Echzeit mit rtai und xenomai unter Linux

Version 0.2

Andreas Waffler
wa-an@rz.fh-augsburg.de

26. August 2009

1 rtai

1.1 Installation

Die Installation des Echtzeiterweiterung rtai auf einem x86 Rechner vollzieht sich in 5 Schritten. Zuerst besorgt man sich einen Linux Kernel, am besten von www.kernel.org, anschließend den rtai Patch. Dieser ist wie der Kernel auch als Tarball verfügbar, unter www.rtai.org. Dabei ist darauf achten das die Versionen von Kernel und rtai Patch zusammenpassen, nicht für jedes Kernelrelease ist auch ein rtai Patch verfügbar. Im Beispiel hier wurden die Versionen linux-2.6.29.4 und rtai-3.7.1 benutzt, beide als tar.bz2-Archiv heruntergeladen und in den Ordner *cd /usr/src* kopiert.
Nun werden die Archive entpackt, und symbolische Links auf die entpackten Ordner erstellt. Das ist vielleicht nicht unbedingt nötig, ist aber durchaus Linux-üblich. Ausserdem lassen sich so Verwechslungen vermeiden.

1. Archive entpacken und vorbereiten
cd /usr/src
tar xjvf linux-2.6.29.4.tar.bz2
tar xjvf rtai-3.7.1.tar.bz2
ln -s linux-2.6.29.4 linux
ln -s rtai-3.7.1 rtai
Als nächstes wir der Patch angewandt. Hier gilt es, den für die Kernelquellen passenden zu wählen. Die Patchdateien liegen im rtai Ordner, hier im Beispiel unter */usr/src/rtai/base/arch/x86/patches/hal-linux-2.6.29.4-x86-2.4-01.patch*. Aus dem Pfad lässt sich die Zielarchitektur und die Kernelversion ablesen.

2. Patch anwenden
cd /usr/src/linux
patch -p1 -b <../rtai/base/arch/x86/patches/hal-linux-2.6.29.4-x86-2.4-01.patch
Jetzt kann (muss) der Kernel konfiguriert werden. Diese Aufgabe ist wohl die Zeitaufwendigste, und hat eigentlich gar nicht mit einer Echtzeiterweiterung zu tun. Natürlich muss auch ein Echtzeitfähiger Kernel für die Zielhardware konfiguriert werden. Um eine 'gute' Konfiguration zu erhalten, gibt es wohl zwei Möglichkeiten. Man könnte die Konfiguration eines 'generic' Kernels nehmen, zum Beispiel die des mit der Distribution kommenden Kernels. Oder man nimmt sich Zeit die Konfiguration eng auf die Zielplattform zu schneidern, was länger dauert, wahrscheinlich nicht beim ersten Mal klappt, aber sicher zu einem leistungsfähigeren (weil genau für das Ziel optimerten) Kernel führt. Die Konfigurationsdatei '.config' wird im Kernelverzeichniss gespeichert. Sie ist entweder schon vorhanden, aus einem anderen Kernel, oder wird in einem langen Prozess durch zB *make menuconfig* erstellt. Wie gesagt, hier ist noch nicht Echtzeitspeziefisches dabei. Entscheidungshilfe, welche Optionen gewählt werden sollten, liefern zB. die Befehle 'cat /proc/cpuinfo' und 'lspci'.
Ist man nun zu einer 'guten' Konfiguration gekommen, sind (endlich) die rtai speziefischen Punkte anzuhandeln. Da betrifft einige Optionen in der Kernelkonfiguration, die sich offenbar nicht mit dem rtai Ansatz vertragen, und daher abgeschaltet, bzw verändert werden sollten.
make menuconfig

```
General Setup -> local version anpassen (z.B: -myrtai)

Enable loadable module support -> Module versioning support = NO
```

```
Processor type and features -> Interrupt Pipline = YES (neue Option
                                                        durch den rtai patch)
                            -> Machine Check Exeptions = NO (vermeidet Kompilierfehler
                                                             ~...smb_thermal_...)

Power management and ACPI options -> Power Management Support = NO (betrifft 'halt',
                                                                    Rechner fährt runter,
                                                                 schaltet nicht mehr aus)
                                  -> CPU Frequency Scaling = NO
                                  -> CPU idle PM support = NO
```

