumpf einer Routine, die Fließkommarechnung nutzt.

```
/*Quellcodeauszug nach Vorlage von P. Mantegazza (mantegazza@aero.polimi.it),
um im Kernel Fließpunktoperationen ausführen zu können.
(Retten und Wiederherstellen der Floating-Point-Register) */

#define save_cr0_and_clts(x) __asm__ __volatile__ ("movl %%cr0,%0; clts" :"=r" (x))
#define restore_cr0(x)       __asm__ __volatile__ ("movl %0,%%cr0": :"r" (x));
#define save_fpenv(x)        __asm__ __volatile__ ("fnsave %0" : "=m" (x))
#define restore_fpenv(x)     __asm__ __volatile__ ("frstor %0" : "=m" (x));


/* Interruptroutine */
void msr_run_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
  /* Variablen für temporäre Sicherung der notwendigen Register */
  unsigned long cr0;
  unsigned long linux_fpe[27];

  /* Register sichern */
  save_cr0_and_clts(cr0);
  save_fpenv(linux_fpe);

  /* ab hier Floating Point Berechnungen */

  ....

  /* bis hier Floating Point Berechnungen */

  /* Register wiederherstellen */
  restore_fpenv(linux_fpe);
  restore_cr0(cr0);
}
```

Floating-Point-Berechnungen im Kernel, die nicht in einer Interruptroutine, sondern im Kontext eines Userprozesses ausgeführt werden, müssen nicht gesondert behandelt werden, da die Register vom Betriebssystem schon beim Taskwechsel für den Userprozess gesichert wurden. Dies gilt beispielsweise für Berechnungen in den Read/Write-Funktionen eines Kernel-Moduls.

### 0.1.6 Parameter

Parameter sind Skalare, Listen oder Aufzählungstypen die seitens eines Userprozesses ausgelesen oder verändert werden können. Die für die Registrierung notwendigen Routinen sind in `msr_reg.c, msr_reg.h` zu finden. Prinzipiell kann jede Variable, die im Echtzeitmodul Verwendung findet als Parameter veröffentlicht werden. Hierzu wird eine Registrierungsfunktion verwendet, deren Name abhängig vom Typ des Parameters ist. In diesem Funktionsaufruf werden zugehörige Eigenschaften einmal für die Ladezeit des Moduls festgelegt. Besonders sorgfältig sollte der "veröffentlichte Name„ gewählt werden, damit der einzelne Parameter später auch in der Liste aller Parameter wiedergefunden werden kann. Für die Hierachisierung wird `/` verwendet. Dies ermöglicht die sinnvolle Gruppierung und Zuordnung von Parametern [5] und ermöglicht eine übersichtliche Darstellung im Nutzerprogramm, wie dies z.B. in Abbildung 4 zu sehen ist.

Für skalare Variablen vom Typ `double, int, unsigned int, unsigned char` gilt nachfolgende Syntax:

```
double out_ampl;

....


msr_reg_dbl_param("/settings/frq/amplitude",    /*veröffentlichter Name*/
   "V",                            /*Einheit*/
   &out_ampl,                      /*Zeiger auf die Variable*/
   MSR_W | MSR_WA | MSR_S,         /*Schreibzugriff,
                                     Schreibzugriff für Administrator, Sicherungsflag*/
   25.0,                           /*Initialisierungswert*/
   0.0,                            /*unterer Grenzwert*/
   100.0,                          /*oberer Grenzwert*/
   NULL,                           /*Funktionszeiger auf eine Funktion,
                                     nach dem Beschreiben der Variablen aufgerufen wird*/
   NULL);                         /*Funktionszeiger auf eine Funktion,
                                                     die vor dem Lesen aufgerufen wird */
```

Für `msr_reg_int_param(),msr_reg_uint_param(),msr_reg_uchar_param()` gilt gleiche Aufrufkonvention. Wichtig ist, dass der wirkliche Typ der Variablen, auf den

[5] Dies gilt auch für Kanäle.

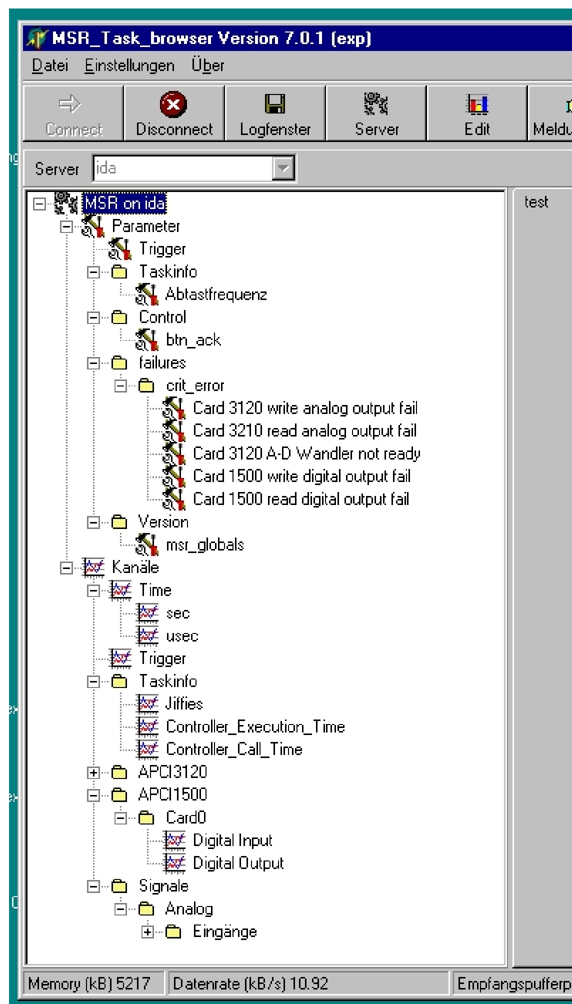


Abbildung 4: Anzeige von Kanälen und Parametern

durch den Zeiger verwiesen wird mit dem angenommenen Typ, der durch den Funktionsnamen festgelegt wird, übereinstimmt. Die Angabe von Grenzwerten ermöglicht es, die Nutzereingaben auf sinnvolle Werte zu beschränken. Die Zugriffbeschränkung erfolgt durch verschiedene Bits eines Flags. Soll nur dem Administrator Schreibzugriff gewährt werden, so ist auch nur das Flag `MSR_WA` zu setzten. Optional besteht die Möglichkeit nach dem Schreiben und vor dem Lesen eines Parameters Funktionen aufzurufen. Hierzu müssen entsprechende Funktionszeiger übergeben werden. Die Rückgabe der Registrierungsfunktion ist positiv, wenn der Funktionsaufruf erfolgreich war und ist die Adresse des zugehörigen Listenelementes vom Typ `int` in der Struktur `struct msr_param_list` [6].

Strings lassen sich mit:

```
char *smstelefonelist = "";
```

[6]siehe `msr_reg.h`

```
msr_reg_str_param("/SMS-Messaging/Phone Book",            /*veröffentlichter Name*/
                   "",                                     /*Einheit*/
                   &smstelefonelist,                       /*Zeiger auf die Variable*/
                   MSR_R | MSR_W | MSR_S,                  /*Schreibzugriff */
                   "0172-12345678,0172-87654321",          /*Initialisierungswert*/
                   NULL,                                   /*Funktionszeiger*/
                   NULL);                                  /*Funktionszeiger*/
```

registrieren.

Desweiteren kann durch das Beschreiben eines Parameters eine Funktion aufgerufen werden. Die notwendige Registrierungfunktion ist:

```
int msr_reg_funktion_call(char *bez,void (*write)());
```

Für den Zugriff auf eindimensionale Arrays stehen die Funktionen:

```
int msr_reg_int_list(char *bez,char *einh,void *adr,
                     unsigned int flags,
                     int anz,
                     int *valid_anz,
                     void (*write)(char *value,void *priv,...),
                     void (*read)());

int msr_reg_dbl_list(char *bez,char *einh,void *adr,
                     unsigned int flags,
                     int anz,
                     int *valid_anz,
                     void (*write)(char *value,void *priv,...),
                     void (*read)());
```

zur Verfügung.

Zum Beispiel:

```
#define AIP_SPEED_TABLE_SIZE 50
int TableRotationMotorSize = 5;

double TableRotationMotorSpeed[AIP_SPEED_TABLE_SIZE]= { 630,   63,   0,   0, -630};

....

msr_reg_dbl_list("/Rotation/Motor/Speedtable/Speed", /*veröffentlichter Name*/
```

```
   "1/min",                          /*Einheit*/
   &TableRotationMotorSpeed,         /*Zeiger auf die Variable */
   MSR_W | MSR_S,                    /*Schreibzugriff */
   AIP_SPEED_TABLE_SIZE,             /*Maximale Größe der Tabelle */
   &TableRotationMotorSize,          /*Zeiger auf eine Variable,
                                       die die aktuelle Größe der Tabelle
                                       angibt */
   NULL,                             /*Funktionszeiger*/
   NULL);                            /*Funktionszeiger*/
```

Tabellen können nicht bei der Registrierung initialisiert werden, sondern dies muss bei der Deklaration geschehen. Eine Tabelle muss nutzerseitig nicht vollständig belegt werden. In der Variablen `valid_anz` wird die aktuelle gültige Anzahl Einträge, die kleiner gleich `anz` sein muss hinterlegt. Um mehrere Tabellen in ihrer Größe sychronisieren zu können, kann diese Variable bei mehreren Tabellen verwendet werden, die dann immer die Größe der zuletzt beschriebenen Tabelle haben. Vorsicht, alte Werte in der Tabelle werden nicht gelöscht.

