Debuggen des Linux-Kernel mittels BDI2000 auf dem Board TechnoTrend MDVBDM Rev. 1.0

Dipl.-Inf. (FH) Arkadius Nowakowski <anowa@freenet.de>

13. Juni 2005

Inhaltsverzeichnis

1 Kompilieren des Kernels

Für das Debuggen des Kernels werden zwei Kernel-Images benötigt. Das erste Image soll in das SDRAM des Entwicklungsboards geladen und anschließend gestartet werden. Das zweite Image dient zum Debuggen und befindet sich auf dem Arbeitsrechner. Das Erstellen beider Images erfolgt in der Sand-Box. Nach dem Entpacken des indimp.tgz-Archivs wird die Sand-Box im Verzeichnis `INDIMP/BuildEnv` mit dem Befehl `make enter` betreten. Für den Fall, dass dieses Kommando nicht als Benutzer root abgesetzt wird, erhält der Benutzer eine Eingabeaufforderung für das root-Passwort. Innerhalb der Sand-Box befinden sich die Kernelquellen im Verzeichnis `/av72demo/linux`. Um ggf. Änderungen an der Kernelkonfiguration vorzunehmen, kann hier der Befehl `make menuconfig` aufgerufen werden. Bevor der Kernel jedoch übersetzt werden kann, ist, für das Hinzufügen von Debugging-Informationen, eine Änderung am Makefile erforderlich [1], welches sich im selben Verzeichnis wie die Kernelquellen befindet. Der Compiler-Parameter `-g` muss dazu an die Variable `CFLAGS` angehängt werden. Ist der Editor vi vorhanden, ist ein direkter Sprung auf die richtige Stelle im Makefile mit dem Aufruf `vi Makefile +/CFLAGS.*:=` möglich. Die Definition dieser Variable befindet sich in den Zeilen 96 und 97 und sieht wie folgt aus:

```
96 CFLAGS := $(CPPFLAGS) -Wall -Wstrict-prototypes -Wno-trigraphs -O2 \
97         -fno-strict-aliasing -fno-common
```

Nach dem Hinzufügen des Parameters `-g` ändern sich diese beiden Zeilen in:

```
96 CFLAGS := $(CPPFLAGS) -Wall -Wstrict-prototypes -Wno-trigraphs -O2 \
97         -fno-strict-aliasing -fno-common -g
```

Im folgenden Schritt ist das Bereinigen des Kernelverzeichnises von sämtlichen zuvor erstellten Objektdateien notwendig. Hierzu dient der Aufruf `make clean dep` im Verzeichnis `/av72demo/linux` innerhalb der Sand-Box. Um schließlich den Kernel zu erstellen muss der Befehl `make zImage` aufgerufen werden. Der fertige Kernel mit Debugging-Informationen befindet sich nach Ablauf dieses Befehls in der Datei `vmlinux` und kann vorsichtshalber per Befehl `cp vmlinux vmlinux.debug` gesichert werden. Diese Datei dient anschließend zum Kernel-Debugging auf dem Arbeitsplatzrechner.

Betrachtet man die Datei `vmlinux.debug` hinsichtlich ihrer Größe genauer, fällt auf, dass diese ca. sieben Megabyte Festplattenspeicher einnimmt, was wiederum die Speicherkapazität des Entwicklungsboards unnötig belastet bzw. sogar überschreitet. Die Dateigröße resultiert aus den zusätzlich hinzugefügten Debugging- Informationen, wofür der Parameter `-g` verantwortlich ist. Für das Ausführen des Kernels auf dem Entwicklungsboard sind diese jedoch nicht erforderlich und können somit entfernt werden. Im Vorfeld ist jedoch das Erstellen einer Kopie des Kernels, per Befehl `cp vmlinux.debug vmlinux.debug.strip`, notwendig. Anschließend kümmert sich der Befehl `armeb-elf-linux-gnu-strip vmlinux.debug.strip` um das Entfernen der überflüssigen Debugging-Informationen und Symbole aus dem Kernel.