Anschießend wirft man den Kompilier- und Installationsvorgang, nach dem üblichen Schema, an
make; make install; make modules_install
Jetzt ist je nach Kernelkonfiguration noch eine 'initramdisk' zu erstellt. Alternativ kann man auch ein Debian Packet von dem Kernel erstellen. Wenn 'make' ohne Fehler gelaufen ist: (Das Arbeitsverzeichniss ist immernoch '/usr/src/linux')
make-kpkg binary-arch
cd ..
dpkg -i linux-image-2.6.29.4ohara-rtai_2.6.29.4ohara-rtai-10.00.Custom_i386.deb
Im Beispiel hier ist die 'local version' des Kernels 'ohara-rtai'. Wenn der Kernel installiert ist,

3. Den neuen Kernel booten

Dazu sollte ein entsprechender Eintrag für den Bootloader erstelt werden. Bei der Variante ein Debian Packet zu erstellen und zu installieren sollte dieser Eintrag automatisch erstellt werden. Ein Eintrag für den Bootloader GRUB sieht zB so aus (*/boot/grub/menu.lst*):

```
title           2.6.29.4.ohara-with-rtai-patch
root            (hd0,0)
kernel          /boot/vmlinuz-2.6.29.4ohara-rtai root=/dev/sda1 ro single vga=791
```

Wenn der neue Kernel läuft, ist die Installation von rtai ist zur Hälfte abgeschlossen. Jetzt müssen noch die rtai-Kernelmodule, Bibliotheken und Testprograme gebaut werden. Man begibt sich in das entpackte rtai Archiv (auf en der Link 'rtai' zeigt). Für den Kompiliervorgang wird ein extra Verzeichniss ('build') angelegt, das scheint wichtig zu sein, da rtai sonst irgendwie durcheinander kommt.

4. kernel module und userspace support installieren

cd /usr/src/rtai
mkdir build
cd build
make -f ../makefile menuconfig
Dabei zeigt sich eine Konfigurationsoberfläche ähnlich der des Kernels. Hier ist die Defaulteinstellung wahrscheinlich erstmal eine gute Sache. Man sollte aber prüfen ob unter dem Punkt 'General' der Pfad zum 'linux kernel tree' stimmt. Nach der Konfiguration startet automatisch der Kompiliervorgang. Nachdem der Kompiliervorgang erfolgreich beendet wurde sollten die neuen Module mit *make install* installiert werden. Anschließend müssen noch device nodes erstellt, bzw kopiert werden:
cp -a /dev/rtai_shm /lib/udev/devices/
cp -a /dev/rtf[1-9] /lib/udev/devices/
Auf den meisten Systemen läuft heutzutage ein 'udev' Deamon, der zur Bootzeit dynamisch devicenodes erstellt. Solange man udev nicht sagt, das die eben erstellten Nodes bei jedem Systemstart erzeugt werden sollen, werden sie nach dem Neustart fehlen. Man müsste also ein udev Regel definieren, die die rtai Nodes berücksichtigt. Natürlich lässt sich udev auch umgehen, durch ein Skript, das nach udev ausgeführt wird.

```
#!/bin/sh -e
if test \! -c /dev/rtai_shm; then
        mknod -m 666 /dev/rtai_shm c 10 254
fi
for n in `seq 0 9`; do
        f=/dev/rtf$n
        if test \! -c $f; then
```

```
                mknod -m 666 $f c 150 $n
        fi
done
```

Dieses (init) Skript kann dann nach einem Zeitpunkt nachdem udev Devices erzeugt hat in den Initablauf eingehängt werden. Um später auch dynamisch gelinkte Programme auszuführen, sollte dem Linker noch mitgeteilt werden, wo sich die rtai Biblioteken befinden. *echo /usr/realtime/lib/ >/etc/ld.so.conf.d/rtai.conf*

5. rtai testen

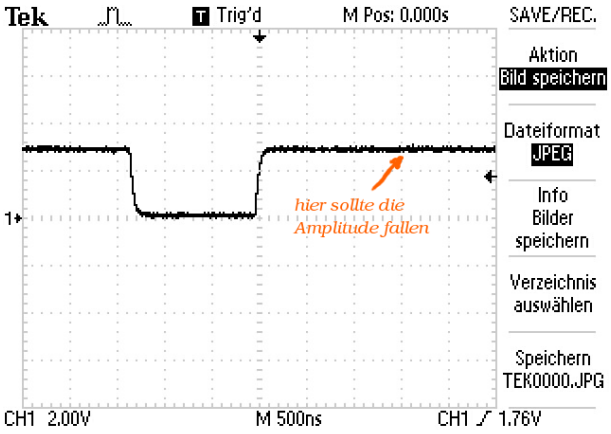
Mit dem rtai Archiv werden auch einige Testprogramme geliefert, sie befinden sich in */usr/realtime/testsuite*. Um zum Beispiel die Latenzzeiten des Systems zu messen, führt man folgenden Test durch:
cd /usr/realtime/testsuite/kern/latency
./run