Beispiel, für synchrone Tabellen:

```
...

msr_reg_dbl_list("/Cylinder/Loadtable/N","N",&TableCylinderN,
                 MSR_R | MSR_W | MSR_S,AIP_LOAD_TABLE_SIZE,&TableLoadSize,NULL,NULL);
msr_reg_dbl_list("/Cylinder/Loadtable/T","N",&TableCylinderT,
                 MSR_R | MSR_W | MSR_S,AIP_LOAD_TABLE_SIZE,&TableLoadSize,NULL,NULL);
msr_reg_dbl_list("/Cylinder/Loadtable/M","Nm",&TableCylinderM,
                 MSR_R | MSR_W |MSR_S,AIP_LOAD_TABLE_SIZE,&TableLoadSize,NULL,NULL);

...
```

Aufzählungstypen werden mit `msr_reg_enum_param()` registriert:

```
enum aip_control_states {
    T_AIP_init,
    T_AIP_ready,
    T_AIP_startup,
    T_AIP_operationmode
};

enum aip_control_states aip_control_status = T_AIP_init;

/* der zugehörige Beschreibungsstring */

char *aip_control_status_str[] = {"Init","Ready","Startup","Operation Mode"};
```

```
#define AIP_CONTROL_STATUS_ANZ (sizeof (aip_control_status_str)/sizeof(int))


...


msr_reg_enum_param("/Control/State",              /*veröffentlichter Name*/
     "",                                /*Einheit*/
     &aip_control_status,               /*Zeiger auf die Variable */
     MSR_R,                             /*Schreibzugriff */
     T_AIP_init,                        /*Initialisierungswert */
     AIP_CONTROL_STATUS_ANZ,            /*Anzahl verschiedener Zustände */
     aip_control_status_str,            /*String, in dem die Zustände bezeichnet werden*/
     NULL,NULL);                        /*Funktionszeiger*/
```

Die Enumeratevariable soll keine Indexzuweisungen haben, da davon ausgegangen wird, dass die Einträge von 0 an aufsteigend im Feld stehen. `*values` ist ein Zeiger auf ein Feld von Strings, in dem die Aufzählungsinhalte noch einmal als String hinterlegt sind. Hier ist von Hand auf Konsistenz zu achten.

### 0.1.7 Kanäle

Variable können auch als Kanal registriert und abgefragt werden `msr_reg.h`, `msr_reg.c`. Kanäle werden immer dann verwendet, wenn Daten mit konstanter Abtastrate vom Echtzeitprozess zur Verfügung gestellt werden sollen. Da die Daten in den Kanälen im Takt der Abtastrate anfallen, verfügt jeder Kanal über einen eigenen Ringpuffer, in dem alle aufgenommenen Werte abgelegt werden.

Die Größe des Ringpuffers, der in Sekunden konfiguriert werden kann, kann einmalig bei der Initialisierung beeinflußt werden durch den Aufruf von `msr_init_kanal_params()`. Üblicherweise ist die höchste zur Verfügung stehende Abtastrate – mit der Kanäle im Puffer abgelegt werden – durch den Takt des Echtzeitinterrupts vorgegeben. In einigen Fällen, z.B. bei der Erfassung von analogen Kanälen per DMA kann jedoch die individuelle Abtastrate einiger Kanäle über der Rate des Echtzeittaktes liegen.

```
 msr_init_kanal_params(int sampling_red,      /*Verhältnis von
                                          schnellstem Kanal zum Echtzeittakt */
                   int sampling_frq,      /*Höchste Abtastfrequenz */
                   int kanal_buffering);  /* Pufferlänge der Kanäle in Sekunden !*/


Beispiel:

msr_init_kanal_params(10,          /* DMA ist 10mal schneller als Echtzeittakt */
```

```
                  MSR_ABTASTFREQUENZ*10,
                  10);                 /* 10 Sekunden Puffer für alle Kanäle */
```

Das Verhältnis von schnellstem Kanal zum Echtzeittakt wird bei `sampling_red` angegeben. Wird `msr_init_kanal_params()` nicht aufgerufen, wird von einer Abtastrate von 1 kHz ausgegangen und eine Pufferlänge von 10000 Werten entsprechend 10 sec Puffer vorgesehen. Die maximal mögliche Pufferlänge für einen Kanal ist auf 2 MB begrenzt [7]. Bei einem Kanal vom Typ `double` sind das immerhin ca. 250.000 Werte. Desweiteren muß der überhaupt zur Verfügung stehende Hauptspeicher berücksichtigt werden, da alle Kanäle immer im Hauptspeicher verfügbar sein müssen.

Kanäle werden mit `msr_reg_kanal() oder msr_reg_kanal2()` registriert. Bezüglich der Wahl der veröffentlichten Kanalnamen siehe Kapitel: 0.1.6.

```
double druck;

...
msr_reg_kanal("/measure/hydraulik/pressure/pp",  /*veröffentlichter Name*/
              "bar",                              /*Einheit*/
              &druck,                            /*Zeiger auf die Variable*/
              TDBL);                             /*Typ der Variablen*/
```

Für Kanäle stehen bisher die Typen `TDBL,TINT,TUINT` zur Verfügung. Es erfolgt keine Überprüfung auf den Typ. Hier muß Übereinstimmung von Typ der Variablen und Angabe in `msr_reg_kanal()` hergestellt sein. Mit `msr_reg_kanal()` werden alle Kanäle initialisiert, die mit der Abtastrate des Echtzeitprozesses erfasst werden sollen.

Für Kanäle, die langsammer oder schneller erfasst werden sollen muß `msr_reg_kanal2()` verwendet werden. Hier kann zusätzlich noch die Untersetzung und ein Alias (der bisher noch nicht verwendet wird) mit angegeben werden.

```
int msr_reg_kanal2(char *bez,
                 void *alias,
                  char *einh,
                   void *adr,
       enum enum_var_typ typ,
                      int red);
```

[7] __get_free_pages(GFP_KERNEL,...) gibt nicht mehr her

Wird ein Kanal schneller abgetastet, als der Echzeittakt muss als Adresse `*adr` ein Array angegeben werden, in dem die während eines Taktes anfallenden Werte des Kanals aufgelistet sind.

Beispiel für einen Kanal, der 10-mal schneller als der Echtzeittakt erfaßt wird.

```
double RotationActuatorTorqueA [APCI3120_DMA_NUM_BLOCKS];

...

msr_reg_kanal2("/Rotation/Actuator/TorqueA",
               "041.0HBM07-0PXA03.MDG.ANV0.",
               "Nm",
               &RotationActuatorTorqueA [0],
               TDBL,
               1);
```

### 0.1.8 Zustandsmaschine

Die Zustandsmaschine beschreibt den logischen Programmablauf. Sie verknüpft die Eingänge eines Prozesses mit den Ausgängen. Während in der Literatur eine große Anzahl Beschreibungen von Zustandsmaschinen zu finden sind, wird hier eine pragmatische und sichere Vorgehensweise in Form von einzelnen Funktionen für die jeweiligen Zustände gewählt. Der Aspekt der Sicherheit ist vorrangig und wir dadurch gekennzeichnet, dass die Ausgänge der Zustandsmaschine jeweils in einem von der zeitlichen Vorgeschichte unabhängigen Zustand sind. Dies wir dadurch erreicht, dass die Zustandsmaschine jeden Taktzyklus durchlaufen wird, dabei zunächst alle Ausgänge in einen sicheren Zustand versetzt (z.B. alle digitalen Ausgänge abgeschaltet und alle analogen zu Null setzt) und in dem jeweiligen Zustand wieder auf den bestimmungsgemäßen Zustand setzt. Da die Ausgänge erst nach dem Durchlauf durch die Zustandmaschine an den jeweiligen Hardwaretreiber übergeben werden, führt das Rücksetzen und Wiedersetzen nicht zu Glitches. Die Abfrage auf Fehler wird ebenfalls in jedem Taktzyklus durchgeführt.