2 Vorbereiten des Kernels

Bevor der Debugging-Vorgang begonnen werden kann, muss der Kernel ohne Debugging-Informationen in den Speicher des Boards geladen und anschließend gestartet werden. Hierfür sind jedoch vorher einige zusätzliche Schritte notwendig. Das Entwicklungsboard bildet sein SDRAM auf den Adressbereich `0x10000000` bis `0x20000000` ab, d.h. eine Anwendung muss sich ebenfalls innerhalb dieser Grenzen befinden. Innerhalb der Sand-Box setzt der Aufruf `. /bin/armeb-cross` einige Umgebungsvariablen, mit denen die ARM-ELF-Toolchain leichter zugänglich ist. Mittels des Aufrufs von `$OBJDUMP -h vmlinux.debug.strip` im Verzeichnis `/av72demo/linux` erhält man die folgenden Informationen.

```
vmlinux.debug.strip:     file format elf32-bigarm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .init         00012000  c0268000  c0268000  00008000  2**5
                  CONTENTS, ALLOC, LOAD, CODE
  1 .text         0012bb1c  c027a000  c027a000  0001a000  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .kstrtab      000054fc  c03a5b1c  c03a5b1c  00145b1c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 __ex_table    000008f8  c03ab020  c03ab020  0014b020  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 __ksymtab     00002790  c03ab918  c03ab918  0014b918  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 __kallsyms    00000000  c03ae0a8  c03ae0a8  00176c40  2**0
                  CONTENTS
  6 .data         00026c30  c03b0000  c03b0000  00150000  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  7 .bss          0003a8c0  c03d6c40  c03d6c40  00176c40  2**5
                  ALLOC
  8 .comment      00002e9c  00000000  00000000  00176c40  2**0
                  CONTENTS, READONLY
```

Die Spalten `VMA` und `LMA` geben die virtuelle und logische Speicheradresse der Kernel-ELF-Binary an. Wie man hier sehen kann beginnt das Segment `.init` an der Adresse `0xc0268000`. Das Programm erwartet, genau an diese hier angegebenen Adressen in den Speicher geladen zu werden.

Diese stimmen jedoch nicht mit den Speicheradressen des Entwicklungsboards überein. Ein Laden des Images an die Adresse `0x10000000` und anschließendes Starten des Kernels würde somit fehlschlagen, da Sprünge auf ungültige Bereiche des Speichers verweisen würden. Zum Beheben dieses Problems befindet sich im Verzeichnis `/av72demo/linux` das Skript `codecomposer-fixup-kernel`. Es erwartet den Dateinamen des Kernels als Parameter und gebraucht den Aufruf `${OBJCOPY} --adjust-vma 0x50000000` zusammen mit dem zuvor angegebenen Dateinamen. Dieser Befehl addiert den Hex-Wert `0x50000000` auf die, in der ELF-Datei verwendeten, virtuellen Speicheradressen und korrigiert somit die Positionen der einzelnen ELF-Segmente, so dass der Kernel problemlos in das SDRAM des Boards an die Adresse `0x10000000` geladen und anschließend gestartet werden kann. Der Dateiname des modifizierten Kernels besteht aus dem Dateinamen, der beim Aufruf als Parameter übergebenen wurde, und der zusätzlich angehängten Endung `.out`, d.h. der Aufruf `./codecomposer-fixup-kernel vmlinux.debug.strip` legt den veränderten Kernel in der Datei `vmlinux.debug.strip.out` ab. Setzt man den Befehl `$OBJDUMP -h vmlinux.debug.strip.out` ab, so erzeugt dieser die folgende Ausgabe.

```
vmlinux.debug.strip.out:     file format elf32-bigarm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .init         00012000  10268000  10268000  00008000  2**5
                  CONTENTS, ALLOC, LOAD, CODE
  1 .text         0012bb18  1027a000  1027a000  0001a000  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .kstrtab      000054fc  103a5b18  103a5b18  00145b18  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 __ex_table    000008f8  103ab020  103ab020  0014b020  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 __ksymtab     00002790  103ab918  103ab918  0014b918  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 __kallsyms    00000000  103ae0a8  103ae0a8  00176c40  2**0
                  CONTENTS
  6 .data         00026c30  103b0000  103b0000  00150000  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  7 .bss          0003a8c0  103d6c40  103d6c40  00176c40  2**5
                  ALLOC
  8 .comment      00002e9c  50000000  50000000  00176c40  2**0
                  CONTENTS, READONLY
```

Hier sind die Änderungen an den Startadressen der einzelnen Sektionen zu erkennen. Der Bereich `.init` beginnt jetzt an der Adresse `0x10268000` anstatt von `0xc0268000`. Somit befinden sich die Adressen im erlaubten Adressraum und der Kernel in der Datei `vmlinux.debug.strip.out` kann in den Speicher des Entwicklungsboards geladen werden.

