Diploma Thesis
University of Applied Sciences Augsburg
Department of Computer Science

Open On-Chip Debugger

Design and Implementation of an On-Chip Debug Solution for Embedded Target Systems based on the ARM7 and ARM9 Family

Submitted by Dominic Rath, summer semester 2005
Examiner: Prof. Dr. Hubert Högl
Examiner: Prof. Burkhard Stork

Diploma Thesis
University of Applied Sciences Augsburg
Department of Computer Science

I affirm that the diploma thesis is my own work, and that it has never been submitted for examination purposes before. All sources and citations used have been quoted as such, and all utilized tools have been mentioned.

_____

Dominic Rath

Open On-Chip Debugger

an On-Chip Debug Solution for Embedded Target Systems based on the ARM7 and ARM9 Family

Dominic Rath

10 09 08 07 06 05 6 5 4 3 2 1

First edition:	11 July 2005
Second impression, with corrections:	18 July 2005

Contents

List of Figures

List of Tables

1 Embedded Systems Debugging

Embedded systems in general, and ARM based system-on-chip (SOC) designs in particular, have seen an immense growth during the past years, with free and open software becoming an integral part of embedded systems development. A survey ran by linuxdevices.com [ELMS05] shows that 43% of the participants have used embedded Linux in their, or their companies, products, and 55% expect to do so within the next two years. The processor architecture used in most designs is ARM, being used by 30% of the developers, prior to x86 which has been used by 28%. Open source tools are the first choice for 59% of the participants, and more than 82% believe the tools available for embedded Linux development are either very good or acceptable.
While free and open source projects offer a high-quality toolchain for ARM development, debugging support is lacking behind, especially as far as system programming is concerned. The GNU Debugger (gdb) offers excellent debugging support, but covers only some areas of embedded systems debugging. Low level tasks require additional hard- and software, and existing open source solutions for these tasks are incomplete or at least partially deficient.
The goal of this diploma thesis is the design and implementation of a free solution for debugging of ARM7 and ARM9 family based SOC designs, making the use of proprietary commercial tools obsolete. The software written as part of this work is initially going to have support for selected members of these processor families, but extensibility to additional cores shall be simplified by an appropriate architectural design. The target interface will be based upon the IEEE Standard Test Access Port and Boundary-Scan Architecture [IEEE1149]. Support for different interfaces between a host PC and an IEEE 1149.1 compatible target is an expressed goal.

Debugging embedded systems is different in many aspects from traditional application debugging. Compared to desktop systems, embedded systems have limited resources, such as main memory, processing power, or input and output capabilities. This makes it inconvenient or even impossible to run a software debugger together with the debuggee on the same system. Depending on the development task, there might be no software running on the target at all, like during bootloader development. In that case, there is usually no debugger on the system, too. During application development, the hardware is expected to be error-free, meaning that every subsystem (CPU, memory, storage, I/O) actually works. On embedded systems, the hardware itself could have errors, like an instable memory interface or untested system components. If the memory system is faulty or just untested, any code could fail, even a debugger.
Because of these restrictions, embedded systems are usually debugged using remote debugging: The debugger is running on a host computer, and controls the target either through hardware, or through a small software running on the target. It's possible for the developer to make use of all the comfort his workstation offers, while the target doesn't have to run a full featured debugger.
The purpose of debugging is to identify and remove defects in software programs. This can be achieved

by either passively watching the code-, and possibly the data flow, or by actively stopping the target at the point of interest. Passive debugging has the advantage of being non-intrusive, and allows program flow and timing to be inspected, while active debugging enables the developer to control the program flow, or alter the contents of target memory.
Some years ago, program code for embedded systems had to be loaded onto memory chips using external programmers. The chips had to be removed from the target, programmed, and put back into the system. Being able to download program code from the host to a target system while this is running greatly simplifies embedded systems development.

The following sections are meant to show some commonly used solutions for embedded debugging in genearal, and an overview of currently available solutions for ARM7/ARM9 debugging in particular. [Asb01] gives an detailed review of the challenges of embedded systems design, the implications for debugging, and the various debug solutions used in embedded systems design.

1.1 Debug Solutions

Logic Analyzers, Trace Hardware

Logic analyzers and dedicated trace hardware, like the *ARM embedded trace macrocell* (ETM) [IHI0014J], allow the program flow to be passively monitored. Logic analyzers monitor the target's data and address bus, and usually generate a listing of executed instructions, possibly annotated with the data accessed. While this works for older microcontrollers, where every instruction executed results in an access to the memory system, it's not possible to trace execution within modern cached architectures. Instructions contained inside the cache wont show up on the memory interface, making a complete trace impossible. Dedicated trace hardware is tightly coupled to the microcontroller core, and keeps track of every instruction executed, without having to rely on the memory interface. The amount of raw data can grow rapidly on systems with high clock rates, making it difficult to get the information out of the target, and hard to find the relevant parts. Advanced solutions like the ETM9 allow triggerpoints, for example instruction addresses, to be defined, at which the trace hardware starts monitoring the core. Filters further limit the amount of data that has to be transfered from the core to the debugging host.