Zur Identifikation der Zustände wird eine Variable eingeführt, die von Typ `enum` ist und für jeden Zustand einen Eintrag erhält.

```
enum tcp_control_states {
    T_TCP_startup,
    T_TCP_bereit,
    T_TCP_betrieb,
    T_TCP_achse_activ,

    ...
```

```
    T_TCP_error,
    T_TCP_handmode
};
```

Da es innerhalb eines Steuerprozesses mehrere Zustandsmaschinen geben kann, die zum einen gleichen Typs als auch voneinander abhängig sind, werden alle relevanten Informationen in einer Struktur gespeichert.

```
struct msr_statemaschine {
    int prevstate;
    int state;
    int flag;
    void *private_data;
    void (*prestate_function)(struct msr_statemaschine *sm);
    void (*state_function)(struct msr_statemaschine *sm);
    void (*poststate_function)(struct msr_statemaschine *sm);
    struct msrState {
        int state;
        void (*sf)(struct msr_statemaschine *sm);
    } msrSFL[];
};
```

Die Elemente haben folgende Bedeutung:

- **prevstate:** Zustand vor dem letzten Zustandswechsel
- **state:** aktueller Zustand
- **flag:** Falls ein Zustandswechsel vollzogen wurde, wird das Flag high gesetzt
- ***private_data:** Bei synchron ablaufenden identischen Zustandsmaschinen kann hier ein Zeiger auf eine zustandsabhängige Datenstruktur gespeichert werden.
- ***prestate_funtion:** Funktionszeiger auf die Funktion, die vor der eigentlichen Zustandsfunktion aufgerufen wird.
- ***state_function:** Funktionszeiger, der bei Zustandswechsel auf die jeweilige Zustandsfunktion gesetzt wird.
- ***poststate_function:** Funktionszeiger auf die Funktion, die im Anschluss der jeweiligen Zustandsfunktion abgearbeitet wird.
- **msrSFL[]:** Strukturarray mit den Zuordnungen Zustand → Zustandsfunktion

Die Initialisierung der Struktur kann folgendermaßen aussehen:

```
static struct msr_statemaschine tcpGlobalStatemaschine =
{
    msrSFL : {{T_TCP_startup,    &tcp_do_startup},
              {T_TCP_bereit,     &tcp_do_bereit},
              {T_TCP_betrieb,    &tcp_do_betrieb},

               ...

              {T_TCP_shutdown,   &tcp_do_shutdown},
              {T_TCP_error,      &tcp_do_error},
              {T_TCP_handmode,    &tcp_do_handmode}},
    prestate_function : tcpGlobalprestate,
    poststate_function : tcpGlobalppoststate,
};
```

Die Zustandsmaschine benötigt zum Starten einen definierten Startzustand. Dies geschieht mit der Funktion `msr_statemaschine_init()`. Dieser Funktion wird der Zeiger auf die Struktur der Zustandsmaschine sowie deren Startzustand übergeben. Gegebenenfalls benötigt die Zustandsmaschine eine `private_data`-Struktur, um mehrere Instanzen einer identischen Zustandsmaschine zu ermöglichen. Dies kann eine beliebige Datenstruktur sein.

```
/*-----------------------------------------------------------*/
void tcp_statemaschine_init()
/*-----------------------------------------------------------*/
{
    msr_statemaschine_init(&tcpGlobalStatemaschine,T_TCP_startup);

    tcpReglerStatemaschineht.private_data = &regler_achse_ht;
    msr_statemaschine_init(&tcpReglerStatemaschineht,T_TCP_CONT_off);

    tcpReglerStatemaschinent.private_data = &regler_achse_nt;
    msr_statemaschine_init(&tcpReglerStatemaschinent,T_TCP_CONT_off);

}
```

Der Zustandswechsel kann mit folgender Funktion initialisiert werden:

```
/*-----------------------------------------------------------*/
void switchToState(struct msr_statemaschine *sm,int newstate)
/*-----------------------------------------------------------*/
```

```
{
    sm->state_function = sm->msrSFL[newstate].sf;
    sm->state = newstate;
}
```

Eine typische zustandsbeschreibende Funktion einer übergeordneten Zustandsmaschine sieht dann z.B. wie folgt aus:

```
static void tcp_do_betrieb(struct msr_statemaschine *sm)

{
    //Setzen von Variablen, die untergeordnete Zustandsmaschinen steuert
    e_pent_allow=TCP_ON;
    e_min_nt_allow=TCP_ON;
    e_min_ht_allow=TCP_ON;
    posregler_ht_output=0.0;
    posregler_nt_output=0.0;


    tcp_do_check_enable();
    //Abfrage von aktuellen Zuständen der untergeordneten Zustandsmaschinen
    if((tcpPowerStatemaschinent.state==T_TCP_POW_activ)&&(is_working_achse_nt))
    {
        switchToState(sm,T_TCP_achse_activ);
    }

    if(!(is_working_min_ht||is_working_min_nt||is_working_pent))
    {
        switchToState(sm,T_TCP_shutdown);
    }

}
```

Für kontrollierte Verzögerungen innerhalb eines Zustandes kann folgendes Macro `AFTERWAIT` verwendet werden. Durch das Konstrukt `do ... while(0)` wird innerhalb des Präprozossormakros ein logischer Block definiert. Dies schützt für falscher Interpretation durch den Präprozessor.

```
#define AFTERWAIT(t,expression) {static int counter = 0;                \
                                 if (sm->Flag == STATE_STEP_IN) counter = 0; \
                                 do { if(counter++>t*MSR_ABTASTFREQUENZ) \
                                      {expression;};                     \
                                    }while(0); }
```

Anbei ein Beispiel für die Verwendung des Macros

```
/*-------------------------------------------------*/
static void aip_shutdown(struct msr_statemaschine *sm)
/*-------------------------------------------------*/
{
    msr_operating = 1;
    SystemGreenFlashLight = 1;
    int_watchdog_switch = 1;

    PanelSetpointPressure=0.0;
    SystemFlagPressurecontrolactive=1; /* Druck geregelt herunterfahren */

    Cylinder1SetPointForce=0;
    Cylinder2SetPointForce=0;
    Cylinder3SetPointForce=0;
    SystemFlagForcecontrolactive=1; //und Kraft geregelt runterfahren

    if(PanelRackPressure < 30.0)
        switchToState(sm,T_AIP_ready);

    AFTERWAIT(10,switchToState(sm,T_AIP_ready));  /* maximal 10 Sek warten */

}
```

Innerhalb der Interruptserviceroutine wird die zentrale Kontrollfunktion aufgerufen (hier : `tcp_do_control`). In dieser Routine werden alle Zustandsmaschinen aufgerufen mit der Funktion `msr_statemaschine_run` mit dem Zeiger auf die zustandsmaschinenabhängige Struktur als Argument. In dieser Funktion werden die einzelnen in der Struktur definierten Funktionszeiger abgearbeitet (prestate_-, state_- und poststate_funtion).

```
/*-------------------------------------------------*/
void tcp_do_control()
/*-------------------------------------------------*/
{

    msr_statemaschine_run(&tcpGlobalStatemaschine);
    msr_statemaschine_run(&tcpReglerStatemaschineht);
    msr_statemaschine_run(&tcpReglerStatemaschinent);
    msr_statemaschine_run(&tcpPowerStatemaschineht);
    msr_statemaschine_run(&tcpPowerStatemaschinent);
    msr_statemaschine_run(&tcpPowerStatemaschinepent);

}
```

Der zyklische Aufruf der Fehlerverwaltung und das Rücksetzen der Ausgänge kann innerhalb der prestate_function einer übergeordneten Zustandsmaschine geschehen.

```
/*-----------------------------------------------*/
void tcpGlobalprestate(struct msr_statemaschine *sm)
/*-----------------------------------------------*/
{
    tcp_reset_all_outputs();
    tcp_check_for_errors(sm);
    tcp_do_init_structures();

}
```

Ein Beispiel für die poststate_function einer übergeordneten Zustandsmaschine ruft die Berechnung der Regler auf. Deren Funktionsweise ist wiederum abhängig von den Zuständen der beteiligten Zustandsmaschinen.

```
/*-----------------------------------------------*/
void tcpGlobalppoststate(struct msr_statemaschine *sm)
/*-----------------------------------------------*/
{
    tcp_calc_controllers(sm->prevstate);
    tcp_reset_all_btns(sm);

}
```

Die Steuerung der Regler sowie weiterer Komponenten erfolgt über globale Flags als auch über die Zustände der beteiligten Zustandsmaschinen.

Beispiele für die Fehlerüberprüfung und das Rücksetzen der Ausgänge sind nachfolgend dargestellt.

```
/*------------------------------------------------*/
static void tcp_check_for_errors(struct msr_statemaschine *sm)
/*------------------------------------------------*/
{
    int  tcp_warn_flag = 0,tcp_error_flag = 0,tcp_crit_flag = 0;

    msr_error_flag = msr_check_for_errors_sub_state(&tcpGlobalStatemaschine);
    msr_error_flag |= msr_check_for_errors_sub_state(&tcpPowerStatemaschineht);
    msr_error_flag |= msr_check_for_errors_sub_state(&tcpPowerStatemaschinent);
    msr_error_flag |= msr_check_for_errors_sub_state(&tcpPowerStatemaschinepent);
    msr_error_flag |= msr_check_for_errors_sub_state(&tcpReglerStatemaschineht);
    msr_error_flag |= msr_check_for_errors_sub_state(&tcpReglerStatemaschinent);
```

```
    tcp_warn_flag = (msr_error_flag >> T_WARN) & 1;
    tcp_error_flag = (msr_error_flag >> T_ERR) & 1;
    tcp_crit_flag = (msr_error_flag >> T_CRIT) & 1;
    if (tcp_crit_flag ) {
        switchToState(sm,T_TCP_critical);
        return;
    }
    if (tcp_error_flag )
        switchToState(sm,T_TCP_error);
}
```

Da mehrere Zustandsmaschinen parallel abgearbeitet werden können, ist die Fehlerüberwachung nicht mehr allein von dem Zustand der übergeordneten Zustandsmaschine abhängig. Der Fehlerbehandlung werden die Strukturpointer der einzelnen Zustandsmaschinen übergeben, um auf Fehler zu prüfen, die auf diese Zustandsmaschine Bezug nehmen (siehe 0.1.9). Somit können Fehlerabfragen realisiert werden, die vom Zustand einer einzelnen Zustandsmaschine abhängig ist.