Anm.: Das Debuggen des Kernels ist mit unterschiedlich kompilierten Kernel-Images nicht möglich. Es könnte fälschlicher Weise angenommen werden, den Kernel für das Entwicklungsboard ohne dem Parameter `-g` und für den Arbeitsplatzrechner mit Debugging-Informationen zu kompilieren und anschließend den Debugging-Vorgang anzustoßen. Jedoch befinden sich die Sektionen der beiden Kernelvarianten an unterschiedlichen virtuellen und logischen Adressen. Somit erfolgt eine falsche Zuordnung der Zeilennummern im Debugger, welcher auf dem Arbeitsplatzrechner läuft, zu den logischen Adressen der einzelnen Anweisungen im Speicher des Entwicklungsboards.

3 Laden und Starten des Kernels

Das BDI2000 bezieht sämtliche Dateien von einem TFTP-Server, u.A. zur Konfiguration, zum Flashen oder für das Laden von Binärdateien in das SDRAM eines Embedded-Gerätes. In der

hier verwendeten Entwicklungsumgebung, läuft dieser Dienst auf dem gleichen Rechner, auf dem sich auch die Sand-Box befindet, d.h. nicht innerhalb der Sand-Box sondern auf dem Basissystem. Als Distribution kam für die Umgebung Debian-Linux (Sarge) zum Einsatz. Das Debian-Sarge-Paket `tftpd` verwendet standardmäßig das Verzeichnis `/boot` als Arbeitsverzeichnis für den TFTP-Dienst. Somit soll für sämtliche Dateien, welche für die Arbeit mit dem BDI2000 vorgesehen sind, das Unterverzeichnis `/boot/bdi2000` dienen. Dies ist auch der Pfad, in dem sich der Kernel befinden muss. Dazu muss die zuvor erstellte Datei `vmlinux.debug.strip.out` in dieses Verzeichnis kopiert werden. Legt man diese Datei als `/boot/bdi2000/vmlinux` ab, erleichtert dies den Ablauf der folgenden Arbeitsschritte. Unter der Annahme, dass sich die Dateien aus dem entpackten Archiv `indimp.tgz` im Verzeichnis `/home/user1` befinden, liegt die Sand-Box an der Stelle `/home/user1/INDIMP/BuildEnv/sandbox` im Dateisystem. Ausgehend von diesem Verzeichnis befindet sich der, auf das Entwicklungsboard zu ladende, Kernel an der Stelle `av7200/linux/vmlinux.debug.strip.out`. Dieser muss nach `/boot/bdi2000/vmlinux` kopiert werden und steht ab diesem Zeitpunkt dem BDI2000 zum Download zur Verfügung.

