uniCONTROL SYSTEMTECHNIK

Embedded Linux

im Automotivebereich

Gliederung

- Vorstellung Unicontrol
- Verwendete Hardware
- Embedded Entwicklungsumgebungen
- Bauen eines Crosscompilers
- Kernel, Treiber und Bootloader
- Bibliotheken und Tools
- Lizenzierung
- Applikationsentwicklung
- NFS vs. Flashen
- Debuggen auf dem Target

Vorstellung Unicontrol

- mittelständisches Unternehmen
- ca. 30 Mitarbeiter
- Zulieferer für Siemens VDO u.a.
- Automotivebereich und Schiffskommunikation
- Kombiinstrumente und Infotainment
- eigene Projekte: DKI, Mediadatenerfassung, Unilogger

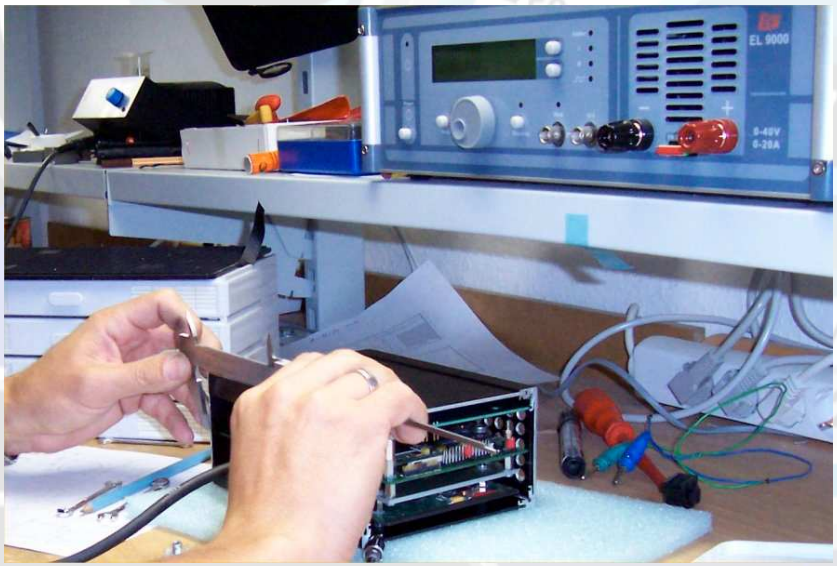
Unicontrol Systemtechnik GmbH
Sachsenburger Weg 34
OT Dittersbach
D-09669 Frankenberg
Tel.: 037206 / 8873 - 0

