Fraunhofer
IPK

Kurzanleitung für EthercatSOEM Treiberkonfiguration unter Linux/ Xenomai 2.6.5

Am Beispiel für den Meßverstärker GSV 8 von ME-Messsysteme

Bearbeitet von

Norman Franke | Studentischer Mitarbeiter
norman.franke@ipk.fraunhofer.de | +49 30 39006-113

Betreut von

Dr.-Ing. Dragoljub Surdilovic | Senior Researcher
dragoljub.surdilovic@ipk.fraunhofer.de | +49 30 39006-172

Fraunhofer-Institut für Produktionsanlagen und Konstruktionstechnik IPK
Pascalstrasse 8-9, 10587 Berlin

Stand: 14. Dezember 2016

Inhaltsverzeichnis

Kapitel 1

Einleitung

In dieser kleinen Kurzanleitung wird die Implementierung der *Simple Open EtherCAT Master Library Version 1.3.0* in ein Linux Kernel 3.10.32 - Xenomai 2.6.5 im LXRT Userspace beschrieben. Die Bibliothek hält sich an die Vorgaben, die im EtherCAT-Standard formuliert wurden. Entgegen fertiger "Plig&Play" Lösungen, wie sie z.B. in Softwareumgebungen wie TwinCat o.ä. zu finden sind, erlaubt diese Bibliothek ein Höchstmaß an Konfiguration und Performanceoptimierung bis hinunter auf einzelne Worte der übertragenen Pakete.
Aus Performancegründen wurde das mitgelieferte Testbeispiel aus SOEM nicht aufgegriffen. EtherCAT besitzt zwei Arten mit Geräten zu Kommunizieren. Es steht eine Mailboxkommunikation (SDO- Service Data Object) und eine Prozessdatenkommunikation (PDO - Process Data Object) zur Verfügung. Die Mailboxkommunikation ist asynchron und wird verwendet, um Prozesstdaten zu Konfigurieren (Antriebsparameter, Abtastraten, usw.), gerätespezifische Konfigurationen vorzunehmen oder eine Diagnose von EtherCAT Slave-Geräten vorzunehmen. Die Prozessdatenkommunikation wird zum zyklischen Datenaustausch verwendet. EtherCAT Master können Informationen versenden und von anderen Mastern empfangen, sofern der Bus per Multi-Master betrieben wird. Es gibt die zwei Übertragungsmodi Pushed Data Exchange (Broadcast) und Polled Data Exchange. Bei Pushed Data Exchange kann jeder Teilnehmer Informationen in seinem eigenen Zyklus senden und alle Teilnehmer können voneinander Informationen empfangen. In diesem Modus werden die GSV8 Messverstärker mit dem Beispielsystem betrieben. Bei Polled Data Exchange gibt es die 1:1 Verbindung und die 1:n Verbindung.
Für den 1:1 Betrieb gilt:

- Ein Teilnehmer sendet seine Daten zyklisch (Client)
- Adressierte Teilnehmer senden eine Antwort (Server)

Für den 1:n Betrieb gilt:

- Ein Teilnehmer sendet seine Daten zyklisch (Client)
- Adressierte Teilnehmer senden eine Antwort (Server)
- "Weiche" Synchronisierung der Teilnehmer

Als Quelle dienen Simple Open EtherCAT Master Library Version 1.3.0 und EtherCAT für die Fabrikvernetzung- EtherCAT Automation Protocol (EAP)

Kapitel 2

Inbetriebnahme

2.1 Gerät identifizieren mit SOEM-slaveinfo.c

Um zu sehen, welche Teilnehmer am Bus angeschlossen sind, lohnt es sich das SOEM Testprogramm *slaveinfo.c* auszuführen. Dazu müssen die RTnet und Xenomai Bibliotheken inkludiert und verlinkt werden.

```
#ifdef RTNET // with xenomai
#include <sys/mman.h>
#include <native/task.h>
#endif
```

Listing 2.1: RTnet in Header einfügen

Ebenso in der Main Methode.