Es muß sorgfältig darauf geachtet werden, dass in `_reset_all_outputs()` alle relevanten Ausgänge zu Null bzw. auf einen definierten Wert gesetzt werden.

```
/*-----------------------------------------------------*/
static void aip_reset_all_outputs()
/*-----------------------------------------------------*/
{

#include "aip_reset_all_outputs.inc"

    int_watchdog_switch = 0;
    int_redflashlight_switch = 0;

    ...

    PanelSetPointValve = -2.0;
    PanelSetPointSafetyValve = -10.0;

    msr_operating = 0;
    msr_trigger = 0;
}
```

### 0.1.9 Fehlerbehandlung durch die Software

Auftretende Fehler, die von der Software erkannt werden können, sind durch ihre Schwere in mehrere Fehlerklassen unterteilt. Zu jeder Fehlerklasse existiert in der Zustandsmaschine ein bestimmter Zustand, der für die Behandlung des aufgetretenen Fehlers zuständig ist.

### Fehlerklasse kritischer Fehler

Fehler, die in diese Klasse eingestuft werden, bewirken die Abschaltung des Watchdogs des IPC's. Dadurch wird Not-Aus ausgelöst. Es werden alle elektrischen Leistungsverbraucher des Aggregates und die Hydraulikventile vom Netz genommen. Der IPC bleibt weiter aktiv. Zur Inbetriebname nach dem Auftreten eines Fehlers muss dieser behoben werden, und der Not-Aus mit dem Start-Taster am Bedienfeld quittiert werden.

### Fehlerklasse Fehler

Diese Fehlerklasse ist für Fehler vorgesehen, auf die mit einem "weichen" Abschalten reagiert werden kann. Nach Beheben des Fehlers und Quittierung des Fehlersignals in der Bediensoftware kann das Aggregat weiter verwendet werden.

### Fehlerklasse Warnung

Fehler, die in diese Klasse eingestuft werden, bewirken eine Warnung an den Benutzer. Der Prozess kann trotz der Warnung weiterbetrieben werden. Der weitere Betrieb läuft auf das Risiko des Benutzers. Die Warnung kann durch Beheben des Fehlers und Quittierung der Warnung in der Bediensoftware abgeschaltet werden. Wird die Warnung ohne Beheben des Fehlers in der Bediensoftware quittiert, wird die Warnung direkt wieder angezeigt.

Ähnlich wie Parameter und Kanäle kann die Überwachung von Variablen ebenfalls registiert werden `msr_error_reg.h`, `msr_error_reg.c`. Dies geschieht mit `msr_reg_chk_failure_sub_state()`.

```
...

msr_reg_chk_failure_sub_state
         (&p_min_ht,                         /*zu überprüfende Variable*/
          TDBL,                              /*Typ der Variablen*/
          T_CHK_MIN,                         /*Überprüfungskriterium*/
          100.0,                             /*Grenzwert*/
          T_CRIT,                            /*Fehlerklasse*/
          "Druckabfall Mineral HT",          /*Ausgabestring*/
          &tcpPowerStatemaschineht,          /*Zustandsmaschine die relevant ist*/
          CHK_INCL,                          /* Überprüfung inclusive der
                                             folgenden Zuständen*/
          2,                                 /* Anzahl der folgenden Zustände*/
          T_TCP_POW_activ,T_TCP_POW_init); /* kommaseparierte Zustandsliste*/
```

Wird eine Variable zur Fehlerüberwachung registriert, erfolgt auf jeden Abtastschritt des Echtzeitprozesses die Überwachung auf die angegebene Bedingung.

Als Fehlerklassen stehen zur Verfügung: `T_INFO,T_WARN,T_ERR,T_CRIT`.

Die Überprüfungkriterien sind: `T_CHK_MAX`: Überwachung auf Überschreitung, `T_CHK_MIN`: Überwachung auf Unterschreitung, `T_CHK_HIGH`: Überwachung auf logisch True (nur Typ int),`T_CHK_LOW`: Überwachung auf logisch False (nur Typ int) `T_CHK_USR`: Fehler wird „händisch“ ausgelöst (siehe unten).

Es wird die Zustandsmaschine an die Funktion übergeben in dessen über die Liste definierten Zustände, die Fehlerüberprüfung ausgelassen `CHK_EXL` oder durchgeführt `CHK_INCL` werden soll.

Beim Auftreten eines Fehlers wird einmalig eine Meldung an alle User-Prozesses verschickt. Der Fehler bleibt jedoch solange anliegen, bis er durch den Aufruf der Funktion `msr_reset_all_errors();` zurückgesetzt wird. Üblicherweise ist dieser Aufruf als Funktionsaufruf für Nutzerprozesse registriert (in `msr_module()`):

```
msr_reg_funktion_call("/Control/btn_ack",&msr_reset_all_errors);
```

Durch den Funktionsaufruf werden alle Fehler zurückgesetzt; Fehler, die noch anliegen treten wieder auf.

Soll ein Fehler nicht durch die interne Überwachung aufgelöst werden, sondern durch einen Funktionsaufruf, kann dies durch die Funktion `msr_raise_error()` erfolgen. Die Fehlerklasse ist dann `T_CHK_USR`.

Beispiel:

```
struct msr_error_list aip_rotation_error;

...

/*Registrieren */
aip_rotation_error =
  *((struct msr_error_list *)msr_reg_chk_failure_sub_state(NULL,
                                                            TINT,
                                                            T_CHK_USR,
                                                            0,
                                                            T_ERR,
                                 "Rotation speed to high for actual Slatangle",
                                                            NULL,
                                                            0,
                                                            0));
```

```
...

/* Auslösen des Fehlers */
msr_raise_error(&aip_rotation_error);
```

Die Abfrage der Fehlerliste geschieht üblicherweise in der Zustandsmaschine durch den Aufruf von: `msr_check_for_errors_sub_state(&AdresseZustandmaschinenstruktur)`, der die zu überprüfende Zustandsmaschine (somit deren Zustände) mitgegeben wird.

```
/*---------------------------------------------------------------*/
static void aip_check_for_errors(struct msr_statemaschine *sm)
/*---------------------------------------------------------------*/
{

    msr_error_flag = msr_check_for_errors_sub_state(&globalstatemaschine);
    msr_error_flag |= msr_check_for_errors_sub_state(&secondstatemaschine);


   /* Wechsel in einen definierten Zustand der globalen
     übergeordneten Zustandsmaschine sofern Fehler vorliegen */
        if (((msr_error_flag >> T_ERR) & 1))
            switchToState(sm,T_AIP_error);

        if (((msr_error_flag >> T_CRIT) & 1))
            switchToState(sm,T_AIP_critical_error);

}
```

### 0.1.10 Regler

Für allgemeine Regelungsaufgaben steht ein PID-Regler mit vielfältiger Überwachungsfunktionalität zur Verfügung (`msr_controller.h, msr_controller.c`). Für die Verwendung müssen die Strukturen `struct msr_controller_states und struct msr_controller_params` initialisiert und verwendet werden. Die Trennung von Parametern und Zustandsgrößen ermöglicht die Verwendung von einem Parametersatz für mehrere Regler bzw. die Verwendung von verschiedenen Parametersätzen für einen Regler. Die Initialisierung der Parameter geschieht mit `msr_controller_param_init()`:

```
int msr_controller_param_init(struct msr_controller_parameter *par,
```

```
   char *name,              /* Basisname für die Erstellung der Parameternamen*/
   char *unit,              /* Einheit des Eingangssignals */
   unsigned int flags,      /* Flags für die Registierung der Parameter MSR_W,....*/
   double abtastfrequenz,   /* Abtastfrequenz des Reglers in Hz */
   double kp,               /* P-Anteil des Reglers V/m */
   double ki,               /* I-Anteil des Reglers V/ms */
   double iog,              /* I-Anteil maximaler Ausgangswert in V */
   double iug,              /* I-Anteil minimaler Ausgangswert in V */
   double kd,               /* D-Anteil des Reglers Vs/m */
   double td,               /* Einstellzeit des D-Anteils in s (der D ist ein DT1!) */
   double max_vel,          /* Maximal zulässige Sollwertänderung in 1/s */
   double tsi,              /* Einstellzeit für den Vergleich soll-ist  in s */
   double max_diff_soll_ist,/* Maximal zulässige Abweichung zwischen Soll- und Istwert:
                               Überschreitung-> Fehler */
   double max_diff_ist,     /* Max. zulässige Abweichung des Istwertes zum Vorherigen */
   double max_diff_ist_ist, /* Maximal Abweichung zwischen 2 Istwerten */
   double max_ist,          /* Maximal zulässiger Istwert: Überschreitung -> Fehler */
   double min_ist,          /* Minimal zulässiger Istwert: Überschreitung -> Fehler */
   double max_out,          /* Maximal zulässiges Ausgangssignal des Reglers in V */
   double min_out,           /* Minimal zulässiges Ausgangssignal des Reglers in V */
   int invert_out,           /* Ausgang invertieren */
   double (*c_skale)(struct msr_controller_states *val)) /* Funktionszeiger auf eine
                               Skalierungsfunktion mit der
                               abhängig von Soll- oder Istwert eine Bewertung
                               im Regler vorgenommen werden kann */
```

Neben der Initialisierung werden auch die notwendigen Parameter und Fehler in dem globalen Listen initialisiert.