Die Steuerung des BDI2000 erfolgt über das Telnet-Protokoll. Hierfür muss zuerst eine Verbindung zwischen Arbeitsplatz und BDI2000 aufgebaut werden. Dazu ist der Aufruf `telnet <ip-address>` erforderlich, wobei die IP-Adresse von der Grundkonfiguration des BDI2000 abhängt. Das BDI2000 ermöglicht das Laden von Daten in das SDRAM des Entwicklungsboards per Befehl `load [<offset>] [<file> [<format]]]`. Der erste Parameter `<offset>` gibt an, an welcher Adresse im Speicher des des Entwicklungsboards, der Ladevorgang angesetzt wird. Die Adresse kann im Hex-Format vorliegen und muss dabei die Zeichen `0x` als Präfix enthalten. Für den Fall, dass der load-Befehl am Anfang des Speichers mit der Adressberechnung beginnen soll, kann als Offset der Wert `0x0` angegeben oder aber auch ganz weggelassen werden. Der zweite Parameter `<file>` steht für den absoluten Pfad auf dem TFTP-Server, an dem sich die zu ladende Datei befindet. Da der Server auf das Verzeichnis `/boot` zugreift und der Kernel somit auf dem Arbeitsrechner an der Stelle `/boot/bdi2000/vmlinux` liegt, ist für `<file>` der Wert `bdi2000/vmlinux` anzugeben. Mit Hilfe des dritten Parameters `<format>` erfolgt die Angabe eines Dateityps. Es stehen hierfür die Kürzel `AOUT`, `BIN`, `COFF`, `ELF` und `SREC` zur Auswahl. Wird ein Typ angegeben, der nicht mit dem Typ, der zu ladenden Datei übereinstimmt, bricht das BDI2000 den Ladevorgang mit einer Fehlermeldung ab. Neben dem Offset-Parameter, hat der Dateityp ebenfalls Einfluss auf die Adresse, an der diese Datei im Speicher abgelegt wird. Der Typ `BIN` z.B. besagt, dass die Daten direkt an die angegebene Offsetadresse geschrieben werden sollen. Beim Typ `ELF` ermittelt das BDI2000 die Startadresse aus den ELF-Sektionen der Datei und addiert diese zum Offset hinzu. Der Linux-Kernel liegt im ELF-Format vor. Somit muss das Kommando `load 0x0 bdi2000/vmlinux ELF` lauten. Nach dem Ladevorgang, befindet sich der Kernel an der Adresse `0x10268000`, da dies die Anfangsadresse der `.init`-Sektion ist und für den den Offset der Wert `0x0` gewählt wurde.

Ruft man den Befehl `load` ohne Parameter auf, greift das BDI2000 auf seine per Konfigurationsdatei definierten Einstellungen zu. Die drei hierfür relevanten Einträge befinden sich in der Sektion `[HOST]`.[1] Der Eintrag `FILE <file>` ist äquivalent zum gleichnamigen Parameter für den `load`-Befehl. Gleiches trifft für den Eintrag `FORMAT <type> [<offset>]` zu. Der dritte Eintrag `LOAD <mode>` steuert das automatische Laden einer Datei und kann entweder mit dem Parameter `AUTO`, für ein sofortiges Laden der Anwendung durch das BDI2000, oder `MANUAL`, für ein manuelles Laden der Anwendung durch den Benutzer, belegt werden. Somit sieht Konfiguration im Bereich `[HOST]` der Konfigurationsdatei wie folgt aus.

```
[HOST]
...
FILE bdi2000/vmlinux
FORMAT ELF 0x0
LOAD MANUAL
...
```

[1] ToDo: hinzufügen der ip vom host.

Das Starten des Kernels erfolgt nach dem erfolgreichen Ausführen des `load`-Befehls, entweder mit oder ohne Parameter, mit dem Kommando `go`.

4 Debuggen des Kernels

Das Kernel-Debugging über das BDI2000 sollte erst erfolgen, nachdem die MMU auf dem Entwicklungsboard initialisiert wurde. Bei dem Aufruf der Kernelfunktion `start_kernel` ist dies bereits der Fall. Aus diesem Grund eignet sich diese besonders gut als erster Breakpoint, der zur Synchronisation mit dem Debugger verwendet werden kann. Für das Setzen des Breakpoints ist jedoch die genaue Adresse der Funktion im virtuellen Adressraum erforderlich. Diese kann innerhalb der Sand-Box, im Verzeichnis `/av72demo/linux`, mit dem Aufruf `nm vmlinux.debug | grep start_kernel` ermittelt werden. Dieser gibt die Meldung `c02685ac T start_kernel` aus und zeigt somit, dass sich die gesuchte Funktion an der Adresse `0xc02685ac` im Speicher befindet.

