Introduction to Embedded Kernel Programming

Low Level Programming for the ARM920T Processor

Version 1.0

Bachelor Thesis of

Gerald Scharitzer

Vienna University of Technology

(Matriculation Number e0127228)

Abstract

This paper shows the fundamental differences between high level application programming in a hosted and homogeneous environment compared to low level kernel programming in a free standing and heterogeneous environment.

First the hardware configuration, which was used in the course of this project, is specified. This way it should be possible for interested readers to conduct the same experiments at home. The next chapter documents the software configuration, that was used to set up the development environment. With exception of the operating system, all required software packages are freely available on the net. Afterwards the development process of cross building with the GNU ARM tool chain is described. This out-of-the-box tool chain is used, so one can concentrate on programming on the "bare metal" rather than the burden of having to build a tool chain before one can start programming at all.

Further on, the architectural implications for developing an operating system kernel for the ARM Architecture are presented. This introduces the basic building blocks and constraints for the implementation. In the next chapter the programming of some representative peripheral devices of the EDB9302 development board is introduced. These examples show simple drivers, which export low level programming interfaces as callable methods.

Finally, lessons learned from this project are summarized and a list of references and useful resources is provided.

Table of Contents

1 Hardware Configuration........ 4
1.1 Host........ 4
1.2 Target........ 4
1.3 Router........ 5
1.4 Connections........ 5
2 Software Configuration........ 6
2.1 Boot Loader........ 6
2.2 Host Operating System........ 6
2.3 Console........ 6
2.4 Telnet........ 8
2.5 UNIX Environment........ 8
2.6 ARM Tool Chain........ 8
2.7 TFTP Server........ 8
3 Development with GNU ARM........ 10
3.1 Compiling........ 10
3.2 Linking........ 11
3.3 Binary Utilities........ 12
3.4 Loading........ 17
3.5 Executing........ 18
3.6 Debugging........ 18
4 Development for the ARM Architecture........ 19
4.1 Execution Modes........ 19
4.2 Registers........ 19
4.3 Procedure Call Standard........ 21
4.4 Exceptions........ 23
4.5 Memory System Endianess........ 24
4.6 Unaligned Memory Access........ 24
4.7 Memory Management Unit........ 24
4.8 Instruction Memory Barriers........ 25
4.9 Physical Memory Mapping........ 26
4.10 Address Space Mapping........ 26
5 Development for the EDB9302........ 28
5.1 Accessing and Controlling Hardware Registers........ 28
5.2 Turning LEDs on and off........ 28
5.3 Communicating via the Serial Interface........ 29
5.4 Resetting via the Watchdog Timer........ 29
6 Summary........ 30
7 References........ 31
8 Resources........ 32

1 Hardware Configuration

The development environment is based on the following hardware configuration. In the first phase, the host computer controls the target board by a console via the serial interface. After that, the host and target systems are configured to communicate over the LAN, which is provided by the router.

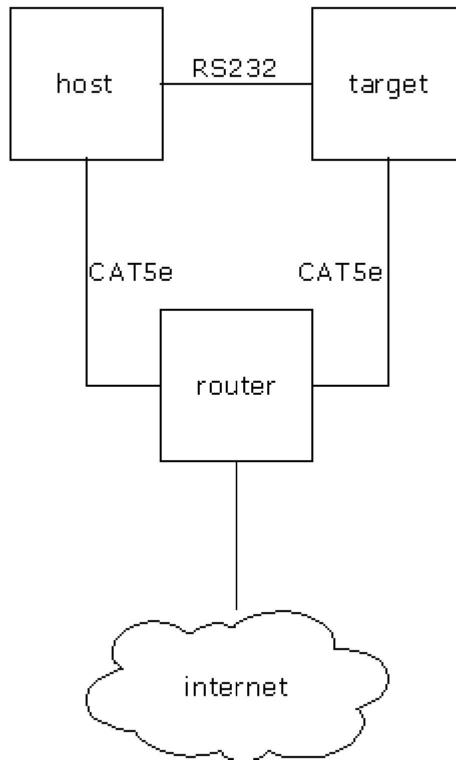

Figure 1: Hardware Configuration

This configuration consists of the following physical components.

- Cirrus Logic EDB9302 Engineering Development Board
- AMD64 Personal Computer
- US Robotics Broadband Router 8000-02
- Serial Null Modem Cable
- 3x CAT5 Cable

1.1 Host

The host computer provides the developing environment and controls the target board. In this context, the host is a personal computer based on an AMD Athlon 64 Processor, which is an implementation of the AMD64 architecture. The serial port (COM1) is used for direct communication with the target, while the LAN is accessed via the ethernet adapter.

1.2 Target

The target is a Cirrus Logic EDB9302 Engineering Development Board, which offers the

following core features.

- EP9302 processor
- 32 MB SDRAM
- 2x UART
- Ethernet MAC

The board also offers numerous other features, which were not exploited in this project though.

The Ethernet MAC subsystem is ISO 802.3 compliant and supports 1, 10 and 100 Mb/s transfer rates.

Before the board is powered on for the first time, make sure that all the jumpers are in their factory default position. These jumper defaults and locations are documented in [EDBTRM].

When the board is powered on, the LEDs POWER ON (red), 10Mb/s (green) and LED1 (green) are turned on, while all other LEDs are turned off. If the board is connected to an active router, the LINK OK LED must turn on. Depending on the configuration, the FULL DUPLEX LED may be active and the board may switch from the 10 Mb/s to the 100 Mb/s LED.

The board can be manually reset with the power-on-reset or the user-reset key.

1.3 Router

The router serves as an internet gateway and enables the LAN. In this context the router is configured to IP address 192.168.123.254 and subnet mask 255.255.255.0.

1.4 Connections

The serial null modem cable is an RS-232 cable with DE-9 connectors and connects the serial port of the host with one serial port of the board. The LAN is built with CAT5e cables for ethernet connections.

2 Software Configuration

2.1 Boot Loader

Originally the board is shipped with Windows CE installed on the flash memory. In the context of this project we installed RedBoot to boot and configure the board.

2.2 Host Operating System

The host operating system is Microsoft Windows XP Professional Service Pack 2. The network configuration of the host is a fixed IP address of 192.168.123.135 with a subnet mask of 255.255.255.0.