In-Circuit Emulators, ROM Emulators

An in-circuit emulator (ICE) replaces the target microcontroller with a special debug variant, that includes hardware debugging facilities. The emulator is connected to a host computer which runs the debugger software. This allows both passive and active debugging, giving a non-intrusive view of the program flow, and allowing fine control over program execution, CPU state and memory contents. Read Only Memory (ROM) emulators substitute target non-volatile memory with dual-ported Random Access Memory (RAM) modules, that can be accessed from a debugger and the target at the same time. Where code has to be run from ROM this allows a debugger to replace instructions with hooks necessary for debug entry, like TRAP or Software Interrupt (SWI) instructions. Code testing is improved, as the memory chips don't have to be programmed with external tools. An ICE might support hardware breakpoints, where address comparators constantly monitor the address bus, and force the system into debug state when an address matches during an instruction fetch. This allows breakpoints to be set on code contained in ROM without using a ROM emulator. If the ICE further provides overlay memory, it's

possible to load code into the target, replacing instructions contained in ROM regions. The ICE watches the accessed memory space, and switches to it's included RAM when an access to overlayed memory occurs.

Debug Stubs

Debug stubs, often called "debug monitors", run on the target system, and connect to a host computer running the debug software. They require working initialization code, that sets up the target clocks, main memory, and a communication channel. This makes a Debug stub unsuitable for early development stages, where initialization code has to be debugged itself.
The stub utilizes an interrupt on the target to take control over program execution, a stub using RS232 communication for example would use the serial interrupt vector. When the host debugger sends data to the debugging stub, an interrupt is generated, giving the stub control over the target. The stub uses some kind of TRAP or BREAKPOINT instruction, or a SWI to replace breakpointed instructions. Once the target hits one of the breakpointed instructions, control is given to the debugging stub, which can inform the debugging host about the breakpoint.
The required initialization and the use of target resources are a major drawback of debugging stubs, but they require only little extra hardware, making them interesting for situations where development tools cost is important.

Integrated Debug Circuitry, On-Chip Debug

Integrated debug circuitry gives the power of in-circuit emulators at a much higher flexibility. Instead of having to replace the target's microcontroller with a special debug version, every chip shipped contains the debug functionality. A serial communication channel, able to operate at high clock speeds, is used to connect the debug circuitry to a host debugger, allowing low pin-count debug connections.

1.2 ARM Debug Solutions

Due to the popularity of SOC designs based on the ARM7 and ARM9 family there several vendors who provide tools to work with the integrated debug circuitry included in all ARM7 and ARM9 based microcontrollers. There are commercial tools available as well as free and open source implementations, offering a wide range of supported functionality. Some are offered as combined solutions, where hardware that interfaces between a host PC and the debug target comes together with debugging software, while others are pure hardware solutions, that can be combined with various software products. Here, debug software doesn't necessarily mean a full-featured debugger, but rather software that talks to the hardware, providing a set of debug functions to a debugging frontend. The following overview isn't meant as a comprehensive listing, but rather to show a few typical designs.

Combined Hardware and Software Solutions

ARM Multi-ICE

`http://www.arm.com/products/DevTools/MultiICE.html`
The *ARM Multi-ICE* allows debugging of a wide variety of ARM based cores and supports all possible functionality, including access to special system-control registers, semihosting and flash programming. It connects to the host computer using a PC parallel port and accesses the target with a JTAG clock of

up to 10 MHz. It comes together with a software called the *Multi-ICE server*, which contains the target specific debug functionality and provides the remote debugging interface (RDI) to a debugger frontend. The Multi-ICE server software requires a PC running a version of Microsoft Windows. Linux or other free operating systems are not supported.

ARM RealView RVI

`http://www.arm.com/products/DevTools/RVI.html`
The *ARM RealView ICE* supercedes the Multi-ICE, providing JTAG clock rates of up to 50 MHz, larger cable lengths, and host connection via Ethernet or universal serial bus (USB), giving greater flexibility. The RVI contains an ARM9 processor which takes care of all the target specific debug functionality. It requires the *RealView Debugger (RVD)* as a frontend, which is available for Microsoft Windows, Linux and Solaris.

Abatron BDI2000

`http://www.abatron.ch/`
The *BDI2000* connects to a host computer via RS232 or 10BASE-T Ethernet, and supports JTAG clock rates up to 16 MHz. The hardware can be configured for a wide variety of target systems, including a configuration for ARM7 and ARM9 targets. The target specific debug functionality is contained inside the BDI2000. Various debugger frontends are supported, like *ARM RealView Tools*, *Metrowerks CodeWarrior* or the free GNU Project Debugger (GDB). An additional telnet interface provides direct access to hardware-specific debug functions. The BDI2000 gives access to almost all possible debug functionality, including system-control registers.

