



University of Applied Sciences –
Fachhochschule Darmstadt,
Fachbereich Informatik

Studiengang Master
Vertiefungsrichtung Technische Systeme



Wintersemester:

2003/2004

Projekt:

InCarMultimedia

Betreuung:

Prof. T.Horsch und Prof. J.Wietzke
M. Pester und M. Müller

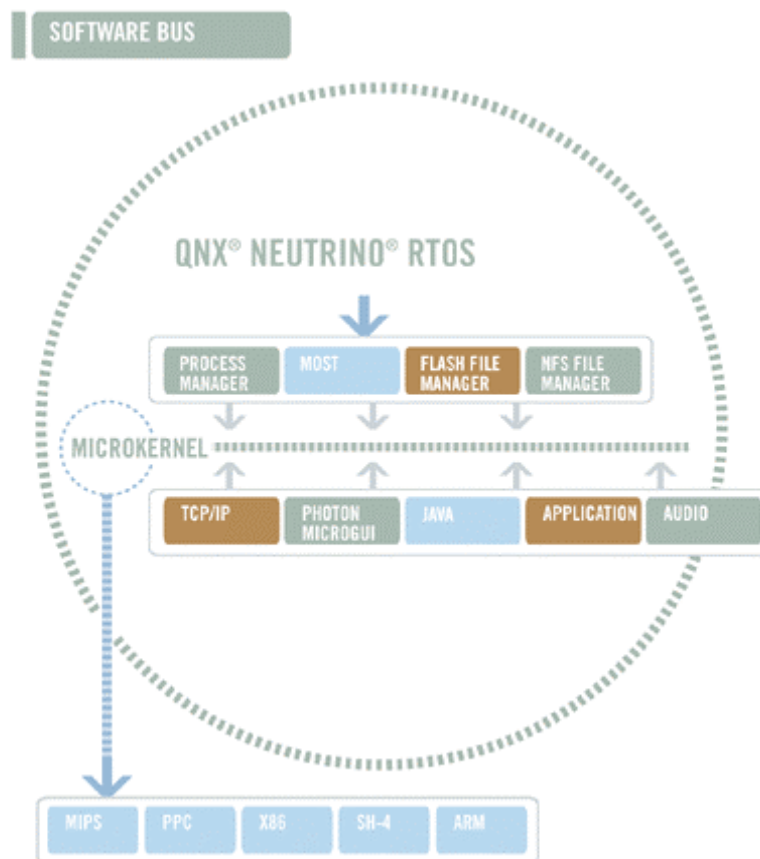
Inhaltsverzeichnis

1. QNX	3
2. Benutzerverwaltung	4
3. Shared Memory	5
4. Prozesse starten.....	7
5. Einführung in den DDD.....	8
6. Debuggen von Prozessen	12
7. MiniCommander	14
8. Commander-Simulator unter Windows	15
9. Embedded System	16
10. MOST	18
11. CAN.....	23
12. Momentics-IDE	24
13. Telnet- und FTP-Zugang	27
14. Serielle Console	28
15. Startscripte	29
16. Automatischer Applikationsstart.....	30
17. Präsentationsapplikation WS03/04 für Hobit.....	31

1. QNX

QNX ist ein von QSL entwickeltes Betriebssystem für Embedded Systeme und wird im ICM-Projekt auf der Zielplattform (SH4) als Betriebssystem sowie auf dem PC als Entwicklungs-plattform verwendet.

QNX ist ein Microkernelbetriebssystem. Darunter versteht man eine Architektur, bei der der Kernel nur die minimal zum Betrieb benötigten Dienste implementiert. Die restlichen Dienste des Betriebssystems werden als separate Userspace-Prozesse implementiert. Die folgende Grafik zeigt eine Auswahl an Diensten wie Most, TCP/IP, Java, usw., die ausserhalb des Kernels liegen. Sie kommunizieren via "synchronous message passing" mit dem Microkernel.



Aufgrund dieser Technologie ist das Betriebssystem sehr modular und stabil.

Modularität ist gerade für Embedded Systeme wichtig, da es hier oft notwendig ist, ein Betriebssystem auf eine neue Plattform zu portieren. Ausserdem existieren bei Embedded Systemen oftmals starke Restriktionen im Bezug auf Speicherplatz. Ein modular aufgebautes Betriebssystem, das optimal an die Einsatzumgebung angepasst werden kann, ist daher in diesem Umfeld bestens geeignet.

Die Stabilität wird dadurch unterstützt, dass die einzelnen Dienste des Betriebssystems in getrennten Userspace-Adressräumen laufen. Gerade die Stabilität ist für ein Betriebssystem im Automotive-Umfeld von grosser Bedeutung.

Weitere Informationen zu QNX können auf der Webseite "<http://www.qnx.org>" in Anspruch genommen werden.

2. Benutzerverwaltung

Die Benutzerverwaltung muss als root durchgeführt werden. Dazu sind im Einzelnen die folgenden Schritte notwendig:

Benutzer hinzufügen:

- * `./etc/group` editieren (Gruppe hinzufügen):
"`<Gruppenname>x:<GruppenID>:`"
Beispiel: "ICM:x:100:" (ohne die Anführungszeichen)
- * GNUbash sollte installiert sein
- * `./passwd <Benutzerlogin>`
User id # (101)
Group id # (100)
Real name () `<Benutzername>`
Home directory (`/home/<Benutzerlogin>`)
Login shell (`/bin/sh`) `/usr/bin/bash`
New password: `<Passwort>`
Retype new password: `<Passwort>`

Benutzer entfernen:

- * alle cron jobs des Benutzers stoppen und entfernen
- * Eintrag in `/etc/passwd` entfernen
- * Eintrag in `/etc/shadow` entfernen
- * Eintrag in `/etc/osshadow` entfernen
- * Eintrag in `/etc/opasswd` entfernen
- * Homeverzeichnis unter `/home/<Benutzerlogin>` löschen
- * mail pool Datei unter `/var/spool/mail` löschen

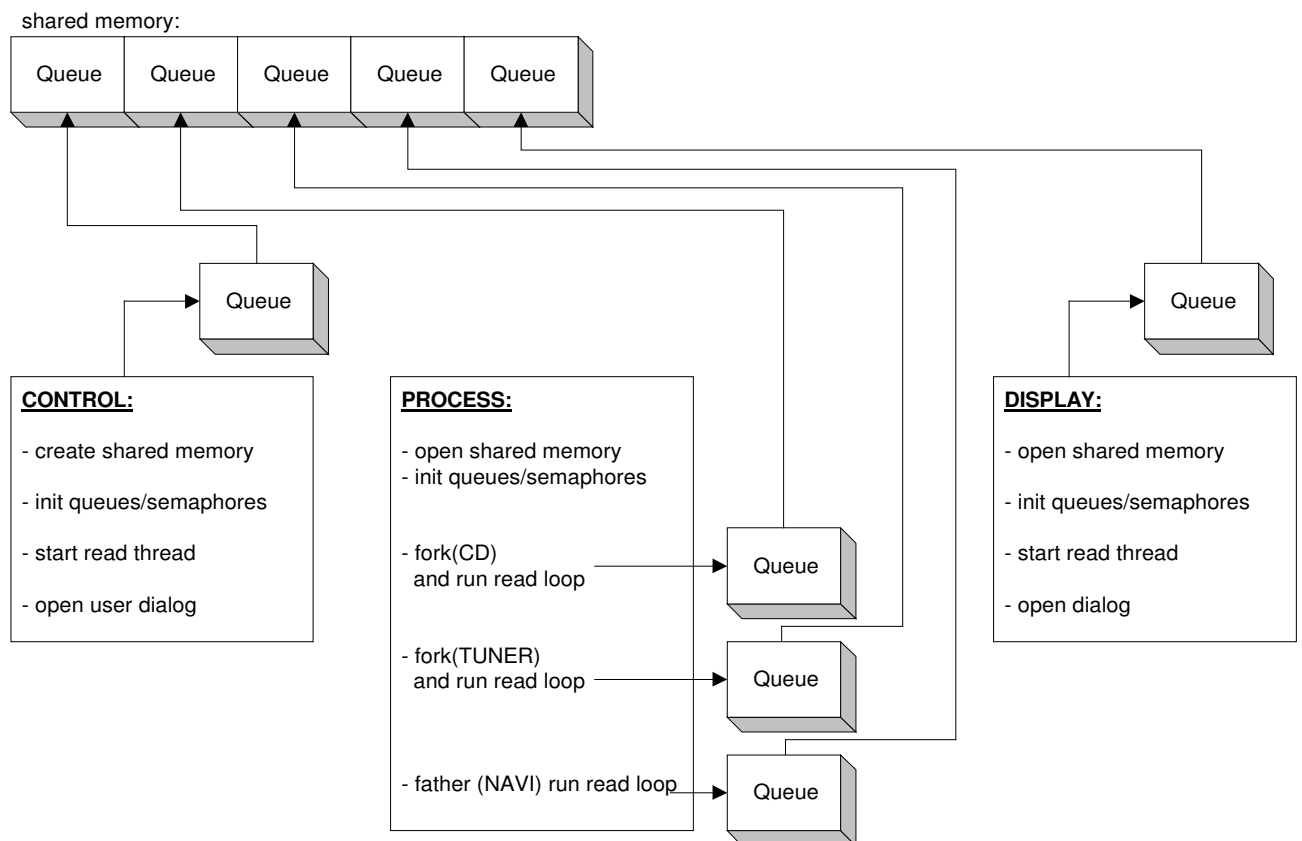
3. Shared Memory

In Embedded Systemen ist die Software oft modular (siehe Kapitel 1 QNX) aufgebaut.

Die Modularisierung kann sich unter anderem dadurch äussern, dass in dem System mehrere Prozesse implementiert sind. Die Kommunikation zwischen Prozessen muss über die Adressgrenze des Prozesses hinaus erfolgen. Somit wird ein Mechanismus benötigt, der über Prozessgrenzen hinaus kommunizieren kann.

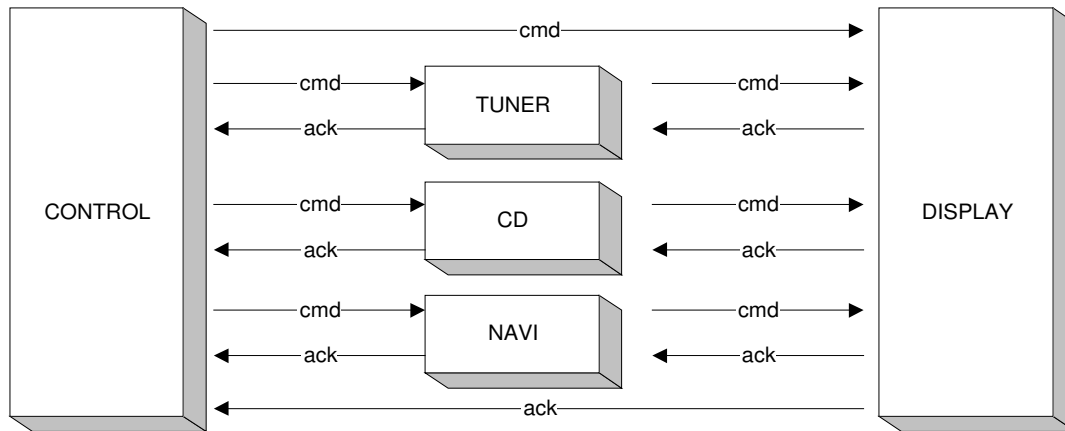
Eine Möglichkeit ist durch den Einsatz von geteiltem Speicher (shared memory) gegeben. Der Speicherbereich wird durch das Betriebssystem verwaltet und ermöglicht so eine Prozesskommunikation.

Im Projekt ICM wurden zu dieser Thematik mehrere Prozesse sowie der shared memory unter QNX implementiert. Die nachstehende Grafik zeigt die Zusammenhänge zwischen den Prozessen und dem shared memory.