2.3 Console

The console session is established using HyperTerminal and is configured as follows.

bandwith................57600 bit/s
data bits................8
parity.....................none
stop bits................1
flow control............none

When the console session is established, a power on reset of the board will cause the boot sequence to appear on the console.

```
+EP93xx - no EEPROM, static ESA, or RedBoot config option.
No network interfaces found

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version v2_0 - built 14:53:37, Nov 10 2005

Platform: Cirrus Logic EDB9302 Board (ARM920T) Rev A
Copyright (C) 2000, 2001, 2002, Red Hat, Inc.

RAM: 0x00000000-0x02000000, 0x00041de8-0x01fdd000 available
FLASH: 0x60000000 - 0x61000000, 128 blocks of 0x00020000 bytes each.
RedBoot>
```

2.3.1 Flash Configuration

The flash configuration (fconfig) can be displayed (-l) with nick names (-n) and full names (-f).

```
RedBoot> fconfig -l -n -f
boot_script: Run script at boot: false
bootp: Use BOOTP for network configuration: true
dns_ip: DNS server IP address: 0.0.0.0
ep93xx_esa: Set eth0 network hardware address [MAC]: false
gdb_port: GDB connection port: 9000
info_console_force: Force console for special debug messages: false
net_debug: Network debug at boot time: false
RedBoot>
```

To enable the board for IP communication, the flash configuration is changed.

```
RedBoot> fconfig
Run script at boot: false
```

```
Use BOOTP for network configuration: false
Gateway IP address: 192.168.123.254
Local IP address: 192.168.123.200
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.123.135
DNS server IP address:
Set eth0 network hardware address [MAC]: true
eth0 network hardware address [MAC]: 0x00:0x00:0x00:0x00:0xE8:0x32
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x60fc0000-0x60fc1000: .
... Program from 0x01fde000-0x01fdf000 at 0x60fc0000: .
RedBoot>
```

Now the new configuration is stored in the flash memory and the board is reset to let the new configuration take effect.

```
RedBoot> reset
... Resetting.
+Ethernet eth0: MAC address 00:00:00:00:e8:32
IP: 192.168.123.200/255.255.255.0, Gateway: 192.168.123.254
Default server: 192.168.123.135, DNS server IP: 0.0.0.0

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version v2_0 - built 14:53:37, Nov 10 2005

Platform: Cirrus Logic EDB9302 Board (ARM920T) Rev A
Copyright (C) 2000, 2001, 2002, Red Hat, Inc.

RAM: 0x00000000-0x02000000, 0x00041de8-0x01fdd000 available
FLASH: 0x60000000 - 0x61000000, 128 blocks of 0x00020000 bytes each.
RedBoot>
```

Now the board is configured to be controlled via the LAN.

```
RedBoot> fconfig -l -n -f
boot_script: Run script at boot: false
bootp: Use BOOTP for network configuration: false
bootp_my_gateway_ip: Gateway IP address: 192.168.123.254
bootp_my_ip: Local IP address: 192.168.123.200
bootp_my_ip_mask: Local IP address mask: 255.255.255.0
bootp_server_ip: Default server IP address: 192.168.123.135
dns_ip: DNS server IP address: 0.0.0.0
ep93xx_esa: Set eth0 network hardware address [MAC]: true
ep93xx_esa_data: eth0 network hardware address [MAC]:
0x00:0x00:0x00:0x00:0xE8:0x32
gdb_port: GDB connection port: 9000
info_console_force: Force console for special debug messages: false
net_debug: Network debug at boot time: false
RedBoot>
```

2.3.2 Network Check

The network configuration is checked with the ping command.

Ping will only work if all firewalls on the host and the router allow the exchange of ICMP Echo Request (8) and Echo Response (0) messages between the host and the board.

The communication from the board to the host is tested with the ping command provided by the boot loader.

```
RedBoot> ping -h 192.168.123.135
Network PING - from 192.168.123.200 to 192.168.123.135
PING - received 10 of 10 expected
RedBoot>
```

The communication from the host to the board is tested from cygwin with the ping command.

```
$ ping 192.168.123.200
PING 192.168.123.200 (192.168.123.200): 56 data bytes
64 bytes from 192.168.123.200: icmp_seq=0 ttl=64 time=0 ms
64 bytes from 192.168.123.200: icmp_seq=1 ttl=64 time=0 ms
64 bytes from 192.168.123.200: icmp_seq=2 ttl=64 time=0 ms
64 bytes from 192.168.123.200: icmp_seq=3 ttl=64 time=0 ms

----192.168.123.200 PING Statistics----
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip (ms)  min/avg/max/med = 0/0/0/0

$
```

Now that the IP network is configured correctly and working the board can be accessed and controlled via the LAN.

2.4 Telnet

When the board is integrated into the LAN it can be controlled through a telnet session. The board provides the telnet service on port 9000.

The telnet session can only be established, when all firewalls on the host and the router allow TCP connections from the host to port 9000 of the board.

```
$ telnet 192.168.123.200 9000

RedBoot>
```

2.5 UNIX Environment

Cygwin is installed on the Windows based host to provide a UNIX environment. The following cygwin packages are used for this project.

bash 3.1-9	The GNU Bourne Again SHell
ping 1.0-1	test IP network connectivity

2.6 ARM Tool Chain

The GNU ARM tool chain enables the host to cross compile for the ARM architecture and provides a set of platform specific tools.

2.7 TFTP Server

The host runs a TFTP server, which makes available the executable program files to the board.

The board can only load programs from the host if all firewalls on the host or the router allow UDP messages between the host port 69 and the board.

When the programs are stored in the root directory of the TFTP server and adhere to the ELF format, they can be loaded with the load command provided by RedBoot.

```
RedBoot> load main
Entry point: 0x00100000, address range: 0x00100000-0x00100150
RedBoot>
```

When RedBoot loads a program via TFTP it reads the transmitted ELF file only as far as it needs to successfully load the program into memory. Therefore the load command will most likely not receive the entire ELF file, which will cause the TFTP server to time out the download and report an unsuccessful transmission. Nonetheless the program was loaded successfully.

```
RedBoot> go 0x100000
ping
```

When the loaded program is executed with the "go" command, it can not return control to RedBoot, because "go" performs a "transfer control" rather than a "call subroutine". Therefore the main routine should, instead of returning to the caller, set the watchdog timer and wait in an empty loop for the reset to occur.

RedBoot's "exec" command is used to boot a linux kernel image and therefore expects an accordingly bound program to be loaded in memory.

3 Development with GNU ARM

The GNU ARM tool chain provides a set of tools to cross build for the target arm-elf. In this project the following tools are used.

arm-elf-gcc	The GNU Compiler Collection
arm-elf-ld	The GNU linker
arm-elf-objdump	Display information from object files
arm-elf-readelf	Display the contents of ELF format files

3.1 Compiling

The compiler is invoked with the following set of options.

-c	compile the source file to an object file
-g	insert debug information
-v	verbose output
-fpic	generate position independent code
-ffreestanding	compile a freestanding application
-mcpu=arm920t	ARM920T processor instruction set
-Wa,-a=<filename>	save assembler listing as <filename>

In this case the -ffreestanding option is rather a comment, than a processing option, because the language semantics of c++ include the allocation and release of heap storage, which is a function provided by hosted environments.

The -mcpu option causes the assembler to stick to the instruction set, which can be executed by the specified processor.

```
arm-elf-gcc -c -g -v -ffreestanding -mcpu=arm920t \
-Wa,a=<asmlist> -o <output> <input>
```

3.1.1 Hosted Environments

In a hosted environment, an entire standard library is available and the program is invoked via its main function, which returns control to the caller and returns an exit code of type int. This is the case for most applications.

The GNU ARM tool chain incorporates the newlib, which is a port of the standard library designed for embedded systems. This library respects the characteristic hardware configuration of such systems.

The newlib can be easily ported to new environments, since it relies only on the implementation of a set of few rather simple functions, which all other library functions are built upon.