1.2 Anwendung

Im Vergleich stehen zwei Userspaceprogramme mit identischer Funktion, eins ohne rtai (rect.c), das andere mit rtai (rect_with_rtai.c). Beide führen ein Schleife aus, in der die Datenpins (D0 -D7) des Paralellports getoggle'd werden.

```c
// gcc -O2 -o rect rect.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/io.h>
#include <signal.h>
#define BASEPORT 0x378

static int end;
static void endme(int dummy){
        end=1;
}
int main(int argc, char **argv) {
        if (ioperm(BASEPORT, 3, 1)) {
                perror("ioperm open");
                exit(1);
        }
        signal(SIGINT, endme);
        outb(0, BASEPORT + 2);
        while (!end) {
                outb(0, BASEPORT);
                outb(255, BASEPORT);
        }
        freeperm();
        exit(0);
}
int freeperm() {
        if (ioperm(BASEPORT, 3, 0)) {
                perror("ioperm close");
        }
}
```

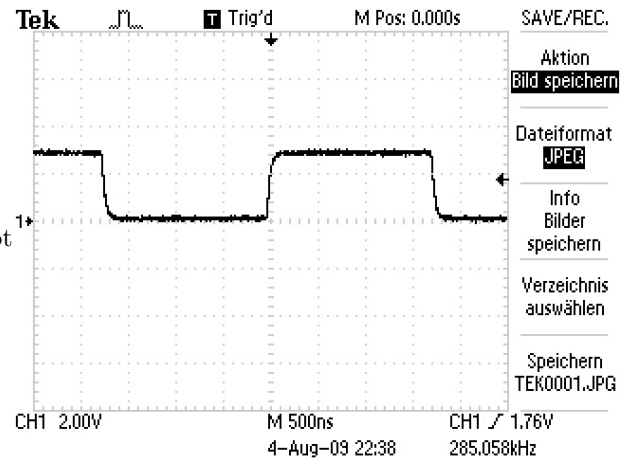
Das Programm erzeugt einen stabiles Rechteck mit ein Frequenz von ca 285 kHz an den Datenpins. Sobald ein anderes Programm gestartet wird ,zB 'updatedb', fängt das Signal stark zu zittern an, und auch die Frequenz bricht stark ein.



Nun das Programm als rtai Prozess. Die rtai speziefischen Erweiterungen sind farbig hinterlegt. Die Dokumentation der rtai API findet sich unter https://www.rtai.org/documentation/magma/html/api/.

```
1  // gcc -I/usr/realtime/include -L/usr/realtime/lib -llxrt -lpthread -O2 -o
       rect_with_rtai rect_with_rtai.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/io.h>
5  #include <signal.h>

5  #include <rtai.h>
6  #include <rtai_sched.h>
7  #define TICK_PERIOD 50000
8  #define TASK_PRIORITY 1
9  #define STACK_SIZE 10000

9  #define BASEPORT 0x378
10 static int end;
11 static void endme(int dummy){
12         end=1;
13 }
14 int main(int argc, char **argv) {
15         if (ioperm(BASEPORT, 3, 1)) {
16                 perror("ioperm open");
17                 exit(1);
18         }
19         signal(SIGINT, endme);
20         outb(0, BASEPORT + 2);

20         RT_TASK * rt_task;
21         rt_set_periodic_mode();
22         rt_task =  rt_task_init(1234, TASK_PRIORITY, 0,0);
23     int tick_period = start_rt_timer(nano2count(TICK_PERIOD));
24         rt_task_make_periodic(&rt_task, rt_get_time() + tick_period, tick_period);

24         while (!end) {
25                 outb(0, BASEPORT);

25                 rt_task_wait_period();

25                 outb(255, BASEPORT);

25                 rt_task_wait_period();

25                 }
26         freeperm();

26         stop_rt_timer();
27 //      rt_task_delete(&rt_task);

27         exit(0);
28 }
29 int freeperm() {
30         if (ioperm(BASEPORT, 3, 0)) {
31                 perror
   ("ioperm close");
32         }
33 }
```

Das vom Programm erzeugte Rechteck Signal bleibt auch unter Last stabil. Wieder wurde 'updatedb' auf einer andrean console gestartet. Da 'updatedb' jedoch kein rtai Prozess ist, kam es auch nur selten zum Zug (während die Festplatte ohne laufenden rtai Prozess kontinuierlich rattert, gibts sie während eines laufenden rtai Prozesses nur selten ein Geräusch von sich).