Listing 2.2: RTnet in main-Methode einfügen

Falls der GSV8 Messverstärker noch ein Firmwareupdate < 1.16 besitzt, fehlt die automatische Konfiguration des GSV8. Dies ist folglich manuell durchzuführen. Dafür werden die Variablen für die analogen und digitalten I/Os global per SDO auf dem GSV8 initialisiert. Für Firmwareversionen $>= 1.18$ geschieht dies seitens des Messverstärkers automatisch und dieses Kapitel kann übersprungen werden.

```
int os;
uint32 ob;
int16 ob2;
uint8 ob3;
int ob4;
```

Listing 2.3: Manuelle Konfiguration

Der GSV8 besitzt die EtherCAT Slave-Adresse 0x6130. In diesem Codebeispiel wird explizit davon ausgegangen, dass sich ein einzelner GSV8 für *slaveinfo* und *simple test* am PC befindet. Soll dies für mehrere Geräte eingefügt werden, so ist eine if-Abfrage mit Unterscheidung der Slave-Adressen einzufügen. Erst bei der generellen Implementierung wird auf n-Geräte eingegangen.
Da unsere Arbeitsgruppe nicht genau aus dem Treiberaufbau schließen konnte, ob mit *ec_slave[0]* alle Slave-Geräte oder der Master gemeint sind, wird jeder Client einzeln ausgelesen und angesprochen. Dies geschieht ab Index 1, also *ec_slave[1], ec_slave[2], etc.*.

```
if(ec_slave[0].state == EC_STATE_PRE_OP){
    //os=sizeof(ob4); ob4 = 0x01;
        //ec_SDOwrite(1,0x1C33,0x01,FALSE,os,&ob4,EC_TIMEOUTRXM);
    os=sizeof(ob4); ob4 = 0x6f72657a;
        ec_SDOwrite(1,0x6125,0x01,FALSE,os,&ob4,EC_TIMEOUTRXM);
        ec_SDOwrite(1,0x6125,0x02,FALSE,os,&ob4,EC_TIMEOUTRXM);
        ec_SDOwrite(1,0x6125,0x03,FALSE,os,&ob4,EC_TIMEOUTRXM);
        ec_SDOwrite(1,0x6125,0x04,FALSE,os,&ob4,EC_TIMEOUTRXM);
        ec_SDOwrite(1,0x6125,0x05,FALSE,os,&ob4,EC_TIMEOUTRXM);
        ec_SDOwrite(1,0x6125,0x06,FALSE,os,&ob4,EC_TIMEOUTRXM);
```

```
        ec_SDOwrite(1,0x6125,0x07,FALSE,os,&ob4,EC_TIMEOUTRXM);
        ec_SDOwrite(1,0x6125,0x08,FALSE,os,&ob4,EC_TIMEOUTRXM);

//printf("setting free run flag...\n");
}
```