3.1.2 Freestanding Environments

In a freestanding environment, there may be no standard library and the program may be invoked at an arbitrary entry point. This is usually the case for an operating system kernel, which sets up his own environment.

3.2 Linking

The object file is the bound by invoking the appropriate linker.

```
arm-elf-ld -static --verbose -T<script> -o <output> <input>
```

3.2.1 The Linker Script

The default linker script is replaced by a user defined script to reduce the size of the executable program file. The main entry point of the program is defined with the ENTRY statement. In this case the entry point is set directly to the main function. This is required by the loader to report, where it has loaded the entry point to.

```
ENTRY(main)
```

The start address is set to 0x100000 to avoid conflicts with the memory allocation of the boot loader. Otherwise the loader will terminate with the following message.

```
*** Abort! Attempt to load ELF data to address: 0x00000000 which is not in
RAM
```

RedBoot prints its RAM allocation at start up and these boundaries are stored in the symbols FREEMEMLO and FREEMEMHI. These can be used to explicitly load programs into available storage areas.

```
RAM: 0x00000000-0x02000000, 0x00041de8-0x01fdd000 available
RedBoot> load -b %{FREEMEMLO} main
Address offset = 0x00042000
Entry point: 0x00042000, address range: 0x00042000-0x00042290
RedBoot>
```

The debug sections are predefined to reserve enough space in the program header. This is necessary if the program is compiled with the -g option to store debug information in the object file. The DWARF debug interface expects the debug sections to start at address 0. This will not interfere with any memory allocation at address 0, because the debug sections are only read by the debugger and not allocated by the loader.

```
SECTIONS {
      . = 0x100000;
      .text   : { *(.text) }
      .data   : { *(.data) }
      .bss    : { *(.bss) }
      .rodata : { *(.rodata) }

      /* stabs debugging sections */
      .stab          0 : { *(.stab) }
      .stabstr       0 : { *(.stabstr) }
      .stab.excl     0 : { *(.stab.excl) }
      .stab.exclstr  0 : { *(.stab.exclstr) }
      .stab.index    0 : { *(.stab.index) }
      .stab.indexstr 0 : { *(.stab.indexstr) }
      .comment       0 : { *(.comment) }
      /* DWARF debug sections
            Symbols in the DWARF debugging sections are relative to the
beginning
            of the section so we begin them at 0.  */
      /* DWARF 1 */
      .debug          0 : { *(.debug) }
      .line           0 : { *(.line) }
```

```
        /* GNU DWARF 1 extensions */
        .debug_srcinfo  0 : { *(.debug_srcinfo) }
        .debug_sfnames  0 : { *(.debug_sfnames) }
        /* DWARF 1.1 and DWARF 2 */
        .debug_aranges  0 : { *(.debug_aranges) }
        .debug_pubnames 0 : { *(.debug_pubnames) }
        /* DWARF 2 */
        .debug_info     0 : { *(.debug_info) }
        .debug_abbrev   0 : { *(.debug_abbrev) }
        .debug_line     0 : { *(.debug_line) }
        .debug_frame    0 : { *(.debug_frame) }
        .debug_str      0 : { *(.debug_str) }
        .debug_loc      0 : { *(.debug_loc) }
        .debug_macinfo  0 : { *(.debug_macinfo) }
        /* SGI/MIPS DWARF 2 extensions */
        .debug_weaknames 0 : { *(.debug_weaknames) }
        .debug_funcnames 0 : { *(.debug_funcnames) }
        .debug_typenames 0 : { *(.debug_typenames) }
        .debug_varnames  0 : { *(.debug_varnames) }
}
```

If the debug sections are not predefined and the program is compiled with the debug option, the linker will issue the following error message.

```
arm-elf-ld: Not enough room for program headers
```

If the debug sections do not start at address 0, then GDB issues the following error message.

```
Dwarf Error: bad offset (0x100000) in compilation unit header
```

The linker is invoked with the -T option to us the specified linker script instead of the default linker script.

```
arm-elf-ld -T<linker-script>
```

The default linker script can be extracted, when the linker is invoked with the --verbose option.

3.3 Binary Utilities

3.3.1 readelf - Display the contents of ELF format files

The readelf tool can be used to analyze the executable file. This will reveal the entry point address, which will be used to initialize the instruction pointer at program invocation. Since ELF is designed to be portable, the header also specifies, whether the data in the file is encoded with big or little endian byte order. An executable file is ready to be loaded and run.

```
$ arm-elf-readelf -a -W main
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            ARM
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x100000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          69944 (bytes into file)
  Flags:                             0x202, has entry point, GNU EABI, software FP
```

```
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           40 (bytes)
  Number of section headers:         18
  Section header string table index: 15
```

Section Headers

The sections flagged with "A" will be allocated in memory by the loader. An operating system kernel must load sections flagged with "X" into executable memory pages and sections flagged with "W" into writable memory pages. For increased safety, read only sections should be loaded into write protected memory pages. This distribution of programs in memory can only be done, if every class of sections is assigned its own set of memory pages.

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00100000 008000 000268 00  AX  0   0  4
  [ 2] .glue_7           PROGBITS        00100268 008268 000000 00  AX  0   0  4
  [ 3] .glue_7t          PROGBITS        00100268 008268 000000 00  AX  0   0  4
  [ 4] .data             PROGBITS        00100268 008268 000000 00  WA  0   0  1
  [ 5] .bss              NOBITS          00100268 010268 000000 00  WA  0   0  1
  [ 6] .rodata           PROGBITS        00100268 008268 000028 00   A  0   0  4
  [ 7] .comment          PROGBITS        00000000 010268 000048 00      0   0  1
  [ 8] .debug_aranges    PROGBITS        00000000 0102b0 000080 00      0   0  1
  [ 9] .debug_pubnames   PROGBITS        00000000 010330 000169 00      0   0  1
  [10] .debug_info       PROGBITS        00000000 010499 000642 00      0   0  1
  [11] .debug_abbrev     PROGBITS        00000000 010adb 0002ec 00      0   0  1
  [12] .debug_line       PROGBITS        00000000 010dc7 000134 00      0   0  1
  [13] .debug_frame      PROGBITS        00000000 010efc 000180 00      0   0  4
  [14] .debug_str        PROGBITS        00000000 01107c 00000e 00      0   0  1
  [15] .shstrtab         STRTAB          00000000 01108a 0000ab 00      0   0  1
  [16] .symtab           SYMTAB          00000000 011408 000330 10     17  29  4
  [17] .strtab           STRTAB          00000000 011738 00015a 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Program Headers

Only executable and shared object files have program headers. The LOAD segments specify the parts of the file, which are to be loaded into memory.

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x008000 0x00100000 0x00100000 0x00290 0x08268 RWE 0x8000

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata

There is no dynamic segment in this file.

There are no relocations in this file.

There are no unwind sections in this file.
```

Symbol Table

The -W option is required to print the entire name of long symbols. In the symbol table, the mangled names of methods can be observed, where the parameter types are encoded in the symbol name.