Die Initialisierung der Zustandsgrößen erfolgt mit `msr_controller_state_init()`:

```
int msr_controller_state_init(struct msr_controller_states *val,
    char *name,                /* Basisname unter dem die Kanäle
                                  registiert werden */
   struct msr_statemaschine* statemaschine; /*Zustandsmaschine, in
                                  dessen Zuständen der Reglerstatus überwacht
                                  wird */
    char *unit,                /* Einheit */
    double *ist1,              /* Adresse: Istwert 1 */
    double *ist2,              /* Adresse: Istwert 2 */
    double *soll,              /* Adresse: Sollwert */
    double *out);              /* Adresse: Ausgang */
```

An diese Routine werden die Adressen der benötigten Variablen übergeben, so dass der Regler dann zyklisch nur durch Aufruf von:

```
int controller_run(struct msr_controller_parameter *par,
                      struct msr_controller_states *val,
                                                  int flag)
```

abgearbeitet werden kann. Der Regler wird durch das `flag` aktiviert und deaktiviert. Die Reglerroutine sollte auch dann aufgerufen werden, wenn der Regler deaktiviert ist (mit flag == 0). Dies ermöglicht dem Regler auf Zustandsänderungen im Flag zu reagieren.

Neben der PID-Funktion verfügt der Regler über folgende weitere Eigenschaften:

- Ist-Ist-Wert Vergleich (Filter über 10 Abtastwerte), Auslösen eines Fehlers
- Überwachung des Ist-Wertes auf Sprünge, Auslösen eines Fehlers
- Steigungsbegrenzung des Soll-Wertes
- Vergleich Soll-Ist-Wert (Einstellbarer Filter), Auslösen eines Fehlers
- Begrenzung des I-Anteils
- Begrenzung des Ausgangssignals
- Überwachung des Ist-Wertes auf Überschreitung von Maximum und Minimum, Auslösen eines Fehlers

Da der Regler die Fehler wie bereits beschrieben über die Funktion `msr_reg_chk_failure_sub_state()` registrieren, muss der Routine `msr_controller_state_init()` die Adresse der Zustandsmaschinenstruktur übergeben werden, die den Regler kontrolliert.

Die Registrierung von reglerinternen Größen als Kanäle kann über eine globale Variable in `msr_controller.c` unterbunden werden:

```
msr_controller_no_channels = 1; //für alle nachfolgenden Reglerregistrierungen
                                //keine Kanäle erzeugen
```

Beispiel für einen Positionsregler für drei Achsen mit gleichen Parametern:

```
/* Strukturen für Positionsregler */
struct msr_controller_parameter posreglerparams;
struct msr_controller_states posregler[3];

...

* Initialisierung Positionsregler (alle drei mit gleichen Parametern)*/
```

```
msr_controller_param_init(&posreglerparams,
    "/Cylinder/Position Controller",  /* Basisname */
    "mm",                      /* Einheit des Eingangssignals */
    MSR_AW,
    MSR_ABTASTFREQUENZ,        /* Abtastfrequenz des Reglers in Hz */
    1.0,                       /* P-Anteil des Reglers V/m */
    1.0,                       /* I-Anteil des Reglers V/ms */
    2.0,                       /* I-Anteil maximaler Ausgangswert in V */
   -2.0,                       /* I-Anteil minimaler Ausgangswert in V */
    0.0,                       /* D-Anteil des Reglers Vs/m */
    0.01,                      /* Einstellzeit des D-Anteils in s (der D ist ein DT1!) */
    200.0,                     /* Maximal zulässige Sollwertänderung in 1/s */
    10.0,                      /* Einstellzeit für den Vergleich soll-ist in s */
    100.0,                     /* Maximal zulässige Abweichung zwischen Soll- und Istwert:
                                  Überschreitung-> Fehler*/
    10.0,                      /* Max. zulässige Abweichung des Istwertes zum Vorherigen */
    20.0,                      /* Maximal Abweichung zwischen 2 Istwerten */
    3000.0,                    /* Maximal zulässiger Istwert: Überschreitung -> Fehler */
    1000.0,                    /* Minimal zulässiger Istwert: Überschreitung -> Fehler */
    10.0,                      /* Maximal zulässiges Ausgangssignal des Reglers in V */
   -10.0,1,NULL);              /* Minimal zulässiges Ausgangssignal des Reglers in V */

    msr_controller_no_channels = 1;

    msr_controller_state_init(&posregler[0],
                              "/Cylinder/1/Position Controller",
                               &tcpReglerStatemaschinent,
                               "mm",
                               &Cylinder1Position,
                               &Cylinder1Position,
                               &Cylinder1SetPointPosition,
                              &Cylinder1SetPointValve);

    msr_controller_state_init(&posregler[1],
                              "/Cylinder/2/Position Controller",
                               &tcpReglerStatemaschinent,
                               "mm",
                               &Cylinder2Position,
                               &Cylinder2Position,
                               &Cylinder2SetPointPosition,
                              &Cylinder2SetPointValve);

    msr_controller_state_init(&posregler[2],
                              "/Cylinder/3/Position Controller",
                               &tcpReglerStatemaschinent,
                               "mm",
                               &Cylinder3Position,
                               &Cylinder3Position,
                               &Cylinder3SetPointPosition,
```

```
                              &Cylinder3SetPointValve);


...


/* Lageregler ausführen-------------------------------------------------------*/
for(i=0;i<3;i++)
  controller_run(&posreglerparams,&posregler[i],SystemFlagPositioncontrolactive);
```

### 0.1.11 Funktionsgenerator

Für die Generierung einfacher periodischer Signale kann ein Funktionsgenerator verwendet werden (`msr_functiongen.h`, `msr_functiongen.c`). Dieser kann sinus-, dreiecks- und rechteckförmige Signale generieren. Um die Funktionalität nutzen zu können muss eine Struktur vom Typ `struct msr_function_gen_parameter` erzeugt und initialisiert werden. Die Initialisierung geschieht einmalig über folgende Funktion:

```
int msr_function_gen_init(struct msr_function_gen_parameter *par,
                          char *name,              /* Basisname */
                          double f0,                 /* frequenz in Hz */
                          double ampl,               /* Amplitude */
                          double offs,               /* Offset */
                          double abtastfrequenz,       /* in HZ */
                          enum msr_func_gen_signals sig);  /* Signaleform */
```

Die hier übergebenen Parameter sind die Startbedingungen des Funktionsgenerators, diese können zur Laufzeit variiert werden. Als Signalform kann zwischen `T_FUNC_SINUS, T_FUNC_SQUARE, T_FUNC_TRIANGLE` gewählt werden.

Anbei ein Beispiel dieser Funktion:

```
  struct msr_function_gen_parameter function_example;

 ...

   msr_function_gen_init(&function_example,
                         "/Funktionsgenerator Beispiel",
                         0.0,
                         0.0,
```

```
0.0,
MSR_ABTASTFREQUENZ,
T_FUNC_SINUS);
```

Um den Funktionsgenerator zu starten, muss dieser zyklisch einmal pro Abtastschritt mit der Funktion `msr_controller_run` aufgerufen werden.

```
double msr_function_gen_run(struct msr_function_gen_parameter *par);
```

Der Rückgabewert der Funktion ist das generierte Signal des Funktionsgenerators.

### 0.1.12 Hardwaretreiber

- ADDI-DATA: für folgende ADDI-DATA-Karten stehen Treiber zur Verfügung:
  - PCI3120: 16 bit, 16-Kanal analog-In, 14 bit, 8-Kanal analog-Out
  - PCI1500: 32 digital I/O
  - PCI1710: Flexibel programmierbare Zählerkarte

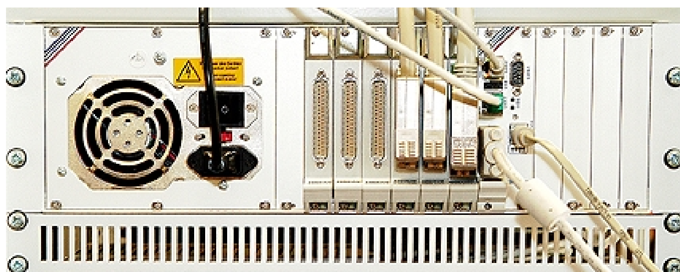

Abbildung 5: Industrie-PC von SMA mit ADDI-DATA Compact-PCI-Steckkarten

- Beckhoff CX1100: Für den Zugriff auf den Beckhoff K-Bus steht ein Treiber zur Verfügung `beckh_cx1100.c,.h` die Funktion ist nachfolgend dokumentiert

### 0.1.13 Beckhoff CX1100-Treiber

Das Beckhoff CX1100- oder CX1000-System basiert auf einer intelligenten CPU-Einheit in Hutschienenbauweise. Dieses Grundsystem kann durch diverse Erweiterungsklemmen aufgestockt werden. Dazu zählen analoge und digitale I/O, Zählerkarten, PT100-Auswerteklemmen, usw.. Die Klemmen sind über den sogenannten K-Bus an den K-Buscontroller gekoppelt. Dieser Controller ist in dem, der CPU-Einheit angeschlossenem Netzteil integriert. Es handelt sich bei dem K-Bus um einen

