OMAP Embedded Linux

Fontys T7 Embedded Linux

27 januari 2009

Inhoudsopgave

1 Introductie

Dit document beschrijft in het kort hoe *Embedded Linux* op een TI OMAP platform gezet kan worden. Het document heeft een technische ondertoon, waardoor enige ervaring met Linux in het algemeen vereist is.

Embedded Linux is in bijna alle opzichten gelijk aan Linux op desktop- en serverhardware. Kennis over deze twee Linuxtoepassingen is bijna altijd toepasbaar op het embedded platform.

Dit document beschrijft enkel de stappen die benodigd zijn om het OMAP platform op te zetten en software te ontwikkelen voor dit platform. Vaak zijn extra stappen benodigd ter voorbereiding. Aangezien deze per Linuxomgeving en -distributie anders zijn, zijn deze stappen niet opgenomen in dit document. Enkel de benodigdheden worden aangekaart. Hoe er aan deze benodigdheden tegemoet gekomen wordt valt dus buiten de scope van dit document.

De nodige voorbereidingen voor het gebruik van Bluetooth en de WiiMote zijn ook opgenomen in dit document. Uiteraard kunnen deze overgeslagen worden wanneer het platform voor andere doeleinden wordt gebruikt.

Commando's worden in de volgende wijze uitgewerkt, en zijn te herkennen aan de dollar-prefix:

```
$ while [ 1 ]
$ do echo "Hello, World"
$ done
```

2 Voorbereiding

2.1 Hardware

Voor het succesvol voltooien van deze guide is een ontwikkelmachine nodig waarop geen toegangsrestricties gelden. Mocht dit wel het geval zijn, zorg dan in ieder geval dat de systeembeheerder bereikbaar is in het geval van problemen.

Het is verstandig een Linuxdistributie te installeren. De correcte werking van andere operating systems met deze guide wordt niet gegarandeerd.

Een werkende RS-232 seriële poort is noodzakelijk. Of deze intern of extern op het systeem aangesloten is, is niet van belang. Ook een vrije RJ-45 netwerkpoort is van belang. De gebruikte netwerkkaart moet kunnen terugschalen naar 10 Mbit/s half duplex modus, aangezien de OMAP dit verwacht.

Deze guide is geschreven voor de OMAP 5912 OSK. Terwijl het met enkele kleine aanpassingen toepasbaar is voor ieder Linuxplatform, is dit bord aangeraden als ontwikkelplatform.

2.2 UNIX-kennis en een Gezond Verstand

Tijdens de bouw van software of het configureren ervan zullen vast en zeker fouten verschijnen. Door middel van gezond verstand en wat kennis over UNIX-achtige operating systems zijn een groot deel van de fouten te achterhalen en op te lossen. Zo veel mogelijk veel voorkomende fouten zijn afgevangen door de meegeleverde scripts. Enkele afhankelijkheden van libraries en tools kunnen echter niet afgevangen worden aangezien deze omgevingsafhankelijk zijn. Kennis over het installeren van tools en libraries voor het systeem waarop de volgende stappen uitgevoerd worden is essentieel.

2.3 Linux Host Toolchain

Alvorens we kunnen gaan ontwikkelen voor het OMAP platform, moeten de ontwikkeltools gebouwd worden. De meeste Linuxdistributies leveren de mogelijkheid om een volledige toolchain te installeren voor het bouwen van *native* applicaties. Dit document gaat er van uit dat de nodige tools voor Linux software-ontwikkeling al aanwezig zijn. Voor het bouwen van de toolchain zijn wat minder gebruikelijke tools ook benodigd, die wellicht nog geïnstalleerd moeten worden.

- Lex
  Lexical analyzer generator
- Bison
  LALR parser generator

- Automake en Autoconf
  Tools voor het genereren van configure- en makefiles
- mkimage
  Een U-Boot tool die van een Linux kernel image een bootable image kan maken

De compiler voor native applicaties moet GCC 4.1 zijn, aangezien nieuwere compilers de toolchain niet kunnen bouwen.

2.4 Bouwomgeving

Om op een nette manier software te bouwen voor de OMAP is een mooie bouwomgeving sterk aan te raden. Dit houdt het hele gebeuren overzichtelijk.

```
$ export SOURCE_DIR=~/source
$ mkdir -p $SOURCE_DIR/build
$ cd $SOURCE_DIR/build
$ wget http://www.jrrzz.net/~jorrizza/src/arm-el/build.sh
$ wget http://www.jrrzz.net/~jorrizza/src/arm-el/Makefile
$ chmod +x build.sh
```

De variabele *SOURCE_DIR* geeft aan waar de source komt te staan. Deze variabele mag veranderd worden naar iedere schrijfbare locatie. Zorg ervoor dat deze variabele niet meer verandert. De variabele wordt in komende hoofdstukken nog gebruikt.

