

Linux auf dem ARM7TDMI System-on-Chip Projekt

Ulf Bartel

22. April 2002

Zusammenfassung

Ziel des Projektes war das Erstellen einer funktionsfähigen Entwicklungsumgebung (unter Linux), sowie eines lauffähigen Linux Systems (inkl. Kernel und Rootfilesystem). Aufgrund zeitlichen Ressourcenmangels kam es aber leider nicht zu einer kompletten Umsetzung des Projekts, obwohl sogar ein konfigurierter und cross-kompilierter Kernel als auch ein Rootfilesystem (Rootfs) erstellt werden konnten. Was nun noch fehlt, ist die Aktivierung des On-Board DRAMs, sowie die Inbetriebnahme des seriellen Ports unter Linux (z.B. als Konsole).

Dabei wurde besonders auf eine günstige und einfache Hardware, sowie lizenzfreie Software geachtet. Das Jtag-Interface ist dabei aus wenigen Bauteilen (für etwa fünf Euro) leicht selber nachzubauen und bei der gesamten Software handelt es sich um Open Source Produkte (fast ausschließlich GNU).

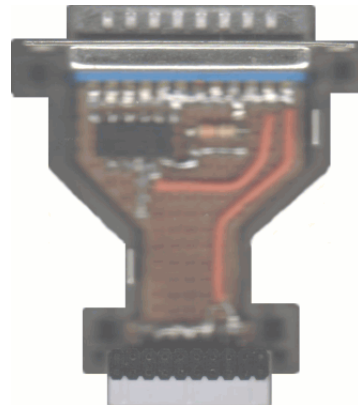
1 Jtag-Interface

Beim Jtag Interface handelt es sich um einen Industriestandard, um z.B. eingebettete Prozessoren und DSPs von einem Host-PC zu debuggen. Dabei ist es möglich, einzelne Register- sowie Speicherinhalte direkt auszulesen oder zu manipulieren. Oft existieren sogar zusätzliche Register (wie hier beim ARM7), zum Beispiel zum erweiterten Debuggen mit Breakpoints.

Da es sich beim Jtag Interface eigentlich nur um eine einfache serielle Schnittstelle handelt, ist die hier vorgestellte Schaltung entsprechend einfach. Man benötigt eigentlich nur einen Treiberbaustein bzw. Pegelkonverter, um das Board direkt mit dem PC verbinden zu können. Das gesamte Protokoll wird dann durch die Software gesteuert.



Gehäuse von oben

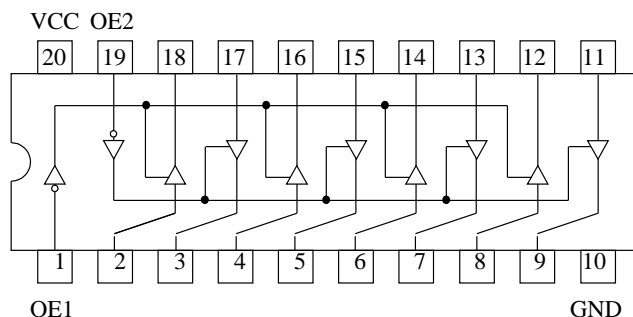


Platinenunterseite (mit 74HC244)

1.1 Schaltung und Aufbau

Als Treiberbaustein kommt ein 8-Bit TriStateBuffer vom Typ 74HC244 zum Einsatz. Verwendet wurde die SMD Variante, welche den Platzbedarf der sowieso sehr kleinen Schaltung noch weiter