2 xenomai

2.1 Installation

Die Installation der Echtzeiterweiterung xenomai gestaltet sich mit 3 Schritten etwas übersichlicher als die der rtai Erweiterung. Auch hier müssen wieder Kernel und xenomai Version zusammenpassen, im Beispiel wurde linux-2.6.30 als Tarball heruntergeladen und in das Verzeichniss */usr/src* kopiert. Die xenomai Quellen können direkt von in einem GIT Repository geklont werden.

1. Quellen besorgen und vorbereiten
cd /usr/src
tar xjvf linux-2.6.30.tar.bz2
mv linux-2.6.30 linux-2.6.30xenomai_patched
git clone git://git.xenomai.org/xenomai-head.git xenomai-head

2. xenomai kompilieren
Das patchen der Kernelquellen übernimmt das Skript *prepare-kernel.sh* das auch im xenomai repository , im Ordner 'scripts' enthalten ist. Da im Repository Patches für alle verfügbaren Kernelversionen vorhanden sind, kann das Skript die meise Arbeit allein erledigen. Zwigned ist die Option *–linux=*, sie beinhaltet den Pfad zu den Kernelquellen, die gepatched werden sollen. Das script frägt dann noch nach der Architekur und nach der patchversion, beide werden per default ermittelt, und müssen nur bestätigt werden. Das readme (xenomai-head/README.INSTALL) sagt auch wie weiter Optionen zum Cross Compiling genutzt werden können. Nachdem die patches angewandt wurden, konfiguriert man dem Kernel.
cd xenomai-head
scripts/prepare-kernel.sh –linux=/usr/src/linux-2.6.30xenomai_patched
cd /usr/src/linux-2.6.30xenomai_patched
make menuconfig

```
<*> Native API (NEW)  --->
<*> POSIX API (NEW)  --->
< > pSOS+ emulator (NEW)  --->
< > uITRON API (NEW)  --->
< > VRTX emulator (NEW)  --->
< > VxWorks emulator (NEW)  --->
< > RTAI emulator (NEW)  --->
[ ] Do not warn about deprecated skin usage (NEW)
<*> Real-Time Driver Model (NEW)  --->
```

Dabei fällt der neue Menupunkt 'Real-time subsystem' auf. Dahinter verstecken sich viele interessante Optionen, wie auch die Wahl der API. Xenomai kann auch Programme ausführen die für andere Echtzeiterweiterungen und sogar für andere Betriebssysteme geschrieben sind. Nachdem den Kernel konfiguriert ist startet man dem Kompiliervorgang mit *make*.

3. user-space support installieren
Das sind Bibliotheken, Headers und verschiedene Testprogramme die auch komiliert und installiert werden müssen um die Erweiterung zu nutzen. Schon **während** der Kernel kompiliert, kann man sich um den userspace support kümmern. Das macht man am besten in einer anderen shell. Dort geht man durch die üblichen Schritte um ein Porgramm aus den Quellen zu installieren. Ein *configure* Skript erfasst alle systemspeziefischen Anhängigkeiten um schreibt ein entsprechendes Makefile. Das Readme (README.INSTALL) verrät optionale Argumente an das *configure* Skript, wobei der default erstmal gut passt.
cd /usr/src/xenomai-head
./configure; make ; make install

Wenn der gepachte Kernel nach einem Neustart läuft, können einige Testprogramme ausprobiert werden. Sie liegen per default in */usr/xenomai/bin*. Dem Linker sollte noch mitgeteilt werden, wo sich die xenomai Biblioteken befinden. Dazu erzeuget man eine Konfigurationsdatei
echo "/usr/xenomai/lib/">/etc/ld.conf.d/xenomai.conf

2.2 Anwendung

Bsp: Programm mit nativer xenomai API
TODO

Bsp: Programm mit rtai API
TODO