```
Symbol table '.symtab' contains 51 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00100000     0 SECTION LOCAL  DEFAULT    1
     2: 00100268     0 SECTION LOCAL  DEFAULT    2
     3: 00100268     0 SECTION LOCAL  DEFAULT    3
     4: 00100268     0 SECTION LOCAL  DEFAULT    4
     5: 00100268     0 SECTION LOCAL  DEFAULT    5
     6: 00100268     0 SECTION LOCAL  DEFAULT    6
     7: 00000000     0 SECTION LOCAL  DEFAULT    7
     8: 00000000     0 SECTION LOCAL  DEFAULT    8
     9: 00000000     0 SECTION LOCAL  DEFAULT    9
    10: 00000000     0 SECTION LOCAL  DEFAULT   10
    11: 00000000     0 SECTION LOCAL  DEFAULT   11
    12: 00000000     0 SECTION LOCAL  DEFAULT   12
    13: 00000000     0 SECTION LOCAL  DEFAULT   13
    14: 00000000     0 SECTION LOCAL  DEFAULT   14
    15: 00000000     0 SECTION LOCAL  DEFAULT   15
    16: 00000000     0 SECTION LOCAL  DEFAULT   16
    17: 00000000     0 SECTION LOCAL  DEFAULT   17
    18: 00000000     0 FILE    LOCAL  DEFAULT  ABS main.cpp
    19: 00100000     0 FUNC    LOCAL  DEFAULT    1 $a
    20: 00100030     0 OBJECT  LOCAL  DEFAULT    1 $d
    21: 00000000     0 FILE    LOCAL  DEFAULT  ABS GPIOE.cpp
    22: 00100034     0 FUNC    LOCAL  DEFAULT    1 $a
    23: 00100058     0 OBJECT  LOCAL  DEFAULT    1 $d
    24: 0010005c     0 FUNC    LOCAL  DEFAULT    1 $a
    25: 00100080     0 OBJECT  LOCAL  DEFAULT    1 $d
    26: 00100084     0 FUNC    LOCAL  DEFAULT    1 $a
    27: 001000d8     0 OBJECT  LOCAL  DEFAULT    1 $d
    28: 001000dc     0 FUNC    LOCAL  DEFAULT    1 $a
    29: 00100130     0 OBJECT  LOCAL  DEFAULT    1 $d
    30: 00000000     0 FILE    LOCAL  DEFAULT  ABS UART1.cpp
    31: 00100134     0 FUNC    LOCAL  DEFAULT    1 $a
    32: 0010017c     0 OBJECT  LOCAL  DEFAULT    1 $d
    33: 00100184     0 FUNC    LOCAL  DEFAULT    1 $a
    34: 00000000     0 FILE    LOCAL  DEFAULT  ABS WatchdogTimer.cpp
    35: 001001cc     0 FUNC    LOCAL  DEFAULT    1 $a
    36: 001001fc     0 OBJECT  LOCAL  DEFAULT    1 $d
    37: 00100200     0 FUNC    LOCAL  DEFAULT    1 $a
    38: 00100230     0 OBJECT  LOCAL  DEFAULT    1 $d
    39: 00100234     0 FUNC    LOCAL  DEFAULT    1 $a
    40: 00100264     0 OBJECT  LOCAL  DEFAULT    1 $d
    41: 00100084    88 FUNC    GLOBAL DEFAULT    1 _ZN6ep93015GPIOE11setGreenLedEb
    42: 00100134    80 FUNC    GLOBAL DEFAULT    1 _ZN6ep93015UART17putCharEc
    43: 00100184    72 FUNC    GLOBAL DEFAULT    1 _ZN6ep93015UART19putStringEPKc
    44: 00100234    52 FUNC    GLOBAL DEFAULT    1 _ZN6ep930113WatchdogTimer7restartEv
    45: 001001cc    52 FUNC    GLOBAL DEFAULT    1 _ZN6ep930113WatchdogTimer7disableEv
    46: 00100200    52 FUNC    GLOBAL DEFAULT    1 _ZN6ep930113WatchdogTimer6enableEv
    47: 00100000    52 FUNC    GLOBAL DEFAULT    1 main
    48: 0010005c    40 FUNC    GLOBAL DEFAULT    1 _ZN6ep93015GPIOE9getRedLedEv
    49: 00100034    40 FUNC    GLOBAL DEFAULT    1 _ZN6ep93015GPIOE11getGreenLedEv
    50: 001000dc    88 FUNC    GLOBAL DEFAULT    1 _ZN6ep93015GPIOE9setRedLedEb

No version information found in this file.
```

The mangled symbol names in the symbol table can be demangled with the nm or c++filt utility to obtain the originating function names.

3.3.2 objdump – Display information from object files

The object files can also be inspected with the objdump tool.

-x display all headers
-d disassemble executable sections

The main difference from readelf is the capability to disassemble sections.

```
$ arm-elf-objdump -x -d main

main:     file format elf32-littlearm
main
architecture: arm, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00100000
```

The output of objdump is a little closer to the ELF format and states, which bits are set rather than interpreting them.

Program Headers

```
Program Header:
    LOAD off    0x00008000 vaddr 0x00100000 paddr 0x00100000 align 2**15
         filesz 0x000002b0 memsz 0x00008288 flags rwx
private flags = 202: [APCS-32] [FPA float format] [software FP] [has entry point]
```

Section Headers

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000027c  00100000  00100000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .glue_7       00000000  0010027c  0010027c  0000827c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .glue_7t      00000000  0010027c  0010027c  0000827c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .data         00000000  0010027c  0010027c  0000827c  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  4 .got          00000000  0010027c  0010027c  0000827c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .got.plt      0000000c  0010027c  0010027c  0000827c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .bss          00000000  00100288  00100288  00010288  2**0
                  ALLOC
  7 .rodata       00000028  00100288  00100288  00008288  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .comment      00000048  00000000  00000000  00010288  2**0
                  CONTENTS, READONLY
  9 .debug_aranges 00000080  00000000  00000000  000102d0  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_pubnames 00000169  00000000  00000000  00010350  2**0
                  CONTENTS, READONLY, DEBUGGING
 11 .debug_info   00000642  00000000  00000000  000104b9  2**0
                  CONTENTS, READONLY, DEBUGGING
 12 .debug_abbrev 000002ec  00000000  00000000  00010afb  2**0
                  CONTENTS, READONLY, DEBUGGING
 13 .debug_line   00000134  00000000  00000000  00010de7  2**0
                  CONTENTS, READONLY, DEBUGGING
 14 .debug_frame  00000180  00000000  00000000  00010f1c  2**2
                  CONTENTS, READONLY, DEBUGGING
 15 .debug_str    0000000e  00000000  00000000  0001109c  2**0
                  CONTENTS, READONLY, DEBUGGING
```