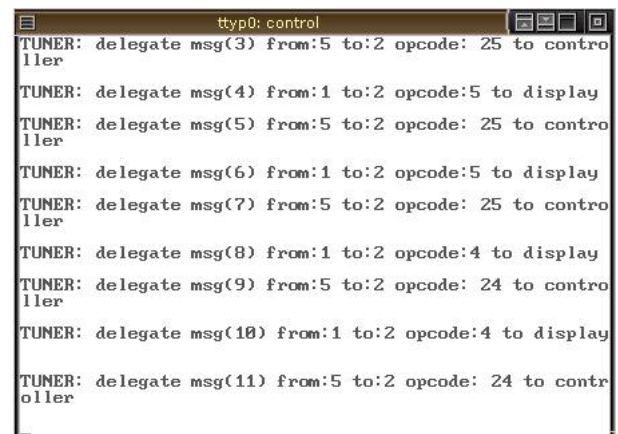
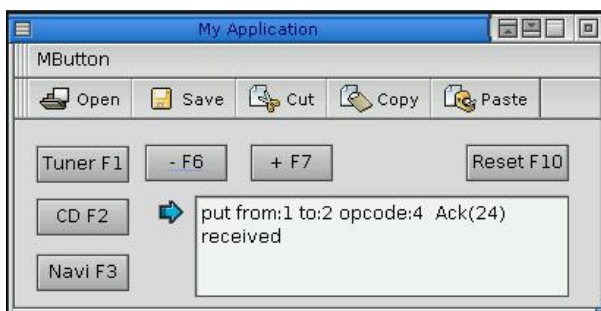
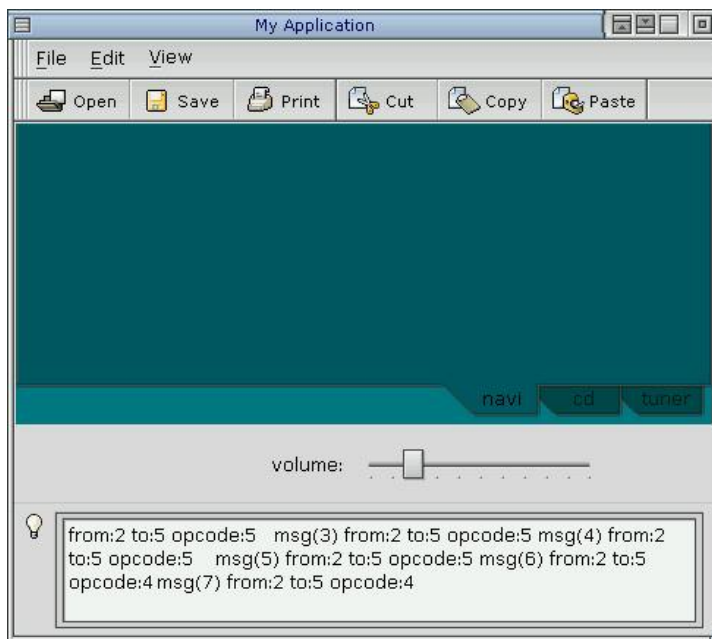


Jeder Prozess besitzt seine eigene Warteschlange (Queue) zur Aufnahme von Nachrichten. Eine Queue ist ein Teilbereich des shared memory. Die Kommunikation der Nachrichten muss z.B. durch Semaphore geschützt bzw. synchronisiert werden. Im einfachsten Fall kennt jeder Prozess die Queues aller anderen Prozesse und kann direkt eine Nachricht in die Queue schreiben. Eleganter ist die Implementierung eines Dispatchers, der für die Zustellung der Nachrichten in die entsprechende(n) Queue(s) sorgt.

Für den Nachrichtenverkehr wurden alle sinnvollen Kombinationen zwischen den Prozessen implementiert und auf dem Entwicklungs-PC unter QNX untersucht. Es ergibt sich der nachstehende Kommunikationsfluss:



Die Entwicklung erfolgte unter Photon. In den nachfolgenden Abbildungen sind die verschiedenen Fenster der Prozesse abgedruckt. Die Sourcen sind im ICM-CVS abgelegt.

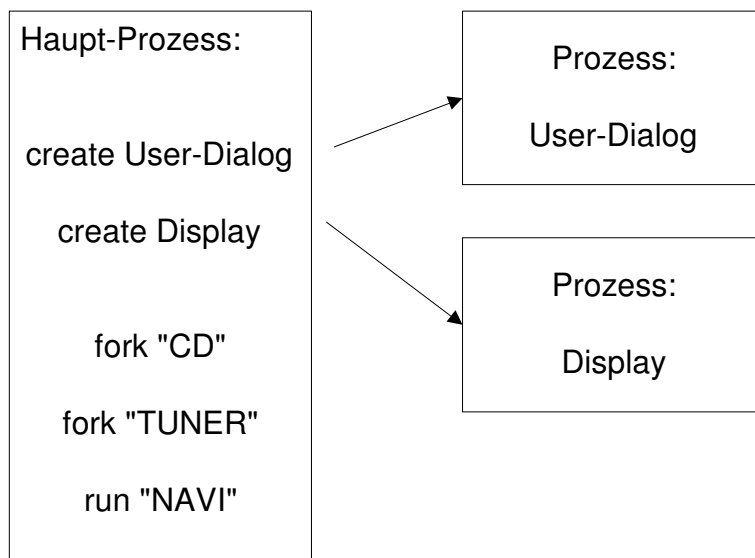


4. Prozesse starten

Die Prozesse für die Kommunikation über den shared memory wurden in der ersten Variante als eigenständige Binaries implementiert, die gegen einen gesharten Sourcecode gelinkt wurden, so dass alle Prozesse die Queues kennen. Eleganter ist unter unixbasierten Betriebssystemen das forken, da hier der Initialisierungsaufwand erheblich gesenkt werden kann.

Um nicht jeden Prozess einzeln zu starten, ist es zweckmässig, diese Aufgabe einem „Master“-Prozess zuzuteilen. Hierbei ist der Prozesskontext zu berücksichtigen und nicht jede Betriebssystemfunktion kann hierfür genutzt werden. Die Funktion `exec()` ist beispielsweise nicht für die genannte Aufgabe nutzbar, so dass `spawnl` genutzt wurde.

mehrere Prozesse starten (spawnl)



```
#define BINARY_LOCATION "."
```

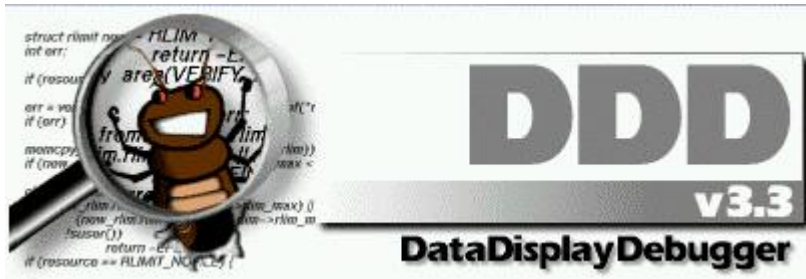
```
int ctrl_pid = spawnl( P_NOWAIT, BINARY_LOCATION"/Control", "Control", NULL );
```

```
if( ctrl_pid == -1 ) {
    cout << "error launching Photon-Control-Process! errorNo:" << errno << endl;
}
```

```
int displ_pid = spawnl( P_NOWAIT, BINARY_LOCATION"/Display", "Display", NULL );
```

```
if( displ_pid == -1 ) {
    cout << "error launching Photon-Display-Process! errorNo:" << errno << endl;
}
```

5. Einführung in den DDD



Data Display Debugger

Vorteile:

- als übersetzte Applikation für QNX und Linux verfügbar
- ausgereifte grafische Oberfläche
- unabhängig von Momentics / IDE
- GDB-Crossdebugger einbindbar (in Verbindung mit GDB-Server)
- direkter Zugriff auf GDB-Kommandos möglich

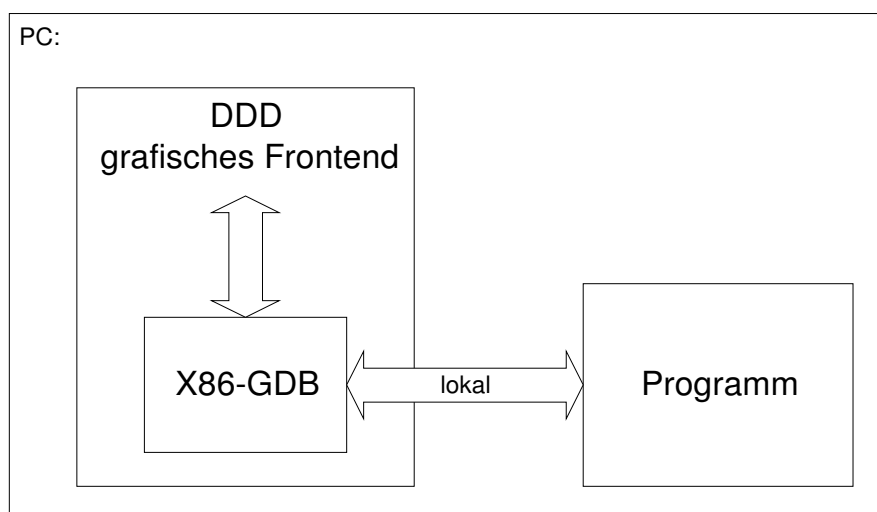
Nachteile:

- nicht in einer IDE (zB. Momentics) integriert

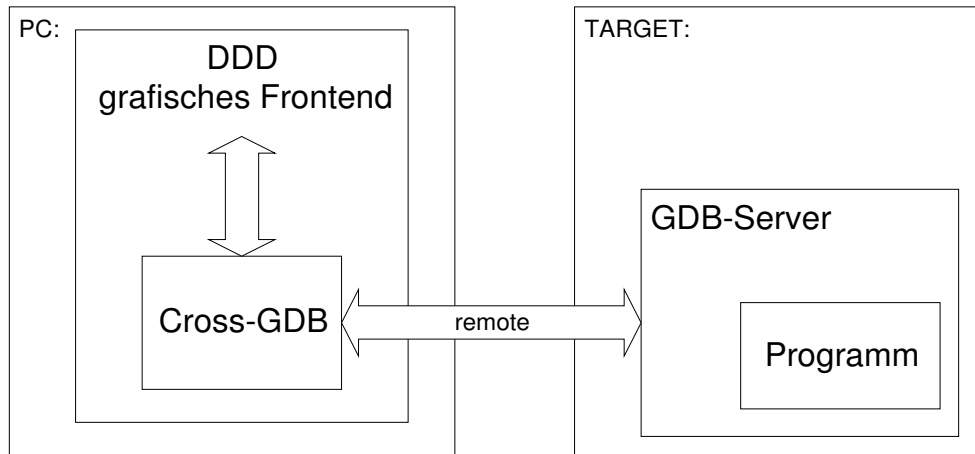
Vergleich zu „Momentics-Debugger“ liegt nicht vor

Für das Debuggen in Embedded Systemen kann man drei Arten unterscheiden:

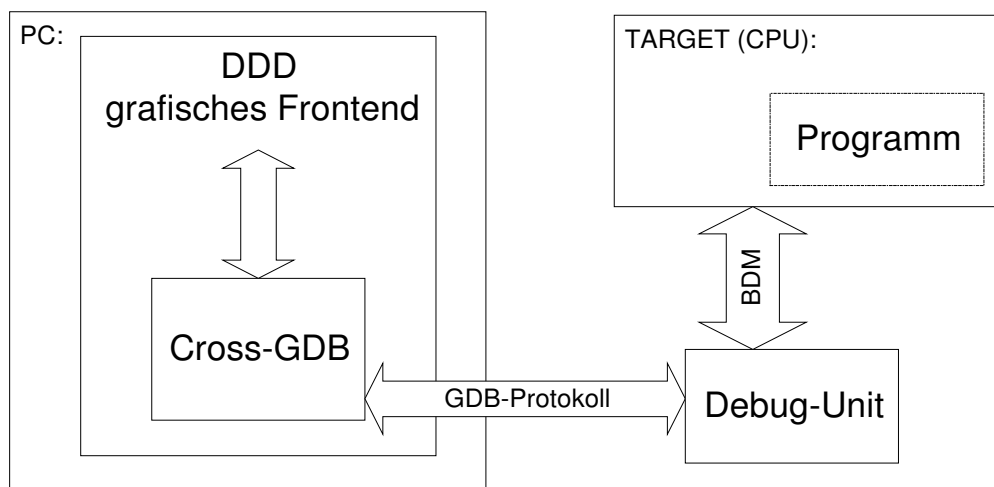
LOKAL DEBUGGEN



REMOTE DEBUGGEN (GDBSERVER)



HARDWARE-DEBUGGING (BDM)

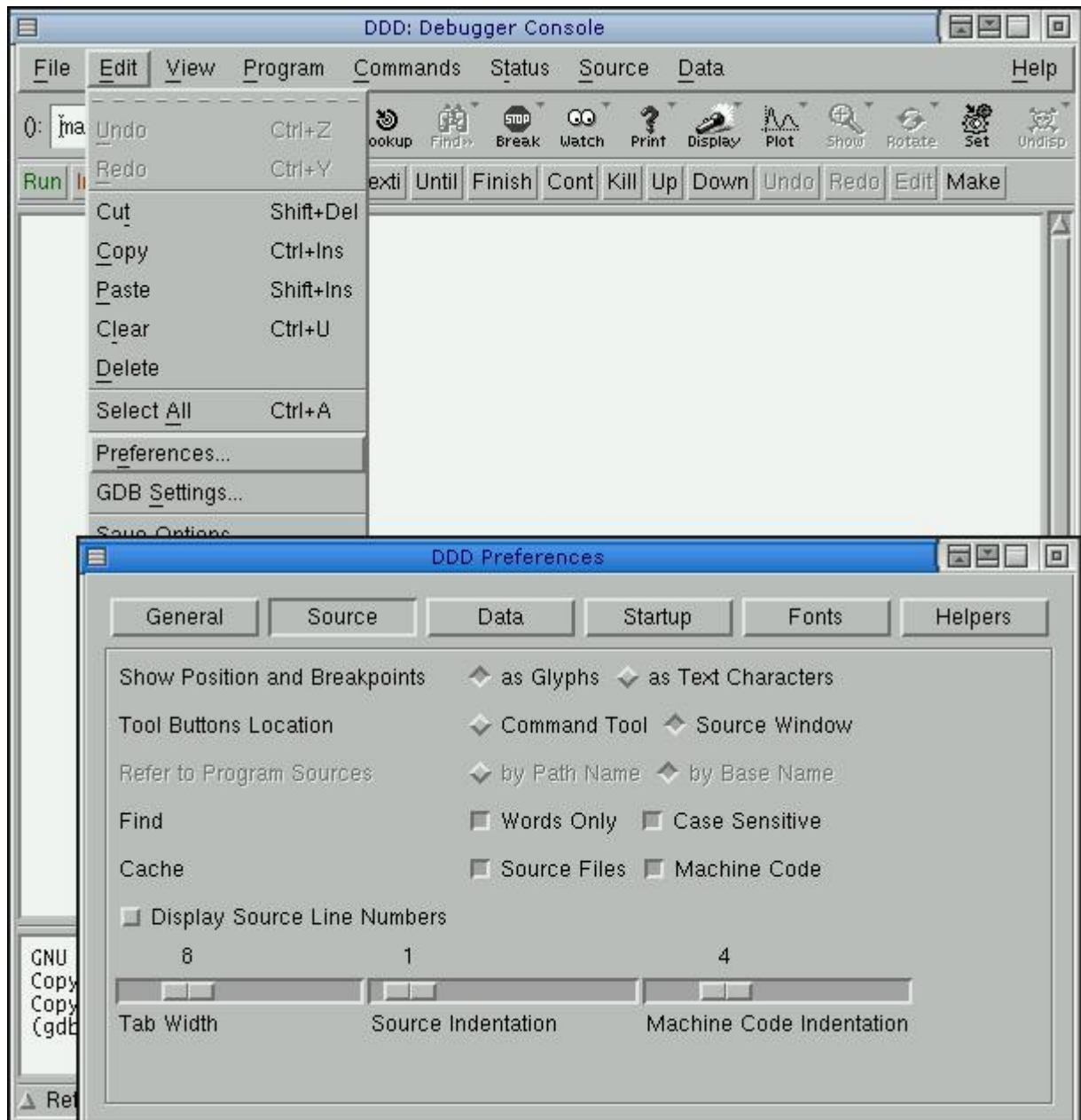


Für das Debuggen eines Targets (Embedded System) sind für den DDD nur die Debuginformation nötig. Es kann somit auf dem Target das als release übersetzte Binary geladen werden.