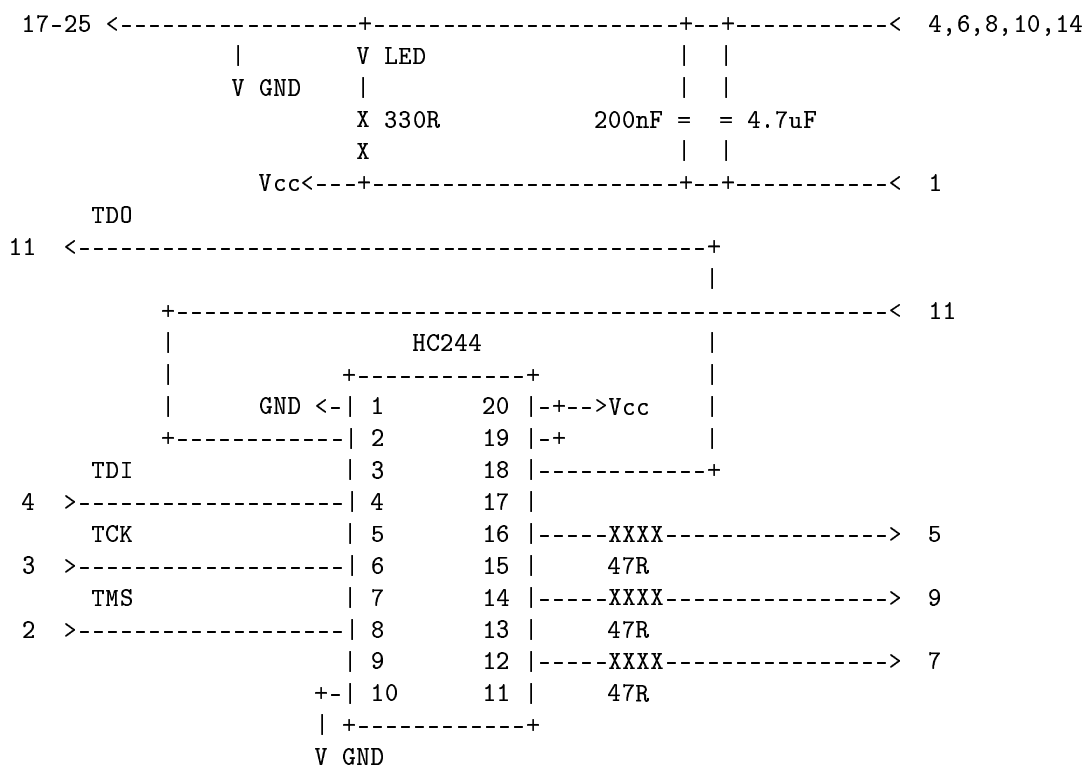
reduziert. Da nur 4 der 8 Treiber des Bausteins benutzt werden, kann fast jeder 2. Pin des Chips entfernt werden, wodurch er sich auch sehr leicht auf 1/10 " Rasterplatten aufbringen laesst. Ich entschied mich, die Pins 3, 5, 7, 9, 11, 13, 15, 17 durch einfaches Abbrechen zu entfernen.



Da die Interfaceschaltung klein ist, kann sie ohne Probleme in das Gehaeuse eines alten RS232-Adaptersteckers integriert werden. Bei der von mir aufgebauten Musterschaltung wurde zusaetzlich eine LED zur Spannungskontrolle des Jtag-Ports integriert. Die Schaltung funktioniert sowohl bei 3.3V als auch bei 5V.

25wayD Male
(PC PRINTER PORT)

14wayIDC
(14-bit JTAG)



Im Folgenden kann man die komplette Pinbelegung der 14- und 20-poligen Pfostenstecker der Jtag-Verbinder sehen. Zum Einsatz kommt hier die 14-polige Variante, wie sie auch auf unserem Developer-Board vorhanden ist.

```

-----
+-----+
| 13   11   9   7   5   3   1 |
| SPU  TD0  TCK  TMS  TDI  NTRST  SPU |
|
| GND  SRST  GND  GND  GND  GND  GND |
| 14   12  10   8   6   4   2 |
+-----+
Top view of 14-bit JTAG header on target

```

```

-----
+-----+
| 19   17   15   13   11   9   7   5   3   1 |
| DBGACK DBGRQ nSRST TD0  RTCK  TCK  TMS  TDI  nTRST VTref |
|
| GND  GND  GND  GND  GND  GND  GND  GND  GND  Vsupply |
| 20   18   16   14   12   10   8   6   4   2 |
+-----+
Top view of 20-bit JTAG header