Symbol Table

```
SYMBOL TABLE:
00100000 l    d  .text  00000000
```

```
0010027c l    d  .glue_7        00000000
0010027c l    d  .glue_7t       00000000
0010027c l    d  .data  00000000
0010027c l    d  .got   00000000
0010027c l    d  .got.plt       00000000
00100288 l    d  .bss   00000000
00100288 l    d  .rodata        00000000
00000000 l    d  .comment       00000000
00000000 l    d  .debug_aranges 00000000
00000000 l    d  .debug_pubnames        00000000
00000000 l    d  .debug_info    00000000
00000000 l    d  .debug_abbrev  00000000
00000000 l    d  .debug_line    00000000
00000000 l    d  .debug_frame   00000000
00000000 l    d  .debug_str     00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  *ABS*  00000000
00000000 l    df *ABS*  00000000 main.cpp
00100288 l       .rodata        00000000 .LC0
00100000 l     F .text  00000000 $a
00100040 l     O .text  00000000 $d
00000000 l    df *ABS*  00000000 GPIOE.cpp
00100048 l     F .text  00000000 $a
0010006c l     O .text  00000000 $d
00100070 l     F .text  00000000 $a
00100094 l     O .text  00000000 $d
00100098 l     F .text  00000000 $a
001000ec l     O .text  00000000 $d
001000f0 l     F .text  00000000 $a
00100144 l     O .text  00000000 $d
00000000 l    df *ABS*  00000000 UART1.cpp
00100148 l     F .text  00000000 $a
00100190 l     O .text  00000000 $d
00100198 l     F .text  00000000 $a
00000000 l    df *ABS*  00000000 WatchdogTimer.cpp
001001e0 l     F .text  00000000 $a
00100210 l     O .text  00000000 $d
00100214 l     F .text  00000000 $a
00100244 l     O .text  00000000 $d
00100248 l     F .text  00000000 $a
00100278 l     O .text  00000000 $d
00100098 g     F .text  00000058 _ZN6ep93015GPIOE11setGreenLedEb
00100148 g     F .text  00000050 _ZN6ep93015UART17putCharEc
00100198 g     F .text  00000048 _ZN6ep93015UART19putStringEPKc
00100248 g     F .text  00000034 _ZN6ep930113WatchdogTimer7restartEv
001001e0 g     F .text  00000034 _ZN6ep930113WatchdogTimer7disableEv
00100214 g     F .text  00000034 _ZN6ep930113WatchdogTimer6enableEv
00100000 g     F .text  00000048 main
00100070 g     F .text  00000028 _ZN6ep93015GPIOE9getRedLedEv
0010027c g     O .got.plt       00000000 _GLOBAL_OFFSET_TABLE_
00100048 g     F .text  00000028 _ZN6ep93015GPIOE11getGreenLedEv
001000f0 g     F .text  00000058 _ZN6ep93015GPIOE9setRedLedEb
```

Disassembly

The following paragraph shows the disassembly of the main function in the .text section. Disassembling ARM instructions is rather straightforward, since the ARM instruction set is a fixed length instruction set and therefore the instruction boundaries are clear throughout the entire disassembly. Even though data words can not be distinguished from code words, the disassembly of data words does not induce any errors, because these instructions are never executed. A variable length instruction set as in the z/Architecture [ZAPOO] can not be correctly disassembled on a per instruction basis. This is because of the data bytes, which can

corrupt the instruction boundaries, when they are interpreted as instructions. In this case, the control flow must be analyzed, to differentiate between data and code.

```
Disassembly of section .text:

00100000 <main>:
  100000:       e1a0c00d        mov     ip, sp
  100004:       e92ddc00        stmdb   sp!, {sl, fp, ip, lr, pc}
  100008:       e24cb004        sub     fp, ip, #4      ; 0x4
  10000c:       e59fa02c        ldr     sl, [pc, #44]   ; 100040 <.text+0x40>
  100010:       e08fa00a        add     sl, pc, sl
  100014:       e59f3028        ldr     r3, [pc, #40]   ; 100044 <.text+0x44>
  100018:       e08a3003        add     r3, sl, r3
  10001c:       e1a00003        mov     r0, r3
  100020:       eb00005c        bl      100198 <_ZN6ep93015UART19putStringEPKc>
  100024:       e3a00000        mov     r0, #0  ; 0x0
  100028:       eb00001a        bl      100098 <_ZN6ep93015GPIOE11setGreenLedEb>
  10002c:       e3a00001        mov     r0, #1  ; 0x1
  100030:       eb00002e        bl      1000f0 <_ZN6ep93015GPIOE9setRedLedEb>
  100034:       eb000083        bl      100248 <_ZN6ep930113WatchdogTimer7restartEv>
  100038:       eb000075        bl      100214 <_ZN6ep930113WatchdogTimer6enableEv>
  10003c:       eafffffe        b       10003c <main+0x3c>
  100040:       00000264        andeq   r0, r0, r4, ror #4
  100044:       0000000c        andeq   r0, r0, ip
[...]
```

The last two instructions are actually read-only constants, which are stored locally in the code segment of the owning function.

3.3.3 nm - List symbols from object files

The mangled method names are translated back, when the objdump or nm tool is invoked with the -C option.

```
$ nm -C main
00100200 T ep9301::WatchdogTimer::enable()
001001cc T ep9301::WatchdogTimer::disable()
00100234 T ep9301::WatchdogTimer::restart()
00100034 T ep9301::GPIOE::getGreenLed()
00100084 T ep9301::GPIOE::setGreenLed(bool)
0010005c T ep9301::GPIOE::getRedLed()
001000dc T ep9301::GPIOE::setRedLed(bool)
00100134 T ep9301::UART1::putChar(char)
00100184 T ep9301::UART1::putString(char const*)
```

3.3.4 c++filt - Demangle encoded C++ symbols

Specific mangled symbols can be demangled with the utility c++filt.

```
$ arm-elf-c++filt _ZN6ep93015UART19putStringEPKc
ep9301::UART1::putString(char const*)
```

3.4 Loading

The loader allocates the memory areas for the program sections, which are required at run time. In doing so, the loader has to respect the attributes of the sections defined in the section headers of the ELF file. The contents of the code (.text, .glue, ...) and data (.data, .rodata, ...) sections are copied from the file into memory. On the contrary, the .bss section is only allocated in memory and then set to all zeros.
Several sections also have an alignment constraint, which informs the loader, that it must

allocate the section on a specific boundary. This boundary is usually an integral power of 2 in bytes. The .text section is allocated on a 4-byte boundary, because this is also the instruction word size.
The section headers also specify, whether they are executable or read-only. This can be respected by the loader by allocating the sections to memory pages with the appropriate permissions set.

3.5 Executing

Execution of the program is initiated by setting the program counter to the entry point of the program. Before this can be done, the execution context has to be set up for the program. First of all, this means setting the stack and heap pointer to the corresponding memory addresses.

3.6 Debugging

3.6.1 gdb - The GNU Debugger

As long as the application running on the target board can not host a gdb session, the board can only be debugged remotely. To debug a program with gdb, the program must be linked with the debug stubs for gdb. These in return require a C runtime environment, which is not always available.

4 Development for the ARM Architecture

4.1 Execution Modes

The ARM architecture defines 7 different execution modes, where 6 of these are privileged modes and 5 are exception modes.

mode	short	privileged	exception	mode bits
user	usr			0b10000
system	sys	x		0b11111
supervisor	svc	x	x	0b10011
abort	abt	x	x	0b10111
undefined	und	x	x	0b11011
interrupt	irq	x	x	0b10010
fast interrupt	fiq	x	x	0b10001

Table 1: Execution Modes

Only privileged modes can execute privileged instructions and are therefore reserved for kernel programs. Exception modes interrupt the normal control flow and link to the established exception handler. The mode bits identify the execution mode in the status registers.