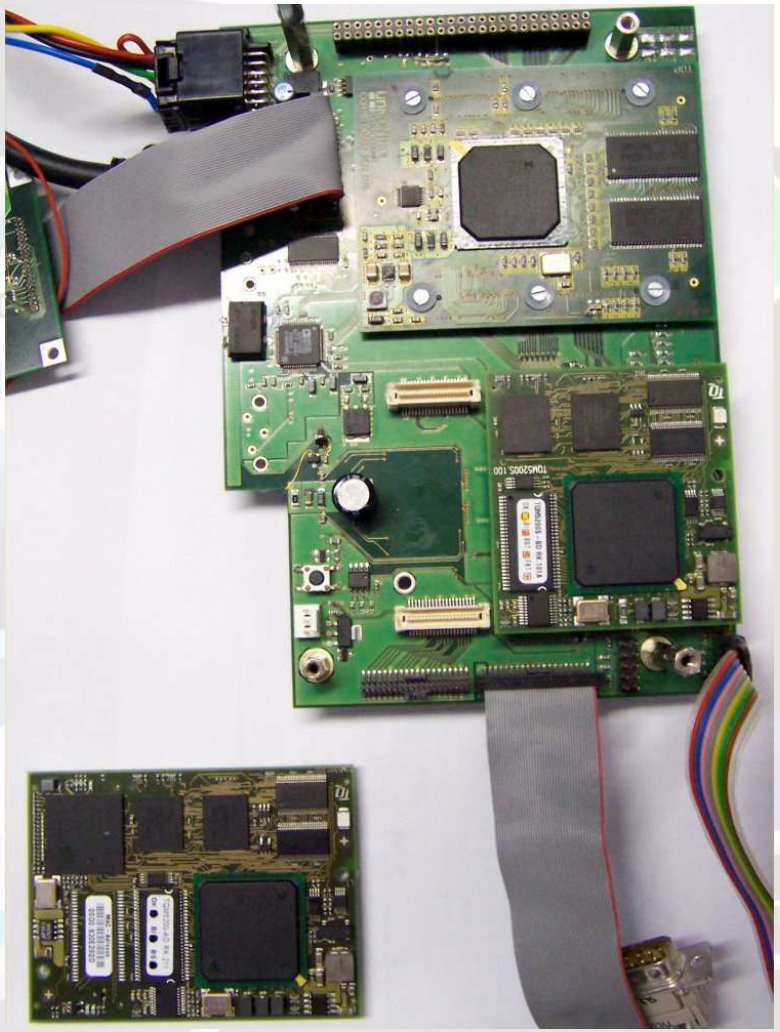
http://www.unicontrol.de

So fing alles an... das Digitale Kombiinstrument

Verwendete Hardware

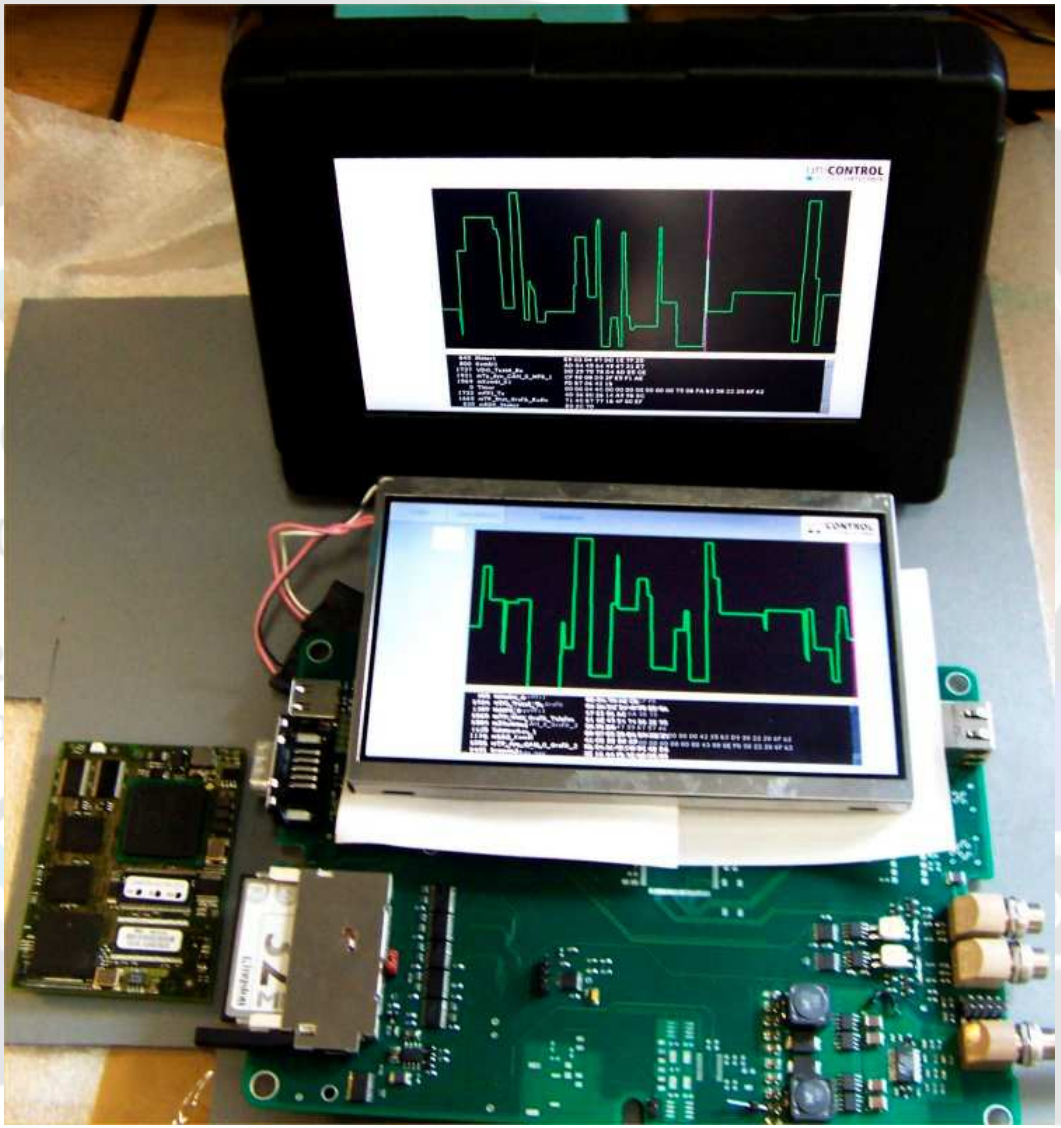
- DKI:
  - MPC5200
  - Coral P
  - Basisplatine