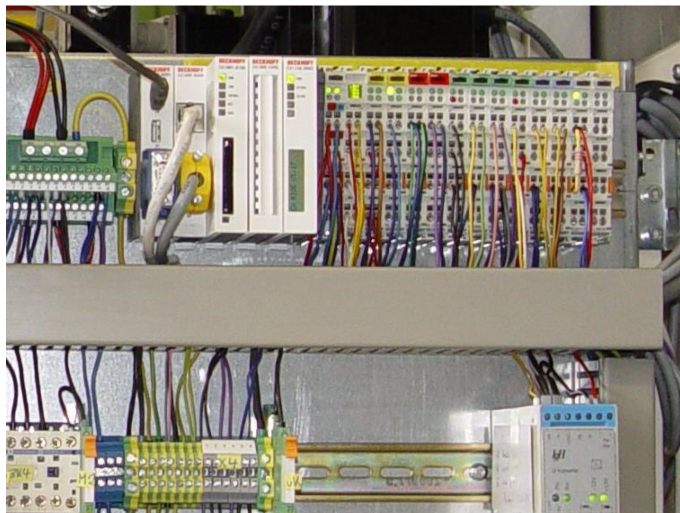

Abbildung 6: Beckhoff CX1100-System mit K-Busklemmen

seriellen Bus, der über Schleifkontakte an den Klemmen geführt ist. So kann der Bus nahezu beliebig erweitert werden. Weiterhin versorgt der Bus die angeschlossenen Klemmen mit einer Versorgungsspannung.

Jede Klemme besitzt abhängig von der Funktion einen mehr oder weniger intelligenten Mikrocontroller. Dieser besitzt mehrere Register, die vom K-Buscontroller ausgelesen oder geschrieben werden können. Diese Register werden per Memorymap in dem Speicher der CPU-Einheit abgebildet.

Derzeit werden folgende Klemmen unterstützt:

| Name | Typ | Anzahl Kanäle | Wandlungs-zeit | Auflösung | Beschreibung |
|---|---|---|---|---|---|
| KL1104 | DI | 4 | 3 ms | | 24 V |
| KL1124 | DI | 4 | 3 ms | | 5 V |
| KL1408 | DI | 8 | 3 ms | | 24 V |
| KL2124 | DO | 4 | 3 ms | | 5 V |
| KL2134 | DO | 4 | 3 ms | | 24 V |
| KL2404 | DO | 4 | | | 24 V |
| KL2408 | DO | 8 | | | 24 V |
| KL2602 | DO | 2 | | | Relaisklemme 230 v 5 A |
| KL3201 | AI | 1 | 200 ms | 12 bit | PT100-Eingangsklemme |
| KL3202 | AI | 2 | 200 ms | 12 bit | PT100-Eingangsklemme |
| KL3051 | AI | 1 | 1 ms | 12 bit | 4-20 mA |
| KL3062 | AI | 2 | 2 ms | 12 bit | 0-10 V |
| KL3064 | AI | 4 | 4 ms | 12 bit | 0-10 V |

| Name | Typ | Anzahl Kanäle | Wandlungszeit | Auflösung | Beschreibung |
|---|---|---|---|---|---|
| KL3403 | AI | 3 | | 12 bit | Leistungsmessklemme |
| KL3404 | AI | 4 | 2 ms | 12 bit | ±10 V |
| KL4002 | AO | 2 | 1.5 ms | 12 bit | 0-10 V |
| KL4004 | AO | 4 | 2 ms | 12 bit | 0-10 V |
| KL4032 | AO | 2 | 1.5 ms | 12 bit | ±10 V |
| KL4034 | AO | 4 | 2 ms | 12 bit | ±10 V |
| KL5111 | COUNTER | 3 | max.200 ns | 16 bit | 24 V |
| KL5111p | COUNTER | 3 | max.200 ns | 16 bit | 24 V |
| KL9010 | SYSTEM | | | | Busendklemme |
| KL9110 | SYSTEM | | | | 24 V Einspeiseklemme, Diagnose |
| KL9187 | SYSTEM | | | | Masseeinspeiseklemme |
| KL9186 | SYSTEM | | | | 24 V Einspeiseklemme |
| KL9210 | SYSTEM | | | | 24 V Einspeiseklemme, Sicherung |
| KL9505 | SYSTEM | | | | 5 V Netzteilklemme |

Der Treiber `beckh_cx1100.c,.h` stellt Funktionen zur Verfügung, um die genannte Hardware kontrolliert anzusprechen. Diese müssen in den entsprechenden Files eingebunden werden.

Ein wesentlicher Punkt zur Initialisierung für den Treiber ist die korrekte Angabe der Reihenfolge der Klemmen auf dem angeschlossenen Bus. Diese Informationen mit dem im Treiber vorliegenden Konfigurationsmerkmalen und der Speichervariablen der einzelnen Klemmen wird durch die Struktur `struct cxklemme busklemmendef[]` repräsentiert. Die Struktur muss wie folgt global deklariert werden:

```
struct cxklemme busklemmendef[] = {KL4004,KL3404,KL1104,KL2134,KL9010};

#define CXNUMKLEMMEN (sizeof(busklemmendef)/(sizeof(struct cxklemme)))
```

Die Klemmen müssen ab der CPU-Einheit bis hin zur Busendklemme aufgezählt werden. Hinter den Klemmenbezeichnungen steht eine Struktur vom Typ `cxklemme`, die alle treiberspezifischen Informationen der Klemme bereithält .

Die Initialisierung dieser Struktur und des Treibers nimmt folgende Funktion vor:

```
cx1100_init(busklemmendef,CXNUMKLEMMEN);
```

Weiterhin legt die Funktion Proc-Verzeichniseinträge an:

- `/proc/cx1100/info`: Allgemeine Informationen zur Treiberkonfiguration und -zustand,
- `/proc/cx1100/DPRamInput` : Direktes Abbild des Mappingspeichers des K-Bus-Controllers,
- `/proc/cx1100/ModulIO` : Registerinhalte der Klemmen, aufgeteilt in Daten- und Status-bereiche.

Abhängig von der Anzahl und Typ der am K-Bus angehängten Klemmen variiert die minimale mögliche Zykluszeit am Bus. Diese wird ebenfalls durch die Initialisierungsroutine gemessen und über die Datei `/proc/cx1100/info` ausgegeben. Zu beachten ist, dass die Zykluszeit nicht die Interruptzykluszeit übersteigt.

Über den Aufruf von

```
cx1100_cleanup();
```

wird der Treiber entladen. Es sind keine weiteren Zugriffe auf die Klemmen mehr möglich.

Der Funktionsaufruf

```
cx1100_trigger_kbus(busklemmendef,CXNUMKLEMMEN);
```

triggert einen K-Buszyklus. Die zu schreibenden Werte werden aus der Struktur `busklemmendef` zur Klemme übertragen. Die zu lesenden Werte werden von den Klemmen in die Struktur `busklemmendef` geschrieben.

Folgende Makros sind zum Zugriff auf die Informationen der Struktur `busklemmendef` definiert:

- `wert=KBUSget(busklemmendef,5,0)`: Liest aus der Struktur `busklemmendef` den gewandelten Wert des ersten Kanals ("0") der sechsten Klemme ("5") im K-Bus und legt ihn in der Variable `wert` ab.
- `status=KBUSgetstatus(busklemmendef,5,0)`: Liest bei Verwendung von intelligenten Klemmen aus der Struktur `busklemmendef` den Status des ersten Kanals ("0") der sechsten Klemme ("5") im K-Bus und legt ihn in der Variable `status` ab.

- `KBUSset(busklemmendef,5,0,wert)`: Schreibt den Inhalt von `wert` in die Struktur `busklemmendef` an die Speicheradresse der Register des ersten Kanals ("0") der sechsten Klemme ("5") im K-Bus. Wird der K-Bus getriggert, wird dieser Wert auf die Klemme übertragen.

### 0.1.14 Sonstige Funktionalitäten

Für folgende Funktionalitäten stehen Softwaremodule zur Verfügung, die zur Zeit nur im Quelltext dokumentiert vorliegen:

- Schrittmotoransteuerung: `msr_stepper.c, .h`
- Funktionsgenerator: `msr_functiongen.c, .h`
- Korrelationfunktion: `msr_korrelation.c, .h`
- PWM-Generator: `msr_pwmgen.c,.h`

### 0.1.15 Kommunikation zwischen Client und Server

Die Kommunikation mit dem Modul erfolgt über ein Character-Device, das von beliebig vielen Prozessen zum Lesen und Schreiben geöffnet werden kann und dabei diese Prozesse jeweils getrennt verwaltet. Dieses Verfahren wird ganz ähnlich für `/dev/tty` angewandt. Der Datenaustausch geschieht rein ASCII-basiert mit einer XML-ähnlichen Syntax. Hierzu verfügt das Kernelmodul über einen vereinfachten XML-Parser und Interpreter. Der Userprozess greift zum Beispiel über simple Standard-IO-Funktionen auf das Device-File zu, das beispielsweise den Namen `/dev/msr` besitzen kann.

Über das Device-File können entsprechend Parameter auf dem Modul gelesen und geschrieben werden sowie das dauerhafte Senden von Meßwerten gestartet und gestoppt werden. Diagnoseinformationen des Moduls werden auch über Dateien im Verzeichnis `/proc` präsentiert.