4.1.1 Supervisor Mode

The supervisor mode is only entered, when a software interrupt or a reset exception occurs. This is also the execution mode, in which the processor begins execution after power on.

4.2 Registers

An ARM processor provides 31 general purpose registers and 6 program status registers.

4.2.1 General Purpose Registers

Among the general purpose registers, there are some, which are more special than the others. There exists a second set of registers 8 to 12, which belong to the fast interrupt mode. This contributes to the implementation of fast interrupts, by keeping the interrupt handler from having to save and restore these registers.

The Stack Pointer

Register 13 is called the stack pointer and points to the current extent of the stack. Every exception mode has a stack pointer of its own, which points to the exception specific stack.

The Link Register

Register 14 is called the link register and stores the return address for calls and exceptions. For every exception mode, there also exists a private link register, which stores the address to return from the exception handler.

The Program Counter

Register 15 is called the program counter, which points to the next instruction to be executed. Therefore, load and store operations have different semantics, when they are performed on the program counter. Reading the program counter returns the address of the read instruction plus an implementation defined offset. Writing to the program counter causes the control flow to

jump to the address just written.

nr	register	description	usr	sys	svc	abt	und	irq	fiq
1	R0	general purpose register 0	x	x	x	x	x	x	x
2	R1	general purpose register 1	x	x	x	x	x	x	x
3	R2	general purpose register 2	x	x	x	x	x	x	x
4	R3	general purpose register 3	x	x	x	x	x	x	x
5	R4	general purpose register 4	x	x	x	x	x	x	x
6	R5	general purpose register 5	x	x	x	x	x	x	x
7	R6	general purpose register 6	x	x	x	x	x	x	x
8	R7	general purpose register 7	x	x	x	x	x	x	x
9	R8	general purpose register 8	x	x	x	x	x	x	
10	R9	general purpose register 9	x	x	x	x	x	x	
11	R10	general purpose register 10	x	x	x	x	x	x	
12	R11	general purpose register 11	x	x	x	x	x	x	
13	R12	general purpose register 12	x	x	x	x	x	x	
14	R13	stack pointer	x	x					
15	R14	link register	x	x					
16	PC	program counter	x	x	x	x	x	x	x
17	R13_svc	supervisor stack pointer			x				
18	R14_svc	supervisor link register			x				
19	R13_abt	abort stack pointer				x			
20	R14_abt	abort link register				x			
21	R13_und	undefined stack pointer					x		
22	R14_und	undefined link register					x		
23	R13_irq	interrupt stack pointer						x	
24	R14_irq	interrupt link register						x	
25	R8_fiq	fast interrupt register 8							x
26	R9_fiq	fast interrupt register 9							x
27	R10_fiq	fast interrupt register 10							x
28	R11_fiq	fast interrupt register 11							x
29	R12_fiq	fast interrupt register 12							x
30	R13_fiq	fast interrupt stack pointer							x
31	R14_fiq	fast interrupt link register							x

Table 2: General Purpose Registers

Dedicated Stacks

Since every execution mode can be interrupted by exceptions with higher priority, every execution mode requires a stack base of its own. On the other hand, the set of stacks for the privileged execution modes is only required for every processor and not every thread of

execution. The only exception mode, which does not require a stack of its own is the reset mode, since it transfers control to the kernel rather than returning to a caller.

4.2.2 Program Status Registers

The control bits of the status registers are especially important to the kernel. The I bit is set to disable interrupts and the F bit is set to disable fast interrupts. The mode bits indicate the current execution mode.

nr	register	description	usr	sys	svc	abt	und	irq	fiq
1	CPSR	current program status register	x	x	x	x	x	x	x
2	SPSR_svc	supervisor saved program status register			x				
3	SPSR_abt	abort saved program status register				x			
4	SPSR_und	undefined saved program status register					x		
5	SPSR_irq	interrupt saved program status register						x	
6	SPSR_fiq	fast interrupt saved program status register							x

Table 3: Program Status Registers

The save program status registers hold the contents of the current program status register, before the exception handler was entered. Before the exception handler returns control, it restores the CPSR by loading it with its copy of the SPSR.

4.3 Procedure Call Standard

The procedure call standard for the ARM architecture [ARMAPCS] further defines the usage of the general purpose registers.

register	symbol	description
0	a1	argument / result / scratch register 1
1	a2	argument / result / scratch register 2
2	a3	argument / scratch register 3
3	a4	argument / scratch register 4
4	v1	variable register 1
5	v2	variable register 2
6	v3	variable register 3
7	v4	variable register 4
8	v5	variable register 5
9	v6 SB TR	platform register
10	v7	variable register 7
11	v8	variable register 8
12	IP	intra procedure call scratch register
13	SP	stack pointer
14	LR	link register
15	PC	program counter

Table 4: Procedure Call Standard

While most of these register allocations are conventions, which must be respected, if one wants to be compatible to the rest of the world, some of these are actually implemented by the hardware. For example, writing to register 15 on an ARM processor will always result in a branch to address written to the register.

4.3.1 Stack Pointer

The stack is an area of contiguous memory which is bounded by the stack base and the stack limit. For a full descending stack the stack pointer is initialized to the stack base and decremented by the frame size for every subroutine call. At all times the following constraints must be met.

- The stack pointer must point within the memory area allocated to the stack.
  stack limit < stack pointer <= stack base
- The stack pointer must be aligned on word boundary.
  stack pointer mod 4 = 0
- The stack may only be accessed within the interval of its current extent.
  [stack pointer, stack base – 1]
- The stack pointer for a public interface must be aligned on double word boundary.
  stack pointer mod 8 = 0

The implementation of a contiguous stack implies, that the distribution of the stacks for several threads in the address space is managed efficiently to avoid collisions, since the only way to extend a stack is to decrease the stack limit. An alternative stack model can be found in [ZOSLEPG] (Chapter 2 – Linkage Conventions) and [ZOSPASG] (Chapter 14 – Stack and Heap Storage), where only stack frames must be contiguous, while stack segments can be scattered

across the address space.

4.3.2 Link Register

The link register stores the return address, which is to be branched to when returning control from the subroutine. It usually contains the address of the instruction (NSI) right after the branch instruction, which invoked the subroutine. NSI is an acronym and stands for "Next Sequential Instruction".

4.3.3 Program Counter

The program counter contains the address, which was initially branched to when the subroutine was invoked and thus marks the entry point of the subroutine.

4.3.4 Argument and Result Registers

The first 4 registers are dedicated for passing argument and return values between the caller and the subroutine. This enables efficient subroutine parameter passing, since the arguments can be passed via the fast registers instead of having to write and read them from the slow memory of the stack.