Da im Release sämtliche Debuginformationen fehlen, ist dieses Binary deutlich kleiner. Alternativ können die Debuginformationen mit Entwicklungstools gestrippt werden. Dadurch kann das Übersetzen von Release entfallen. Diese Variante ist dann interessant, wenn das Compilieren sehr zeitintensiv ist.

Auf dem X86-System kann der DDD direkt als Package installiert werden und arbeitet mit dem X86-GDB zusammen. Für das Debuggen mit einem Crosscompiler ist der DDD anzupassen.

Eine sinnvolle Einstellung im DDD ist das automatische Variablen-Popup (Source Indentation). Hierdurch kann dem DDD mitgeteilt werden, dass Variablenwerte angezeigt werden sollen, wenn der Mauscursor über einer Variable im Quelltext steht.



Beim Debuggen sind folgende Schritte nötig:

- Programm laden
- Breakpoint setzen
- RUN
- CONTINUE
- Stoppt bei Breakpoint
- NEXT

```

}

int main()
{
    int errno;
    int fd = shm_open(MAIN_SH_MEM, O_RDWR | O_CREAT | O_EXCL, FILE_MODE);
    int success = true;
    int count=0;
    char *ptr=0;

    if (fd == -1)
    {
        if (errno != EEXIST)
        {
            cout<<"can not open shared mem\n" << endl;
            success = false;
        }
        else
        {
            fd = shm_open(MAIN_SH_MEM, O_RDWR, FILE_MODE);
            if (fd == -1)
            {
                cout<<"can not open shared mem\n"<<endl;
                success = false;
            }
        }
    }
}

```

```

(gdb) file /root/icm/bin/control
(gdb) break main
Breakpoint 1 at 0x8048e62: file control.cpp, line 84.
(gdb) run
(gdb) cont

Breakpoint 1, main () at control.cpp:84
(gdb)

```

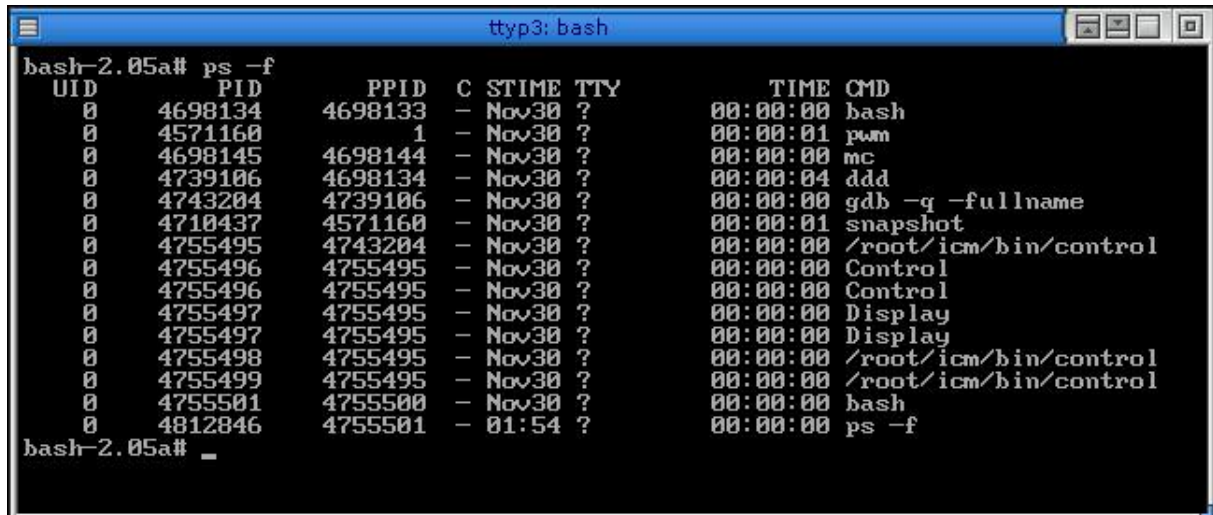
▲ Breakpoint 1, main () at control.cpp:84

Die Grafik zeigt die Oberfläche des DDD. Im unteren Fenster kann man direkt mit dem GDB agieren und oberhalb dieses Fensters sind die grafischen Elemente zur Steuerung des Debuggens.

6. Debuggen von Prozessen

Beim Debuggen mehrerer Prozesse ist Kenntnis über die PID & PPID notwendig, um die Prozesse anhand dieser Informationen zu attachen.

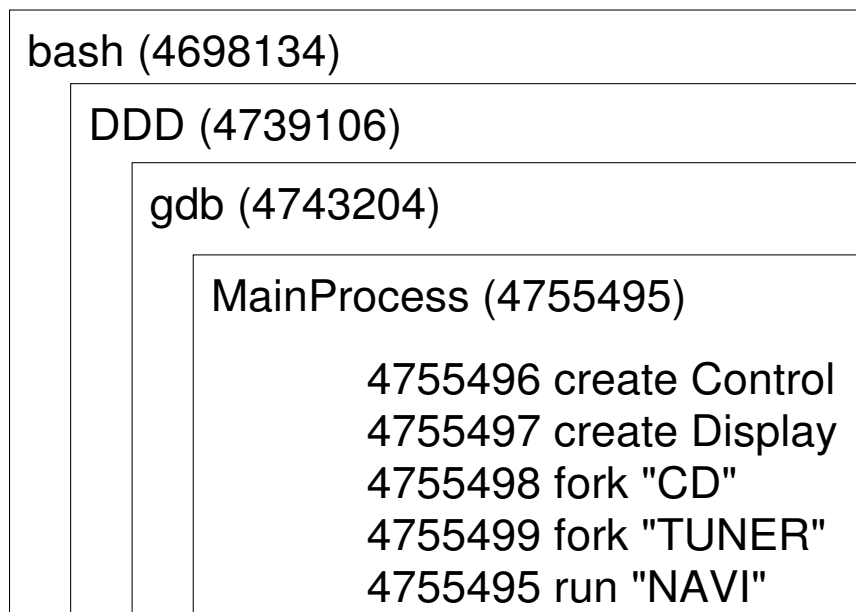
Um die Informationen zu erhalten, kann der Befehl „ps“ genutzt werden.



```

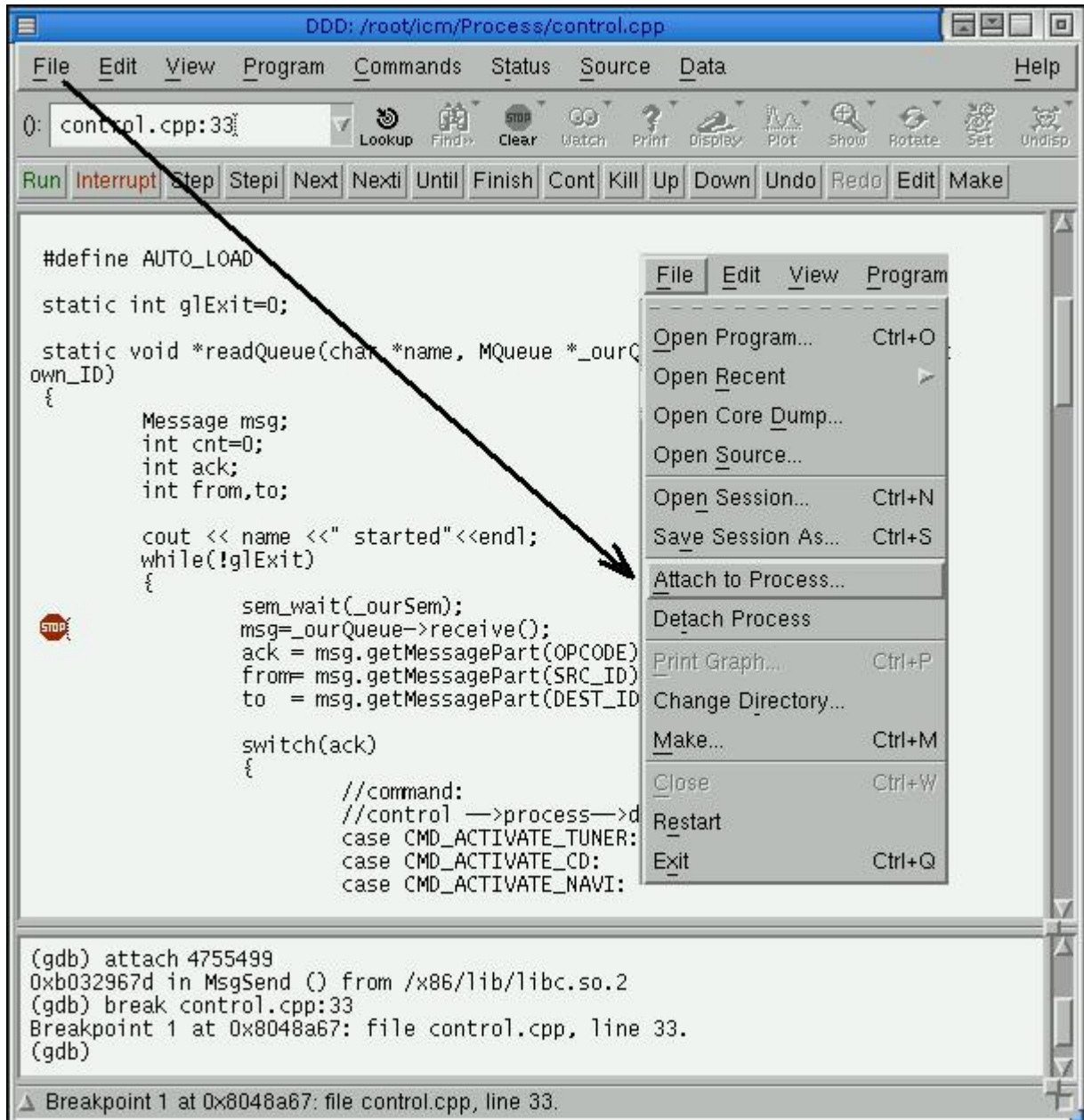
bash-2.05a# ps -f
  UID      PID      PPID  C  STIME TTY      TIME  CMD
    0     4698134  4698133  -  Nov30 ?        00:00:00 bash
    0     4571160      1  -  Nov30 ?        00:00:01 pwm
    0     4698145  4698144  -  Nov30 ?        00:00:00 mc
    0     4739106  4698134  -  Nov30 ?        00:00:04 ddd
    0     4743204  4739106  -  Nov30 ?        00:00:00 gdb -q -fullname
    0     4710437  4571160  -  Nov30 ?        00:00:01 snapshot
    0     4755495  4743204  -  Nov30 ?        00:00:00 /root/icm/bin/control
    0     4755496  4755495  -  Nov30 ?        00:00:00 Control
    0     4755496  4755495  -  Nov30 ?        00:00:00 Control
    0     4755497  4755495  -  Nov30 ?        00:00:00 Display
    0     4755497  4755495  -  Nov30 ?        00:00:00 Display
    0     4755498  4755495  -  Nov30 ?        00:00:00 /root/icm/bin/control
    0     4755499  4755495  -  Nov30 ?        00:00:00 /root/icm/bin/control
    0     4755501  4755500  -  Nov30 ?        00:00:00 bash
    0     4812846  4755501  -  01:54 ?        00:00:00 ps -f
bash-2.05a# _
  
```

Mit der Ausgabe des Befehles „ps“ kann die nachstehende Abhängigkeit zwischen den Prozessen abgelesen werden.