Hardware-only Solutions

Macraigor Wiggler, Parallel Port Wigglers

`http://www.macraigor.com/wiggler.htm`
The *Macraigor Wiggler* is a simple device that connects a PC parallel port to the target JTAG interface. The host PC simulates the target interface by switching signals on and off, a technique often called "bit-banging". It acts as a signal buffer, providing necessary level translation between the PC (5V Transistor-Transistor Logic (TTL)) and the target (1.5V...5V). Schematics for wiggler-compatible clones are freely available on the net and can be adjusted to special requirements, like adding or removing signals that are optional in the JTAG standard, but required by some targets. Figure 1.1 shows the functional equivalent of a complete wiggler clone. The Wiggler's speed is limited by the PC's parallel port, which requires a minimum of about 1 μs per `in` or `out` instruction [Rs00]. A complete clock cycle requires at least 2 μs, limiting the maximum frequency to 500 kHz.

Macraigor Raven

`http://www.macraigor.com/raven.htm`
Like a Wiggler, the *Macraigor Raven* connects to the host using a PC parallel port, but it uses the enhanced parallel port (EPP) protocol and higher-level commands. Logic inside the Raven translates the parallel data from the host to a serial bitstream at up to 8 MHz. The internal design of the Raven is proprietary, schematics are not available. Binary drivers are available for Microsoft Windows and Linux.

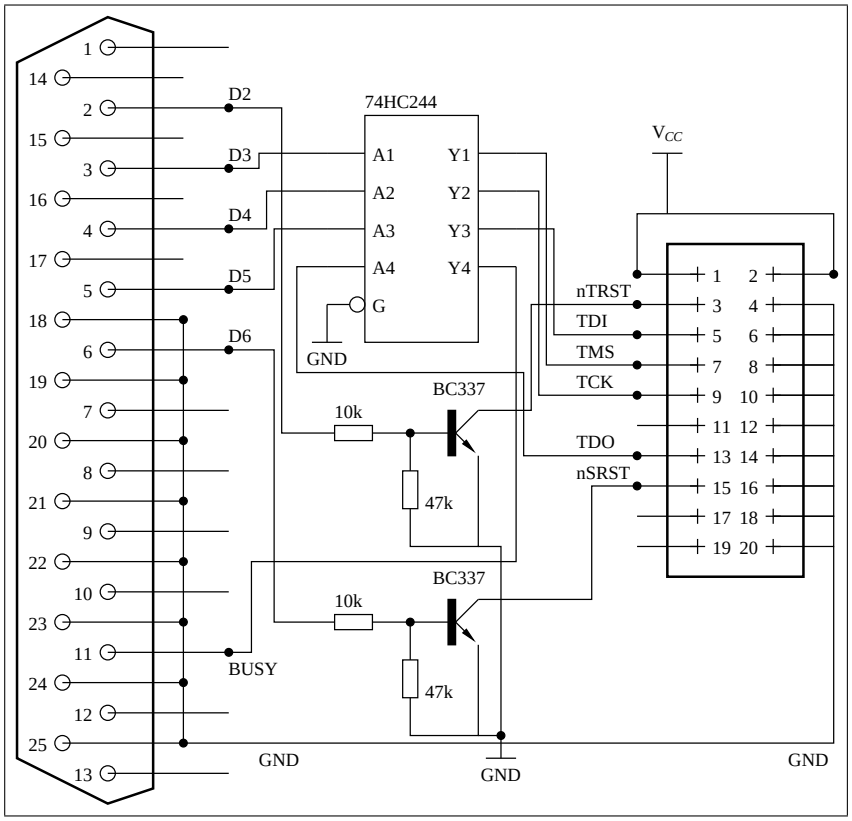

Figure 1.1: Wiggler schematics

Amontec Chameleon POD

`http://www.amontec.com/chameleon.shtml`
The *Amontec Chameleon POD* is based on a Xilinx Coolrunner (XPLA3) XCR3128XL-VQ100 complex programmable logic device (CPLD) (`http://www.xilinx.com`). It connects to the host using a PC parallel port, and supports many different configurations, including emulations of the Macraigor Wiggler and Raven. The configurations, that may be downloaded for free, are programmed into the Chameleon using proprietary software available only for Microsoft Windows.

USBJTAG-1

The device designed by Hubert Högl around a FTDI2232C (`http://www.ftdichip.com/FTProducts.htm#FT2232C`), connects to a host PC using a USB 1.1 Full-Speed (11 Mbit) interface. Using its Multi-Protocol Synchronous Serial Engine (MPSSE) [AN2232C-01], the FTDI chip is capable of JTAG clocks between 6 MHz and 93 Hz.

Figure 1.2 shows an example implementation using the DLP-2232M evaluation kit, available at `http://www.ftdichip.com/Products/EvaluationKits/DIPModules.htm#DLP-2232M`. The evaluation kit contains all circuitry necessary for the USB functionality, making it ideally suited for prototyping.