```

1.2 Test und Inbetriebnahme

Auf der Softwareseite kommt dann das freie (GNU) Programm `armtool`¹ (`/test/armtool/`) von [1] zum Einsatz, welches sich hervorragend auch zur Automatisierung von Abläufen in Shell-Skripten (z.B. zum Initialisieren und Booten) eignet. Aus der erstellten Hardware wurde dann die folgende Konfiguration abgeleitet, welche im File `jtag_io.c` einzutragen ist:

```

/* Parallel port i/o addr. (LPT1) */
#define DATA_PORT    0x378
#define STATUS_PORT  0x379

/* Masks for the parallel port pins */

#define MASK_TMS_BIT  0x01
#define MASK_TCK_BIT  0x02
#define MASK_TDI_BIT  0x04
#define MASK_TDO_BIT  0x10

/* Polarity of signals */
#define INVERT_TMS 0
#define INVERT_TCK 0
#define INVERT_TDI 0
#define INVERT_TDO 0

```

Es beherrscht eigentlich nur die folgenden vier grundlegende Funktionsaufrufe. Einen zum Lesen des Speichers, zwei zum Schreiben und schliesslich einen Aufruf zum Ausführen. Wie wir im Folgenden sehen werden, reicht diese Funktionalitaet aber schon in vielen Situationen aus.

¹Das auf [1] als `tar.gz` erhaeltliche `armtool` hatte einen Fehler (Zeile 117/118 in `arm.c`). Dort war anstatt `MASK_TDO_BIT` die Bit-Maske fest eincodiert, so dass meine ersten Versuche fehlschlagen. Die aktuelle, fehlerbereinigte und erweiterte Version ist im CVS von `anoncvcs@kabel.home.at:/u/cvs/uclinux` zu haben. Sie liegt auch unter `/others/arm-boot(cvs)/`.

```
Usage: ./armtool r{ead}  adr words {filename} ... read from target
       ./armtool w{rite} adr value           ... write a single word to target
       ./armtool w{rite} adr words filename  ... write file contents to target
       ./armtool x{ecute} adr               ... execute program at 'adr'
```

Um die Funktionsfaehigkeit des Interfaces nun endlich zu testen, kann man z.B. versuchen den IDCode des Prozessors auszulesen. Das ist auch dann moeglich, wenn die Xilinx-FPGAs noch nicht programmiert wurden. Der IDCode unserer Prozessors (und scheinbar auch vieler anderer ARM7TDMI-Cores) ist 0x1F0F0F0F. Dazu kommentiert man Zeile 50 von armtool.c ein oder man benutzt direkt 'gdbice' (/test/others/gdbice/tp), welcher unter [3] zu erhalten ist. Ist der Test erfolgreich verlaufen, kann man von einem funktionsfaehigem Interface inklusive korrekt konfigurierter Software ausgehen.

Ein kleines Shellskript (/test/armtool/led1.sh²) kann jetzt auch die korrekte Anbindung des Daten-/Adressbussen des ARM7TDMI an die konfigurierte Hardware zeigen. Dazu kam das .mcs-File aus der Uebungsaufgabe (/test/projekt_dateien/led_ctrl.mcs) zum Einsatz.

```
#!/bin/sh
# clear led_reg
sleep 1 ; ./armtool w 0x000b0004 0x00000000
# set to custom led mode
./armtool w 0x000b0008 0x00000001

#set led_reg
sleep 1 ; ./armtool w 0x000b0004 0x00000001 # set first led
sleep 1 ; ./armtool w 0x000b0004 0x00000003
sleep 1 ; ./armtool w 0x000b0004 0x00000007
sleep 1 ; ./armtool w 0x000b0004 0x0000000f
sleep 1 ; ./armtool w 0x000b0004 0x0000001f
sleep 1 ; ./armtool w 0x000b0004 0x0000003f
sleep 1 ; ./armtool w 0x000b0004 0x0000007f

# set to normal speed
./armtool w 0x000b0000 0x00120000
# go into automatic mode
./armtool w 0x000b0008 0x00000000
```

Zuerst wird das **adr_led_register**, welches den aktuellen Staus der Leuchtdioden enthaelt, geloescht. Die Adresse **2'h01** dieses Registers aus regdef.h bedeutet bei 32-bittigem Datenbus, dass man 4 Bytes zur Basisadresse (**addr_misc3**) hinzuaddieren muss. Daraus entsteht die Adresse 0xb0004 (0xb0000 + 1*4).Danach wird die Schaltung durch Setzen des **adr_mod_register** (0xb0000 + 2*4) auf 1 in den manuellen Modus versetzt. Anschliessend wird eine LED nach der anderen im Sekundentakt angeschaltet. Nachdem alle sieben LEDs leuchten, wird die Schaltung wieder in den automatischen Lauflichtmodus bei normaler Geschwindigkeit zurueckversetzt.