Wenn die benötigte ID des zu debuggenden Prozesses vorliegt, dann ist

- das Programm zu laden (mit Debug-Infos)
- attach auf den Prozess (ID)
- setzen des Breakpoints



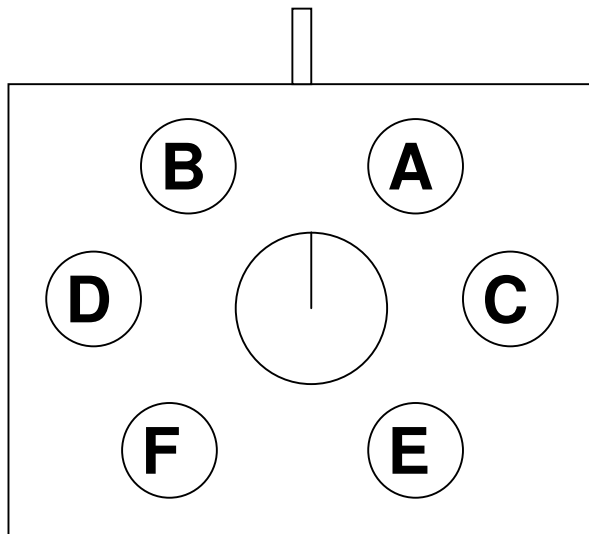
Beim Debuggen der Prozesskommunikation über den shared memory ist es sinnvoll, den Breakpoint wie im obigen Bild hinter eine Blockade (z.B. Semaphore) zu setzen.

Wird eine Nachricht an den Prozess geschickt, so „feuert“ der breakpoint und man kann an der gewünschten Stelle schrittweise den Programmverlauf verfolgen.

7. MiniCommander

In Embedded Systemen sind die Ressourcen des Systemes begrenzt zB. I/O-Leitungen. Dadurch ist die Eingabe meist auf die serielle Schnittstelle oder wenige Tasten eingeschränkt. In dem Embedded System des ICM-Projektes wird daher ein Eingabemodul genutzt, welches als MiniCommander bezeichnet wird und an die serielle Schnittstelle (57600Baud) angeschlossen wird.

Das Eingabemodul besitzt 6 Tasten und einen Drehgeber der auch als Taste genutzt werden kann. Da in dem Modul ein eigener Microcontroller ist, benötigt dieser eine Versorgungsspannung von 12V. Die Aufteilung der Tasten und des Drehgebers ist im nachstehenden Bild zu sehen.



A+B schwarze Taster
D+C rote Taster
F+E grüne Taster

Drehgeber

Die Tasten sind in einigen Bytes kodiert und es ergeben sich die nachstehenden Werte:

	drücken	loslassen
Taste F:	0BF055208E	0BF05500AE
Taste D:	0BF05508A6	0BF05500AE
Taste B:	0BF05502AC	0BF05500AE
Taste E:	0BF05510BE	0BF05500AE
Taste C:	0BF05504AA	0BF05500AE
Taste A:	0BF05501AF	0BF05500AE
Taste Drehgeber:	0BF05540EE	0BF05500AE

Drehgeber right step
0BF0990163

Drehgeber left step
0BF0AA0150

Beispiel Hexdump:

	0B		F0		55		10		BE		
0x02	0x30	0x42	0x46	0x30	0x35	0x35	0x31	0x30	0x42	0x45	0x0D
STX	length		source		cmd		data		Chsum		ETX

chsum=0xBE=0x0B xor 0xF0 xor 0x55 xor 0x10

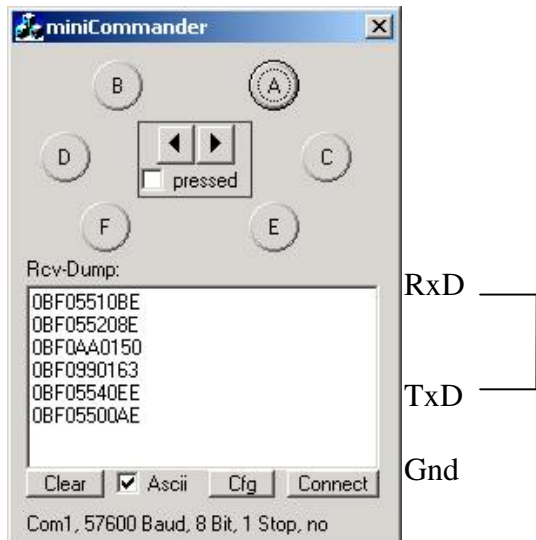
STX und ETX dienen als Start- und Endekennung eines Kommandos.

--> Der Status Taste/Drehgeber bleibt gedrückt wird nicht mit übertragen,
somit muss der Auswerter die States merken !!!

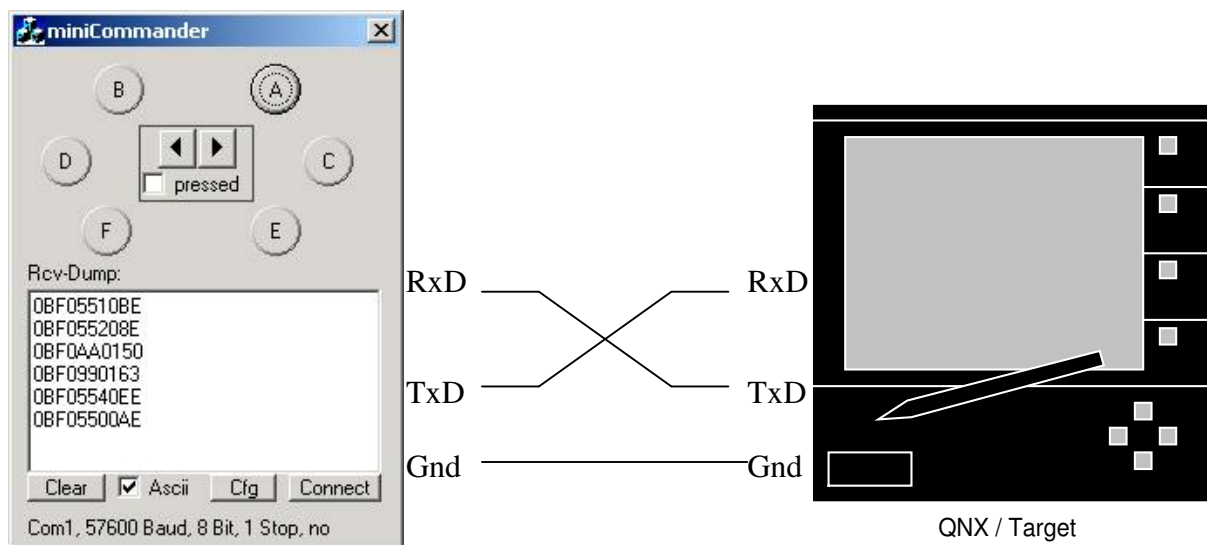
8. Commander-Simulator unter Windows

Für das im letzten Kapitel vorgestellte Hardwaremodul wurde unter Windows (MFC) ein Simulator geschrieben, der als Ersatz genutzt werden kann. Auf Tastendruck werden die Kommandos per serieller Schnittstelle versendet (wie die des Hardwaremodules).

Anwendungsbeispiele:



Zum Test mit einem Simulator wird TxD mit RxD verbunden, so dass die eigenen Kommandos wieder empfangen werden. Dies kann man am PC leicht realisieren, in dem man z.B mit einer Jumperbrücke Pin2 und Pin3 des 9-poligen seriellen Anschlusses brückt.



Wenn der Simulator im QNX-Umfeld eingesetzt werden soll, so benötigt man ein serielles Kreuzkabel (Null-Modem) sowie einen Windows-PC für den Simulator selbst. Der Windows-PC wird per Kreuzkabel mit dem Target oder QNX-PC verbunden.

Alle Quellen für den MiniCommander/Simulator sind im ICM-CVS abgelegt.

9. Embedded System

Das Embedded System im ICM-Projekt besitzt folgende Features:

- QNX als Betriebssystem
- Schnittstellen: Seriell, MOST, CAN und PCMCIA
- ATAPI-Schnittstelle mit Festplatte und DVD-Laufwerk
- Farbdisplay (320x96 Pixel)
- PCI-Erweiterung mit LAN-Karte (RJ45)
- Tuner und Verstärker über MOST
- GPS für Navigation
- Hitachi SH4 CPU sowie 32MB Flash und 32 MB RAM

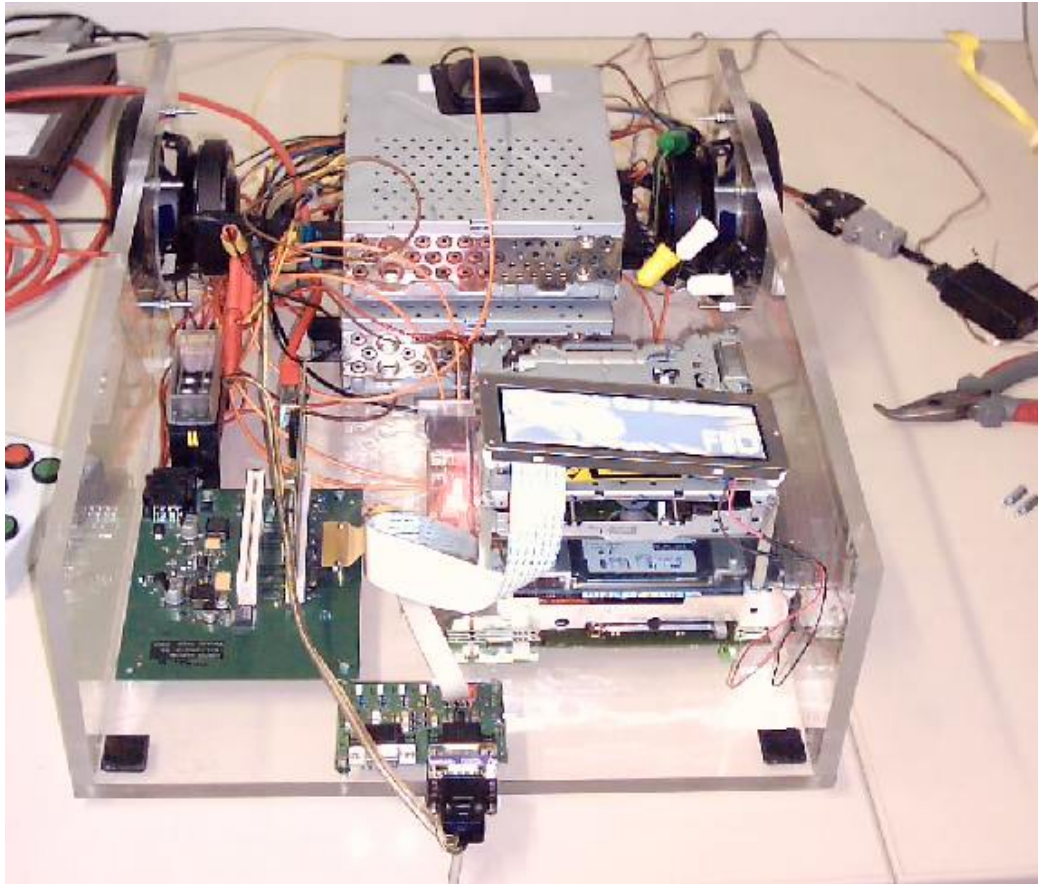
In der nachstehenden Grafik ist das System im „Prototype“-Zustand abgebildet. Die braune Box ist der Optolyzer, der zum Tracen von MOST-nachrichten eingesetzt wird. In der Metallbox, die sich darunter befindet, ist das Gateway, der Tuner sowie Verstärker enthalten. Die PCI-Erweiterung inklusive LAN-Karte ist im vorderem Bereich des Bildes zu sehen. Das Display und die seriellen Schnittstellen sind rechts im Bild.

Das gesamte System wird mit 12V versorgt und hat etwa eine Stromaufnahme von 4 Ampere. Abhängig von der Firmware des Gateways ist eine Aktivierung über die CAN-Schnittstelle (Zündungssignal) notwendig. Die restlichen Komponenten, Festplatte und Hauptplatine sind unter dem DVD-Laufwerk.