De twee files *build.sh* en *Makefile* zijn handigheidjes voor het bouwen van software. De twee zorgen ervoor dat software geconfigureerd, gebouwd en geïnstalleerd wordt door middel van één commando. Het shellscript is verantwoordelijk voor de omgevingsvariabelen zodat de toolchain gebruikt wordt. De Makefile wordt vervolgens aangeroepen. In de Makefile staan de locaties van de software, de configuratie ervan en bouwcommando's. Iedere keer wanneer deze files gebruikt worden, wordt in het kort uitgelegd wat er gebeurt.

De tree wordt in de komende hoofdstukken alsvolgt opgebouwd:

```
$SOURCE_DIR/
  ./build/
    ./build.sh
    ./Makefile
    ./target/
    ./<source dirs>/
  ./crosstool/
  ./crosstool_arm/
```

3 Toolchain

Voor dit project is besloten om de Crosstool toolchain te gebruiken. Deze toolchain is opgebouwd uit standaard GNU tools. Hij is ondertussen al enkele jaren oud, maar is gestandardiseerd door Fontys. Deze toolchain levert GCC 4.1 met Glibc 2.3.2.

Er is een aangepast build script aangeleverd om de gewenste toolchain te bouwen. Het enige dat veranderd is, is de target.

De volgende stappen resulteren in een werkende toolchain.

```
$ cd $SOURCE_DIR
$ svn co http://crosstool.googlecode.com/svn/trunk/src crosstool
$ cd crosstool/
$ wget http://www.jrrzz.net/~jorrizza/src/arm-el/demo-arm-custom.sh
$ sh demo-arm-custom.sh
```

De toolchain is nu geïnstalleerd in de *crosstool_arm* directory onder je *SOURCE_DIR*. De directory *crosstool* bevat de sources en bouwomgeving die gebruikt zijn door crosstool.

4 Linux

Een niet onverwacht aspect van Embedded Linux is de Linux kernel zelf. In dit hoofdstuk wordt een poging gedaan een werkende Linux kernel te bouwen.

Er is voor Linux kernel versie 2.6.24 gekozen. Dit is de nieuwste versie die in het voorgaand onderzoek met Crosstool tot een werkende kernel gecompileerd kon worden. In het onderzoek is de range 2.6.18-2.6.27 onderzocht.

De rol die de Linux kernel gaat spelen in het systeem is applicatie-afhankelijk. Een vrij uitgebreide configuratie is voorgeleverd. Het aanpassen van de configuratie is natuurlijk toegestaan, en sterk aanbevolen voor specifieke toepassingen.

```
$ cd $SOURCE_DIR/build
$ ./build.sh linux
```

Het script downloadt eerst de Linux source code van Montavista's GIT repository. Vervolgens wordt een standaardconfiguratie toegevoegd. Dit tezamen levert een bouwbaar geheel dat vervolgens tot een image gecompileerd wordt. Door middel van de mkimage tool worden er U-Boot specifieke headers toegevoegd. Uiteindelijk wordt de image en eventuele modules in de target directory geïnstalleerd.

5 Busybox

Busybox is een applicatie die bijna alle userspace taken centraliseert in één applicatie. Busybox levert init, een shell, wat file management tools en nog veel meer goodies. Het bouwen en installeren van een voorgeconfigureerde Busybox gaat alsvolgt:

```
$ cd $SOURCE_DIR/build
$ ./build.sh busybox
```

Het script downloadt en configureert Busybox. Vervolgens wordt het gecompileerd en geïnstalleerd in de target directory.

De voorgeleverde configuratie is enorm uitgebreid. Dit maakt Busybox relatief groot. Voor een ontwikkelomgeving is dit prima, maar voor een uiteindelijke userspace is dit een te logge oplossing. Het is verstanding Busybox uiteindelijk te minimaliseren tot de functies die daadwerkelijk gebruikt worden.

6 Configuratie met KConfig

Waarschijnlijk voldoet de voorgeleverde configuratie van zowel de Linux kernel als Busybox niet aan de wensen van het project. Daarom is het handig om te weten hoe de configuratie aan te passen valt.

Zowel Linux als Busybox maken gebruik van dezelfde configuratiemethode alvorens ze overgaan tot het daadwerkelijk bouwen van de software. De configuratieparameters worden opgeslagen in een configuratiebestand, *.config* genaamd, door kconfig. Deze file bevat alle configuratie-opties in Bash-achtige definevorm. Er zijn vier mogelijkheden waarop een configuratie-optie genoteerd kan zijn.