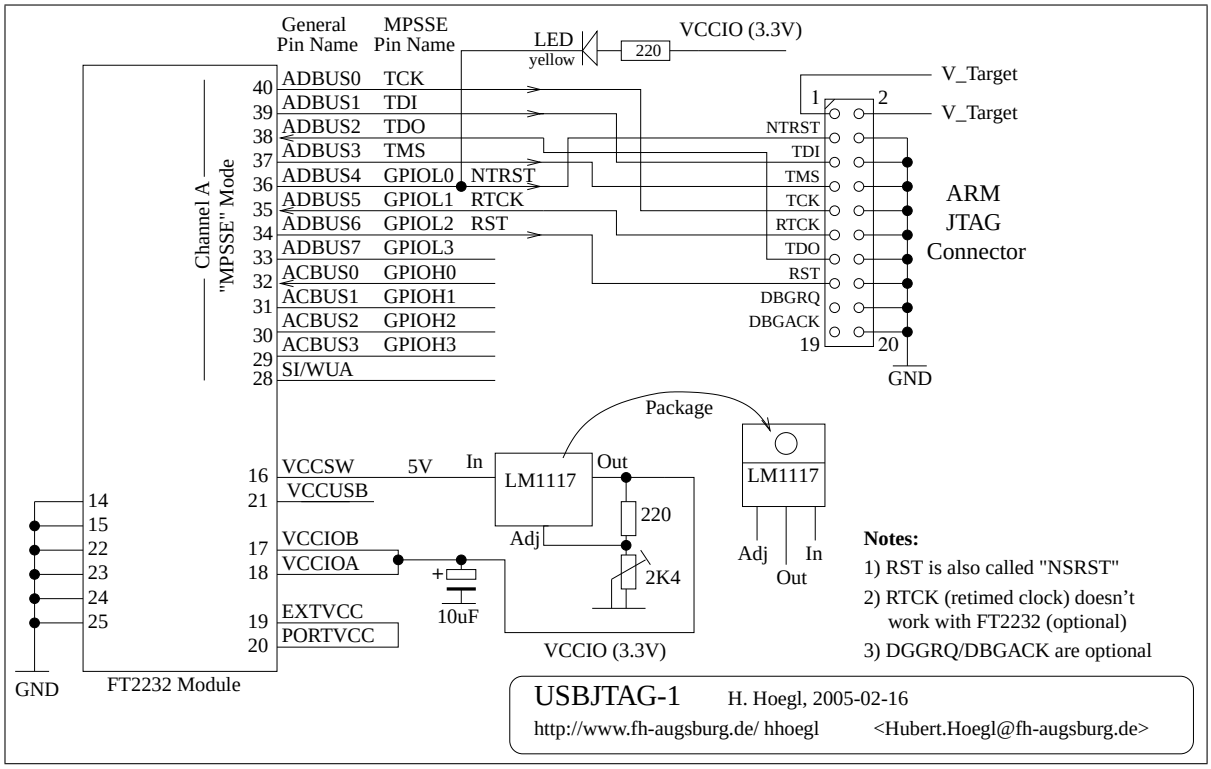

Figure 1.2: USBJTAG-1 schematics

Software Solutions

Macraigor OCD Commander, OCDRemote

`http://www.macraigor.com/ocd_cmd.htm`, `http://www.macraigor.com/full_gnu.htm`
The Macraigor software packages are intended to be used with the Macraigor Wiggler, Raven, usbDemon and mpDemon devices, but it's possible to use the software licensed as freeware together with Wiggler and Raven clones, too. Supported cores include many ARM7 and ARM9 members, as well as MIPS, Motorola PowerPC, and Intel XScale. Access to special registers is limited. There's no information available about the extent of cache and MMU handling. The OCD Commander is a complete graphical debugger, while the OCDRemote is a console application that interfaces the Macraigor hardware with the a GDB client. Both are available for Microsoft Windows and Linux. The Linux software comes with binary-only objectfiles that are linked into the provided kernel module. Writing Flash memory is not possible using the freeware programs, but Macraigor offers a (non-free) software called "Flash Programmer" that is able to program flash chips on ARM7 and ARM9 targets.

Open Source Software

There are a few open source projects to support ARM7 and ARM9 debugging, all licensed under the GNU General Public License (GPL). Functionality is limited compared to the available commercial solutions, and all but the gdb-jtag-arm seem to be unmaintained or no longer under active development. The

only JTAG hardware interfaces supported are Wiggler and compatibles. None of the projects provides handling of the MMU or caches found on cores like the ARM720t or ARM920t.

- JTAGER by Rongkai Zhan
  http://jtager.sourceforge.net/
  JTAGER supports ARM7TDMI, ARM720t, and ARM920t based targets. Flash memory write support is included for SST39LF/VF160 (http://www.sst.com/) and MBM29LV650 (http://www.spansion.com/) chips. A command line interface is implemented for user interaction, GDB support isn't included. Version 0.3.0 was released on October, 17th 2004, and is still in an early development stage. Bugs and shortfalls of the code can lead to target system crashes and memory inconsistencies.