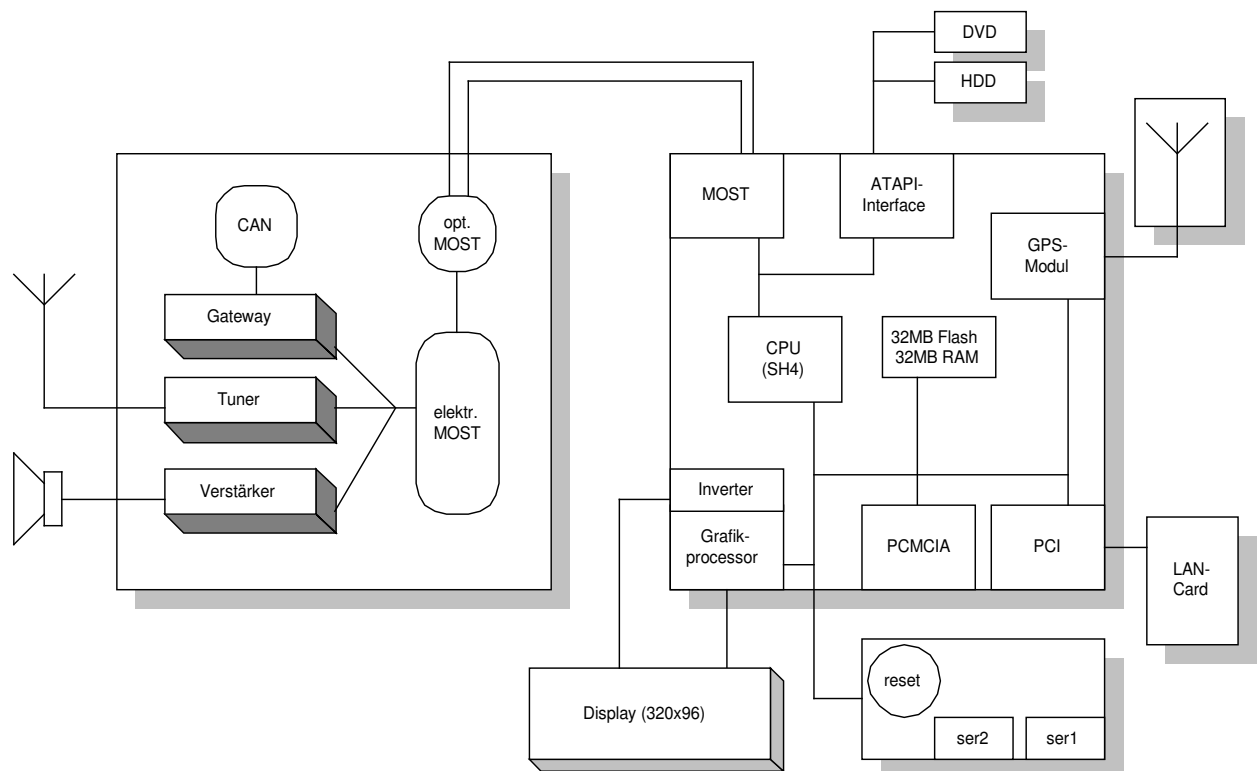


Auf der nächsten Seite ist eine schematische Darstellung des Gesamtsystemes abgedruckt. In der Schemadarstellung kann man gut die beiden Hauptkomponenten GatewayBox incl. Tuner und Verstärker sowie die Hauptplatine (Headunit) erkennen.

Hardwareaufbau im Plexiglasgehäuse



Interner Aufbau der Hardwareplattform:



10. MOST

MOST (Media Oriented Systems Transport) ist ein Bus-System für Fahrzeuge. MOST wurde ursprünglich von OASIS entworfen. Heute wird MOST durch die MOST-Cooperation weiterentwickelt.

MOST stellt einen highspeed, multimedia Bus bereit. Über diesen Bus können angeschlossene Geräte beliebig miteinander kommunizieren (peer to peer). Das heisst zum Beispiel, dass ein über MOST angeschlossenes Telefon, die Sprachwiedergabe über die ebenfalls per MOST angeschlossenen Lautsprecher der Stereoanlage durchführen kann.

MOST verwendet als physikalisches Übertragungsmedium einen glasfaserartigen Ring. Dafür wird „Plastic Optical Fiber“ (POF) verwendet, da der Bus im Auto auch einigen physischen Belastungen standhalten muss und ausserdem leicht sein muss.

MOST erlaubt das einfache Anschliessen neuer Geräte durch Plug&Play. Geräte identifizieren sich beim Anschliessen selbst und werden automatisch initialisiert. An einen Bus können bis zu 64 Geräte angeschlossen werden.

MOST unterstützt Datenübertragungen bis 25Mbps. Zukünftige Versionen sind mit bis zu 150 Mbps geplant.

MOST ist für den Transport folgender Traffic-Arten ausgelegt:

- Realtime Video/Audio und Telefongespräche, zum Beispiel das Abspielen von Cds/DVs.
- Übertragung von Dateien. Dies wird zum Beispiel benötigt, wenn die Navigationsanwendung ihre Kartendaten von einer CD im CD-Player lädt.
- Übertragung von Kontrollanweisungen. Über den MOST-Bus ist es auch möglich angeschlossene MOST-Geräte zu steuern (Bsp. Start der Wiedergabe von Track 1 auf der CD im CD-Player).

Die Hauptkriterien bei der Entwicklung von MOST waren es eine zugleich billige als auch sehr stabile Lösung zu finden. Beides Kriterien die typisch sind für den Einsatz in Embedded Systemen.

MOST soll die bisherige Verkabelung in Fahrzeugen ersetzen. Gegenüber der traditionellen Verkabelung hat MOST folgende Vorteile:

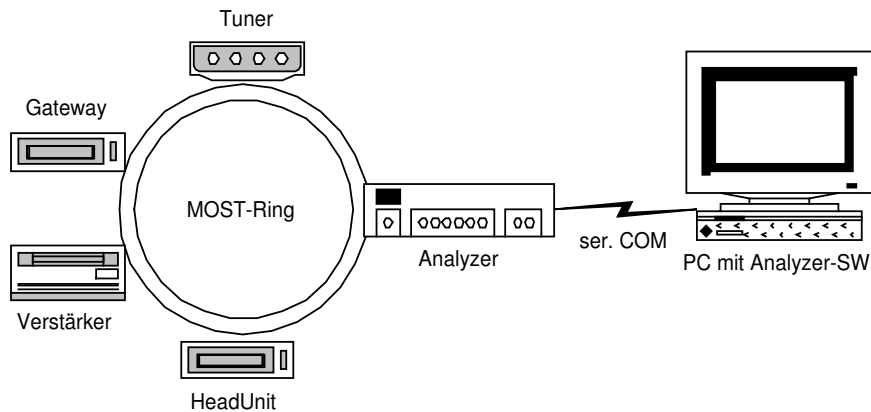
- MOST hat eine bessere Performance als andere Bus-Systeme da durch die Verwendung von Glasfaser als Übertragungsmedium eine deutlich grössere Bandbreite zur Verfügung steht.
- MOST ist Platz sparer. Es wird für alle Geräte nur ein Kabel benötigt, statt einem grossen Kabelbaum.
- Durch die Einsparung an Kabeln und Komponenten ist MOST auch billiger als traditionelle Ansätze.
- MOST hat bessere elektrische Eigenschaften. Die Signale auf dem BUS werden nicht durch andere elektrische Systeme beeinflusst.

Weitere Informationen zu MOST:

- Auf der Website der Firma OASIS findet man alle Standards zum Thema MOST. <http://www.oasis.com>
- Einen guten Überblick über das MOST-System bietet das „MOST Specification Framework“. http://www.oasis.com/support/downloads/mosttechnology/MOSTSpecification_Framework_1_V1.pdf
- Skript zur Vorlesung In-Car Multimedia Systeme von Prof. J.Wietzke

Most-Analyzer installieren und in Betrieb nehmen

allgemeiner Aufbau:



Wichtig:

Der MOST-Kommunikationsring muss geschlossen sein und der Analyzer entsprechend konfiguriert sein, je nach dem, ob er selbst Nachrichten versenden darf oder nur empfangen soll.

Nachfolgend die Schritte zur Installation und Inbetriebnahme des Analyzers.

1) Installation von CD

Folgende Software ist von der CD-ROM zu installieren:

- Optolyzer4Most Professional
- MostRadar

Auf dem Desktop sollte dieses Zeichen:



erscheinen.

2) Software freischalten

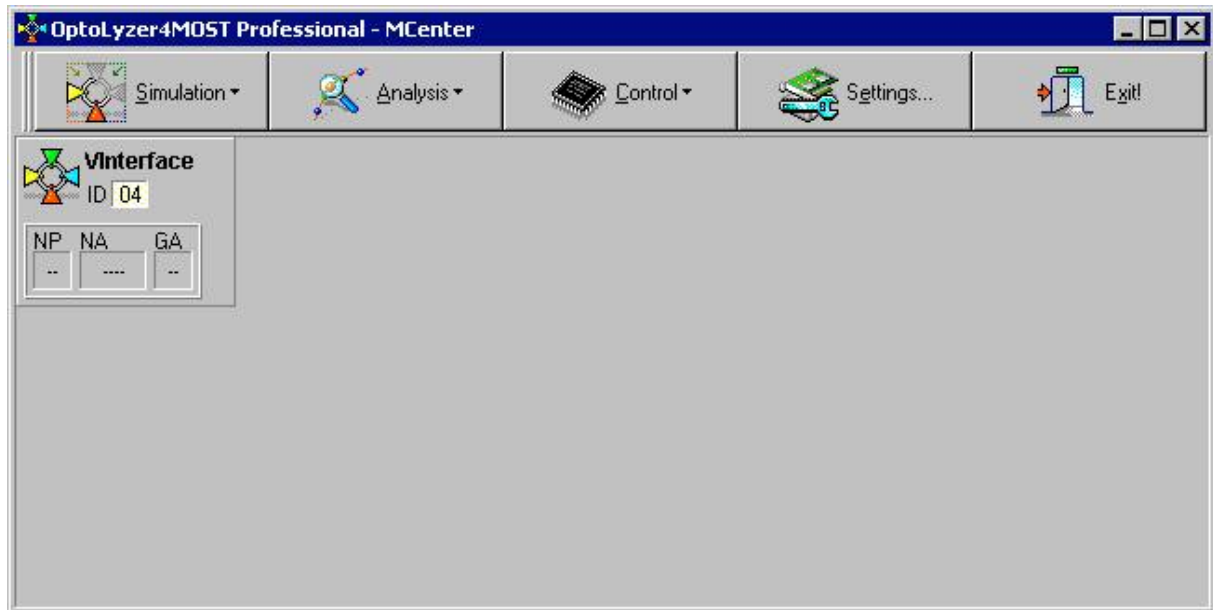
Es ist folgender Key in dem geöffneten Optolyzer4Most einzugeben

43D8-8858-0B3D-BFF7

OPL4V42438

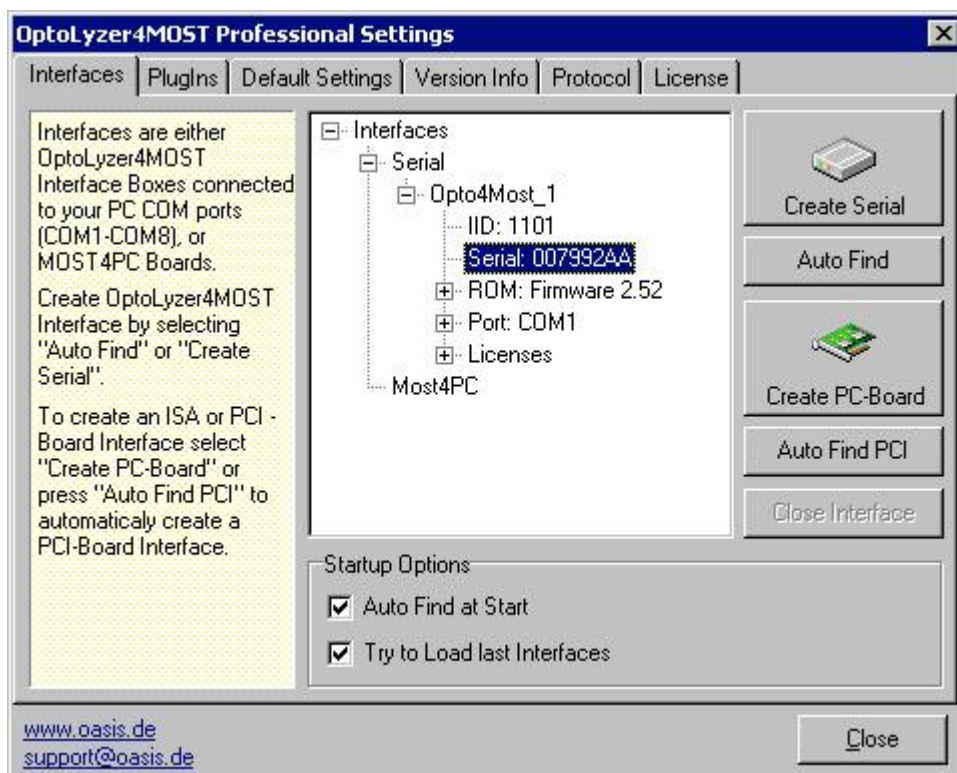
3) Analyzer Suchen

Menü->Settings

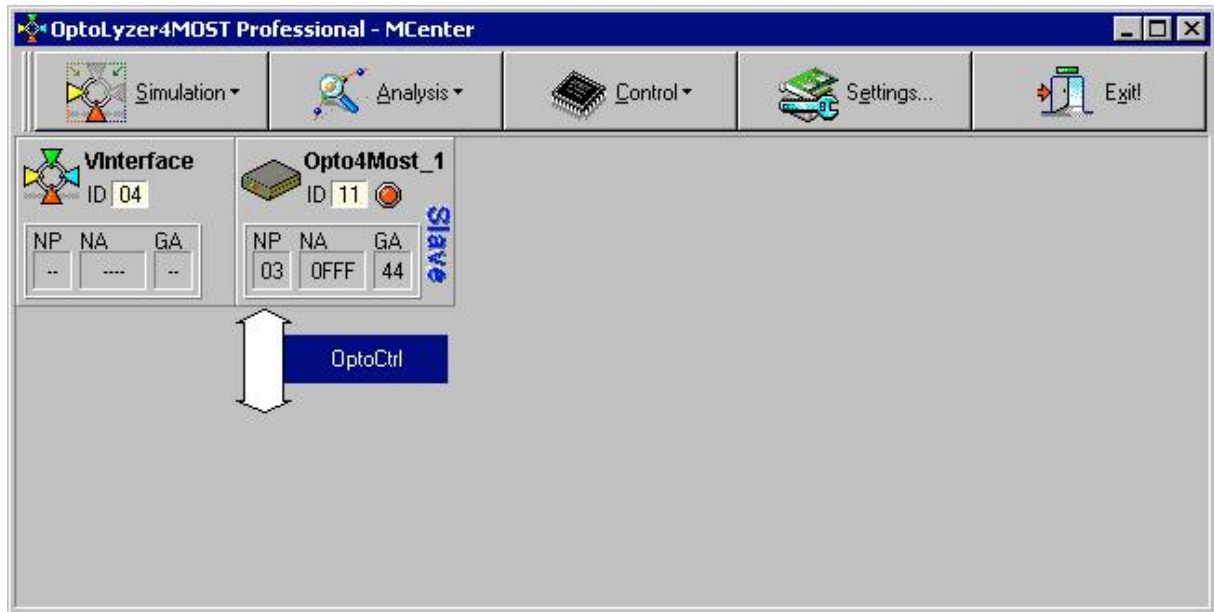


(bild1)