Nun folgt der zum Beispiel (/test/projekt_dateien/FPGA/led_ctrl.vhd) gehoerende Ausschnitt aus regdef.v :

```
parameter addr_misc3 = 32'h000b0000;

// Area addr_misc3
```

²Es existiert dort noch ein aehnliches Shellskript 'led2.sh', welches zum weiteren Testen auch gut geeignet ist.

```
parameter adr_pre_register = 2'h00;
parameter adr_led_register = 2'h01;
parameter adr_mod_register = 2'h02;
parameter adr_out_register = 2'h03;
```

Nach diesem Beispiel kann nun auch von einer funktionierenden FPGA Hardware und korrekter Adress- und Datenanbindung ausgegangen werden. Um aber eigenstaendige Programme direkt auf dem ARM ausfuehren zu koennen, benoetigt man einen Compiler, Linker und weitere Tools, welche direkt Binaercode fuer diesen Prozessor erstellen koennen.

2 Crosscompiler

Zum schnellen Testen und zur Initialisierung der Hardware eignet sich `armtool` sehr gut. So kann man z.B. den DRAM-Controller aktivieren, bevor man ein Programm in den Speicher laedt. Um dieses Programm zu erstellen, benoetigt man einen funktionsfaehigen Cross-Compiler (hier: `gcc` Version 2.95).

2.1 Allgemeines Vorgehen

Als erstes muss man sich die Source-Pakete vom `gcc` und den `binutils` (enthalt `as`, `ar`, `objcopy`, `strip`, ...) von [5] besorgen. Soll ein Linux-System erstellt werden benoetigt man weiterhin mindestens eine C Runtime-Library (`uClibc` od. `newlib`) und den Kernel-Source.

Da die im Kapitel 2.2 beschriebene Variante sehr viel einfacher und problemloser war, werde ich mich hier auf das wesentliche beschraenken.

Nach dem Entpacken aller Pakete beginnt man mit den `binutils` : `./configure --target=arm-uclinux ; make ; make install`. Danach kommt der `Bootstrap-gcc`: `./configure --target=arm-uclinux --enable-languages=c ; make all-gcc CFLAGS+=-Dinhibit_libc ; make install`. Nach dem Setzen der `PATH`-Environmentvariable kann mit dem Uebersetzen der Library oder anderer Programme begonnen werden.

Eine gute und ausfuehrliche Anleitung findet sich aber in Form der PowerPoint-Praesentation von John E. Andrews[12] (`/test/doukuments/sanjose01_triscend.ppt`). Eine HTML-Version davon, sowie die zugehoerigen File (`sanjose01_triscendscript.tar.gz`) befinden sich auch unter `/test/doukuments/`.

2.2 Midori³ fuer ARM7[1]

Bei dieser Minidistribution handelt es sich um eine nicht offizielle Portierung fuer den ARM7TDMI, welche auch von [1] downzuloaden ist. Sie ist sehr einfach zu handhaben, und enthaelt neben dem `gcc` als Crosscompiler, die `binutils` und die `uClibc`[7] (eine abgespeckte Variante der normalen `libc/glibc`). Als Userspaceprogramme unter Linux bietet das Paket unter anderem `'init-1.1'`, `'sash-1.1.1'` (eine kleine Shell), `'telnetd-1.9'`, sowie natuerlich die Multi-Call-Binary `'busybox-0.47'`.