- armtool by Erwin Authried (part of Midori Linux)
  http://home.at/cgi-bin/viewcvs.cgi/midori/sources/armtool/
  Armtool only supports ARM7TDMI based targets, is able to read and write target memory, and allows code downloaded to the target to be executed. It's suitable for batch usage, but doesn't allow user interaction, neither through a command line interface nor using a debugging frontend. Flash support isn't included.

- jtag-arm9 by Simon Wood
  http://jtag-arm9.sourceforge.net/
  Jtag-arm9 only supports ARM9 based targets, and contains a command line interface for user interaction. It is able to halt and resume the target, read and modify target registers, and supports memory read and write operations. Flash support isn't included.

- gdb-jtag-arm by Tobias Lorenz
  http://gdb-jtag-arm.sourceforge.net/
  Gdb-jtag-arm is the only open source project that supports the GNU Debugger (gdb) as a debugging frontend. It is based on jtag-arm9, fixing some, but not all, of the original software's bugs. Target system crashes or failures writing target memory can result from these defects.

2 IEEE 1149 - JTAG

The Joint Test Access Group (JTAG) was formed in 1985 to create printed circuit board (PCB) and integrated circuit (IC) test standards. The latest version of their proposal was approved by the Institute of Electrical and Electronics Engineers (IEEE) as IEEE Std. 1149.1-2001 [IEEE1149], *IEEE Standard Test Access Port and Boundary-Scan Architecture*. The standard was created to support testing of component functionality, component interconnections and component interaction on assembled products. Subsequently, the term JTAG shall refer to the mentioned IEEE standard unless otherwise noted. This chapter is intended to give the reader enough understanding of the standard necessary for the operation of a JTAG based debugger. The main objective is therefor the design of a bus master, not the connected devices.

2.1 Test Logic

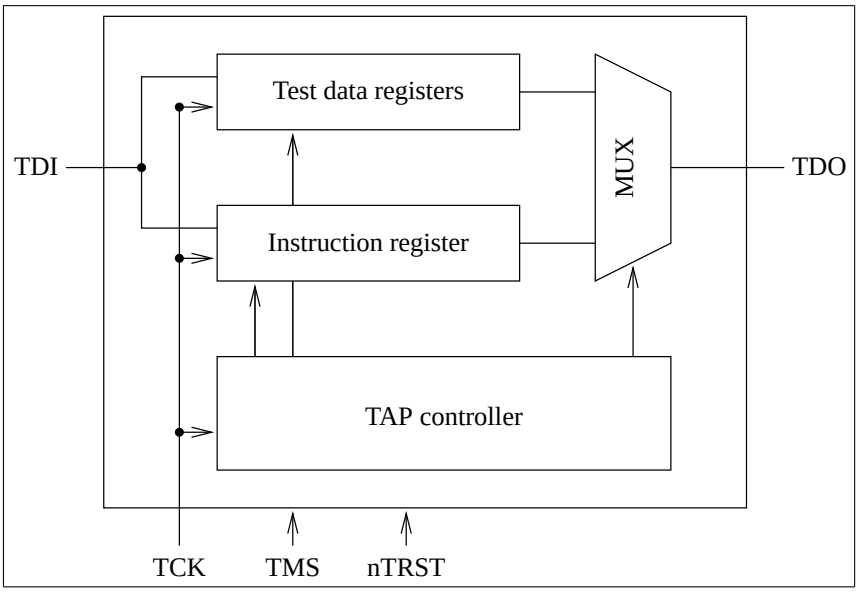

Figure 2.1: Test logic

A device that conforms to the JTAG standard contains one instruction register (IR), a number of test data registers (DR) and a test access port (TAP) controller which handles all test operations. The boundary-scan technique uses scan cells connected to a component's inputs and outputs, forming a serial

shift register. Test patterns are shifted (or "scanned") by a TAP bus master through the TAP into a component and apply known values to the scan cells. The scan cells' previous content is shifted out of the component and can be captured by the TAP bus master. The instruction register is required to be at least two bits long, and is used to control test functionality. The data registers are specific to a particular device, but the standard demands at least two register, a one-bit long bypass register, and a boundary-scan register. Figure 2.2 shows two components containing test logic like the conceptual example shown in Figure 2.1 connected to a single bus master.