„Auto Find“ starten (findet Optolyzer am seriellen Port)



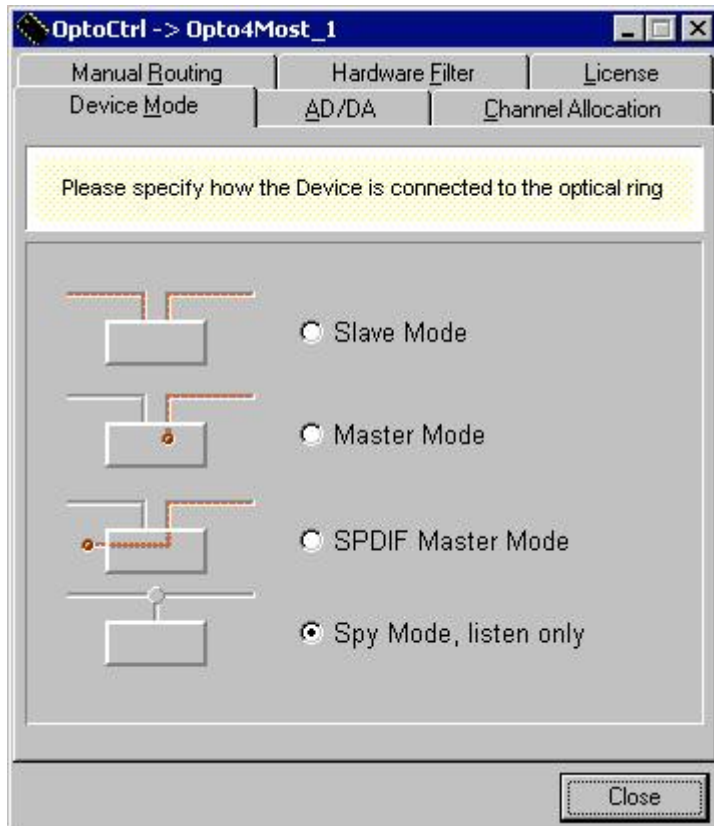
(bild2)



(bild3)

Der Analyzer wurde gefunden

- 4) Modus einstellen
Menü->Control->DeviceMode

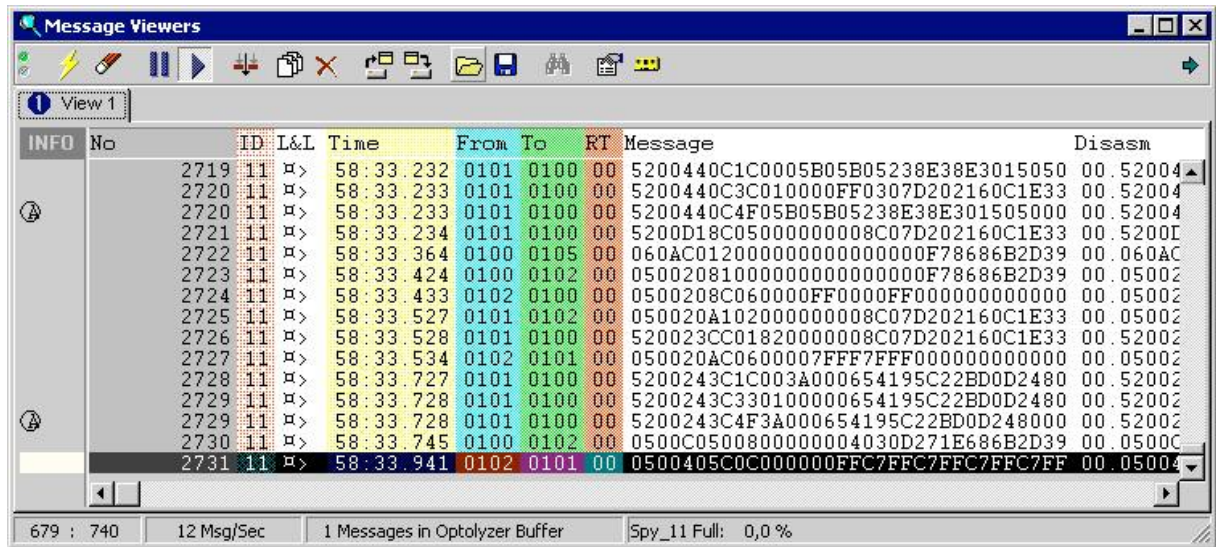


(bild4)

Im einfachsten Fall ist der Analyzer auf SPY-Mode zu stellen. In diesem Mode hat der Analyzer keine ID und ist ein passiver Teilnehmer im MOST, der nur die Nachrichten empfängt.

5) Nachrichten empfangen

Menü->Analyse->Viewer

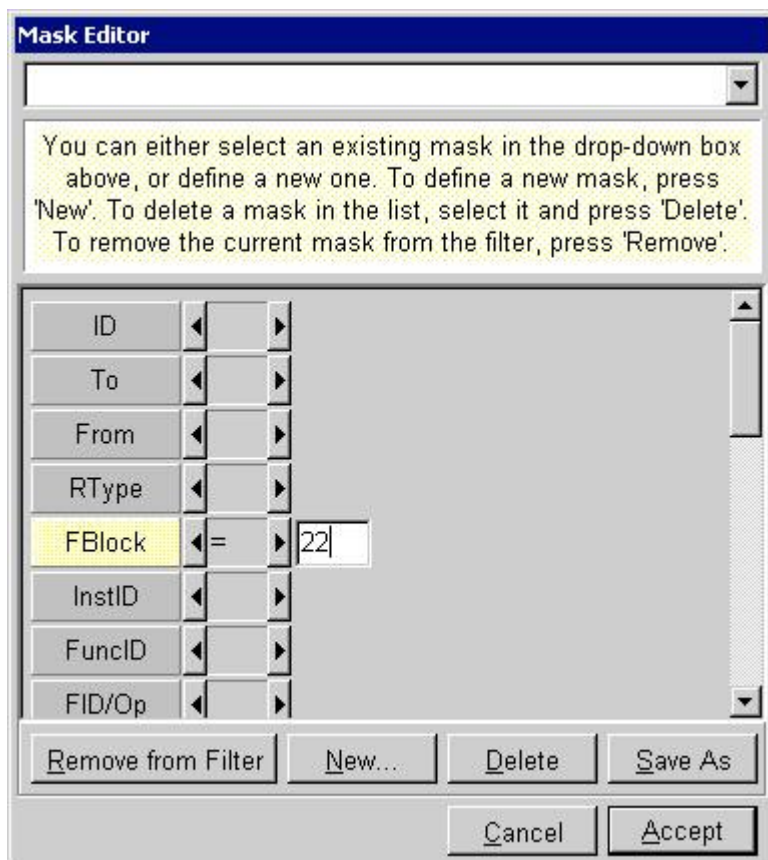


(bild5)

6) Filter setzen

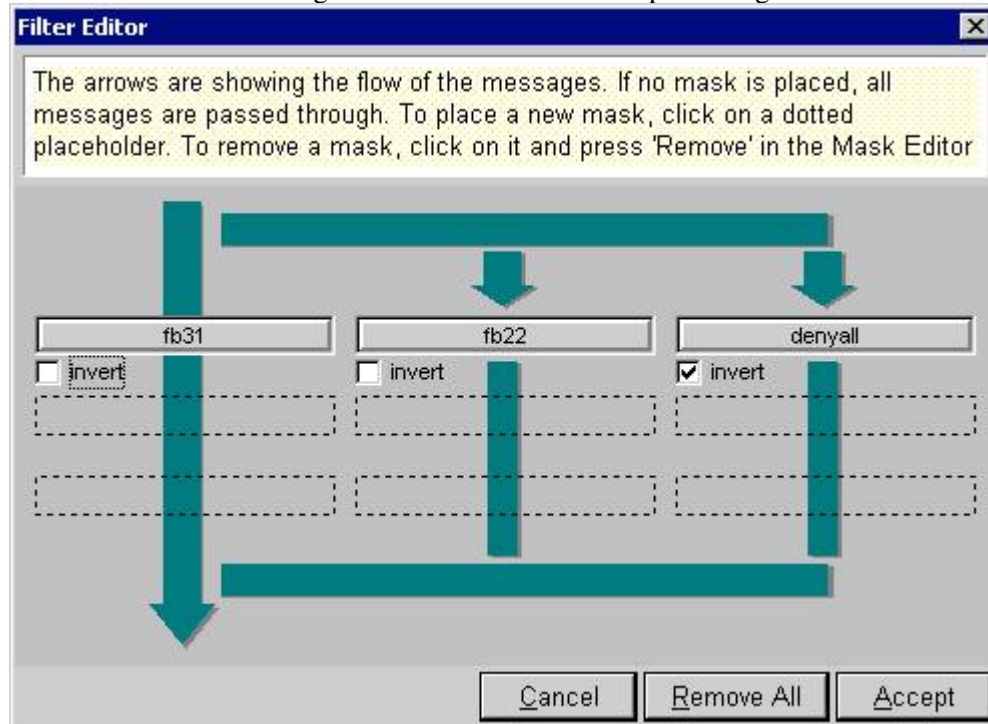


Dieses Symbol klicken, um Filter zu setzen.



(bild6)

Jeden Filter einzeln eintragen und mit SaveAs & Accept bestätigen



(bild7)

Im 3. Filter „denyall“ sind die Defaulteinstellung belassen wurden. Es wurde also nur „SaveAS“ und „Accept“ für diesen Filter ausgeführt.

Der 3.Filter ist als „invert“ zu aktivieren, um nur die eingestellten Filter (fb31 & fb22) anzuwenden.

11. CAN

Das Controller Area Network (CAN) verbindet mehrere gleichberechtigte Komponenten (Knoten, Node) über einen 2-Draht Bus miteinander. Das CAN-Protokoll wurde 1983 von Bosch für den Einsatz in Kraftfahrzeugen entwickelt.

Weitere Information sind unter <http://www.me-systeme.de/canbus.html> zu finden.

Wie schon erwähnt, benötigt das Gateway je nach Firmware zum hochfahren eine Nachricht (Zündung) auf dem CAN.

Diese Nachricht lautet für Zündung an:

ID

0x000 04 03 00 00 00

Diese Nachricht muss ca. alle 100ms kommen.

Für Zündung aus:

ID

0x000 04 00 00 00 00.



Zur Simulation dieser Nachricht kann in dem Embedded System eine CAN-BOX oder ein Modul mit einem Microcontroller genutzt werden, der den o.g. Nachrichtenverkehr erzeugt.

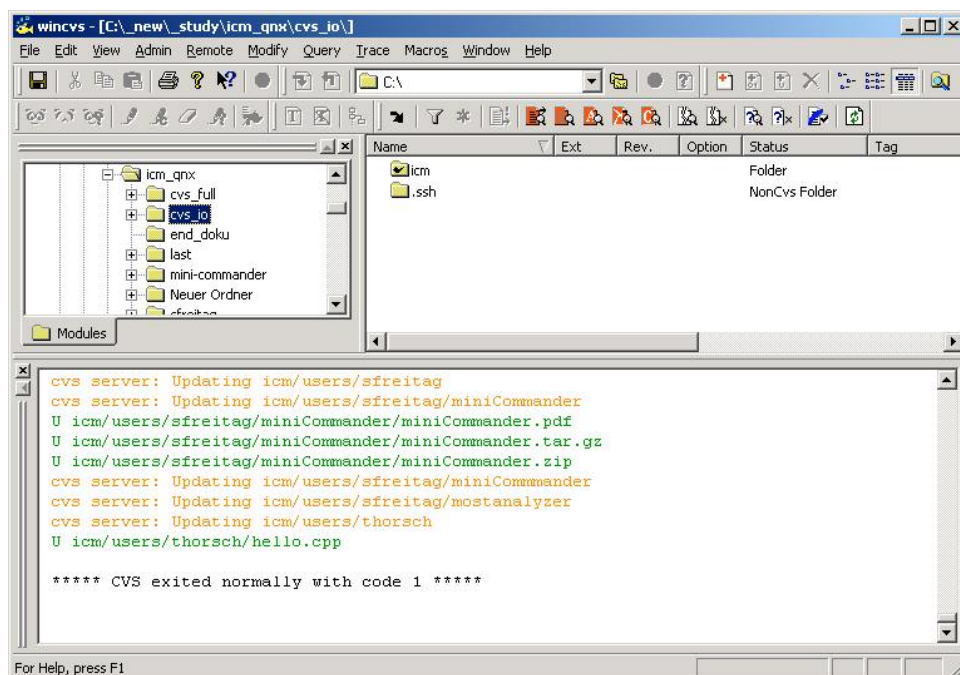
Das Modul ist im obigen Bild dargestellt sowie in der Abbildung „Hardwareaufbau im Plexiglasgehäuse“. Bei CAN ist noch zu beachten, dass es zwei abweichende PIN-belegungen für PEAK (Pin 2+7) und für PON (Pin 8 + 7) für den 9poligen Anschluss gibt.