Nun kann man das BDI2000 zusammen mit dem Entwicklungsboard einschalten. Wichtig hierbei ist, dass zuvor auf dem Board die Jumper JP1, zwischen LED und IR-Empfänger, und JP2, ca. in der Mitte des Boards, gesetzt sind, da sonst das BDI2000 keine Möglichkeit hat, das Board in einen Haltezustand zu versetzen. Ist das BDI2000 aktiviert, kann eine Telnet-Verbindung per Befehl `telnet bdi2000` aufgebaut werden, wobei hierfür in der Datei `/etc/hosts` ein Eintrag für den Rechnernamen `bdi2000` vorhanden sein muss. Anschließend wird der Kernel per `load` geladen, ein Breakpoint an die soeben ermittelte Adresse mit Hilfe des Befehls `bi 0xc02685ac` gesetzt und der Kernel mittels `go` gestartet. Unmittelbar danach, bleibt das System direkt an dieser Adresse stehen und der Breakpoint kann wieder mit dem Befehl `ci` gelöscht werden.

Innerhalb der Sand-Box kann ab diesem Zeitpunkt eine Verbindung mit dem BDI2000 per Debugger aufgebaut werden. Als Hilfestellung, kann hierzu die Debugger-Konfigurationsdatei `/av72demo/linux/.gdbinit` mit folgendem Inhalt erstellt werden:

```
set endian big
define conav
        target remote bdi2000:2001
end
```

Die erste Zeile stellt den Debugger ein, sodass er das Zahlenformat BIG-Endian gebraucht. Die restlichen drei Zeilen definieren ein neues Kommando namens `conav`. Es dient als Abkürzung für die relativ umständliche Eingabe `target remote bdi2000:2001`, die eine Verbindung zwischen Debugger und BDI2000 mittels RDI (Remote Debugging Interface) herstellt. Im nächsten Schritt wird der Befehl `. /bin/armeb-cross` ausgeführt und im Verzeichnis `/av72demo/linux` der Debugger per `armeb-elf-linux-gnu-gdb vmlinux.debug`[2] gestartet. In der Eingabeaufforderung des GDB erfolgt der Verbindungsaufbau mit dem Befehl `conav` und die folgende Meldung wird ausgegeben:

```
start_kernel () at init/main.c:358
358             printk(linux_banner);
warning: shared library handler failed to enable breakpoint
```

Die Zeilennummer 358 gehört zur Funktion `start_kernel`. Sobald die Verbindung besteht, ist die Eingabe von GDB-Befehlen wie `next`, `step`, `break` etc. möglich.

5 NFS-Root

Allgemein gilt der Einsatz von NFS als großer Vorteil bei der Entwicklung von Software für Embedded-Linux-Systeme. Im Vergleich zu anderen Lösungen, bei denen die zu testende Software zuerst per FTP, serieller Verbindung oder ähnlichen Methoden auf das Entwicklungsboard übertragen wird, gewährt NFS einen enormen Zeitvorteil. Unnötige Wartezeiten bei der Übertragung von Daten werden somit überbrückt. Gleiches gilt für die Entwicklung von Linux-Treibern.

[2] TODO: vmlinux.debug.out erzeugen und damit den debugger starten.

5.1 Konfiguration des NFS-Servers

Das Verzeichnis `/av72demo/rootfs_files` innerhalb der Sand-Box beinhaltet die komplette Verzeichnisstruktur, das sog. Root-Filesystem, welches auch im Standalone-Betrieb der Set-Top-Box im Flash-Speicher vorzufinden ist. Dieses Verzeichnis eignet sich somit für den NFS-Export. Der NFS-Server läuft auf dem Basisbetriebsystem und dessen Konfiguration erfolgt in der Datei `/etc/exports`. Für den Export dieses Verzeichnisses ist hier folgender Eintrag notwendig, der sich aus drei Teilen zusammensetzt, welche sich in der gleichen Zeile befinden müssen:

```
/home/user1/INDIMP/BuildEnv/sandbox/av72demo/rootfs_files \
      192.168.124.10(ro,no_root_squash)
```