Die Installation geht, wie u.a. im `Linux Enterprise`[6] ausfuehrlich beschrieben, sehr einfach von Statten. Nach dem Entpacken des Hauptpakets (`'midori-1.0.0-beta3.tar.gz'`) werden alle Applikationen im `.mlz`-Format ins `'apps'`-Verzeichnis kopiert. Ein `'make webconfig'` startet nun die Konfiguration, welche ueber einen Webbrowser stattfindet. Ein anschliessendes `'make'` (bzw. `'make root'`) startet dann die vollautomatische Kompilation des `gcc`, der `binutils`, sowie aller benoetigten Libraries und Pakete.

Im Folgenden werde ich beim Erstellen von Binaries von der Benutzung dieses Tools ausgehen. Entsprechend sind die Pfade in `'/test/path'` auch gesetzt. Man muss mit dem Befehl `'source ./path'` vor dem Kompilieren die Umgebungsvariablen setzen.

³Ein Produkt der Firma Transmeta fuer embedded X86-Prozessoren, welches mittlerweile eingestellt wurde.

2.3 Test der Crossdevelopmenttools

Zum Test der frisch erstellten Tools kommt `/others/c/led/led.c` zum Einsatz. Zu Beginn des Files sind die gleichen Registeradressdefinitionen wie in `regdef.v` zu erkennen. Man haette hier natuerlich auch die erzeugte `regdef.h` als Header-File direkt per `'include'` einbinden koennen.

Die leere Funktion `__gccmain()` ist leider noetig um das Programm fehlerfrei (ohne Warnings) kompilieren zu koennen. `'mwait()'` ist nur eine kleine Warteschleife. Die eigentliche Arbeit wird in der Unendlichschleife im Hauptprogramm `'main()'` geleistet. Diese realisiert ein simples Laufflicht, bei der eine LED nach der anderen ueber die Variable `'i'` angeschaltet wird. Die Adressierung erfolgt dabei ueber `'led_reg'` - ein Pointer auf `'LED_BASE'` mit dem Typ `'long'`. Da `'long'` der 32-bit Breite des Datenbusses entspricht, kann man durch ein einfaches Referenzieren von `'led_reg'` als Array, die entsprechenden Register der Hardware direkt ansprechen.

```
#define LED_BASE 0x000b0000 // base adress (addr_misc3)
#define LED_PRE_REG 0x0 // prescaler
#define LED_LED_REG 0x1 // led-bits
#define LED_MOD_REG 0x2 // modus

__gccmain(){

void mwait(int msec){
    long x;
    for (x=1; x<65*msec;x++) ; // delay
}

int main (void){
    int i;
    unsigned long out;
    volatile unsigned long * led_reg;

    led_reg = (unsigned long *) LED_BASE;

    led_reg[LED_MOD_REG]=0x00000001; // set custom led mode

    while (1){ // never stop !
        out=1;
        for (i=0 ; i<7 ; i++){
            led_reg[LED_LED_REG]=out;
            out=out*2;
            mwait(100);
        }
    }

    led_reg[LED_MOD_REG]=0x00000000; // automatic mode
    led_reg[LED_PRE_REG]=0x00120000; // standard speed

return(0);
}
```

Hier nun noch der interessante Teil des dazugehoerigen Makefiles. Nach dem eigentlichen Kompilieren, Linken und Ueberfuehren in den Binaercode (aus der ELF Datei), folgt noch ein Erstellen des zugrundeliegenden ARM-Assemblercodes. Dies geschieht zum einen mittels `'objdump'` aus den `'binutils'` und zum Anderen mittels des `'disarm'[8]` (`/test/disarm`) Disassemblers⁴. Auch hier lohnt

⁴Um das File entsprechend umzuformatieren, habe ich das Tool `'cat_disarm.c'` geschrieben. Es liegt fertig kompiliert im gleichen Verzeichnis wie `'disarm'` selbst.

mal ein Blick in die erzeugten Files.

```
PREFIX = arm-m1-elf-
```

```
all:
```

```
$(PREFIX)as -o setup.obj setup.S
$(PREFIX)gcc -S -o led.S led.c
$(PREFIX)as -o led.obj led.S
$(PREFIX)ld -Ttext 0x380 -o led.elf -Map led.map setup.obj led.obj
$(PREFIX)objcopy -S -O binary led.elf led.bin

$(PREFIX)objdump -d led.elf > led.dis
cat led.bin | cat_disarm | rundisarm > led.disarm
```

Der Upload des frisch erstellten 'led.bin' geschieht nun wie ueblich mittels armtool. Das passende Skript 'go' liegt im selben Verzeichnis.