#### 0.1.15.1 Schnittstelle zum Kernelmodul

Dieser Abschnitt beschreibt den Befehlssatz, der Anwenderprogrammen zur Kommunikation mit dem Kernelmodul zur Verfügung stehen. Die Kommunikation erfolgt über das serielle device `/dev/msr`. Auf dieses Device kann lesend und schreiben von beliebig vielen Clients zugegriffen werden. Für den Netzwerkzugriff auf das Kernelmodul steht der Port 2345 zur Verfügung (siehe hierzu Kapitel 0.1.17).

Befehle und Daten werden in beide Richtungen im ASCII-Format ausgetauscht und folgen weitgehend einer XML-Syntax:

```
<tag option1="Wert1" option2="Wert2" ...>
```

Für viele Kommandos gibt es eine Kurzform (Alias).

**Ping:**

Funktion: Kontakt zum Echtzeitprozess überprüfen

Tag: `<ping>`

Optionen: -

Alias: -

Antwort des Prozesses:`<ping>`

**Parameter lesen:**

Funktion: lesen die Liste aller Parameter. Tag: `<read_parameter>`

Optionen: `name="Name",short="true",id="ID"`

Alias: `<rp>`

Beispiel: `<read_parameter>`
Antwort des Prozesses:

```
<parameters>
<parameter name="/Trigger" unit="" index="0" typ="TINT" flags="3" value="1" init="0" ll="0" ul="1"/>
<parameter name="/Taskinfo/Abtastfrequenz" unit="Hz" index="1" typ="TDBL" flags="1" value="1000.0000" init="1000.0000" ll="50.0000" ul="5000.0000"/>
<parameter name="/Control/btn_ack" unit="" index="2" typ="TFCALL" flags="3" value=>
<parameter name="/failures/crit_error/Card 3120 write analog output fail" unit="" index="3" typ="TINT" flags="3" value="0" init="0" ll="0" ul="0"/>
<parameter name="/failures/crit_error/Card 3210 read analog output fail" unit="" index="4" typ="TINT" flags="3" value="0" init="0" ll="0" ul="0"/>
<parameter name="/failures/crit_error/Card 3120 A-D Wandler not ready" unit="" index="5" typ="TINT" flags="3" value="0" init="0" ll="0" ul="0"/>
<parameter name="/failures/crit_error/Card 1500 write digital output fail" unit="" index="6" typ="TINT" flags="3" value="0" init="0" ll="0" ul="0"/>
<parameter name="/failures/crit_error/Card 1500 read digital output fail" unit="" index="7" typ="TINT" flags="3" value="0" init="0" ll="0" ul="0"/>
<parameter name="/Version/msr_globals" unit="" index="8" typ="TSTR" flags="1" value=""/>
</parameters>
```

Beispiel: `<read_parameter name="/Trigger">`: liest den Parameter */Trigger* aus.

```
<parameter name="/Trigger" unit="" index="0" typ="TINT" flags="3" value="1" init="0" ll="0" ul="1"/>
```

Beispiel: `<read_parameter name="/Trigger" short="true">`: liest den Parameter */Trigger* aus und läßt die Limits weg.

Antwort des Prozesses:

```
<parameter name="/Trigger" index="0" value="1"/>
```

Beispiel: `<rp name="/Trigger" id="ax1234">`: liest den Parameter */Trigger* aus und schickt eine *id* mit. Die ID kann für die Quittierung eines individuellen Lesezugriffs verwendet werden.

Antwort des Prozesses:

```
<parameter name="/Trigger" unit="" index="0" value="1" id="ax1234" ...>
<ack id="ax1234"/>
```

**Parameterwerte lesen:**

Funktion: lesen der Parmeterwerte in Kurzform

Tag:`<read_param_values>`

Alias: -

Options: -

Beispiel:`<read_param_values>`

Antwort des Servers:`<param_values value="1|1000.0000||0|0|0|0|0|"/>`
Die Werte aller Parameter werden übertragen.

**Parameter schreiben:**

Funktion: einen einzelnen Parameter beschreiben

Tag: `<write_parameter>`

Alias: `<wp>`

Options: `name="Name", value="Value"`

Beispiel:`<wp name="/Trigger" value="1">`: beschreibt die Variable */Trigger* mit dem Wert *1*.

Hinweis: Für das Beschreiben eines Parameters müssen Schreibrechte vorhanden sein (siehe hierzu Kapitel: 0.1.6). Ist das Schreiben eines Parameters nicht erfolgreich, schickt der Meßprozess eine Fehlermeldung an den Client zurück. Der Client kann, wenn er die Fehlermeldung nicht auswertet, durch Lesen des Parameters überprüfen, ob der korrekt gesetzt wurde.

**Kanalliste lesen:**

Funktion: lesen der Kanaleigenschaften

Tag:`<read_kanaele>`

Alias:`<rk>`

Options:`name="Name"`

Beispiel:`<rk>`

Antwort des Prozesses:

```
<channels>
<channel name="/Time" unit="s" alias="" index="0" typ="TDBL" bufsize="5000" HZ="1000" value="1097511627.2284"/>
<channel name="/Time/sec" unit="s" alias="" index="1" typ="TULINT" bufsize="5000" HZ="1000" value="1097511627"/>
<channel name="/Time/usec" unit="us" alias="" index="2" typ="TULINT" bufsize="5000" HZ="1000" value="228434"/>
<channel name="/Trigger" unit="" alias="" index="3" typ="TINT" bufsize="5000" HZ="1000" value="1"/>
<channel name="/Taskinfo/Jiffies" unit="" alias="" index="4" typ="TULINT" bufsize="5000" HZ="1000" value="456761362"/>
<channel name="/Taskinfo/Controller_Execution_Time" unit="us" alias="" index="5" typ="TUINT" bufsize="5000" HZ="1000" value="65"/>
<channel name="/Taskinfo/Controller_Call_Time" unit="us" alias="" index="6" typ="TUINT" bufsize="5000" HZ="1000" value="1001"/>
<channel name="/APCI3120/Card0/ADWandler/Kanal0" unit="inc" alias="" index="7" typ="TUINT" bufsize="5000" HZ="1000" value="0"/>
<channel name="/APCI3120/Card0/ADWandler/Kanal1" unit="inc" alias="" index="8" typ="TUINT" bufsize="5000" HZ="1000" value="0"/>
<channel name="/APCI3120/Card0/ADWandler/Kanal2" unit="inc" alias="" index="9" typ="TUINT" bufsize="5000" HZ="1000" value="0"/>
<channel name="/APCI3120/Card0/ADWandler/Kanal3" unit="inc" alias="" index="10" typ="TUINT" bufsize="5000" HZ="1000" value="0"/>
</channels>
```

**Zyklische Datenübertragung beginnen:**

Funktion: Starten der zyklischen Datenübertragung zum Client. Die Übertragung der Daten erfolgt in Blöcken für die jeweiligen Kanäle, wobei jeweils das Blockende einen Zeitstempel erhält. Die Abtastrate für die Übertragung wird durch den Parameter `Reduction` eingestellt und die Blockgröße durch `Blocksize`. Reduction * Blocksize dürfen die Puffergröße des einzelnen Kanals nicht übersteigen. Mit `sync` und `quiet` kann zunächst auf der Echtzeitseite eine Übertragungsliste zusammengestellt und dann die Übertragung gleichzeitig (bei der gleichen Zeitmarke) gestartet werden.

Tag: `<xsad>`

Alias: -

Optionen:

```
channels="Kanalliste",
reduction="Untersetzung",
blocksize="Blocksize",
coding="ascii,Base64",
presision="0..9",
sync,
quiet
```

Beispiel 1: `<xsad channels="1,2" reduction="100" blocksize="10">`

Datenstrom:


```
<data time="1097571970.5014">
<F c="1" d="1097571969,1097571969,1097571969,1097571969,1097571970,1097571970,1097571970,1097571970,1097571970,1097571970"/>
<F c="2" d="601268,701270,801269,901269,1269,101348,201364,301380,401396,501412"/>
</data>
<data time="1097571971.5015">
<F c="1" d="1097571970,1097571970,1097571970,1097571970,1097571971,1097571971,1097571971,1097571971,1097571971,1097571971"/>
<F c="2" d="601428,701444,801460,901476,1492,101508,201524,301540,401556,501572"/>
</data>
<data time="1097571972.5017">
<F c="1" d="1097571971,1097571971,1097571971,1097571971,1097571972,1097571972,1097571972,1097571972,1097571972,1097571972"/>
<F c="2" d="601588,701603,801620,901635,1651,101667,201683,301699,401715,501731"/>
</data>
```


Beispiel 3: `<xsad channels="1,2" reduction="100" blocksize="10" "quiet">`
`<xsad channels="49" reduction="20" blocksize="5" precision="4" "sync">`