- Ingecompileerd

  `CONFIG_PENGUIN=y`

  Dit zorgt ervoor dat het object waarnaar de configuratie-optie verwijst in de target binary wordt gelinkt. Dit heeft als voordeel dat dit in veel opzichten een snelheidswinst oplevert. De target binary zelf wordt echter wel groter op deze manier, waardoor er initieel meer geheugen gereserveerd wordt. Het is dus zaak alleen die dingen als *y* te configureren die daadwerkelijk nodig zijn in het project waarvoor deze .config wordt gefabriceerd.

- Gemodulariseerd

  `CONFIG_PENGUIN=m`

  Een gemodulariseerd object wordt gelinkt tegen de target binary tijdens runtime. Deze objecten staan bij de Linux kernel bekend als *kernel modules*. Het laden van een kernel module tijdens boot time is not done, tenzij de module uit gaat van de module_init en module_exit functies om een ACPI sleep state te overleven. Dit komt embedded bijna nooit voor, dus het is veilig en aangeraden geen modules tijdens boot time te hoeven laden. De meest voorname reden waarom deze functionaliteit geleverd wordt is de mogelijkheid om een relocatable binary te bouwen. Er hoeft dan niets gecompileerd te worden op een volkomen andere machine. De gewenste functionaliteit wordt in dat geval tijdens runtime geladen. Het incompileren van alle configuratie-opties resulteert simpelweg in een veel te grote binary, en dus de nodige geheugenproblemen.
  Developers kunnen gebruik maken van kernel modules om het mogelijk te maken om modules te vervangen met eigen code, of compleet nieuwe modules te schrijven, zonder de volledige kernel te hoeven hercompileren. Een door de Linux kernel build gebouwde module kan worden geladen door middel van *modprobe*, een object file (.ko) kan worden geladen door middel van *insmod*. Modprobe is enkel een slimmere insmod en weet waar

de .ko files te vinden zijn. De geladen modules, en afhankelijkheden van elkaar, kunnen worden bekeken door middel van de *lsmod* tool. Modules kunnen, mits niet in gebruik, verwijderd worden met de *rmmod* tool.

```
$ insmod ./penguin.ko
    - of -
$ modprobe penguin
$ lsmod
Module                  Size  Used by
penguin                 1024  0
daemon                   666  penguin
$ rmmod penguin
```

- Constante

  ```
  CONFIG_PENGUIN_DEFAULT_RETURN=42
  CONFIG_PENGUIN_DOMINATION="world"
  ```

  Deze constantes worden gebruikt door modules, goed vergelijkbaar met standaard C defines.

- Ontbrekend

  ```
  # CONFIG_PENGUIN is not set
  ```

  Wanneer een optie niet gezet is, wordt de module helemaal niet meegenomen in het buildproces. Niet iedere optie krijgt zo'n comment line. Aangezien kconfig configuratie doorgaans in een boomstructuur werkt, is het aangeven van een enkel knooppunt als comment voldoende om alle functionaliteit onder dit knooppunt uit te schakelen.
  Het is de kunst bij het configureren van software om zoveel mogelijk weg te laten en alleen het werkelijk nodige over te houden.

In de source tree kun je KConfig over het algemeen aanroepen met de volgende commando's:

```
$ cd $SOURCE_DIR/build/linux-omap-2.6
$ make config
    - of -
$ make menuconfig
    - of -
$ make xconfig
    - of -
$ make oldconfig
```

De vier opties voor configuratie config, menuconfig, xconfig en oldconfig zijn methodes om door middel van KConfig de .config file aan te passen.

config Door middel van vragen op de command line wordt de .config file opgebouwd. Standaard antwoorden, alsmede opties worden aangegeven tussen blokhaken.

menuconfig Een ncurses interface waarin met een gebruiksvriendelijke methode de configuratie ingesteld kan worden. Zorg ervoor dat de ncurses development files geïnstalleerd zijn alvorens menuconfig te starten.

xconfig Een X11 applicatie, gelijkwaardig aan menuconfig. Dit maakt gebruik van Qt, en heeft dus Qt development files nodig.

oldconfig Dit is exact als config, maar vraagt alleen de vragen die nieuw zijn in vergelijking met de al aanwezige .config file. Dit is vooral handig bij een nieuwe kernelversie op dezelfde hardware. Het kan zelfs goed zijn dat oldconfig geen vragen stelt. Dat betekent dat de huidige .config voldoet voor de nieuwe kernelversie.

6.1 Standardisatie

Voor .config files zijn aparte targets aangemaakt in de Makefile. Deze zorgen ervoor dat met een clean build standaard configs worden gedownload die vervolgens door de builds van Busybox en Linux gebruikt kunnen worden. Deze locaties zijn te herkennen aan de *URL_ *_ CONFIG* variabelen.