Listing 2.4: Manuelles Konfigurieren der I/Os des GSV8

```
//set all required setting for GSV8
//by writing the needed values in the according registers
if (( ec_slavecount >= 1 ) &&
             (strcmp(ec_slave[i].name,"GSV-8") == 0))
                  {
  for (i = 1; i <= ec_slavecount; i++){
           printf("Set new TxPDO for slave %i.\n", i);
      os=sizeof(ob4); ob4 = 0x00;
           ec_SDOread(i,0x1A00,0x00,FALSE,&os,&ob4,EC_TIMEOUTRXM);
      printf("found %i in si0, state: %i, %i\n", ob4, ec_slave[i].state,
    EC_STATE_PRE_OP);

      os=sizeof(ob4); ob4 = 0x00;
           ec_SDOwrite(i,0x1A00,0x00,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61300120;
           ec_SDOwrite(i,0x1A00,0x01,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61300220;
           ec_SDOwrite(i,0x1A00,0x02,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61300320;
           ec_SDOwrite(i,0x1A00,0x03,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61300420;
           ec_SDOwrite(i,0x1A00,0x04,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61300520;
           ec_SDOwrite(i,0x1A00,0x05,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61300620;
           ec_SDOwrite(i,0x1A00,0x06,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61300720;
           ec_SDOwrite(i,0x1A00,0x07,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61300820;
           ec_SDOwrite(i,0x1A00,0x08,FALSE,os,&ob4,EC_TIMEOUTRXM);

      os=sizeof(ob4); ob4 = 0x61500108;
           ec_SDOwrite(i,0x1A00,0x09,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61500208;
           ec_SDOwrite(i,0x1A00,0x0A,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61500308;
           ec_SDOwrite(i,0x1A00,0x0B,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61500408;
           ec_SDOwrite(i,0x1A00,0x0C,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61500508;
           ec_SDOwrite(i,0x1A00,0x0D,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61500608;
           ec_SDOwrite(i,0x1A00,0x0E,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61500708;
           ec_SDOwrite(i,0x1A00,0x0F,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x61500808;
           ec_SDOwrite(i,0x1A00,0x10,FALSE,os,&ob4,EC_TIMEOUTRXM);
      os=sizeof(ob4); ob4 = 0x10;
           ec_SDOwrite(i,0x1A00,00,FALSE,os,&ob4,EC_TIMEOUTRXM);
      printf("Set new TxPDO for slave %i. Done.\n", i);
  }
}
```

Listing 2.5: manuelles Konfigurieren der I/Os des GSV8

Die Adresse für **ob4** ist der Bedienungsanleitung des GSV8 zu entnehmen.

2.2 Daten Auslesen mit SOEM-simple_test.c

Auch *simple_test.c* ist wieder für RTnet zu konfigurieren. Außerdem ist das Testprogramm für pthreads zu konfigurieren.

```
#include <unistd.h>
#include <pthread.h>

#ifdef RTNET
#include <sys/mman.h>
#include <native/task.h>
#endif
```

Listing 2.6: *<sys/mman.h>* für RTnet und *<native/task.h>* für Xenomai

Wie es auch in der main-Methode zu erkennen ist.

```
int main(int argc, char *argv[]) {
  int iret1;
  printf("SOEM (Simple Open EtherCAT Master)\nSimple test\n");

#ifdef RTNET
  mlockall(MCL_CURRENT | MCL_FUTURE);
  RT_TASK task;
  rt_task_shadow(&task, "SimpleTest", 99, T_JOINABLE);
#endif

  if (argc > 1) {
    // create thread to handle slave error handling in OP
    iret1 = pthread_create(&thread1, NULL, (void *) &ecatcheck,
        (void*) &ctime);
    // start cyclic part
    simpletest(argv[1]);
  } else {
    printf("Usage: simple_test ifname1\nifname = eth0 for example\n");
  }

  printf("End program\n");
  return (0);
}
```

Listing 2.7: Main Methode in simple_test

In der Vielzahl der SOEM-Versionen ist uns aufgefallen, dass es mehrere Arten der Implementierung des *simple_test* gibt. Wenn man sich durch die Struct-Definitionen von *ec_slave, ec_group, ODlist und OElist* gelesen hat, wird schnell klar, dass mit diesen Strukturen sogenannte SDO Objekte (Service Data Objects) und PDO Objekte (Process Data Objects) abgebildet werden. Die SDOs sind für die Mailboxkommunikation gedacht. In der ursprünglich verwendeten SOEM-Version werden die Listen ODlist und OElist erstellt und durch Iteration die dort eingetragenen Teilnehmer abgearbeitet und dann die Daten ausgelesen.

```
void si_sdo(int cnt) {
  int i, j;
  ODlist.Entries = 0;
  memset(&ODlist, 0, sizeof(ODlist));
  if (ec_readODlist(cnt, &ODlist)) {
    for (i = 0; i < ODlist.Entries; i++) {
      ec_readODdescription(i, &ODlist);
      while (EcatError)
        printf("%s", ec_elist2string());
      if (ODlist.Index[i] == 0x6130) {
        memset(&OElist, 0, sizeof(OElist));
        ec_readOE(i, &ODlist, &OElist);
        while (EcatError)
          printf("%s", ec_elist2string());
        printf("sensor values of slave %i:, ", cnt);
        //ODlist.MaxSub[i] + 1 or 7
        for (j = 1; j < 7; j++)
        {
          if ((OElist.DataType[j] > 0) && (OElist.BitLength[j] > 0)) {
            if ((OElist.ObjAccess[j] & 0x0007)) {
              //printing of actual sensor values
```

```
                int os;
                float ob4;
                os = sizeof(ob4);
                ob4 = 0x00;
                ec_SDOread(cnt, ODlist.Index[i], j, FALSE, &os,
                    &ob4, EC_TIMEOUTRXM);
                printf("%f ", ob4);
              }
            }
          }
        }
      }
      printf("\n");
    } else {
      while (EcatError)
        printf("%s", ec_elist2string());
    }
}
```