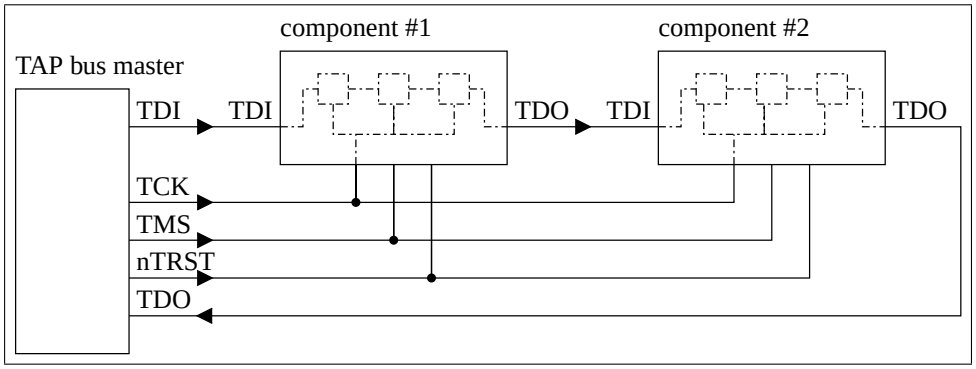

Figure 2.2: JTAG TAP / Boundary-Scan Architecture

2.2 JTAG/TAP Signals

Table 2.1: JTAG interface signals

Name (abbreviation)	Description	Direction
Test Clock (TCK)	Serial clock signal	out
Test Mode Select (TMS)	Controls movement of the JTAG state machine	out
Test Data Input (TDI)	Serial data fed into tested equipment	out
Test Data Output (TDO)	Serial data read back from tested equipment	in
Test Reset (nTRST)	Optional signal to asynchronously initialize test equipment	out

Table 2.1 shows the signals defined by the JTAG standard and their direction from the bus master's point of view. The TCK signal allows data to be scanned into multiple components independently from component specific system clocks. TCK may be stopped at 0 for an indefinite time, while test components are guaranteed to retain their current state, but not necessarily stopped at 1, which is permissible but not required by the standard.
The TMS signal selects the path taken in the JTAG state machine (see Figure 2.3). This signal is sampled at the rising edge of TCK, and is expected to be changed by the TAP bus master on the falling edge of TCK. The state machine is designed in a way that allows the Test-Logic-Reset state to be reached after five TCK cycles with TMS held high from every possible state. The standard requires circuitry to apply a logic 1 to TMS when the signal is undriven, ensuring normal operation when no test equipment is connected.

TDI transmits serial data shifted from the bus master to connected TAP controllers. Like TMS it is sampled on the rising edge of TCK, and in case of an undriven signal circuitry shall apply a logic 1 to TDI, too. Serial data from connected TAP controllers is shifted out of a component using the TDO signal. It changes its state on the falling edge of TCK, and should be sampled by the bus master on the rising edge of TCK.