Datenstrom:


```
<data time="1097572127.5264">
<F c="1" d="1097572126,1097572126,1097572126,1097572126,1097572127,1097572127,1097572127,1097572127,1097572127,1097572127"/>
<F c="2" d="626270,726347,826362,926379,26395,126410,226427,326443,426459,526474"/>
<F c="49" d="4.1138,-3.9829,4.7730,-3.3194,5.3431"/>
</data>
<data time="1097572127.6264">
<F c="49" d=-2.6532,4.1660,-1.9843,2.9877,-1.3174"/>
</data>
<<data time="1097572127.7265">
<F c="49" d=1.8042,-0.6488,0.6179,0.0177,-0.5688"/>
</data>
<data time="1097572127.8265">
<F c="49" d=0.6881,-1.7572,1.3553,-2.9398,2.0184"/>
</data>
<data time="1097572127.9265">
```

```
<F c="49" d=-4.1181,2.6871,-5.2922,3.3497,-4.7514"/>
</data>
<data time="1097572128.0265">
<F c="49" d=4.0107,-4.0879,4.6714,-3.4208,5.3285"/>
</data>
<data time="1097572128.1265">
<F c="49" d=-2.7552,4.3488,-2.0886,3.1675,-1.4197"/>
</data>
s<data time="1097572128.2265">
<F c="49" d=1.9867,-0.7550,0.8041,-0.0869,-0.3836"/>
</data>
<data time="1097572128.3266">
<F c="49" d=0.5819,-1.5677,1.2494,-2.7530,1.9153"/>
</data>
o<data time="1097572128.4266">
<F c="49" d=-3.9323,2.5791,-5.1054,3.2447,-4.8603"/>
</data>
d<data time="1097572128.5266">
<F c="1" d="1097572127,1097572127,1097572127,1097572127,1097572128,1097572128,1097572128,1097572128,1097572128,1097572128"/>
<F c="2" d="626490,726506,826522,926538,26554,126570,226586,326602,426618,526634"/>
<F c="49" d=3.9060,-4.1938,4.5646,-3.5294,5.2235"/>
```


Beispiel 3: (BASE64-Kodierung der Rohdaten, Format: little endian)
`<xsad channels="49" reduction="20" blocksize="5" coding="Base64">`

Datenstrom:


```
<data time="1097572239.7443">
<F c="49" d="1ARqAjWB6j+AEEAIIAQAwMC0X9ov7fc/TO2l9lJ7CcCLM8WZ4kwBQA=="/>
</data>
<data time="1097572239.8443">
<F c="49" d="DNGF6EJ0EcA1dho7jZ0GQCloFDQKGhXA4J/vz/fnC0AUzYnmRHMSwA=="/>
</data>
<data time="1097572239.9444">
<F c="49" d="hV5CL6GXEEB9V74r35UPwBrtjHZG0xNAUggphBRCCsCpg9RB6iAVQA=="/>
</data>
<data time="1097572240.0444">
<F c="49" d="KL4T34nvBMAEt4HbwG0QQPq1/Fp+Lf+/vKfd0+5pB0Ck/lH/qH/0vw=="/>
</data>
<data time="1097572240.1444">
```


Die Übertragung kann für jeden Kanal individuell eingestellt werden und jederzeit durch erneutes Senden des `xsad`-Tags geändert werden.

**Zyklische Datenübertragung beenden:**

Funktion: Stopp der zyklischen Datenübertragung für alle oder ausgewählte Kanäle

Tag: `<xsod>`

Alias: -

Optionen: `channels="1,2,3,4,....''`

Beispiel: `<xsod channels="1,2">` stoppt die Datenübertragung für die Kanäle 1 und 2

Beispiel: `<xsod>` stoppt die Übertragung aller Kanäle

**Zugriffsbeschränkung:**

Funktion: Standardmäßig hat ein Prozess, der ein Device zum Kernelmodul öffnet, für Parameter nur lesenden Zugriff. Diese Beschränkung kann durch einen Befehl aufgehoben werden.

Tag: `<remote_host>`

Alias: -

Options:`access="allow", isadmin="true"`

Beispiel:`<remote_host access="allow" isadmin="true">` Setzt alle Schreibrechte auf Variablen für User und Administratoren (MSR_W, MSR_WA).

**Meldungen des Servers:**

Funktion: Mitteilungen des Prozesses. Diese werden ohne Anforderung vom Prozess verschickt.

Tag(vom Prozess): `info,warn,error,crit_error,broadcast`

Optionen: -

Alias: -

Antwort des Prozesses z.B.:
`<warn time="1097512770.728503" text="Parametername nicht vorhanden.>"`

**Broadcast:**

Funktion: Mitteilungen an alle angeschlossenen Clients senden

Tag: `<broadcast>`

Alias: -

Options: `action="Action",text="Text"`

Beispiel: `<broadcast text="Hallo Clients">` schickt die Meldung *Hallo Clients* an alle angeschlossenen Clients.

Antwort des Prozesses:
`<broadcast time="1097512770.728503"  text="Hallo Clients">`

**id:**

Funktion: Bestätigung eines Kommandos. Jedes Kommando kann mit einer ID versehen werden, die nach Ausführung des Kommandos zum Client zurückgesendet wird.

Tag: `<tag ...... id="myID">`

Alias: -

Options: -

Beispiel: `<ping id="123"/>`

Antwort: `<ping id="123"/><ack id="123"/>`

### 0.1.16 Organisation der Dateien

Für die Organisation der Echzeitsoftware hat sich folgende Dateiablage als sinnvoll herausgestellt: Im jeweiligen Projekt befinden sich alle C-Quelldateien in einem Unterverzeichnis z.B. `/rt`.

Alle Dateien, die mit `msr_` beginnen, oder sich im Unterverzeichnis `rt_lib` [8] sind Libraries und Treiber. Alle Dateien, die mit `projekt_` beginnen, sind speziell für das Projekt geschrieben.

| | |
|---|---|
| msr_param.h: | Definition der Abtastrate |
| msr_lists.c: | Funktionen für das Characterdevice, welches von dem Modul zur Verfügung gestellt wird |
| msr_reg.c: | Funktionen um Parameter und Kanäle zu registrieren |
| msr_interpreter.c: | Der Interpreter, der an das Device geschickte Kommandos verarbeitet (XML) |
| msr_charbuf.c: | Ringpuffer für lange Zeichenketten |
| msr_messages.c: | Routine zur Formatierung von Meldungen in log/messages und zum User |
| msr_error_reg.c: | Routine zur Registrierung von Fehlerüberwachungsfunktionen und zur Erzeugung von Fehlermeldungen in Messages und zum User |
| msr_proc.c: | Schnittstelle zum proc-Verzeichnis |
| msr_stepper.c: | Schrittmotoransteuerung |
| msr_controller.c: | PID-Regler |
| | |
| apci1710.c: | Treiber von Addidata (Modifiziert von IgH) |
| apci1500.c: | Treiber von Addidata (Modifiziert von IgH) |
| apci3120.c: | Treiber von Addidata (Modifiziert von IgH) |
| | |
| msr_io.c: | Schnittstelle zu den Kartentreibern (zum Teil projektspezifisch) |
| msr_module.c: | Schnittstelle zum Kernel, Initialisierung (zum Teil projektspezifisch) |
| msr_control.c: | Schnittstelle zu projektspezifischen Routine |
| | |
| projekt_statemaschine.c: | Zustandsmaschine und Regler |
| | |
| projekt_globals.c: | Registrierung aller Parameter und Kanäle sowie Skalierung der Rohdaten auf physikalische Werte und Überprüfung der Werte auf Verletzung von Grenzwerten, Berechnung der Ausgänge |
| projekt_globals.h: | Hier sind fast alle projektspezifischen Konstanten und Macros definiert! |
| Makefile: | ohne Kommentar |
| msr_load,msr_unload, msr_reload: | Shellscripte zum manuellen Laden und Entladen des Kernelmoduls (nur Root) |

[8] dies sollte ein Link auf die aktuelle Version der hier beschriebenen Library sein.

### 0.1.17 Netzwerk

Die Veröffentlichung von /dev/msr über einen TCP/IP-Port kann auf zwei verschiedene Weisen erfolgen:

- der msrd ist ein C-Programm welches die Ein- und Ausgaben von /dev/msr an STDIN und STOUT verfügbar macht. Es kann daher als Client des inetd verwendet werden. Dies ist auch der bestimmungsgemäße Gebrauch. Hierzu muss der inetd installiert sein und die Dateien /etc/inetd.conf um folgenden Eintrag ergänzt werden

  ```
  # IgH msr-server auf 2345 hm
  dbm stream tcp nowait root /opt/msr/bin/msrd msrd
  ```

  wenn davon ausgegangen wird, dass der msrd im angegebenen Verzeichnis (/opt/msr/bin) liegt.

- eine zweite Möglichkeit ist das Programm /opt/msr/bin/msr_serv.pl welches eigenständig verwendet werden kann und zusätzlich eine schwache Authentifizierung auf Grund des anfragenden Rechners durchführen kann. Dabei haben alle Clientrechner, die sich mit dem Steuerrechner verbinden, lesenden Zugriff. Der schreibende Zugriff wird über die Datei /opt/msr/etc/hosts.auth gesteuert, die eine Liste der gültigen IP-Adressen enthält.

Ein einfacher Test der TCP/IP Verbindung kann z.B. unter Unix/Linux mit dem Programm netcat durchgeführt werden. Das Beispiel zeigt die Kontaktaufnahme mit dem Rechner ida vom Rechner euler aus über den Port 2345.

```
user@euler:~> netcat ida 2345
<connected name="MSR" host="ida" version="131840"/>
<info time="1097531725.569731" text="new connection"/>

...
```