Verwendete Hardware

- Unilogger:
  - MPC5200
  - integrierter Grafikchip
  - Basisplatine



Embedded Entwicklungsumgebungen

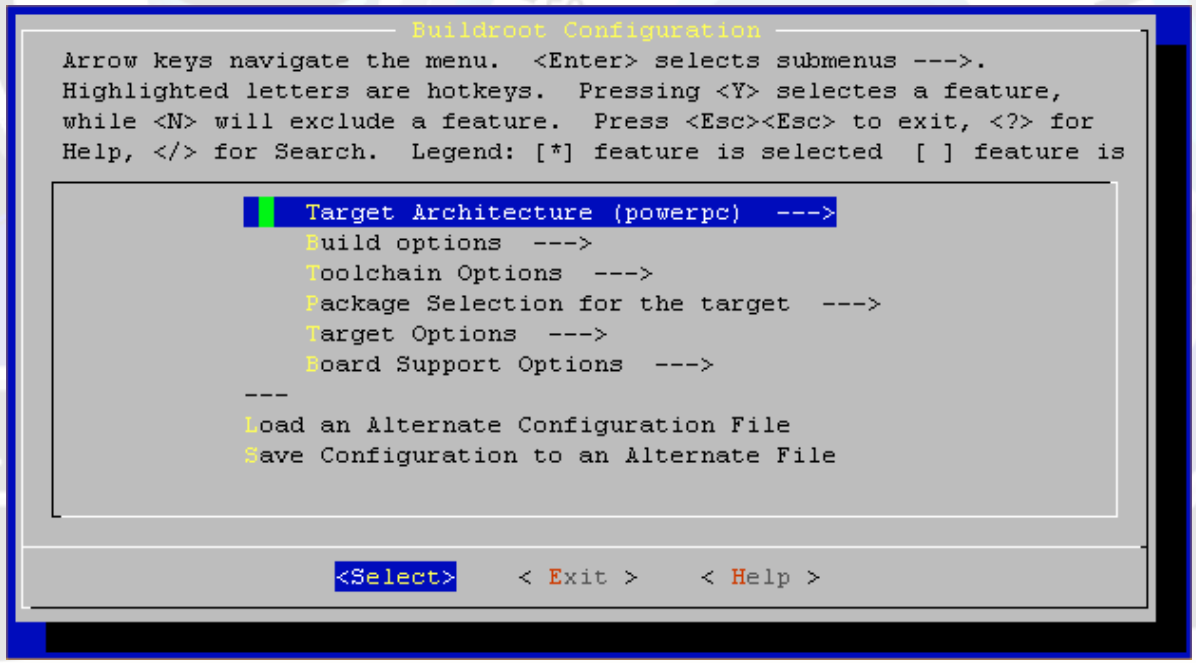
- GHS, WindRiver
- Integrity, VxWorks, QNX, WinCE, OSEK
- SysGo, MontaVista, Denx
- Multi 2000, Eclipse, make, gcc
- glibc, uClibc
- Hardware-Emulator/-Debugger
- Netzwerk (NFS)
- Flashen
- eigene Entwicklungsumgebung

Eigene Entwicklungsumgebung

- historisch gewachsen
- spezielle Anforderungen:
  - mehrere Targets
  - soll für PC und Target bauen
  - alles in Versionsverwaltung
- Realisierung:
  - System aus Konfigurationsdateien und -variablen (platforms, projects, configs)
  - Trennung in Kernel, Tools+Libs und eigene Software (generische + spezielle Anteile)
- compilieren mit gcc + make

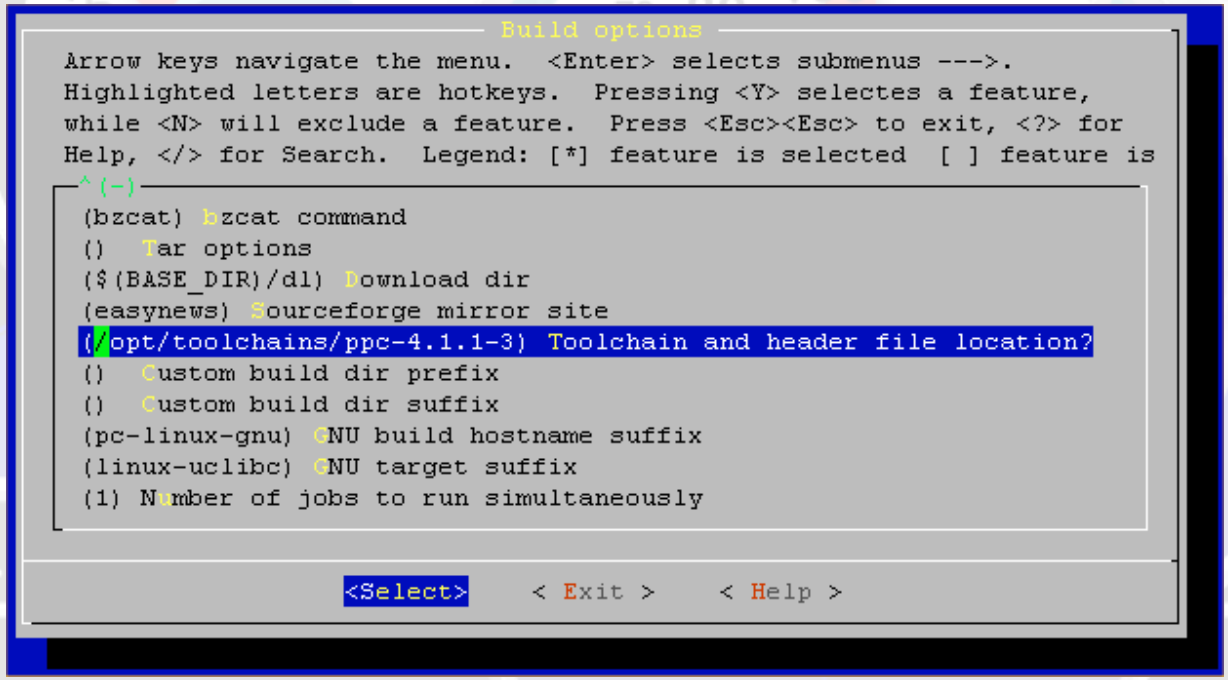
Bauen eines Crosscompilers I

- buildroot daily snapshot aus Subversion (oder als fertiges Paket)
- Patches einpflegen (z.B. gcc-Bugs bei C++-Unterstützung auf PowerPC)
- make menuconfig