Listing 2.8: Konfigurationsvariablen

Dies dann Zyklisch in einer Schleife, wenn 'alle' Slave-Geräte einsatzfähig sind.

```
      if (ec_slave[0].state == EC_STATE_OPERATIONAL)
```

Listing 2.9: Konfigurationsvariablen

Der Hauptloop im Programm ist dann also

```
      do {
        ec_send_processdata();
        ec_receive_processdata(EC_TIMEOUTRET);
        ec_statecheck(0, EC_STATE_OPERATIONAL, 5000);
      } while (chk-- && (ec_slave[0].state != EC_STATE_OPERATIONAL));
      printf("Number of output: %dbyts, input %dbyts\n",
          ec_slave[1].Obytes, ec_slave[1].Ibytes);
      if (ec_slave[0].state == EC_STATE_OPERATIONAL)
      {
        printf("Operational state reached for all slaves.\n");
        inOP = TRUE;
        for (i = 1; i <= 10000; i++) {
          ec_send_processdata();
          wkc = ec_receive_processdata(EC_TIMEOUTRET);
          if (wkc >= expectedWKC) {
            for (slv = 1; slv <= ec_slavecount; slv++)
            {
              si_sdo(slv);
            }
            needlf = TRUE;
          }
        }
        inOP = FALSE;
      } else { //...
```

Listing 2.10: Konfigurationsvariablen

Die Kraftsensorwerte sind auf der Konsole auszugeben. So kann schnell validiert werden, ob die Konfiguration und die angezeigten Werte Sinn ergeben. Printouts ergeben auf dem Xenomai-System eine konstante durchschnittliche Zugriffszeit von 31 ms. Dies ist darauf zurückzuführen, dass die verwendeten SDO-Listen nur Service Objekte sind und bei Anfragen durch reduzierte Priorität auf das Gerät zugegriffen wird. Dies erzeugt hohe Latenzen. Das aktuelle Beispiel aus SOEM 1.3.1 verzichtet auf diese Listen und greift direkt auf die PDOs der Slave-Teilnehmer zu. Ein Vergleichstest wurde nicht angestellt. Jedoch ist dieser Sachverhalt eine sinnvolle Ergänzung zum Verständnis des Themas EtherCAT.