3 Kernel und Rootfs

Ein betriebsbereites System besteht, neben dem Betriebssystem-Kern (Kernel), aus einer Ansammlung von Tools, die die eigentlichen Aufgaben der Applikation kontrollieren. Diese Programme werden gegen die diversen Bibliotheken (auch crosscompiliert) gelinkt. Bei MMU-losen Linux Systemen ist dieses derzeit leider nur statisch moeglich. Anschliessend werden die Dateien dann aus dem ELF-Format in das wesentlich kompaktere FLAT-Format mittels 'elf2flt' uebersetzt. Alle notwendigen Dateien werden zum Schluss in einem fuer den Kernel mountbaren Format, dem Rootfs, abgelegt.

3.1 Linux Kernel

Um einen Kernel fuer den ARM7 kompilieren zu koennen, muss man zuerst die Originalsources[10] mit dem dazu passendem .diff[5] patchen. Anschliessend kann man dann z.B. mit 'make menuconfig' (in /test/linux-2.4.6-nommu/) alle benoetigten Komponenten auswaehlen. Bei embedded Sytemen mit wenig Speicher sind moeglichst wenige Komponenten sicherlich sinnvoll. Mit 'make dep && make' werden nun die Dependencies und danach der Kernel erstellt. Der kernel 'linux' wird dann mit 'arm-m1-elf-objcopy -Obinary linux linux.bin' ins passende Format uebersetzt. Der ungepackte Kernel erreicht so immerhin bei mir die beachtliche Groesse von knapp 450 KB.

3.2 Rootfs

Ein minimales root Filesystem (als romfs) koennte folgendermassen aussehen (/test/root/):

```
/bin/sh
/bin/init
/bin/df
/bin/ls
/dev/ttyS0
/proc
```

Dabei startet der 'init'-Prozess (/bin/init) - der erste Prozess ueberhaupt - die Shell (/bin/sh) als Konsole auf dem seriellen Port. Ein Login ist dann nicht erforderlich. Dazu muss man dem Kernel vor dem Start aber noch entsprechende init-Informationen uebergeben⁵.

⁵Eine fuer unser Board geeignete Boot-Option koennte folgendermassen aussehen:
root=/dev/rom0 rom=0x?????? console=ttyS0 init=/bin/sh

Ungepackt belegen die Dateien etwa 84 KBytes an Speicher. Etwa ebenso gross werden dann die Rootfs-Image-Dateien, egal ob sie als romfs (mit /test/buildroot/genext2), ramfs oder z.B. ext2 (mit /test/genromfs) formatiert sind. Als weitere Option wuerde sich vor allem das 'compressed ramfs' (cramfs) anbieten.

Bei Benutzung von Midori zum Erstellen des Rootfs, muss man fast ausschliesslich die Dateien aus /test/midori-1.0.0-beta3/cache/root mittels 'genext2fs' oder 'genromfs' in ein Image packen. Vorher sind nur einige Kleinigkeiten zu erledigen, wie z.B. fehlende Device-Dateien oder Bootup-Skripte anlegen.

Den Upload des Kernel und Rootfs kann dann ein kleines Skript erledigen. Eine Beispieldatei fuer den Atmel EB01 ist in test/eb01/ zu finden. Dort kann man auch sehen, wie die Parameterruebergabe zum Kernel funktioniert.

4 Debugging

Wichtig ist eine Moeglichkeit zum Debuggen, die auch dann funktioniert, wenn der Kernel nicht vollstaendig bootet oder die serielle Schnittstelle noch nicht funktionsfaehig (oder ueberhaupt vorhanden) ist. Hier wird die wahrscheinlich einfachste Methode des Debuggens durch printen benutzt. Dabei werden einfach alle Nachrichten in einem vorher reservierten RAM Bereich gespeichert. Das Auslesen erfolgt dann ueber das Jtag-Interface.