12. Momentics-IDE

Die Softwareentwicklung in ICM erfolgt unter QNX in der IDE Momentics. Das nachstehende Bild zeigt die Version, die eingesetzt wird:

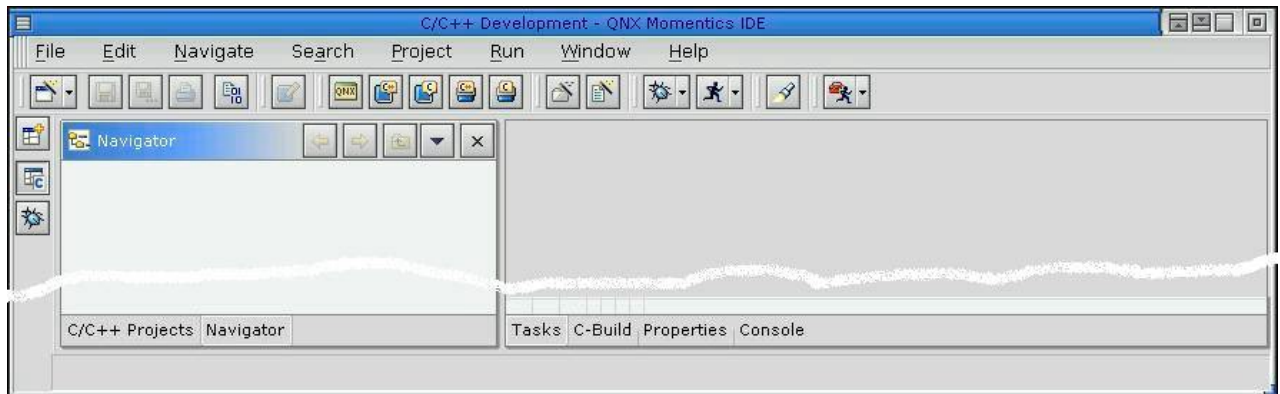


Das zu bearbeitende Projekt liegt auf dem ICM-CVS und kann über Momentics ausgecheckt werden, da ein CVS Zugang enthalten ist. Für den CVS-Zugang unter Windows ist eine ausführliche Dokumentation von Karsten Klein für den Zugang mit WINCVS über SSH erstellt wurden (CVSROOT: `username@www.kosi.fh-darmstadt.de:/home/cvs`).

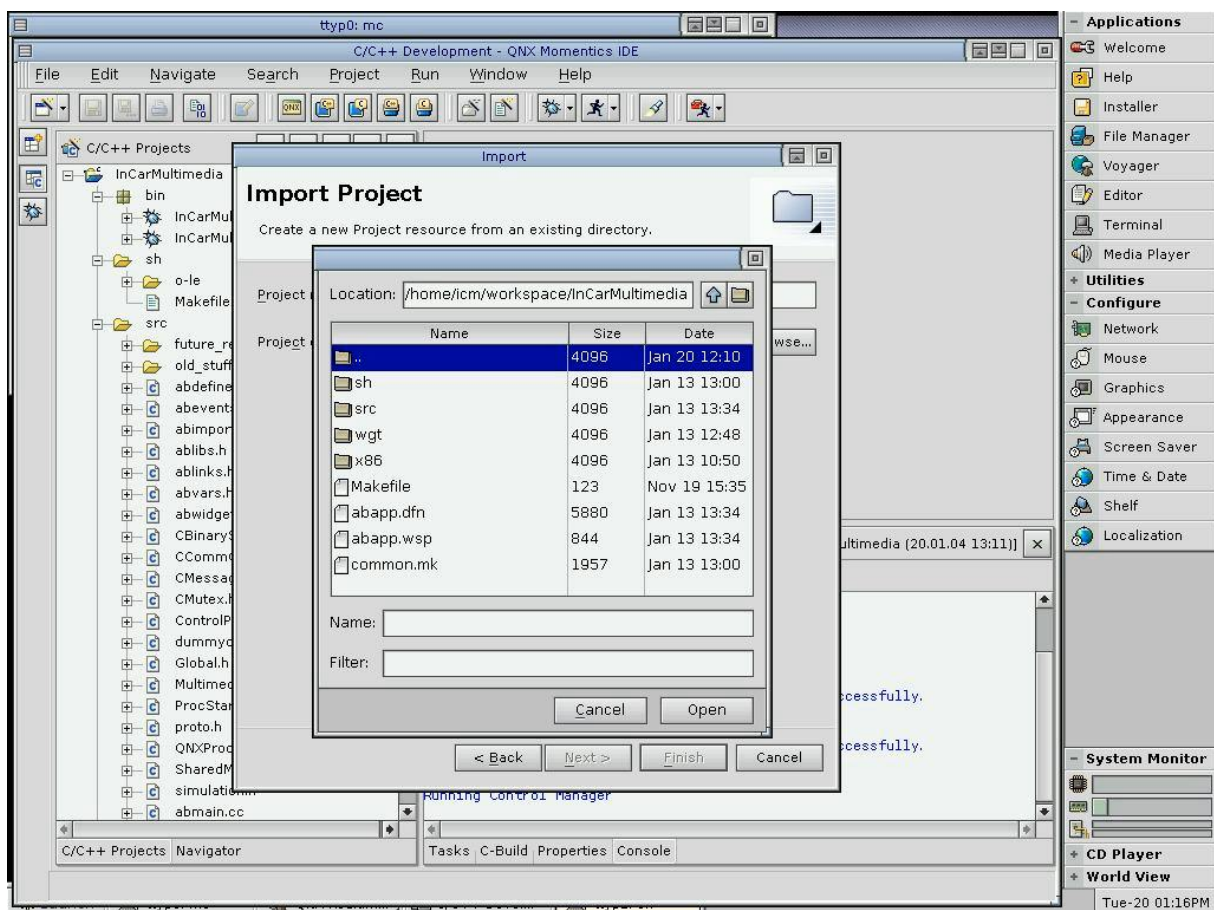


In der obigen Abbildung ist WINCVS (1.3.14.1 Beta) abgebildet und es wurde der gesamte Inhalt aus dem CVS gelesen.

Sind die Sourcen bzw. das aktuelle Projekt vom CVS geladen, so kann das Projekt in der Momentics-IDE geladen werden. Dazu sind ggf. geladene Projekte im Navigator zu entladen.



Das Projekt wird geladen über „File->Import...“ und als Quelle ist „Existing Project into Workspace“ zu wählen. Anschliessend öffnet sich der Dialog „Import Project“:

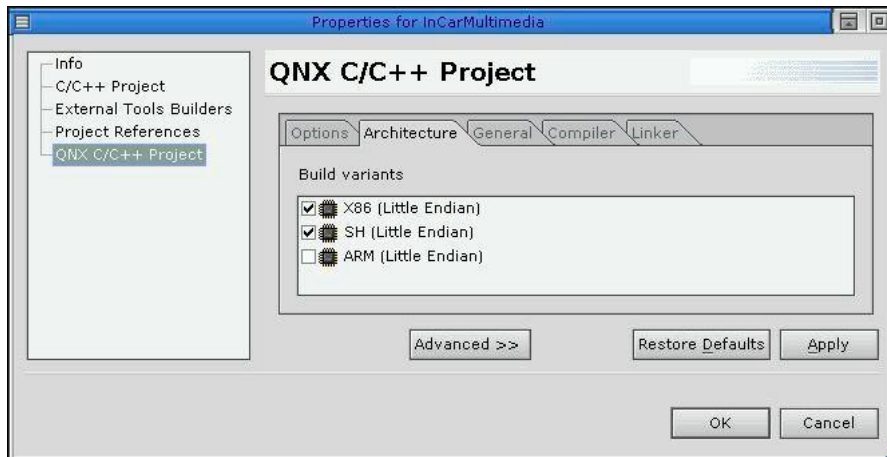


Mit dem „Browse..“-Button kann wie in der Grafik gezeigt das gewünschte Projekt geöffnet werden.

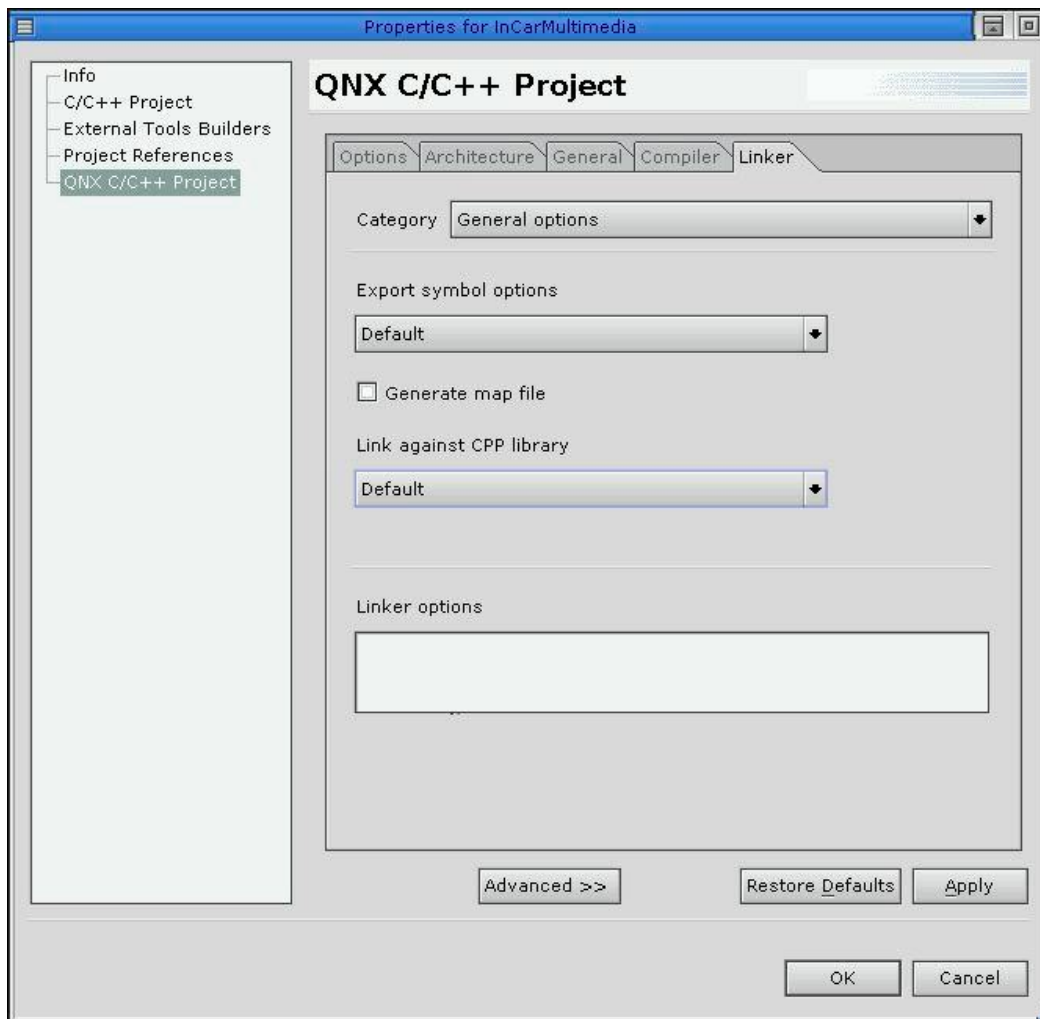
Im Dialog „Import Project“ werden die Felder (Project name, Project contents) gefüllt und dieser kann mit dem Button „Finish“ geschlossen werden. In der Hauptansicht der IDE sollte jetzt im Fenster „Navigator“ das gerade geladene Projekt erscheinen:



Zum Übersetzen der Projektquellen sollten einige Momentics-Einstellungen (Preferences) geprüft bzw. gesetzt werden. Zum einen ist die Coderzeugung für X86 (PC) und SH (Target) zu aktivieren.



Um die Codegröße für das Target zu verringern sollten zum anderen die Exceptions deaktiviert werden.



Der „Default“ sollte auf „Dinkum Abridged without exceptions“ gesetzt werden.

Zu beachten ist bei Projekten mit Photonwidgets, in denen Prozesse geforkt werden, das nach dem „Generate Sources“ noch manuell eingegriffen werden muss. Nach dem Generieren sind in der Funktion main() im File „abmain.cc“ alle Zeilen ausser init(), exit() und return() zu löschen. Die main() sollte dann so aussehen:

```
int main ( int argc, char *argv[] )
{
    init( argc, argv );
    PtExit( 0 );
    return 0;
}
```

Die erstellte Applikation kann jetzt auf dem Entwicklungs-PC gestartet werden. Hierzu ist einfach das X86-Binary zu starten. Dies kann von der Console oder über den „Run“-Button von Momentics aus geschehen. Im vorliegenden Beispiel soll die Applikation für die Hobit (InCarMultimedia) ausgeführt werden. Dazu wird in der Console „/home/icm/workspace/InCarMultimedia/x86/o/InCarMultimedia“ gestartet. Da bei dieser Applikation keine Eingabe implementiert ist, kann die Applikation nur „gekillt“ werden.