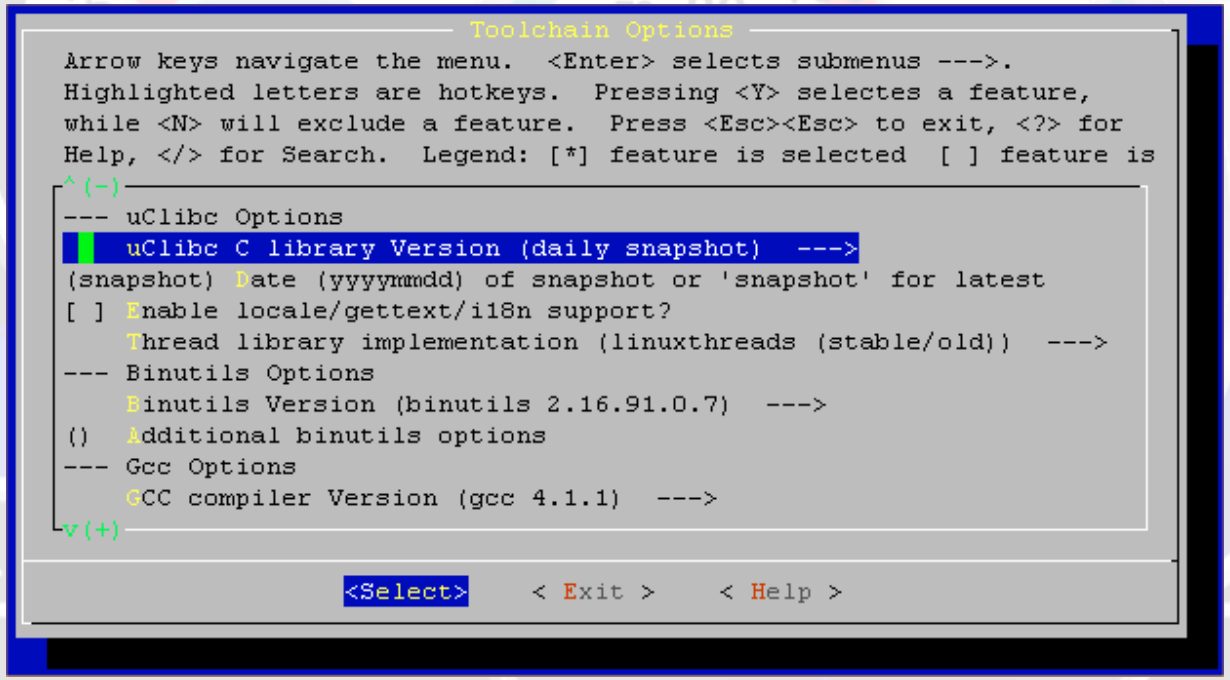
Bauen eines Crosscompilers II

- Wichtig: Zielpfad einstellen!
- gcc benutzt den Pfad unter bestimmten Umständen
- sonst Header an falscher Stelle gesucht