Voor het vervangen van de standaard configuratie zijn de volgende stappen benodigd, dit werkt hetzelfde voor zowel Linux als Busybox.

```
$ cd $SOURCE_DIR/build/linux-omap-2.6
$ make menuconfig
$ cp .config ../linux-omap-2.6.config
$ gzip linux-omap-2.6.config
```

Het originele config bestand zal vervangen worden en zal voortaan gebruikt worden door het build script als standaardconfiguratie. Wanneer de gepubliceerde standaardconfiguratie weer benodigd is, is het simpelweg verwijderen van de configuratie voldoende. De originele configuratie wordt vervolgens opnieuw gedownload.

7 BlueZ

Voor deze applicatie is een Bluetooth-stack benodigd. Naast support voor Bluetooth in de kernel is er software in userspace nodig om met Bluetooth aan de gang te kunnen. BlueZ is de officiële Bluetooth stack onder Linux.

Voordat BlueZ gebouwd kan worden zal er eerst aan de dependencies moeten worden voldaan. De twee belangrijkste zijn GLib en DBUS.

7.1 GLib

Deze library vindt zijn oorsprong in het GTK+ en GNOME project. Het implementeert veelgebruikte dingen zoals lists, trees en strings in C.

Bouwen en configureren van GLib is eenvoudig te doen met de volgende reeks aan commando's:

```
$ cd $SOURCE_DIR/build
$ ./build.sh glib
```

Het script downloadt de GLib source eerst. Vervolgens wordt deze geconfigureerd. Er wordt gebruik gemaakt van een config cache file om bepaalde tests over te slaan. Sommige tests compileren testcode. Aangezien er ARM binaries worden gecompileerd, en de host waarschijnlijk geen ARM is, zullen deze tests falen. De resultaten van de tests op een ARM systeem worden opgeslagen in het config cache bestand zodat de tests voor lief worden genomen door het configure script. Uiteindelijk wordt de source code gecompileerd en geïnstalleerd in de target directory.

7.2 DBUS

DBUS is een message bus syteem. Het is een volledig in userspace uitgewerkte methode voor Inter Process Communication (IPC). BlueZ maakt gebruik van DBUS bij het communiceren tussen de BlueZ library en de daemon, die vervolgens weer toegang verschaft aan de hardware.

Een vereiste van DBUS is een XML library. Deze zal eerst aangeleverd moeten worden.

7.2.1 Expat

Expat is een veelgebruikte XML library. Deze is eenvoudig in zijn opzet, dus eenvoudig te crosscompilen:

```
$ cd $SOURCE_DIR/build
$ ./build.sh expat
```

Expat wordt in de volgende stappen gedownload, geconfigureerd en gebouwd. Expat is een goed voorbeeld van een nette library die door middel van Autoconf eenvoudig te crosscompilen is. In de praktijk is dit echter een uitzondering.

7.2.2 DBUS Build

Nu Expat aangeleverd is, kan DBUS gebouwd worden.

```
$ cd $SOURCE_DIR/build
$ ./build.sh dbus
```

Alvorens DBUS gebouwd kan worden is er een kleine patch benodigd. In de toolchain die in deze guide gebruikt wordt is PIE ondersteuning gebroken. PIE staat voor Position Independent Executable. Dit zorgt ervoor dat de machine-instructies onaangepast goed uitgevoerd worden, zelfs wanneer de positie van de code verandert. Dit is enkel noodzakelijk zonder MMU, dus het zal weinig problemen opleveren. De patch haalt PIE uit de compiler instructies.

Het configure script van DBUS faalt in het juist detecteren van de paden naar de headerfiles. Tijdens de configuratie van DBUS worden de paden naar GLib en DBUS include directories hard meegegeven. Ook een config cache wordt gegenereerd voor het juist configureren van DBUS.

Vervolgens wordt DBUS geconfigureerd en geïnstalleerd in de target directory.

7.3 BlueZ Build

Nadat de dependencies tegemoet gekomen zijn kan BlueZ gebouwd worden. Als het goed is gaat dit probleemloos.

```
$ cd $SOURCE_DIR/build
$ ./build.sh bluez
```

Tijdens de configuratie van BlueZ worden de nodige functionaliteiten uitgeschakeld. Wanneer er meer functionaliteit benodigd is, kan de *CONFIG_BLUEZ* variabele in de makefile aangepast worden.

BlueZ wordt geconfigureerd en geïnstalleerd in de target directory.

8 CWiid

Deze library verschaft gemakkelijke toegang tot functionaliteit van de WiiMote. Door middel van deze library kunnen custom tools geschreven worden in een kortere tijd dan zonder deze library.