Zue Demonstration der Funktionsweise existiert 'arm-text.c' (/test/others/c/arm-text/). Dieses repraesentiert das Hauptprogramm dar, welches die "print"-Funktion aus dem folgenden 'example.c' benutzt.

```
unsigned long *out_size;
unsigned long out_pos;

str_init(unsigned long adr)
{
    out_size = (unsigned long*) adr;
    *out_size = 0;
    out_pos=0;
}

str_add(char *s)
{
    char *p;

    p = (char*) out_size+4;
    p += out_pos;
    while (*s){
        *p++ = *s++;
        out_pos++;
    }
    *out_size = (out_pos+3)>>2; // nr. of words
    *p++='\n';
    *p++='\n';
    *p++='\n';
}

```

Nach dem Initialisieren('str_init()') kann man direkt 'str_add' zum Speichern von Strings benutzen. Nach der Kompilation kann mit dem Upload und Start des Programms beginnen (Shellskript 'go'). Danach zeigt das folgende Skript arm-text.sh (eigentlich ein Perl-Skript) die Nachrichten an.

```

#!/usr/bin/perl

$ARMT00L="/test/arm-boot/armtool";

# print captured textbuffer
# 2100000 textlength in words
# 2100004 start of text

$adr='$ARMT00L r 0x00000800 1';$adr=~s/[\n\ ]//gis;$adr=~s/^\.*:\ ?/0x/;

$len=$adr;$len=$len*1;
printf ("len des buffers : $len\n\n");
print '$ARMT00L r 0x00000804 $adr /dev/stdout'

```

Auf die gleiche Weise kann man diese Methode im Kernel benutzen, indem man die Kernel print-Funktion ('kprintf') entsprechend manipuliert.

Diese Art des Debuggens laesst sich natuerlich auch in anderen Projekten verwenden, bei denen Linux nicht zum Einsatz kommt.

5 Weiterfuehrende Schritte

Um das Projekt jetzt noch zu beenden, muesste man zuerst den DRAM inkl. Controller und den seriellen Port mittels der FPGA realisieren. Mit der in Kapitel 4 entwickelten Methode sollte es relativ leicht fallen einen standart UART unter Linux zu aktivieren. Wenn dann die Konsole ueber die serielle Schnittstelle laeuft, kann man mit einer einfachen Terminalemulation wie 'kermit' oder 'minicom' das gesamte System steuern.

6 Weitere Ideen

Nun folgen noch ein paar interessante Ideen, denen man noch nachgehen koennte:

- Einsatz von gcc-3.0, welcher auch den kompakteren Thumb-Befehlssatz untestuetzt. Ein Problem ist allerdings, dass dieser fuer MMU-lose ARM7 Prozessoren noch nicht "stable" ist.
- Aktivierung des Flash-Speichers als Blockdevice, welches unter der Rubrik 'Memory Technology Devices' (MTD) im Kernelconfig aktiviert werden kann. Dann waere auch ein Booten ohne Host-PC moeglich.
- benutzen von addr2line beim Kerneldebuggen
- bei sehr wenig Speicher eignet sich vielleicht ein Kernel der 2.0 Serie besser
- evtl die Verwendung der 'newlib' anstatt der 'ulibc'
- Journalling-Flash-Filesystem (JFFS2)

7 Anhang (Dateibaum der CD)

Nun folgt abschliessend die Erklaerung des Verzeichnisbaumes der CD. Dieser sollte nach /test kopiert werden, damit die Pfade der Shellskripte (inkl. Environmentvariablen) stimmen. Es werden noch ein paar weitere Programme aufgelistet, die im Text keine Erwaehnung fanden, welche man sich aber auf jeden Fall einmal anschauen sollte. Ein Beispiel ist z.B. 'jtag-arm9'[4] welches auch weiteren Beispielcode enthaelt.