2.3 Daten Auslesen mit SOEM - Einbindung in ein Programm

In diesem Beispiel wurde die Firmware aktualisiert und der GSV8 teilt seine Konfiguration automatisch dem EtherCAT-Master mit. Eine manuelle Konfiguration aus Listing 2.5 fällt damit aus.
Der *#include ethercat.h* der SOEM Library ist essentiell. Nachdem die Struktur des SOEM verstanden wurde, ist es an der Zeit durch PDOs auf die Daten zuzugreifen. Um n-Geräte per Broadcast zu erreichen, müssen diese wie folgt initialisiert werden. Zuerst sind alle Teilnehmer festzustellen. Dann werden diese in den Zustand OPERATIONAL gesetzt. Es wird eine Anfrage zum Lesen der Daten gesendet, um zu schauen, ob alle Teilnehmer funktionieren. Dies wird solange wiederholt, bis der Slave sich OPERATIONAL meldet. Durch alle Slave-Geräte iteriert, wird am Ende in dem von uns eingeführten Arbeitsschritt die Liste aller angeschlossenen Geräte auf der Konsole angezeigt. Es ist anzumerken, dass die Variable *EC_TIMEOUTRET* durch die globale Variable *EC_TIMEOUTRET_C4G* ersetzt wurde, da die Robotersteuerung eine eigene Variable für den Timeout festlegt.

```
      for (int g = 0; g <= ec_slavecount; g++) {
        ec_slave[g].state = EC_STATE_OPERATIONAL;
      }
      for (int slv_cnt = 0; slv_cnt <= ec_slavecount; slv_cnt++) {
        // send one valid process data to make outputs in slaves happy
        ec_send_processdata();
        ec_receive_processdata(EC_TIMEOUTRET_C4G);
        // request OP state for all slaves
        ec_writestate(slv_cnt);
        chk_ = 40;
        // wait for all slaves to reach OP state
        ec_statecheck(slv_cnt, EC_STATE_OPERATIONAL, EC_TIMEOUTSTATE * 4);
        do {
          printf(
              "Wait to reach Operational state of slave ec_slave[%i]",
    slv_cnt);
          printf("\n");
          ec_send_processdata();
          ec_receive_processdata(EC_TIMEOUTRET_C4G);
          ec_statecheck(slv_cnt, EC_STATE_OPERATIONAL, 5000);
        } while (chk_-- && (ec_slave[slv_cnt].state != EC_STATE_OPERATIONAL
    ));

      }
      // final state check
      uint8_t ready_slaves = 0;
      for (int g = 0; g <= ec_slavecount; g++)
      {
        if (ec_slave[g].state == EC_STATE_OPERATIONAL) ready_slaves++;
      }
      // we count ec_slave[0], too! So this should be the master,
      // but not clearly documented
      // See SOEM/simple_test_rt
      if (ready_slaves ==  ec_slavecount+1)
      {
        printf("Operational state reached for %i of %i slaves.\n",
    ready_slaves,ec_slavecount+1 );
        return true;
      }else {
        printf("Not all slaves found! ready_slaves vs ec_slavecount is %i
    vs %i", ready_slaves, ec_slavecount);
        printf("\n");
        return false;
```

Listing 2.11: SOEM mit n-Geräten Konfigurieren

Jetzt kann man in einem Main Loop zyklisch die Messwerte auslesen. Dabei werden die Messverstärker per Daisychain an eine RTnetfähige Netzwerkkarte angeschlossen. Um die Ausfallsicherheit einer Netzwerkkarte zu erhöhen, kann das System auch per *cable redundancy* bzw. *redundancy ring* das Ende der Slave-Kette mit einer weiteren Netzwerkkarte verbinden und diese per warm stand-by mitlaufen lassen. Eine geeignete Behandlung bei Ausfall einer Netzwerkkarte ist hierbei zu implementieren.

```
1
   ec_send_processdata();
3  wkc_ = ec_receive_processdata(EC_TIMEOUTRET_C4G);
   if (wkc_ >= expectedWKC_) {
5    for (int l = 1; l <= ec_slavecount; l++) {
       if (l == 2)        //GSV8#1 is Slave number 2
7          {
         for (unsigned int j = 0;
             (j * sizeof(float)) < (ec_slave[l].Ibytes)
                 && (j < ft_length); j++) {
11       ft[j] = ((float*) ec_slave[l].inputs)[j];
         }            // end for loop #1
13     }            //end GSV8#1
       if (l == 1)
15         {
         for (unsigned int j = 0;
17           (j * sizeof(float)) < (ec_slave[l].Ibytes)
                 && (j <= ft_length_2); j++) {
19
           if (j < ft_length_2)
21            {
             ft_2[j] = ((float*) ec_slave[l].inputs)[j];
23         }
           if (j == ft_length_2)
25          {
            shm_ptr->raw_GasThrottle_s =
27          ((float*) ec_slave[l].inputs)[6]; //index starts a "0"
            }
29       } //end for loop#2

31     } // end GSV8#2
     }      //slaves loop
33 }        // wkc loop
```