4.4 Exceptions

The handling of exceptions is controlled by their priorities and their disabling via the current status register.

exception	execution mode	priority	vector
reset	supervisor	1 (highest)	0x00000000
data abort	abort	2	0x00000010
fast interrupt	fast interrupt	3	0x0000001c
interrupt	interrupt	4	0x00000018
prefetch abort	abort	5	0x0000000c
undefined instruction	undefined	6 (lowest)	0x00000004
software interrupt (SWI)	supervisor	6 (lowest)	0x00000008

Table 5: Exceptions

Whenever an exception is raised, normal interrupts are disabled. Fast interrupts are only disabled by a reset or a fast interrupt.

4.4.1 Exception Vectors

The exception vectors are the first 8 4-byte addresses, which are branched to, when the corresponding exception occurs. Therefore the instructions at the exception vectors must be branch instructions to the appropriate exception handlers.

```
B   reset      ; branch to reset handler
B   undefined  ; branch to undefined instruction handler
B   swi        ; branch to software interrupt handler
B   prefetch   ; branch to prefetch abort handler
B   data       ; branch to data abort handler
B   reset      ; this exception vector is reserved
B   interrupt  ; branch to interrupt handler
B   fast       ; branch to fast interrupt handler
```

4.4.2 Exception Priorities

Whether an exception can occur is controlled via the "fast interrupts disabled" and "interrupts disabled" bit in the current program status register. All other exceptions can occur at any point in time. The exception priorities control, in which sequence the corresponding exception handlers are invoked.

4.4.3 Nested Exceptions

exception execution	reset	data abort	fast interrupt	interrupt	prefetch abort	undefined SWI
user	x	x	x	x	x	x
system	x	x	x	x	x	x
supervisor	x	x	1		x	x
interrupt	x	x	2		x	x
fast interrupt	x	x			x	x
abort	x	x	2		x	x
undefined	x	x	2		x	x

Table 6: Execution Modes vs Exception Priorities

1. Fast interrupts are disabled, when the supervisor mode is entered because of a reset exception. When the supervisor mode is entered because of a software interrupt, then the fast interrupt enabled bit remains unchanged.
2. Fast interrupts are only disabled, if a reset or fast interrupt occurs. Otherwise the fast interrupts enabled bit remains unchanged.

4.5 Memory System Endianess

If the standard system control processor is used, which supports both big and little endian memory systems, then the endianess bit in register 1 of control processor 15 must be set correctly, before any halfword or byte access is performed on the memory system.

```
MRC p15,0,r0,c1,c0 ; r0 = cp15 r1
ORR r0,r0,#0x80    ; set big endian bit
MCR p15,0,r0,c1,c0 ; cp15 r1 = r0
```

4.6 Unaligned Memory Access

The result of an unaligned memory access is implementation dependent and may even be unpredictable. The meaning of "unpredictable" is constrained, such that it may not cause any security holes or cause any part of the system to halt. Otherwise all programs executed on the system would have to be trusted. Nonetheless the kernel must check addresses from untrusted components, such that it does not produce unpredictable results while in kernel mode.

4.7 Memory Management Unit

The memory is partitioned by the MMU into sections and pages, where 3 different page sizes are supported. Sections can be mapped with a single table lookup, while page mapping requires a two step translation.

1 section = 1 MB
1 large page = 64 KB
1 small page = 4 KB
1 tiny page = 1 KB

The first level translation table requires 16KB and must be allocated on a 16KB boundary. Second level translation tables can be coarse page tables of 1KB size or fine page tables of 4KB size. Therefore one has to make a trade off between coarse memory mapping and wasting physical memory for large translation tables.

4.8 Instruction Memory Barriers

Whenever memory is modified before execution, the code is called self modifying code. This is the case for every operating system, that supports loading programs into memory. Before such modified code is executed, an instruction memory barrier (IMB) must be executed. This guarantees, that the modification of the instruction memory does not interfere with the instruction prefetch.

4.8.1 Global IMB

The execution of a global IMB guarantees, that all accessible memory locations are eligible for instruction fetching. For a global IMB the recommended instruction is

```
SWI 0xF00000
```

and the C signature of the function should look like this.

```
void IMB(void);
```

4.8.2 Local IMB

The execution of a local IMB ensures valid instruction fetches of previously modified memory locations only for a specified range of addresses. For a local IMB the recommended instruction is

```
SWI 0xF00001
```

and the C signature should look like this,

```
void local_IMB(unsigned long start, unsigned long end);
```

where "start" and "end" denote the boundaries of a half open interval of addresses. The specified memory area ranges from "start" (inclusive) to "end" (exclusive).

4.9 Physical Memory Mapping

The previous chapters define the following physical memory mapping.

address	offset + 0x0	offset + 0x4	offset + 0x8	offset + 0xc	area
0x00000000	`B reset`	`B undefined`	`B swi`	`B prefetch`	exception
0x00000010	`B data`	`B reset`	`B interrupt`	`B fast`	vectors
reset	reset	handler			exception
undefined	undefined	instruction	handler		handlers
swi	software	interrupt	handler		
prefetch	prefetch	abort	handler		
data	data	abort	handler		
interrupt	normal	interrupt	handler		
fast	fast	interrupt	handler		
kernel	kernel	module			
global_imb	global	instruction	memory	barrier	software
local_imb	local	instruction	memory	barrier	interrupts
16 KB block on 16 KB boundary	primary	address	translation	table	
4 KB blocks on 4 KB boundary	fine	address	translation	tables	secondary address
1 KB blocks on 1 KB boundary	coarse	address	translation	tables	translation tables
data stack	data	abort	handler	stack	exception
fast stack	fast	interrupt	handler	stack	handler
interrupt stack	normal	interrupt	handler	stack	stacks
prefetch stack	prefetch	abort	handler	stack	
undef swi stack	undefined	instruction	and SWI	stack	
kernel stack	supervisor	execution	mode	stack	

Table 7: Physical Memory Mapping

The first 8 words contain pointers to the exception handlers. These handlers along with their stacks are also allocated in physical memory. The nucleus also contains the system calls of the kernel. If virtual memory is to be used, then at least one 16 KB block is required for the primary address translation table. Fine grained virtual memory allocation additionally requires one or more secondary address translation tables.

These physical memory areas must be set up, before virtual memory is turned on. Furthermore these memory locations are kernel level objects and must be protected by the MMU.

4.10 Address Space Mapping

An address space is split into common and private areas. The common areas are mapped to the same physical addresses in all address spaces. These addresses are used to communicate between address spaces via shared memory and to invoke system calls provided by the kernel. The private areas contain the code and data, which belong to the address space.

address	length	content
0x00000000	4	branch to reset handler
0x00000004	4	branch to undefined instruction handler
0x00000008	4	branch to software interrupt handler
0x0000000c	4	branch to prefetch abort handler
0x00000010	4	branch to data abort handler
0x00000014	4	branch to reset handler
0x00000018	4	branch to interrupt handler
0x0000001c	4	branch to fast interrupt handler
reset		reset handler
undefined		undefined instruction handler
swi		software interrupt handler
prefetch		prefetch abort handler
data		data abort handler
interrupt		interrupt handler
fast		fast interrupt handler
imb		instruction memory barriers

Table 8: Address Space Mapping

An address space switch can be done by modifying the address translation tables during runtime. This way the private areas of different address spaces can be mapped to different physical addresses.

5 Development for the EDB9302

The development for the EDB9302 board is based on the EP9301 User's Guide [EP9301UG]. In this example we access the following peripheral interfaces.