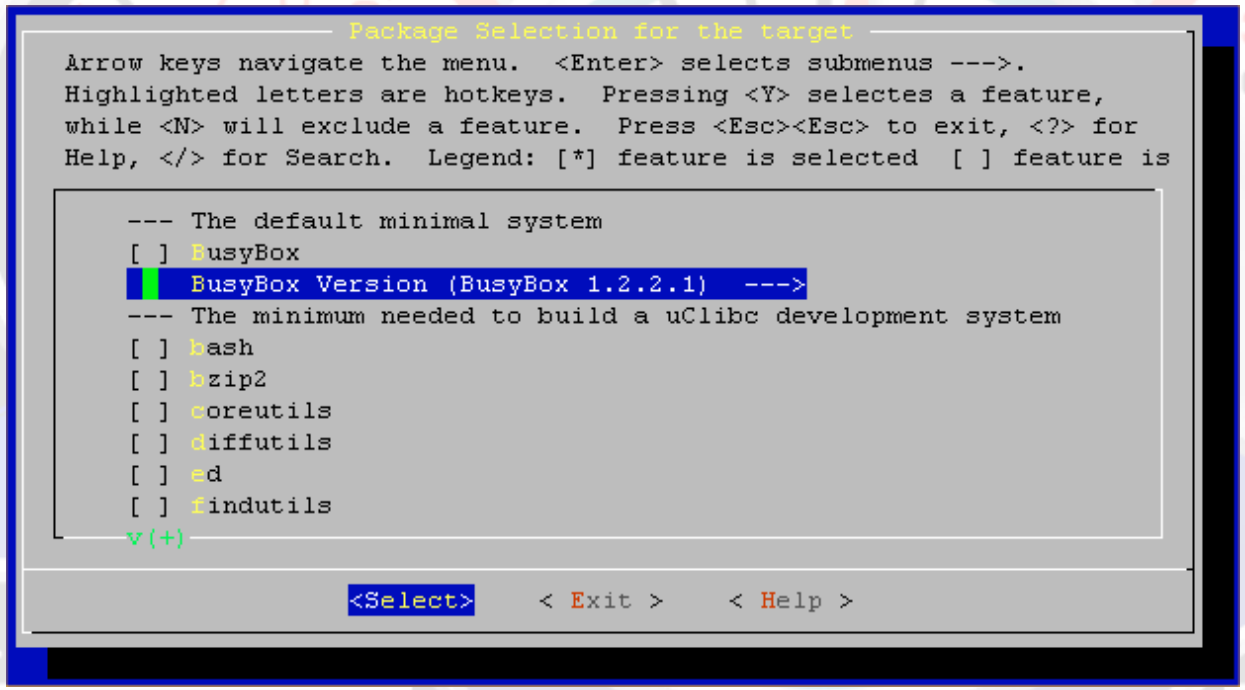
Bauen eines Crosscompilers III

- Architektur, Pfade, Versionen
- gcc, binutils, Kernel, uClibc
- gcc 4.1.1, uClibc Snapshot



Bauen eines Crosscompilers IV

- zusätzlich können Busybox u.a. Utils eingebunden werden
- bei uns wird Busybox später gebaut



Bauen eines Crosscompilers V

- Start mit make
- baut in angegebenes Verzeichnis
- lädt alle benötigten Pakete herunter und baut sie
- bei Fehlern am besten von vorn anfangen :-) (zumindest auf PowerPC)
- am Ende liegt im Zielverzeichnis der fertig gebaute Compiler mit der uClibc

Kernel, Treiber und Bootloader I

- aktuellster Kernel (2.6.20 aus git)
- vor allem wegen Unterstützung von Embedded OpenFirmware
- Architektur: powerpc (früher: ppc)
- Einbinden der Patches von den Maintainern für die Prozessorfamilie MPC52xx
- eigene Anpassungen (inkl. Bugreports)
- make menuconfig
- make uImage

Kernel, Treiber und Bootloader II

- Grafiktreiber Fujitsu Coral P (auch von Denx und ALT Software (OpenGL) unterstützt)
- CAN-Treiber mit Transportprotokoll als optionales Modul
- Diagnosetreiber (über Transportprotokoll)
- GPIO-Treiber
- GPL-Lizenzierung (bis auf wenige Ausnahmen)
- mit eigenem Buildsystem gebaut

Kernel, Treiber und Bootloader III

- U-Boot Snapshot (git) oder fertiges Paket
- Hauptentwicklung bei Denx, Anteil von Secret Lab
- unterstützt Embedded OpenFirmware

```
cis login:

U-Boot 1.2.0-gaf2b1157 (Feb  2 2007 - 09:09:44)

CPU:   MPC5200 v1.2, Core v1.1 at 396 MHz
       Bus 132 MHz, IPB 132 MHz, PCI 66 MHz
Board: TQM5200 (TQ-Components GmbH)
       on a UniControl Rev2 carrier board
I2C:   85 kHz, ready
DRAM:  64 MB
FLASH: 32 MB
Video: CoralP
Enable LVDS
In:    serial
Out:   serial
Err:   serial
Net:   FEC ETHERNET
SRAM:  512 kB
VGA:   SMI501 (Voyager) with 8 MB
Hello gray_rat VWPassat_Rev2_Dev. Auto : run nfsboot

Hit any key to stop autoboot:  0
=>
```

Kernel, Treiber und Bootloader IV

- eigene Anpassungen an unsere Hardware → GPL
- Unterstützung des Coral make *boardname*_config (uc5200, basiert auf tqm5200)
- make erzeugt uboot.bin
- Flashen mit U-Boot → Henne-Ei-Problem
- problematisch, wenn Flashen fehlschlägt
- auch per JTAG - JTAG-Tools (openwince) von Unicontrol auf HW portiert (Prozessor + Pinbelegung Flashanbindung)

Bibliotheken und Tools

- Pakete runterladen oder Snapshot
- Patches einpflegen (Busybox z.B. tftp)
- bei uns automatisch (ähnlich zu buildroot)
- Busybox: make menuconfig
  .config speichern und in Zukunft benutzen
- sonst:
  ./configure --host=powerpc-unknown-linux --target=host --prefix=...; make && make install
  (eigentlich powerpc-uclibc-linux, aber viele configures erkennen das nicht richtig und brechen ab)
- aufpassen bei benutzten Headerfiles!
- am Ende erhält man Bibliotheken und Tools