Listing 2.12: Fallunterscheidung für n-Slaves mit zwei Fällen

Hier werden nun zwei Messverstärker angesprochen, bei dem zum Einen 6 Kanäle und zum Anderen 7 Kanäle ausgelesen werden. Die Adresse geht Float-Weise durch das Datenarray des EtherCAT-Pakets. Im einfachen Daisychained Modus ist zu beachten, dass der erste Slave am PC die letzte *ec_ slave[ID]* besitzt.

Um auf Nummer sicher zu gehen, ruft der zyklische Task ebenfalls nur die Daten ab, wenn alle Teilnehmer bereit sind. Hierbei wird sich auf die Zuverlässigkeit des GSV8 berufen und lediglich auf Timeouts gegengeprüft. Eine sichere softwareseitige Kontrolle der Zustände aller Teilnehmer wird mit der Methode *ecatcheck()* zur Verfügung gestellt. Dies ist mit der Performance noch zu vergleichen. Innerhalb des Q4/2016 oder Q1/2017 wird dies evaluiert werden.

```
1    if ( ec_slave[0].state == EC_STATE_OPERATIONAL && ec_slave[1].state ==
     EC_STATE_OPERATIONAL && ec_slave[2].state == EC_STATE_OPERATIONAL)
     receiveSensorData(forces, sizeof(forces), forces_Handgrip,
3        sizeof(forces_Handgrip));
```

Listing 2.13: Explizite zyklische Abfrage von 2-Geräten

Die Performance der hier gezeigten Implementierung beträgt seitens des PCs 2ms, seitens zwei angeschlossener GSB8 Messverstärker über EtherCAT 60µs für beide Bus-Teilnehmer.

2.3.1 Aussicht

Im Rahmen von Folgeprojekten wird unsere Arbeitsgruppe IPK/AUT eine Portierung von EtherCAT auf Xenomai 3 vornehmen. Dies steht zum Jahr 2017 an.

Kapitel 3

Betrieb

Die SOEM-Bibliothek stellt nach den Vorgaben des EtherCAT-Standards ein Fehlermodell. Dies erlaubt während des Betriebs die Feststellung eines fehlerhaften Betriebs. Dazu gehört das Auslesen der Fehlermeldungen der angeschlossenen Geräte, wie auch die ausbleibende Antwort eines angeschlossenen Gerätes (alive-/ "ich lebe noch"). Im Beispiel *simple_test.c* wird dazu ein separater Thread gestartet, der parallel zum Hauptprogramm den Status der Slave-Geräte überprüft und je nach Problemfall das Programm anhält oder beendet.

```
int main(int argc, char *argv[]) {
  int iret1;
  printf("SOEM (Simple Open EtherCAT Master)\nSimple test\n");

#ifdef RTNET
  mlockall(MCL_CURRENT | MCL_FUTURE);
  RT_TASK task;
  rt_task_shadow(&task, "SimpleTest", 99, T_JOINABLE);
#endif

  if (argc > 1) {
    // create thread to handle slave error handling in OP
    iret1 = pthread_create(&thread1, NULL, (void *) &ecatcheck,
        (void*) &ctime);
    // start cyclic part
    simpletest(argv[1]);
  } else {
    printf("Usage: simple_test ifname1\nifname = eth0 for example\n");
  }

  printf("End program\n");
  return (0);
}
```

Listing 3.1: Erzeugen des pthreads für ecatcheck

Anmerkung: Die Befehle, wie z.B. $\mathrm{rt}_t ask_s hadow()$, *sindinderXenomaiDokumentationzufinden*.