7.1 Struktur und Beschreibung des Filesystems (/test/source_tree.txt):

```
/test/arm-boot/          - Programm zur Ansteuerung des Arm7 Jtag-ports
                        (enthalt armtool sowie die Testprogramme
                        led1.sh* und led2.sh*)
/test/buildroot/        - Tool zum Erstellen eines rootfs (ext2)
                        - generate*
/test/disarm/           - ARM7 Disassembler (enthaelt rundisarm,
                        cat_disarm* sowie disarm*)
/test/documents/       - Dokumente zum ARM7 und Jtag
/test/documents/uni_documents/ - Dokumentationen und Datenblaetter aus der Uni
/test/genromfs/        - Tool zum Erstellen eines rootfs (romfs)
/test/linux-2.4.6-nommu/ - Linux-Kernel Source (schon fertig gepatcht und
                        konfiguriert)
/test/midori-1.0.0-beta3/ - Midori-Entwicklungsumgebung fuer ARM7
/test/midori-1.0.0-beta3/cache/root/ - Enthaelt den von Midori erstellen Verzeichnis-
                        baum fuer das rootfs
/test/others/arm-boot(cvs)/ - aktuellere Version von arm-boot (aus dem CVS)
/test/others/asm/      - Beispiel eines init-files (init.S) in Assembler
/test/others/gdbice/   - weiterer Jtag-Debugger ARM7TDMI (kann
                        Breakpoints/Watchpoints setzen sowie alle
                        Register inkl. ICEBreaker Register aendern)
/test/others/jtag-arm9/ - Jtag-Debugger fuer den ARM9. Die Hard- und
                        Software ist fast identisch mit der fuer den
                        ARM7 (siehe auch Unterverzeichnisse example)
/test/others/c/armtext/arm-text(.c)* - Demonstriert die Funktionsweise der print-
                        Ersatzfunktion
/test/others/c/armtext/example.c     - Kern des Beispielprogramms. Enthaelt die
                        Funktionen 'str_init' und 'str_add'
/test/others/c/armtext/go*           - Upload von arm-text.bin und start
/test/others/c/armtext/arm-text.sh*  - Download des Speicherbereichs und Ausgabe der
                        gespeicherten Nachrichten
/test/others/c/led/go                - Upload von led.bin und start
/test/others/c/led/led(.c, .S, .bin) - Beispielsourcen
/test/projekt_dateien/              - Dateien aus der Windows Entwicklungsumgebung
                                    (led_ctrl.mcs, .bit, .vhd und regdef.v)
/test/projekt_doku/                 - diese Dokumentation
/test/root/                         - Dateien eines minimalen Rootfilesystems
/test/sourcen/                      - Sourcen (meist .tar.gz) der verwendeten Tools
                                    (disarm,genromfs,buildroot ...), sowie des
                                    Kernels (incl. .diff-Files)
/test/sourcen/midori                - Komplette Entwicklungsumgebung der Midori
                                    Distribution fuer ARM7
/test/sourcen/linux_binaries        - weitere Pakete (busybox, tthttpd, tinylogin)
/test/sourcen/eb01                  - Tools fuer den ARM7TDMI basierten ATMEL EB01
/test/path*                          - initialisiert die Umgebungsvariablen (ist je
                                    nach Installationsort anzupassen)
/test/mount*                        - mountet den Samba-Server des Armlab
```

* Eigenentwicklungen

Zum Mounten des Samba-Servers (mittels 'mount') der Uni ist natuerlich ein DHCP-Client wie z.B. 'dhclient' notwendig.

Literatur

- [1] <http://linux.home.at>
- [2] <http://www.redhat.com/embedded/technologies/resources/deblaquiere.pdf>
- [3] <http://sourceforge.net/projects/gdbice>
- [4] <http://jtag-arm9.sourceforge.net/>
- [5] <http://www.uclinux.org/>
- [6] Linux Enterprise, Ausgabe 01/2002, Seite 50, "Das Feintuning stimmt" von Erwin Authried
- [7] <http://uclibc.org/>
- [8] <http://www.banana.demon.co.uk/kevsoft/disarm.zip>
- [9] <http://cvs.lineo.com/>
- [10] <http://kernel.org>
- [11] <http://www.beyondlogic.org/uClinux/>
- [12] <http://www.rtcgroup.com/elinuxexpo/sanjose/conferences2.shtml>