Lizenzierung

- Dilemma: Applikationssoftware soll nicht veröffentlicht werden (Know-How, Sicherheitsbedenken)
- Konsequenz: GPL-Bibliotheken scheiden aus
- GPL-Tools und LGPL-Bibliotheken weitgehend unproblematisch
- aber: Nutzer muss Möglichkeit haben, im System diese Bibliotheken/Tools auszutauschen
- schwierig bei Embedded Systemen
- theoretisch besteht bei uns diese Möglichkeit, allerdings wird Zugriff auf Flash aus Sicherheitsgründen unterbunden
- im Kernel bestimmte Funktionen nur für GPL-Treiber verfügbar → wir benutzen GPL

Applikationsentwicklung

- ...um dem Target einen Sinn zu geben

- Unicontrol-Entwicklungen:
  - DKI - Digitales Kombiinstrument
    - generische (fahrzeugunabhängige) Anteile
    - fahrzeug- / projektabhängige Anteile
  - UDAG - Universelles Datenerfassungsgerät
    - Förderantrag nach Marktanalyse mittels stark eingeschränktem Testmodell „Unilogger“
    - Linux auf dem Target
    - Konfigurationsapplikation mit C# auf Windows
    - selbe Codebasis für beide (C++, Boost)

  → beide benutzen dasselbe Buildsystem

NFS vs. Flashen I

- endgültige Version:
  - Schreiben in ROM oder Flash → „Flashen“
  - unabhängig vom Entwicklungsrechner
- zur Entwicklung:
  - Laden in RAM per Debugger/Emulator (JTAG)
  - Netzwerk-Filesystem, z.B. NFS
  - abhängig vom Entwicklungsrechner

NFS vs. Flashen II

- in /etc/exports Verzeichnis freigeben:
  /tftproot/ *(rw,no_root_squash,async,nohide)
- make nfs kopiert alles nach /tftproot/TARGET/
- selbe Verzeichnisstruktur wie auf Target
- Kernel als Rootpath mitgeben
  (bei uns im U-Boot durch Konfigurationsvariablen)
- Kernel per TFTP gebootet
  (selbes Verzeichnis wie NFS): nfsboot im U-Boot
  (eigene Anpassung)

```
=> print
ramboot=run ramargs addip;bootm 0xFC0A0000 - 0xFC080000
nfsboot=rarp 350000 /${target}${ofdtbfile};rarp 200000 /${target}${bootfile}
bootdelay=2
preboot=set stdout serial;echo Hello ${hostname} ${target}. Auto : ${bootcmd
rootpath="/tftproot/"
bootfile="/boot/uImage"
netdev=eth0
hostname=gray_rat
ramargs=setenv bootargs ${bootargs} root=/dev/mtdblock4
nfsargs=setenv bootargs ${bootargs} root=/dev/nfs rw nfsroot=${serverip}:${r
addip=setenv bootargs ${bootargs} ip=${ipaddr}:${serverip}:${gatewayip}:${ne
u-boot=/u-boot.bin.tq
ofdtbfile=/boot/Rev2.dtb
splashimage=FC820000
update.fw=protect off FC000000 FC05FFFF;erase FC000000 FC05FFFF;cp.b 200000
load.fw=rarp 200000 ${u-boot}
ethact=FEC ETHERNET
ethaddr=00:D0:93:0E:29:2A
target=VWPassat_Rev2_Dev
ipaddr=10.0.0.2
serverip=10.0.0.1
netmask=255.255.255.0
gatewayip=10.0.0.1
```

NFS vs. Flashen III

- make flash_image baut Images aus allen benötigten Teilen
- mehrere Images, digital signiert:
  - OFDTB (OpenFirmware DeviceTree Binary)
  - Kernel, RootFS, OptFS
  - Splashscreen (U-Boot), Designs (VarFS)
- Flashen über verschiedene Methoden:
  - CAN (per Diagnose über TP)
  - Netzwerk (von Applikation auch über Menü)
  - JTAG, seriell, ...
- Image im U-Boot über ramboot gestartet (automatisch nach Timeout)
- Timeout auf 0 setzbar, schaut nach Tastendruck, später GPIO

Debuggen auf dem Target I

- Zugriff auf HW:
  - seriell (minicom)
  - Netzwerk zum Aufstecken (telnet)
  - JTAG (zusätzliche HW)
- Kernel-Debugging:
  - kdb (direkt im Kernel mit eigener Oberfläche)
  - kgdb (Kernelpatch, Connect mit gdb)
- Applikation evtl. für Host compilieren und dort debuggen
- bei den Tools auch gdb für Target → Debugging auf Target
- aber: Binaries zu groß → besser: Remote-Debugging (z.B. mit -O3 und strip 6,7MB, mit -g -ggdb3 ca. 120MB)
- gdb + gdbserver mit buildroot gebaut (massiv gepatcht)

Debuggen auf dem Target II

- Target:
  - gdbserver /dev/ttyS0|*host:port*
    (*host*: erlaubter Client, wird nicht unbedingt geprüft)
  - per seriell problematisch, wenn Konsole die belegt
  - mögliche Szenarien:
    - gdbserver seriell, Konsole Netzwerk
    - gdbserver Netwerk, Konsole seriell
    - beide Netzwerk
    - beide seriell auf verschiedenen seriellen Interfaces