Het kan gecrosscompiled worden door middel van de inmiddels bekende reeks commando's:

```
$ cd $SOURCE_DIR/build
$ ./build.sh cwiid
```

Er zijn enkele aanpassingen benodigd binnen de source tree. De patch die over de source wordt toegepast fixt een aantal dingen. Ten eerste worden de testapplicaties verwijderd uit de build. Ten tweede wordt er een fout opgelost in de makefiles waardoor een target altijd hardcoded werd uitgevoerd. Ten slotte is een BlueZ API call veranderd, en nog niet doorgevoerd in de CWiid source. Verder zijn de makefiles erg slecht geschreven waardoor compiler flags in een minder nette wijze doogegeven moeten worden.

Uiteindelijk, na het repareren van de source, kan de library gebouwd worden. Ook deze wordt geïnstalleerd in de target directory.

9 Booten

Het opstarten, of booten, van het OMAP bordje kan op veel manieren. Om development te vergemakkelijken wordt in deze guide gekozen voor het netbooten van het bordje.

Dit houdt in dat het bordje alle benodigde data over het netwerk binnenhaalt. De volgende stappen worden ondernomen tijdens het booten van de OMAP:

1. OMAP powerup
2. Flash ROM wordt naar RAM gekopieerd
3. U-Boot start
4. DHCP request en lease
5. BOOTP request en answer
6. TFTP transfer van Linux kernel naar RAM
7. Linux kernel start
8. Mount NFS root filesystem
9. Init start
10. Apps en eventueel shell starten

9.1 U-Boot

U-Boot zal zo geconfigureerd moeten worden dat het bovenstaande stappenplan wordt gevolgd. Ook moeten de boot parameters voor de Linux kernel zo ingesteld worden dat het bootproces verloopt volgens plan.

Om de U-Boot configuratie aan te passen is het zaak om het ontwikkelbord via een seriële link aan te sluiten. Een gebruiksvriendelijke client onder Linux is *gtkterm*. De standaardconfiguratie van de link zou alsvolgt gedefinieerd moeten zijn:

- Speed: 115200
- Parity: none
- Bits: 8
- Stopbits: 1
- Flow control: none

Terwijl U-Boot start kan het onderbroken worden door op een toets te drukken. Door middel van de U-Boot command line kunnen enkele instellingen goed gezet worden. Let op dat bij het derde commando geen enters gebruikt mogen worden tussen de regels in.

```
OMAP5912 OSK# setenv fileaddr 10000000
OMAP5912 OSK# setenv bootcmd dhcp\; bootm $(fileaddr)
OMAP5912 OSK# setenv bootargs console=ttyS0,115200n8 mem=30M
              noinitrd root=/dev/nfs rw ip=dhcp
              nfsroot=192.168.1.2:/share/nfs/rootfs2.6,nolock single
OMAP5912 OSK# saveenv
```

Deze instellingen maken stap 1-10 mogelijk aan de kant van de OMAP. Op dit moment zal de OMAP via DHCP blijven wachten op een reactie. De betekenis van alle bootargumenten zijn op te zoeken in de Linux documentatie. Deze set bootargumenten zorgen ervoor dat de seriële poort als console wordt gebruikt, niet meer dan 30MB aan geheugen wordt gebruikt, geen initrd wordt gebruikt, dat hij van NFS read/write gebruik maakt als root filesystem, waar de NFS share te vinden is, dat hij niet lockt op NFS I/O en dat hij is single user mode start.

9.2 NFS

Het OMAP bordje zal gaan booten met een NFS root filesystem. Aan de kant van de OMAP is alles nu geregeld. Aan de kant van de ontwikkelmachine moet nog het een en het ander geregeld worden. Een NFS-server is eenvoudig op te zetten.

Het installeren van een NFS-server is erg distributiespecifiek. Er zijn twee smaken aan NFS-servers. De meest gebruikte is de NFS kernel server. Deze server maakt gebruik van kernelfunctionaliteit voor het delen van bestanden. Dit is doorgaans sneller en logischer. Mocht de kernel van de productiemachine geen NFS-serverondersteuning aanbieden, is de tweede smaak een oplossing. Een NFS-server in userspace werkt doorgaans hetzelfde, maar heeft alle code in userspace draaien.

De configuratie is wel redelijk gestandardiseerd. Het maakt over het algemeen niet veel uit welke smaak aan NFS-server er draait. De configuratiefile */etc/exports* zou de volgende regel toegevoegd moeten krijgen:

```
/share/nfs/rootfs2.6        192.168.1.50(rw,no_root_squash)
```

Dit zorgt ervoor dat de directory /share/nfs/rootfs2.6 gedeeld wordt voor de machine met IP-adres 192.168.1.50. De machine mag zowel lezen als schrijven. Normaal is het zo dat root op de clientmachine geen rechten heeft om bestanden

als root aan te maken op de servermachine. Dit heeft te maken met veiligheid van de servermachine. Aangezien dit een volledig filesystem gaan worden, en root de enige gebruiker is, is het zaak root wel toegang te geven tot het filesystem. Daarvoor is de *no_root_squash* flag toegevoegd. Na het aanpassen van het bestand is het noodzakelijk de NFS-server te reloaden.