Dazu sind die nachfolgenden Schritte notwendig:

```
$ ps -e
1486884 ?    00:00:00 pterm
725029 ?    00:00:00 voyager
725030 ?    00:00:00 vserver
1556519 ?    00:00:00 ps
1548328 ?    00:00:00 /home/icm/workspace/InCarMultimedia/x86/o/InCarMul
1486889 ?    00:00:00 /bin/sh
1548330 ?    00:00:00 /home/icm/workspace/InCarMultimedia/x86/o/InCarMul
1552427 ?    00:00:00 pv
$ kill 1548328
$ kill 1548330
```

Die Applikation ist nun beendet.

13. Telnet- und FTP-Zugang

Ist die Applikation auf X86 weitreichend geprüft worden, so kann nun das Verhalten auf dem Target untersucht werden. Dazu wird ein Zugang zum System auf dem Target benötigt. Den Eingabeprompt (Console) erhält man über Telnet und der Datenaustausch ist mit ftp zu erledigen. Für beide Zugänge muss das Target mit einem PC über Netzwerk verbunden werden. Das Target hat die IP-Adresse 172.16.166.123 und die Subnetmaske 255.255.0.0 .

Dem PC ist eine entsprechende IP-Adresse für dieses Netzwerk zuzuweisen.

Unter Windows kann per ftp mit dem Explorer zugegriffen werden. Dazu ist im Explorer folgendes einzugeben:

<ftp://172.16.166.123>

```
user:      root
password:  root
```

In einer Windows-Console kann mit „telnet 172.16.166.123“ eine telnet-Verbindung hergestellt werden. Es gelten die gleichen Zugangsdaten wie unter ftp.

Mit „ping 172.16.166.123“ kann die Netzwerkverbindung zum Target geprüft werden, falls kein ftp oder telnet möglich ist, um zu prüfen, ob das Target im Netzwerk erreicht werden kann.

14. Serielle Console

Unter unixbasierten Systemen kann die Ein-/Ausgabe über die serielle Schnittstelle umgelenkt werden. Bei Embedded Systemen sollte dies „von Haus aus“ eingestellt sein, da hier keine geeigneten Anzeigeeinheiten (grosser Monitor) verfügbar sind. Das Embedded System des ICM-Projektes meldet sich ebenfalls mit der Console auf der ersten seriellen Schnittstelle.

Für den Zugang über die serielle Console wird ein PC mit Terminalprogramm (z.B. Minicom) sowie ein 1:1 Kabel benötigt, mit dem das Target und der PC verbunden werden. Die Baudrate des Terminals ist mit 57600 Baud einzustellen. Wenn das Target gebootet wird so meldet es sich mit folgenden Ausgaben:

```
System page at phys:0800a000 user:0800a000 kern:8800a000
Starting next program at v88037ad8
  57.735 ms IFS pon2-sa
  87.555 ms starting slogger
 146.214 ms sersci
wait4c ~  0 msec for /dev/ser1
# 298.153 ms starting PCI server
wait4c ~ 12 msec for /dev/pci
 423.066 ms Starting start2.sh
 508.211 ms powermgr
 570.302 ms Starting Photon
 628.545 ms io-gfx
wait4c ~ 66 msec for /dev/phfont
 878.225 ms screen1
4080.424 ms starting Flash driver
wait4c ~ 3400 msec for /dev/shmem/gonet
4234.986 ms starting network driver
4273.765 ms start2.sh done, now network stuff
6753.229 ms Starting EIDE
ifconfig en0 172.16.166.123
6892.374 ms starting devc-pty
6964.815 ms pipe
7023.773 ms qconn
7095.544 ms inetd
Path=0 - XILINX OSGI
target=0 lun=0      CD-ROM(5) - PHILIPS APM DVD-M2   Rev: 0203
target=1 lun=0     Direct-Access(0) - IC25N020ATMR04-0 Rev: OMO1
wait4c ~ 1000 msec for /dev/hd0t77
8226.474 ms mounting HDD as /
Starting /hddstart.sh in background
.....
```

Der Zugriff über die serielle Console ist vergleichbar mit dem Telnetzugang. Jedoch benötigt die serielle Console keine weiteren Dienste wie z.B. inetd oder telnetd. Wenn also der Zugriff über telnet oder ftp nicht möglich ist, kann man an der seriellen Schnittstelle im Regelfall noch auf das Target zugreifen.

Wichtiger HINWEIS:

Zu beachten ist, das der Minicommander auf der gleichen seriellen Schnittstelle arbeitet und beim automatischen Starten (siehe Kapitel „Automatischer Applikationsstart“) einer Anwendung, die den Minicommander einsetzt, dadurch der Zugang auf die serielle Schnittstelle gesperrt wird. Es ist also zu beachten, das man sich den Zugang zum Target nicht selbst abschneidet.

15. Startscripte

Um die eigene Applikation auf dem Target starten zu können, müssen auf dem Target noch einige Einstellung gesetzt werden, damit die Applikation korrekt ausgeführt werden kann. Dazu werden auf dem Target sogenannte Scripte genutzt, die das Target einstellen und dann die Applikation starten. Auf dem Target kann das Script /qnc/fh3.sh gestartet werden, um das System zu prüfen. In dieser Applikation können alle Komponenten über den MiniCommander angesprochen werden (zB. MOST, GPS, DVD-Laufwerk...). Für die eigenen Applikation kann ein einfacheres Startscript genutzt werden. Dieses Script „ICMDemoHobbit.sh“ ist nachstehend abgedruckt.

```
/hbbin/dev-most-mops.last -a110 -v -r -s -D 0x561 &

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/lib/hbllib:/proc/boot/l
export PATH=/sbin:/usr/bin:/usr/sbin:$PATH
#export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/lib:/usr/lib:/hbllib:/proc/boot/l
#export PATH=/sbin:/usr/bin:/usr/sbin:$PATH

#echot "serial IFACE"
#devc-sersci -c60000000/16 sci scif
slay -shup ser-sh
slay -shup ser-sh
#/hbbin/hb-mini-commander -d /dev/ser1 -t1 -j >/dev/null &
#/hbbin/hb-mini-commander -d /dev/ser1 -t1 -j &
stty -ihflow -ohflow par=even 38400 </dev/ser2

mqueue

cd /hbbin
export MMEDIA_OVERLAY=0
export MMEDIA_OFFSCREEN=0
ln -svP /pon/ /flash/pon

#./pon_mmi &

#set
#sleep 20

#for sensoric
/hbbin/hbs -D 0x561 &

io-audio -dmops-j
ln -sP /mnt/hd0/sprache/ /usr/local/sprache
/hbbin/spl >/dev/null &

echo "starting CTL"
cp /qnc/* /dev/shmem
cd /dev/shmem
# /hbbin/pon_ctl_navi >/dev/null &
# /hbbin/pon_ctl_naviprev &
#SFr /hbbin/pon_ctl_navi &
#added SFrk
/icm/ICMDemoHobbitV11

sleep 2
echot "start HDD"
echot mount
mount -r /dev/hd0t79 /mnt/hd0
mount /dev/cd0 /mnt/cd0

ln -sP /mnt/hd0/pon3/Birdview /hbbin/Birdview

echot "fstart.sh wait4b fully"
wait4c -p 100 /dev/shmem/fully 40
echot "NAVI is fully operable"
```

16. Automatischer Applikationsstart

Für die Präsentation des Embedded Systems auf der Hobbit ist es wünschenswert wenn das System selbständig hochfährt und die Applikation autonom startet. Um dieses Verhalten zu erreichen, ist ein spezielles Script auf dem Target zu erstellen. Der Name des Scriptes ist mit /hddstart.sh zu benennen. Zu beachten ist, das die Rechte für das Script korrekt gesetzt sind (executable für root). Befehle hierfür sind:

- ls -al
- chmod
- chgrp

Zum Anlegen des Scriptes kann der VI-Editor oder echo mit Umlenkung genutzt werden. Das Startscript wird beim booten des Systems ausgeführt. Zu diesem Zeitpunkt ist die Festplatte bereits gemounted.

Hier nun das Script, wie es für die Hobbit 2004 genutzt wurde:

```
cat /hddstart.sh  
/qnc/ICMDemoHobbit.sh
```

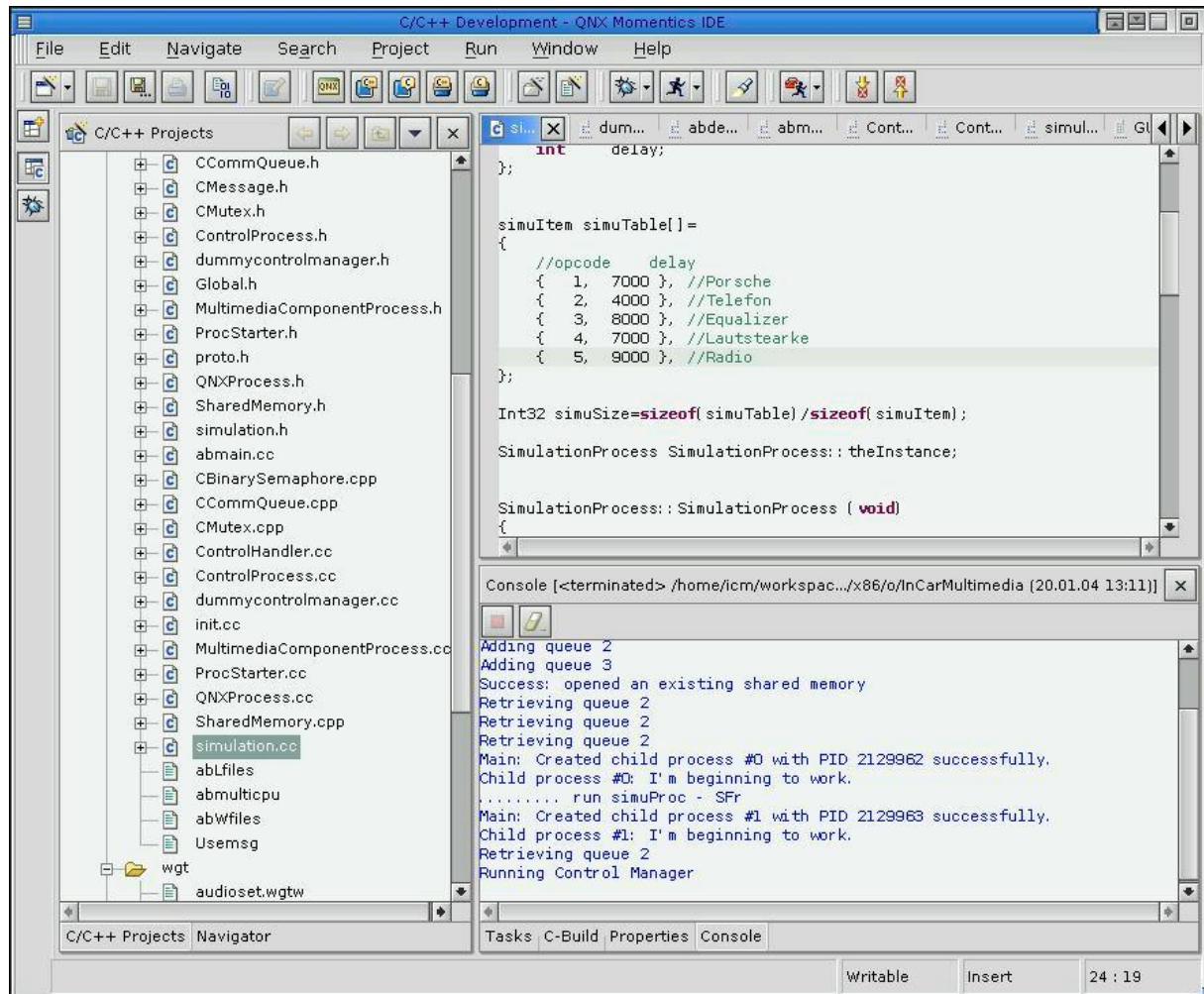
Es wird also nur das Script „/qnc/ICMDemoHobbit.sh“ nach dem Hochfahren des Systems gestartet und muss nicht per Console „von Hand“ gestartet werden.

Wichtiger HINWEIS:

In Kapitel „Serielle Console“ wurde schon darauf hingewiesen, das man sich ggf. durch Einträge im hddstart-Script den Zugang zum System abschneiden kann, und somit keine Möglichkeit mehr hat, Einstellungen auf dem System zu ändern.

17. Präsentationsapplikation WS03/04 für Hobit

Bei der Hobit wird eine funktional sehr eingeschränkte Applikation präsentiert, da bis zu diesem Termin keine Funktionalität mehr implementiert werden kann. Diese Funktionalität wird im weiteren Verlauf des ICM-Projektes schrittweise implementiert werden. Die Hobit-Version ist somit eine reine Präsentationsapplikation, in der verschiedene Screens auf dem Display umgeschaltet werden. Die Umschaltung der Screens wird durch einen Prozess angestoßen, der entsprechende Nachrichten über den shared memory versendet. In der nachfolgenden Grafik ist das Hobit-Projekt in Momentics geöffnet dargestellt.



Es sind fünf verschiedene Screens implementiert, die anhand der Daten in der Tabelle `simuTable[]` umgeschaltet werden.

Auf der nächsten Seite sind die fünf Displayinhalte abgebildet.



Diese Displaydarstellungen geben einen Ausblick darauf, welche Funktionalität implementiert werden soll. Hierzu zählen insbesondere die Kommunikation über MOST und die Navigation.