Der erste Teil ist der Pfad zum Root-Filesystem, welcher komplett angegeben werden muss. Der zweite Teil gibt die IP-Adresse des Rechners an, der Zugriff auf das Verzeichnis erhalten soll. In dem Fall ist dies die IP-Adresse des Entwicklungsboards. Direkt im Anschluss folgt, ohne Leerzeichen, der dritte Teil. Es handelt sich dabei um eine in Klammern eingeschlossene und mit Komma getrennte Liste mit Optionen für den NFS-Export. Das Kürzel `ro` besagt, dass nur ein lesender Zugriff auf die Daten erlaubt ist. Bindet ein Client das exportierte Verzeichnis in seinen Verzeichnisbaum ein, erfolgt in diesem Verzeichnis standardmäßig eine Änderung des Dateieigentümers `root` in eine anonyme User-ID oder in die User-ID der Kennung `nobody`. Damit soll verhindert werden, dass der Benutzer `root` auf dem Client-System root-Rechte für die Daten auf dem Server-System erhält. Dieses Verhalten nennt sich *root squashing* und kann mit der Option `no_root_squash` deaktiviert werden. Bei einem Diskless-Client ist *root squashing* jedoch nicht erwünscht. Somit ist diese Option in der Konfiguration für den NFS-Export des Root-Filesystems erforderlich. Nach dem Eintragen der Konfiguration, ist ein Neustart des NFS-Servers notwendig.

5.2 Konfiguration des Kernels

Der Client soll direkt nach dem Starten des Kernels sein Wurzelverzeichnis über NFS einbinden. Für diesen Zugriff ist eine entsprechende Konfiguration erforderlich, die innerhalb der Sandbox im Verzeichnis `/av72demo/linux` nach dem Aufruf von `make menuconfig` einzutragen ist. Es müssen drei Einstellungen vorgenommen werden. Die erste befindet sich an folgender Stelle:

```
System Type -> AV7200 Options -> Bootup ip settings -> Bootup IP Configuration:
```

Dieser Eintrag stellt die IP-Konfiguration unmittelbar nach dem Kernelstart ein. Er besteht aus sechs Teilen, die durch einen Doppelpunkt voneinander getrennt sind:

```
<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<autoconf>
```

Die genaue Beschreibung der einzelnen Teile befindet sich in der Kerneldokumentation innerhalb der Sandbox unter `/av72demo/linux/Documentation/nfsroot.txt`. Die folgende Konfigurationszeile zeigt die Einstellungen für einen Client mit der IP `192.168.124.10`, der sich mit einem NFS-Server mit der IP `192.168.124.1` verbinden soll. Beide Rechner befinden sich im gleichen Subnetz mit der Make `255.255.255.0`. Hierbei wird davon ausgegangen, dass das Basissystem und somit der NFS-Server unter der IP-Adresse `192.168.124.1` erreichbar ist. Ferner lautet der Hostname des Clients `avbox` und die Schnittstelle, welche mit der IP-Adresse des Clients belegt wird, `eth0`.

```
192.168.124.10:192.168.124.1::255.255.255.0:avbox:eth0:
```

Die zweite Kerneloption befindet sich an folgender Stelle:

```
System Type -> AV7200 Options -> Bootup ip settings -> NFS root:
```

Hier wird der NFS-Server und der Pfad zum freigegebenen Verzeichnis in der Form `[<server-ip>:]<root-dir>[,<nfs-options>]` eingetragen. Die Bedeutung der einzelnen Elemente ist ebenfalls unter `/av72demo/linux/Documentation/nfsroot.txt` zu finden. Beim Auslassen dieser Option, greift der Client auf die soeben eingestellte IP-Adresse des Servers und

auf den Pfad `/tftpboot/%s` zu, wobei `%s` für die Adresse des Clients steht. Wie oben bereits erwähnt, lautet die IP-Adresse des NFS-Servers `192.168.124.1` und der Pfad zum root-Verzeichnis `/home/user1/INDIMP/BuildEnv/sandbox/av72demo/rootfs_files`. Somit ergibt sich der folgende Eintrag:

```
192.168.124.1:/home/user1/INDIMP/BuildEnv/sandbox/av72demo/rootfs_files
```

Die dritte Option muss identisch zur vorhergehenden eingestellt werden. Sie ist im Konfigurationsmenü an folgender Stelle zu finden:

```
System Type -> AV7200 Options -> NFS root fallback -> NFS root:
```