9.2.1 Root File System

Tijdens deze guide is een groot gedeelte van het root filesystem gegenereerd. Dit is echter nog niet voldoende. Omdat het hier om een ontwikkelplatform gaat, en voldoende ruimte beschikbaar is op de NFS host, is het handig wat extra functionaliteit toe te voegen aan het filesystem.

Als eerste wordt de basis gebruikt zoals die door Montavista is voorgeleverd. Dit is niet zo verschrikkelijk nodig, maar het biedt tenminste een werkend filesystem voor het embedded device. Het is distributiespecifiek of een root shell door middel van *su* of *sudo* bereikt wordt.

```
$ su -
    - of -
$ sudo sh
# mkdir -p /share/nfs/
# cd /share/nfs/
# wget -q -O - \
# http://linux.omap.com/pub/filesystem/rootfsosk.tar.bz2 | tar jxv
# chown -R <user>:<group> rootfs2.6
# exit
```

De chown zorgt ervoor dat alle bestanden van de gebruiker *<user>* en groep *<group>* zijn. Vervang *<user>* en *<group>* door de gebruikersnaam van de user die de ontwikkelomgeving gaat gebruiken.

In de volgende stap worden de ontwikkeltools in het filesystem gezet, zodat er op de target gecompileerd en gedebugd kan worden. Ook wordt de C library gekopieerd waardoor er dynamisch gelinkte applicaties gebruikt kunnen worden.

```
$ cp -rv $SOURCE_DIR/crosstool_arm/gcc-4.1.0-glibc-2.3.2/\
$ arm-unknown-linux-gnu/arm-unknown-linux-gnu/* \
$ /share/nfs/rootfs2.6/
```

Tenslotte kunnen de binaries die in de afgelopen hoofdstukken gebouwd zijn overgeheveld worden naar het kersverse filesystem.

```
$ cp -rv $SOURCE_DIR/build/target/* \
$ /share/nfs/rootfs2.6/
```

Het filesystem is nu compleet. Er is enkel nog een probleem dat opgelost dient te worden. Sommige applicaties hebben tijdens het compileren de *PREFIX*-variabele meegenomen en zullen nu zoeken op deze locatie naar bestanden. Deze variabele is gezet op *$SOURCE_DIR/target/*. Dit resulteert doorgaans in een pad als /home/gebruiker/<iets>/target/. Natuurlijk bestaat deze locatie nog niet op het embedded device.

```
$ mkdir -p /share/nfs/rootfs2.6/$SOURCE_DIR/build
$ cd /share/nfs/rootfs2.6/$SOURCE_DIR/build
$ ln -s / target
```

Deze commando's zorgen ervoor dat een zoekactie op de originele *PREFIX* resulteren in een zoekactie op de root, ofwel /.

Officieel moet dit opgelost worden met een nette chroot tijdens het bouwproces, maar dat idee is opgeofferd ten behoeve van de reductie van complexiteit.

9.3 DHCP

Een DHCP-server heeft in deze opstelling twee afzonderlijke functies. Zowel stap 3 als stap 4 worden uitgevoerd door de DHCP-server. In dit voorbeeld wordt dnsmasq gebruikt als zowel DHCP- als TFTP-server. Dit is een klein en lichtgewicht server, ideaal geschikt voor deze opstelling.

Het installeren van dnsmasq is distributieafhankelijk. Vaak heet het pakket gewoon dnsmasq. Veel dependencies heeft het niet.

Het belangrijkste is natuurlijk de configuratie. De belangrijkste configuratievariabelen zijn de volgende, en zijn doorgaans te vinden in */etc/dnsmasq.conf*:

```
interface=eth0
dhcp-range=192.168.1.50,192.168.1.50,12h
dhcp-boot=uImage
```

Zorg ervoor dat *eth0* het netwerkapparaat is waarmee de ontwikkelmachine *alleen* aan de OMAP verbonden is. Deze configuratie geeft iedere machine aangesloten op de netwerkpoort het IP 192.168.1.50. Het draaien van een DHCP-server op een publiek netwerk wordt doorgaans niet getolereerd. Let er dus op dat dnsmasq niet draait wanneer de netwerkpoort gebruikt wordt voor andere doeleinden.

Wanneer een client voor een BOOTP-bestandsnaam vraagt zal dnsmasq uImage terug geven. Deze wordt vervolgens op de TFTP-server gezocht.