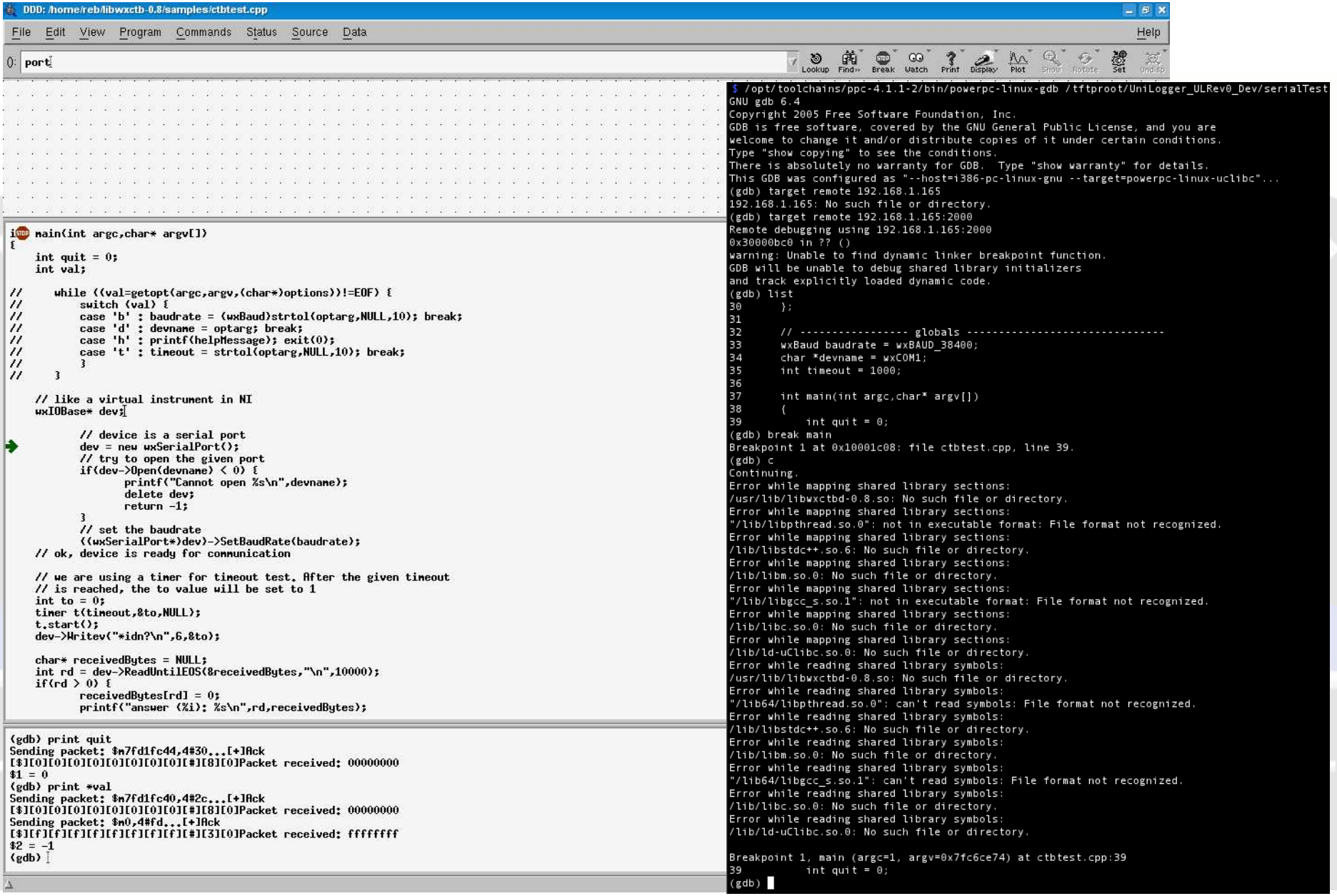
```
# /usr/bin/gdbserver heka:2000 serialTest
Process serialTest created; pid = 439
Remote debugging from host 192.168.1.84
```

Debuggen auf dem Target III

- Host:
  - GDB aufrufen:
    powerpc-linux-gdb binary (mit Debugsymbolen)
    gdb> target remote /dev/ttyS0|host:port
    gdb> cont (nicht run!)
  - gdb (aus Toolchain!) läuft auf Host, aber verarbeitet Symbole vom Target (strip ebenso)
  - auf Target und Host muss Binary nicht dasselbe sein, nur Abarbeitungsreihenfolge → gestripptes Binary auf Target möglich

Debuggen auf dem Target IV

- DDD:
  - Aufruf im gdb in Skript verlagern:
    ```
    # cat target-debug
    target remote host:port
    ```
  - ddd mit diesem Skript aufrufen:
    ```
    ddd -x target-debug --debugger \
    /opt/toolchains/ppc-4.1.1-2/bin/powerpc-linux-gdb binary
    ```
  - ddd-Konsole durch Output nahezu unbrauchbar
  - Backtrace ebenso
  - deswegen gdb auf Kommandozeile vorzuziehen
- Alternativen?
  - kdbg (KDE), aber kann kein Remote-Debugging