3.1 Fehlermodell

Die Genauigkeit des Fehlermodells eines Protokolls hängt von den Fehlerarten ab, die behandelt werden können. In der Datei *ethercattype.h* sind die Rahmenbedingungen für die Robustheit der SOEM Bibliothek definiert. Es werden Variablen für "weiches" Timeout, "hartes" Timeout, maximale Wiederholversuche bei Timeout, etc. festgelegt. In Abhängigkeit der Umgebung wird angeraten, die Parameter zu ändern. So sind die Angaben für maximal 100 kabelgebundene Slaves gemacht. Für schnurlose Verbindungen müssen ggf. Tests gemacht werden, um statistisch aussagekräftige Ergebnisse über die Zuverlässigkeit der Verbindung machen zu können.
Der Status der Slaves wird in der Liste *ec_slave[slave no.]* abgespeichert. Die Auswertung der Einträge kann entweder direkt nach den jeweiligen *printf()* Aufrufen erfolgen, oder die Liste wird in einer eigenen StatusMethode ausgewertet.

```
  while (1) {
    if (inOP
        && ((wkc < expectedWKC) || ec_group[currentgroup].docheckstate)) {
      if (needlf) {
        needlf = FALSE;
        printf("\n");
      }
      //* one ore more slaves are not responding */
      ec_group[currentgroup].docheckstate = FALSE;
      ec_readstate();
      for (slave = 1; slave <= ec_slavecount; slave++) {
        if ((ec_slave[slave].group == currentgroup)
            && (ec_slave[slave].state != EC_STATE_OPERATIONAL)) {
          ec_group[currentgroup].docheckstate = TRUE;
          if (ec_slave[slave].state
              == (EC_STATE_SAFE_OP + EC_STATE_ERROR)) {
            printf(
                "ERROR : slave %d is in SAFE_OP + ERROR, attempting ack.\n"
,
                slave);
            ec_slave[slave].state =
                (EC_STATE_SAFE_OP + EC_STATE_ACK);
            ec_writestate(slave);
          } else if (ec_slave[slave].state == EC_STATE_SAFE_OP) {
            printf(
                "WARNING : slave %d is in SAFE_OP, change to OPERATIONAL.\n
",
                slave);
            ec_slave[slave].state = EC_STATE_OPERATIONAL;
            ec_writestate(slave);
          } else if (ec_slave[slave].state > 0) {
            if (ec_reconfig_slave(slave, EC_TIMEOUTMON)) {
              ec_slave[slave].islost = FALSE;
              printf("MESSAGE : slave %d reconfigured\n", slave);
            }
          } else if (!ec_slave[slave].islost) {
            //* re-check state */
            ec_statecheck(slave, EC_STATE_OPERATIONAL,
                EC_TIMEOUTRET);
            if (!ec_slave[slave].state) {
              ec_slave[slave].islost = TRUE;
              printf("ERROR : slave %d lost\n", slave);
            }
          }
        }
        if (ec_slave[slave].islost) {
          if (!ec_slave[slave].state) {
            if (ec_recover_slave(slave, EC_TIMEOUTMON)) {
              ec_slave[slave].islost = FALSE;
              printf("MESSAGE : slave %d recovered\n", slave);
            }
          } else {
            ec_slave[slave].islost = FALSE;
            printf("MESSAGE : slave %d found\n", slave);
          }
        }
      }
```

```
        if (!ec_group[currentgroup].docheckstate)
58        printf("OK : all slaves resumed OPERATIONAL.\n");
      }
60    usleep(10000);
    }
62 }

64 int main(int argc, char *argv[]) {
```

Listing 3.2: Methode ecatcheck für Linux/Xenomai

Achtung: Das Abfangen der Fehler obliegt in der Verantwortung des Programmierers! Das Beispiel aus *simple_test.c* macht nichts weiter, als den Status festzustellen und ihn auf der Konsole auszugeben. Die Übersicht der definierten Fehler kann in den Structs der *ethercattype.h* nachgeschaut werden. Besonders sind hier *ec_err* und *ec_state* hervorzuheben.

Mit freundlichen Grüßen und happy coding.