9.4 TFTP

Een TFTP-server is een van de meest eenvoudige bestandsservers die tegenwoordig nog gebruikt worden. De eerste letter staat niet voor niets voor *trivial*.

De volgende regels moeten toegevoegd worden aan */etc/dnsmasq.conf*:

```
enable-tftp
tftp-root=/share/tftp
```

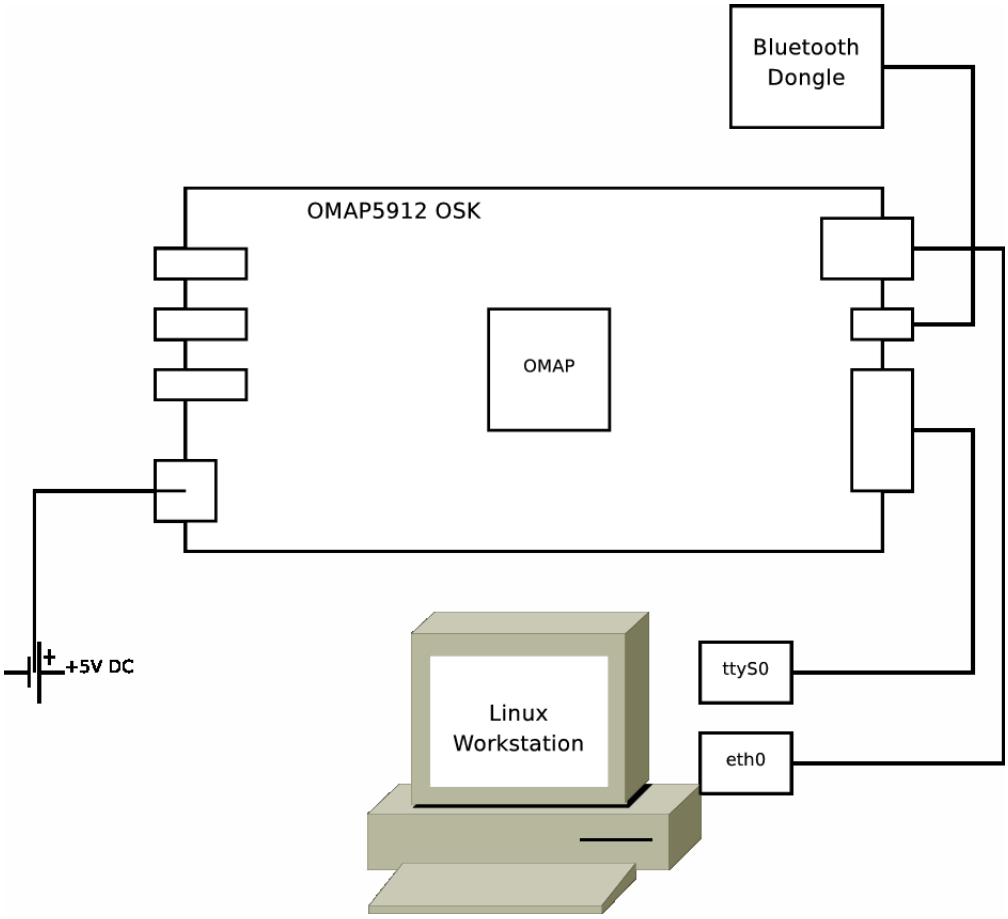
De TFTP-server wordt op deze manier ingeschakeld, en bestanden worden vanuit /share/tftp gehost. Herstart dnsmasq na het aanpassen van de configuratie. Het enige wat nog rest is het aanmaken van deze locatie, en het aanleveren van de uImage Linux kernel.

```
$ su -
    - of -
$ sudo sh
# mkdir -p /share/tftp
# chown <user>:<group> /share/tftp
# exit
$ cp $SOURCE_DIR/build/linux-omap-2.6/arch/\
$ arm/boot/uImage /share/tftp
```

Hetzelfde als voorheen geldt weer voor de <*user*> en <*group*> aanpassingen die gedaan moeten worden.

9.5 Hardware en Touwtjes

Als het goed is, kan de OMAP nu starten. Voor de volledigheid volgt nog een schema hoe de OMAP aangesloten dient te worden.



10 Ontwikkelomgeving

Voor het gemak worden er twee methodes aangeboden. De eerste methode laat de keus voor editor en projectopbouw compleet vrij. Dit is vooral voor de gevorderde programmeurs die bekend zijn met UNIX software development. De tweede optie is het gebruik maken van een IDE, waarin met een grafische tool het schrijven van software gemakkelijker zou moeten worden.