6 Debuggen von Modulen

Bevor ein Treiber bzw. Modul debuggt werden kann, muss es wie gewohnt mit dem Flag `-g` übersetzt werden. Zusätzlich gilt es, weitere Parameter beim Kompileraufruf zu übergeben. Übersetzt man ein Modul für den 2.4er Kernel sind die Parameter bzw. Definitionen `-D__KERNEL__`, `-DMODULE` und `-DLINUX` erforderlich. Außerdem muss der Pfad zu den Kernel-Headern per `-I /av72demo/linux/include` mit angegeben werden. Unter der Annahme, dass sich das zu übersetzende Modul in der Quelldatei `hello.c` befindet, sieht der Kompileraufruf wie folgt aus:

```
armeb-elf-linux-gnu-gcc g -D__KERNEL__ -DMODULE -DLINUX \
        -I /av72demo/linux/include -c hello.c
```

Ein erfolgreicher Kompilerdurchlauf liefert hier die Objektdatei `hello.o`, die wiederum in ein Unterverzeichnis des exportierten NFS-Verzeichnisses kopiert werden kann. Hierfür kann z.B. das Unterverzeichnis `/usr` verwendet werden, d.h. der Kopierbefehl lautet somit `cp hello.o /av72demo/rootfs_files/usr/`. Im nächsten Schritt kann das BDI2000 zusammen mit dem Entwicklungsboard gestartet werden. Nach dem erfolgreichen Laden und Starten des Kernels mittels der BDI2000-Kommandos `load` und `go` sollte das Board auf die NFS-Freigabe auf dem Basissystem zugreifen. Zur Kontrolle, bietet es sich an, diesen Vorgang mit dem Befehl `tail -f /var/log/messages` zu beobachten. Erscheint hier keine Meldung über das Einbinden der Freigabe, liegt ein Fehler vor. Ist das Mounten des Wurzelverzeichnisses über NFS korrekt abgelaufen, kann man sich per Telnet mit dem laufenden Linux-System auf dem Entwicklungsboard verbinden. Damit die Verbindung zwischen Arbeitsrechner und Entwicklungsboard zustande kommen kann, muss die IP-Adresse des Embedded-Systems eingestellt werden. Dies erfolgt in der Datei `/av72000/rootfs_files/etc/rc.boot`. In Zeile 53 wird die Variable `MYIP` gesetzt, die mit der gleichen IP-Adresse zu belegen ist, wie die des soeben konfigurierten NFS-Clients, d.h. die Zeile muss `MYIP=192.168.124.10` lauten. Anschließend kann der Befehl `telnet 192.168.124.10` aufgerufen werden.

Das Listing 1 zeigt die oben genannte Datei `hello.c`, die den Quellcode für das Beispielmodul `hello.o` enthält. In Zeile 18 erfolgt das Erstellen eines Eintrags `hello` im proc-Dateisystem. Bei einem lesenden Zugriff auf `/proc/hello` wird dabei die in Zeile 4 definierte Funktion `proc_print_hello` aufgerufen, welche den String *Hello to you!* ausgibt. Entscheidend für das Debuggen von Modulen ist, dass die zu debuggenden Funktionen als `static` zu deklarieren sind. In diesem Beispiel soll das Debuggen der Funktion `proc_print_hello` demonstriert werden.

```
1 #include <linux/proc_fs.h>
2 #include <linux/module.h>
3
4 static int proc_print_hello(char *buf, char **start,
5         off_t offset, int count, int *eof, void *data)
6 {
7     int len = 0;
8
```

```
 9       len = sprintf( buf, "Hello to you!\n");
10
11       *eof = 1;
12       return len;
13   }
14
15   static int init_module(void)
16   {
17       printk( "<1> Creating /proc/hello ...\n");
18       create_proc_read_entry( "hello", 0, NULL, proc_print_hello , NULL);
19       return 0;
20   }
21
22   static void cleanup_module(void)
23   {
24       printk( "<1> Removing /proc/hello ...\n");
25       remove_proc_entry("hello", NULL );
26   }
```

Listing 1: 'Inhalt der Datei hello.c'