- General Purpose IO (GPIO) port E
- Universal Asynchronous Receiver Transmitter (UART) 1
- Watchdog Timer

These interfaces are accessed via physical addresses, which do not map to any memory locations, but instead specify the hardware registers of the peripheral units.

5.1 Accessing and Controlling Hardware Registers

Hardware registers must be accessed through pointers to volatile fields, because they do not follow the read and write semantics of exclusive memory locations. The volatile declaration causes the compiler to really read the register, whenever it is referenced and write the register, whenever it is assigned. This is necessary, because hardware registers can change independently of the program flow and accessing them can trigger the underlying hardware components. Similar concepts are used for Direct Memory Access (DMA) and memory mapped IO.

There exist hardware registers, which can be accessed correctly only in an integral manner. This means, that the entire register must be read or written in a single instruction rather than processing it as a sequence of bytes. Examples are single LOAD and STORE instructions of half words, full words or double words instead of multiple LOADs and STOREs of single bytes.

Furthermore, the maximum address resolution on many architectures is on byte boundary. Therefore, whenever single bits must be modified without affecting the other register contents, the register must be set in a read-modify-write operation.

5.2 Turning LEDs on and off

The green and red LEDs (GRLED and RDLED) are driven by pins 0 and 1 of the GPIO port E. They are turned on and off by simply setting and resetting the corresponding bits 0 and 1 of the GPIO port E data register, which is assigned to address 0x80840020.

```
// General Purpose IO Port E Data Register
#define PEDR (*((volatile unsigned int *) 0x80840020))
#define GRLED 0x1
#define RDLED 0x2

void GPIOE::setGreenLed(bool b) {
  if (b)
    PEDR |= GRLED; // turn green led off
  else
    PEDR &= ~GRLED; // turn green led off

}

void GPIOE::setRedLed(bool b) {
  if (b)
    PEDR |= RDLED; // turn red led off
  else
    PEDR &= ~RDLED; // turn red led off
}
```

5.3 Communicating via the Serial Interface

When the transmit FIFO buffer is not full, then a write to the UART1Data register will cause the least significant byte of the register to be transmitted over the serial interface.

```
// UART 1 Data Register
#define UART1Data (*((volatile unsigned int *) 0x808c0000))

// UART 1 Flag Register
#define UART1Flag (*((volatile unsigned int *) 0x808c0018))
// Transmit FIFO Full
#define TXFF 0x20

void UART1::putChar(const char c) {
  while (UART1Flag & TXFF); /* polling */
  UART1Data = c & 0xff; /* transmit byte */
}
```

Instead of wasting CPU cycles with polling the UART1 flag register, the communication can be performed with interrupts or direct memory access.

5.4 Resetting via the Watchdog Timer

When the watchdog timer is enabled and not restarted within its time out period, then a reset pulse is generated. The watchdog register is assigned to address 0x80940000 and enabled by writing 0xaaaa to it.

```
// watchdog control register
#define Watchdog (*((volatile unsigned int *) 0x80940000))
#define ENABLE_WATCHDOG  0xaaaa
#define DISABLE_WATCHDOG 0xaa55
#define RESTART_WATCHDOG 0x5555

void WatchdogTimer::restart(void) {
  Watchdog = RESTART_WATCHDOG;
}

void WatchdogTimer::enable(void) {
  Watchdog = ENABLE_WATCHDOG;
}

WatchdogTimer::restart();
WatchdogTimer::enable();
while(1); // loop while waiting for reset
```

6 Summary

The operating system kernel manages the allocation of hardware resources and is therefore inherently architecture dependent. On top of the kernel, the operating system can implement a hardware abstraction layer to provide a consistent and architecture independent programming interface to applications.

Several features provided by the operating system kernel require explicit hardware support. If the operating system wants to make sure, that it will receive control again after passing it to an application, it will need timer based interrupts. To make the kernel trustworthy, it has to be protected from erroneous or malicious application code. For this it will need a memory management unit, that controls all memory accesses.

The hardware architecture and communication protocols define the framework in which the kernel can be implemented. From the bottom, the hardware architecture defines the atomic building blocks (instructions, registers, ...), which are available to the implementer. Top down, the kernel interface defines, which algorithms and data structures are to be implemented by the underlying hardware.

The operating system must provide an execution environment to application programs. An application program is compiled and linked for a specific runtime environment. When invoked, the program simply starts executing, because it is assumed, that the runtime environment is already set up by the operating system. This runtime environment consists of all the resources, which were allocated to the application. Since resource allocation may only occur in supervisor mode, this is the job of the operating system. Application programs may only be executed in problem mode to protect the operating system and other applications.

7 References

[ARM920T] ARM920T (Rev 1) Technical Reference Manual
http://www.arm.com/pdfs/DDI0151C_920T_TRM.pdf

[ARM9TDMI] ARM9TDMI (Rev 3) Technical Reference Manual
http://www.arm.com/pdfs/DDI0180A.zip

[ARMABI] ARM Application Binary Interface
http://www.arm.com/products/DevTools/ABI.html

[ARMARM] ARM Architecture Reference Manual
http://www.arm.com/miscPDFs/14128.pdf

[ARMAPCS] ARM Architecture Procedure Call Standard
http://www.arm.com/miscPDFs/8031.pdf

[EDB9302] EDB9302 Documentation
http://arm.cirrus.com/files/schematics/edb9302/

[EDBTRM] EDB9302 Technical Reference Manual
http://arm.cirrus.com/files/schematics/edb9302/2_Technical%20Reference%20Manual/Technical%20Reference%20Manual%20EDB9302.pdf

[EP9301UG] EP9301 User's Guide
http://www.cirrus.com/en/pubs/manual/EP9301_User_Guide.pdf

[L4KA] L4KA Pistachio Microkernel
http://l4ka.org/projects/pistachio/

[L4RM] L4 Version X.2 Reference Manual
http://l4ka.org/projects/pistachio/l4-x2-r5.pdf

[RBUG] RedBoot User Guide
http://ecos.sourceware.org/docs-latest/redboot/redboot-guide.html

[RVCTAG] RVDS 3.0: RVCT 3.0 Assembler Guide
http://www.arm.com/pdfs/DUI0204G_rvct_assembler_guide.pdf

[ZAPOO] z/Architecture Principles of Operation
http://publibz.boulder.ibm.com/epubs/pdf/a2278324.pdf

[ZOSLEPG] z/OS V1R8.0 Language Environment Programming Guide
http://publibz.boulder.ibm.com/epubs/pdf/ceea2170.pdf

[ZOSPASG] z/OS V1R8.0 MVS Programming Assembler Services Guide
http://publibz.boulder.ibm.com/epubs/pdf/iea2a660.pdf

8 Resources

ARM tool chain	GNU ARM Toolchain http://gnuarm.org/files.html
boot loader	RedBoot http://sourceware.org/redboot/
L4 kernel	L4KA Pistachio http://l4ka.org/download/
Linux for Windows	Cygwin http://cygwin.com/
TFTP server	Solar Winds TFTP Server http://www.solarwinds.net/freetools.htm