10.1 Kleinschalig C(++)-project

10.1.1 Preparatie

Een skeletonproject wordt aangeboden. Dit skeletonproject is een voorbeeld van een "Hello, World!" project. Het downloaden van dit startproject is vrij eenvoudig.

```
$ cd $SOURCE_DIR/build/
$ wget -q -O - \
$ http://www.jrrzz.net/~jorrizza/src/arm-el/skel.tar.gz |\
$ tar zxv
```

10.1.2 Programmeren

Code schrijven met behulp van deze toolchain is ook eenvoudig. De Makefile is bijna gelijk aan de veel voorkomende Makefiles in C(++) projecten. Een paar aanpassingen zijn gemaakt, waardoor enkele dingen ontbreken.

Omdat deze Makefile niet direct wordt aangeroepen, zijn er enkele variabelen niet opgegeven. Variabelen zoals *CC*, *LD*, *CFLAGS* en *LDFLAGS* worden aangeleverd door het buildsysteem. Om lokale flags op te geven is in dit voorbeeld de *_LOCAL* suffix gebruikt, maar een += operator werkt natuurlijk ook.

Deze variabelen zijn beschikbaar binnen de Makefile:

- CC
  De C-compiler
- LD
  De linker
- AS
  De assembler
- AR
  De archiver

- CXX
  De C++-compiler
- STRIP
  De binary-stripper
- RANLIB
  De archive indexer
- CFLAGS
  Standaard flags voor de C-compiler
- LDFLAGS
  Standaard flags voor de linker
- HOSTCC
  De C-compiler om binaries te maken die draaien op het hostplatform

10.1.3 Bouwen

Voor het bouwen van custom projecten heeft het buildscript een standaard prefix aangemaakt. Elke keer wanneer een target met de prefix my_ wordt aangeroepen, verwijst het buildscript naar die directory:

```
$ cd $SOURCE_DIR/build
$ ./build.sh my_skel
```

Dit zal de Makefile in de directory *$SOURCE_DIR/build/skel/* aanroepen en de software crosscompilen.

Voor het schoonmaken van de source directory kun je de *clean* target aanroepen. Dit kan simpelweg door het volgende commando uit te voeren:

```
$ cd $SOURCE_DIR/build
$ ./build.sh un_skel
```

10.1.4 Testen

Net als in de voorgaande hoofdstukken gaat het testen van eigengemaakte software ook eenvoudig. Wie had dat gedacht.

Wat kennis over gdb is erg meegenomen op dit punt. Zorg dat de flag *-g* in de *CFLAGS_LOCAL* voor komt bij het bouwen van de source.

Kopieer het binary product van de build naar de target:

```
$ cp $SOURCE_DIR/build/skel/debug \
$ /share/nfs/rootfs2.6/bin/
```

Vervang *skel/debug* natuurlijk even met het product van het echte progsel.

Boot de OMAP en start je binary in de debugger:

```
$ gdbserver 192.168.1.2:2345 debug
```

Vervang debug weer door de echte applicatie. Ja, inderdaad, op deze manier is iedere binary op de OMAP te debuggen.

Op het bouwsysteem kan er een gdb client gestart worden, waarmee de applicatie gedebugd kan worden.

```
$ gdb $SOURCE_DIR/build/skel/debug
(gdb) remote 192.168.1.50:2345
(gdb) start
```

Uiteraard kan *skel/debug* weer vervangen worden. De gdb gebruikt de binary voor het genereren van zinnige meldingen. Deze meldingen zijn er door de *-g* flag in gezet. Zonder deze flag kan gdb alleen machine-leesbare gegevens laten zien.

10.2 Integrated Development Environment

10.2.1 Setup

In deze guide wordt uitgegaan van een bestaande Eclipse-installatie. Zorg ervoor dat de CDT van Eclipse geïnstalleerd is. Eclipse is te vinden op de site, eclipse.org, of in de package manager van je operating system. Eclipse zal ook in je pad aanwezig moeten zijn.

10.2.2 IDE C(++)-project

Er is in de file skel.tar.gz ook een Eclipse-project toegevoegd. Je kunt dit project openen met eclipse. Standaard werkt Eclipse niet samen met het door jezelf gebouwde buildsysteem. Het voorbeeldproject bevat de instellingen waardoor Eclipse *build.sh* gebruikt voor het compilen van de software.

De instellingen van het voorbeeldproject maken gebruik van de omgevingsvariabelen die door het buildsysteem worden ingesteld. Het starten van eclipse gaat dus ook alsvolgt:

```
$ ./build.sh eclipse
```

Let wel: Eclipse is op het moment van schrijven ernstig gebroken op enkele punten en laat zonder het schrijven van een eigen Eclipse-toolchain-plugin het niet toe losse onderdelen te compileren. Ook include-paden kunnen niet ingesteld worden. Code completion zal dus niet of nauwelijk werken.