Nun kann das Modul per `insmod -m /usr/hello.o`, über die Telnet-Verbindung mit dem Embedded-System, geladen werden. Durch den Parameter `-m`, liefert der Aufruf zusätzlich die Adressen aller Sektionen und Symbole des Moduls im virtuellen Hauptspeicher des Systems:

```
Using /usr/hello.o
Sections:       Size      Address   Align
.this           00000060  c5ca5000  2**2
.text           0000014c  c5ca5060  2**2
.rodata         00000058  c5ca51ac  2**2
.data           00000000  c5ca5204  2**0
.kstrtab        00000068  c5ca5204  2**0
.bss            00000000  c5ca526c  2**0
.plt            00000028  c5ca5270  2**3
__ksymtab       00000018  c5ca5298  2**2

Symbols:
c5ca5108 t cleanup_module
c5ca51ac r .rodata
c5ca5060 t proc_print_hello
c5ca5000 d __this_module
c5ca5060 T __insmod_hello_S.text_L332
c5ca51ac R __insmod_hello_S.rodata_L88
c5ca526c d .bss
c5ca5000 D __insmod_hello_O/usr/hello.o_M425E360B_V-1
00000000 a hello.c
c5ca5060 t .text
c5ca5138 t create_proc_read_entry
c5ca5204 d .data
c5ca50bc t init_module
```

In der zweiten Hälfte der Ausgabe befindet sich eine Liste mit den Symboladressen des geladenen Moduls. Der dritte Eintrag lautet `c5ca5060 t proc_print_hello`. Dies deutet darauf hin, dass die Funktion `proc_print_hello` an der Adresse `0xc5ca5060` zu finden ist. Als nächstes muss an dieser Adresse ein Breakpoint gesetzt werden. Dazu ist zuerst das BDI2000 per Befehl

`halt` anzuhalten und der Breakpoint p er `bi 0xc5ca5060` zu setzen. Das stehende System kann anschließend wieder per `go` angewiesen werden, weiter zu arbeiten. Wie bereits erwähnt, erfolgt ein Aufruf der Funktion `proc_print_hello` sobald die Datei `/proc/hello` gelesen wird. Somit hält das System, durch das Absetzen des Befehls `cat /proc/hello` über die Telnet-Verbindung, am soeben gesetzten Breakpoint an. Dieser ist ab diesem Zeitpunkt nicht mehr notwendig und kann über die Eingabeaufforderung des BDI2000 per Kommando `ci` entfernt werden.

An dieser Stelle kommt der Debugger in der Sandbox zum Einsatz. Dieser muss im Verzeichnis `/av72demo/linux` mit dem Kernel als Parameter aufgerufen werden:

```
armeb-elf-linux-gnu-gdb vmlinux.debug
```

Da das neue Modul nicht Teil des Kernels ist, benötigt der Debugger die fehlenden Symbolinformationen, die mit dem GDB-Kommando `add-symbol-file` hinzugefügt werden können. Dieses benötigt zwei Parameter. Der erste ist der Pfad zur Objektdatei, welche die Symbolinformationen enthält und der zweite die Adresse des Textsegments. Letztere ist aus der Ausgabe von `insmod -m /usr/hello.o` heraus zu lesen. Die Zeile `c5ca5060 t .text` deutet darauf hin, dass sich dieses Segment an der Adresse `0xc5ca5060` befindet. Somit lautet der GDB-Befehl `add-symbol-file hello.o 0xc5ca5060`, unter der Voraussetzung, dass sich die Datei `hello.o` im gleichen Verzeichnis befindet wie der Kernel. Wenn anschließend der selbst definierte Befehl `conav` abgesetzt wird, kann der GDB sofort nach dem Verbindungsaufbau zum BDI2000 die aktuelle Speicheradresse zur Zeilennummer der Funktion `proc_print_hello` zuordnen. Ab hier kann mit dem eigentlichen Debuggen des Moduls begonnen werden.

Literatur

[1] Debugging Linux with the BDI2000 & bdiGDB
http://www.ultsol.com/PDFS/Tool_Talk_04-002-Debugging_Linux_BDI2000.pdf

[2] bdiGDB - JTAG interface for GNU Debugger
http://www.abatron.ch/Files/ManGDBArm-2000C.pdf