DDD: /home/reb/libwxctb-0.8/samples/ctbtest.cpp
File Edit View Program Commands Status Source Data Help
port
Lookup Find Break Watch Print Display Plot Show Rotate Set Undisp
main(int argc,char* argv[])
{
int quit = 0;
int val;
// while ((val=getopt(argc,argv,(char*)options))!=EOF) {
// switch (val) {
// case 'b' : baudrate = (wxBaud)strtol(optarg,NULL,10); break;
// case 'd' : devname = optarg; break;
// case 'h' : printf(helpMessage); exit(0);
// case 't' : timeout = strtol(optarg,NULL,10); break;
// }
// }
// like a virtual instrument in NI
wxIOBase* dev;
// device is a serial port
dev = new wxSerialPort();
// try to open the given port
if(dev->Open(devname) < 0) {
printf("Cannot open %s\n",devname);
delete dev;
return -1;
}
// set the baudrate
((wxSerialPort*)dev)->SetBaudRate(baudrate);
// ok, device is ready for communication
// we are using a timer for timeout test. After the given timeout
// is reached, the to value will be set to 1
int to = 0;
timer t(timeout,&to,NULL);
t.start();
dev->Writev("*idn?\n",6,&to);
char* receivedBytes = NULL;
int rd = dev->ReadUntilEOS(&receivedBytes,"\n",10000);
if(rd > 0) {
receivedBytes[rd] = 0;
printf("answer (%i): %s\n",rd,receivedBytes);
(gdb) print quit
Sending packet: $m7fd1fc44,4#30...[+]Ack
[$][0][0][0][0][0][0][0][0][#][8][0]Packet received: 00000000
$1 = 0
(gdb) print *val
Sending packet: $m7fd1fc40,4#2c...[+]Ack
[$][0][0][0][0][0][0][0][0][#][8][0]Packet received: 00000000
Sending packet: $m0,4#fd...[+]Ack
[$][f][f][f][f][f][f][f][f][#][3][0]Packet received: ffffffff
$2 = -1
(gdb)
$ /opt/toolchains/ppc-4.1.1-2/bin/powerpc-linux-gdb /tftproot/UniLogger_ULRev0_Dev/serialTest
GNU gdb 6.4
Copyright 2005 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i386-pc-linux-gnu --target=powerpc-linux-uclibc"...
(gdb) target remote 192.168.1.165
192.168.1.165: No such file or directory.
(gdb) target remote 192.168.1.165:2000
Remote debugging using 192.168.1.165:2000
0x30000bc0 in ?? ()
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
(gdb) list
30 };
31
32 // ---------- globals ----------
33 wxBaud baudrate = wxBAUD_38400;
34 char *devname = wxCOM1;
35 int timeout = 1000;
36
37 int main(int argc,char* argv[])
38 {
39 int quit = 0;
(gdb) break main
Breakpoint 1 at 0x10001c08: file ctbtest.cpp, line 39.
(gdb) c
Continuing.
Error while mapping shared library sections:
/usr/lib/libwxctbd-0.8.so: No such file or directory.
Error while mapping shared library sections:
"/lib/libpthread.so.0": not in executable format: File format not recognized.
Error while mapping shared library sections:
/lib/libstdc++.so.6: No such file or directory.
Error while mapping shared library sections:
/lib/libm.so.0: No such file or directory.
Error while mapping shared library sections:
"/lib/libgcc_s.so.1": not in executable format: File format not recognized.
Error while mapping shared library sections:
/lib/libc.so.0: No such file or directory.
Error while mapping shared library sections:
/lib/ld-uClibc.so.0: No such file or directory.
Error while reading shared library symbols:
/usr/lib/libwxctbd-0.8.so: No such file or directory.
Error while reading shared library symbols:
"/lib64/libpthread.so.0": can't read symbols: File format not recognized.
Error while reading shared library symbols:
/lib/libstdc++.so.6: No such file or directory.
Error while reading shared library symbols:
/lib/libm.so.0: No such file or directory.
Error while reading shared library symbols:
"/lib64/libgcc_s.so.1": can't read symbols: File format not recognized.
Error while reading shared library symbols:
/lib/libc.so.0: No such file or directory.
Error while reading shared library symbols:
/lib/ld-uClibc.so.0: No such file or directory.
Breakpoint 1, main (argc=1, argv=0x7fc6ce74) at ctbtest.cpp:39
39 int quit = 0;
(gdb)

Vielen Dank für Ihre Aufmerksamkeit

Unicontrol Systemtechnik GmbH
Sachsenburger Weg 34
OT Dittersbach
D-09669 Frankenberg
Tel.: 037206 / 8873 - 0

http://www.unicontrol.de

- Weiterführende Links:
  - http://www.unicontrol.de
  - http://www.digitales-kombiinstrument.de
  - http://www.linuxdevices.com
  - http://www.busybox.net
  - http://uclibc.org
  - http://u-boot.sourceforge.net
  - http://openwince.sourceforge.net