2.3 JTAG State Machine

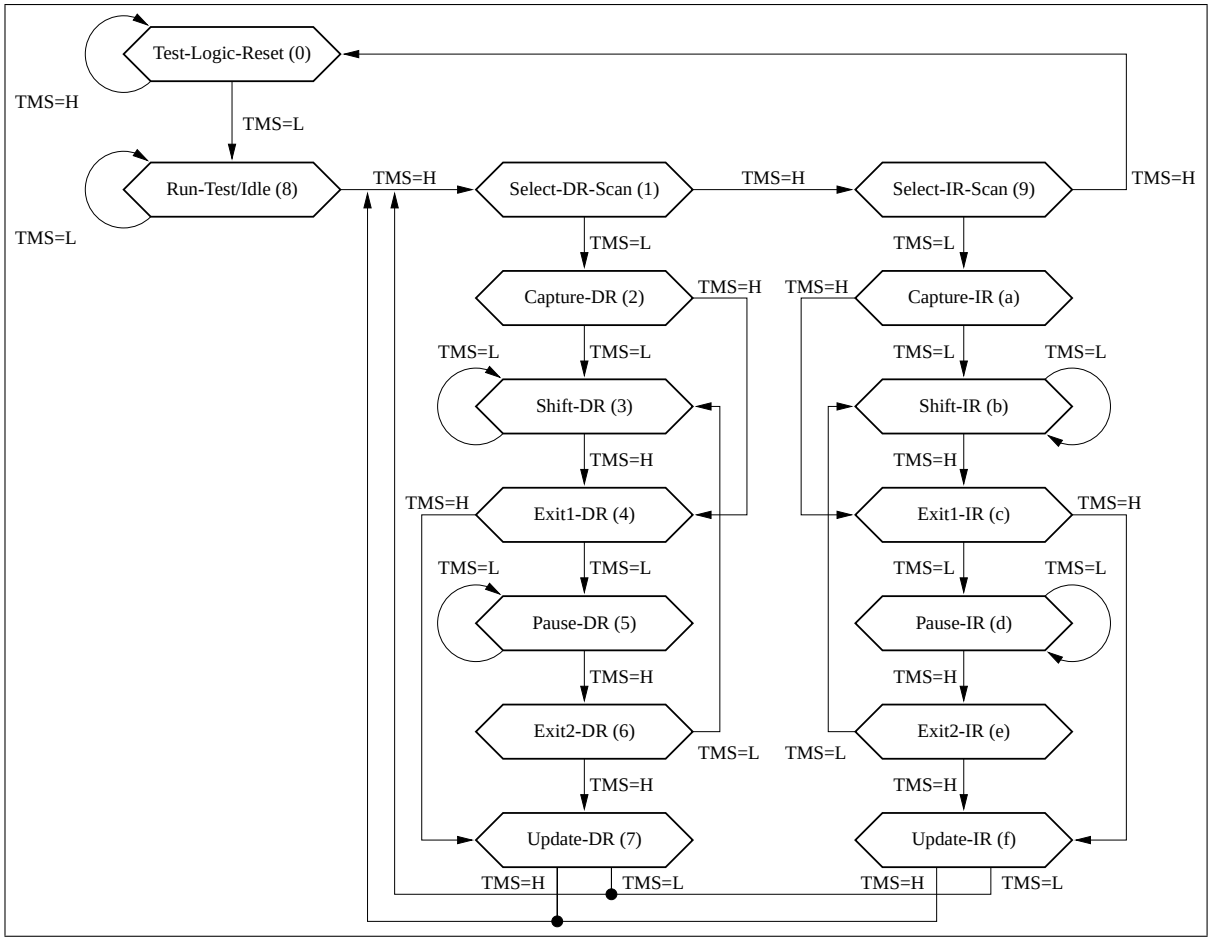

Figure 2.3: JTAG state machine

All JTAG operations are controlled through a state machine implemented in the TAP controller. The state machine is driven by TMS and is clocked by the rising edge of TCK. When a test session is initiated, the bus master has to initialize all connected TAP controllers by putting them into Test-Logic-Reset (TLR) state. TLR is reached by either forcing nTRST low or by executing five TCK cycles with TMS kept high. Once in TLR, the device identification register (IDCODE) or the bypass register (BYPASS) is selected, and all test functionality is reset. If TMS is low on a rising edge of TCK in TLR, the state

machine enters the Run-Test/Idle state. Depending on the currently selected instruction, test operations can be executed in the (RTI) state, or the test logic is left idle, and no operations occur. From RTI, Select-DR-Scan (SDS) is reached. SDS is, like Select-IR-Scan (SIS), Exit1-DR/IR (E1D, E1I) and Exit2-DR/IR (E2D, E2I), a temporary states where no test operations occur, used to select different paths through the state machine. In Capture-DR (CD), the currently selected test data register may be parallel loaded if appropriate, or left unchanged if the register doesn't have a parallel input or if the current instruction doesn't require the current value to be captured. During Capture-IR (CI), a fixed value of b01 is loaded into the least significant two bits of the IR, and design specific values may be put into any remaining IR bits. Once Shift-DR or Shift-IR is reached, the TAP bus master takes TMS low and starts outputting the desired value on each falling edge of TCK. The device under test will sample TDI on the rising edge and stay in Shift-IR while TMS is kept low. Pause-DR/IR may be used to indefinitely idle during - or between - scan operations. No test logic operations occur while a TAP controller is in Pause state. On the falling edge of TCK in Update-DR, the current value of the serial shift register is latched onto parallel outputs, if this is required for the currently selected test data register. Similarly, the current value of the instruction serial shift register is latched onto parallel outputs on the falling edge of TCK in Update-IR. Latching a new value on the IR parallel outputs makes this value the new current instruction.
Figure 2.4 shows an example session where a value of b0100 is scanned into a 4-bit long instruction register. During Capture-IR, a value of b0001 was loaded into the IR and can be captured by the bus master during the scan. Data register scans are similar, only a different path in the state machine is taken. The data register accessed depends on the current value of the instruction register and possibly on test operations executed earlier.

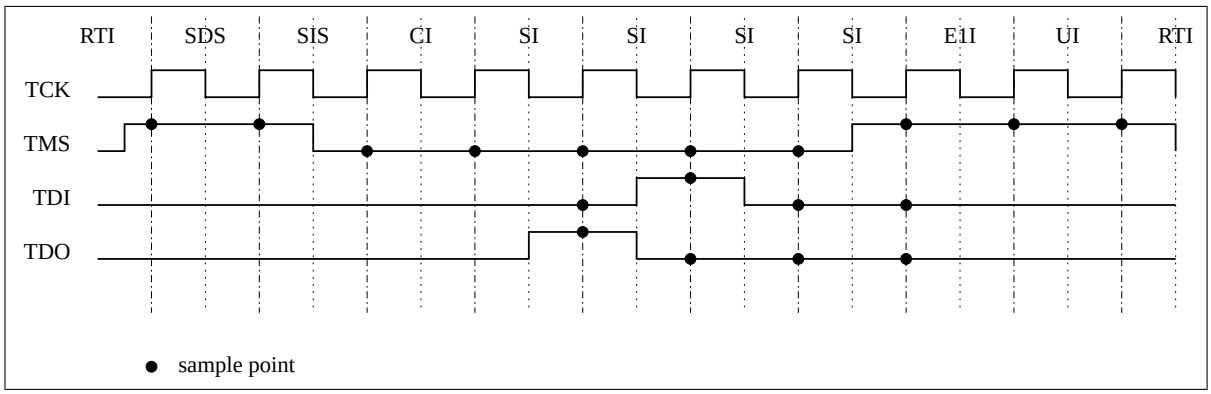

Figure 2.4: JTAG signals and states

2.4 JTAG Instructions

The JTAG standard requires several instructions to be available on a device compliant to standard IEEE 1149.1, but most of these are unimportant for the purpose of ARM debugging. The following instructions are used in debugging ARM7/ARM9 based systems:

Table 2.2: JTAG instructions (subset)

Name	Description
BYPASS	When the BYPASS instruction is selected on a device, the 1-bit wide bypass register is connected as the current test data register. This allows the scan chain in configurations with multiple successive devices to be shortened, making accesses faster. A device in BYPASS mode should not perform any test operation. One binary code for BYPASS shall be all ones (e.g. b1111 or 0xf for a device with a 4-bit wide instruction register), but additional codes may map to BYPASS, too.
EXTEST	The mandatory EXTEST instruction selects the boundary-scan register as the current test data register. Signals that are driven from outside of the component are loaded into the boundary-scan register during the falling edge of TCK in Capture-DR state, and signals that are driven from the component are loaded from the boundary-scan register on the falling edge of TCK in Update-DR state. This allows signals from the system to the component to be captured, and known values to be applied to signals driven from the component to the system. The binary code of the EXTEST instruction may be chosen by the component designer.
INTEST	The optional INTEST instruction also selects the boundary-scan register, but is used to capture signals driven out of the component, and known values to be applied to signals driven into the component. The binary code of the INTEST instruction may be chosen by the component designer.
IDCODE	IDCODE is an optional instruction that selects a device identification register as the current test data register. While IDCODE is selected, no other test data register shall be selected. The binary code of the IDCODE instruction may be chosen by the component designer.

3 ARM7 / ARM9 Architecture

This aim of this chapter is to describe the architecture implemented by ARM7 and ARM9 family targets, with a focus on aspects relevant for a debugger. These two core families share a great deal of debug functionality, making it possible to support both with a single debug solution.

3.1 Core Families

Currently available ARM7 family members, the ARM7TDMI, ARM710T, ARM720T, and ARM740T, are based on an ARM7TDMI core, with the exception of the ARM720T Rev 4, which is based on an ARM7TDMI-S synthesizable core. Older ARM7 members like the ARM700 or ARM750 are beyond the scope of this work and thus, ARM7 shall only refer to the above mentioned cores for the remainder of this document. The ARM9 family is based on the ARM9TDMI core, which is not available separately, but only as part of an ARM920T, ARM922T or ARM940T. Other ARM9 cores, like the ARM926EJ-S, ARM946E-S and ARM966E-S are based on the synthesizable ARM9E-S or ARM9EJ-S core, and contain slightly different debug functionality. This document is going to indicate these differences, but the prototype software resulting from this diploma thesis will be limited to ARM7TDMI, ARM720T and ARM920T cores. ARM9 cores with the letter E form a family of their own, but for the purposes of this document ARM9 shall refer to both families. See Table 3.1 for a list of letters used in ARM core and architecture names and their meaning.

The ARM architecture version implemented by the ARM7TDMI and ARM9TDMI based cores is

Table 3.1: ARM core features

Letter	Description
T	Thumb mode support (compressed 16-bit instruction set)
D	Debug support
M	Enhanced Multiplier (multiply with 64 bit)
I	Embedded-ICE
E	ARM 'Enhanced' DSP instruction set
J	Jazelle Java acceleration technology

ARMv4T, while the newer ARM9E(J)-S based cores implement ARMv5TE or ARMv5TEJ. The major difference between the two architecture versions is the support of the ARM 'Enhanced' DSP instruction set, which is available on all ARMv5 cores, and the Jazelle Java acceleration technology, available only on ARMv5TEJ cores. From the debugger's point of view, the added DSP instructions don't require any special handling, as they don't affect the processor state, but if debug state is entered from Jazelle state, the core has to be switched to ARM state before the core and system state may be examined. All ARMv5

User	FIQ	IRQ	Supervisor	Abort	Undefined	System
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12	R12
R13	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und	R13
R14	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und	R14
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und	

Figure 3.1: ARM banked registers

cores also support a software breakpoint instruction (BKPT) that forces the core into debug state when it's executed. On cores without support for this instruction, the software breakpoint behavior has to be simulated using a data dependent breakpoint, that triggers once a certain instruction is fetched from memory.

The basic execution context is the same for ARMv4T, ARMv5TE and ARMv5TEJ, and has to be restored by a debugger before control can be transfered back to system software.

- 31 general purpose registers, including the program counter (PC). Only 16 registers are accessible at any time, the remaining registers are banked registers available only from within a particular processor mode. Figure 3.1 shows the association between processor modes and banked registers. Note that System mode shares all registers with User mode. Register R15 is the PC, and its use is subject to special restrictions. Register R14 is the link register which stores the return address on function calls or exception entry. Register R13 is often used as a stack pointer, although this is not mandatory. The remaining registers may be used at the developer's or the compiler's choice.

- 6 status registers. The current program status register (CPSR) contains information about the current processor mode and state, while the saved program status registers contain the saved state from which an exception mode was entered. Only exception modes have a saved program status register. see Figure 3.2 for the program status register format in architecture versions up to ARMv5TEJ.

- The current processor mode is one of User (USR), Fast Interrupt (FIQ), Interrupt (IRQ), Supervisor

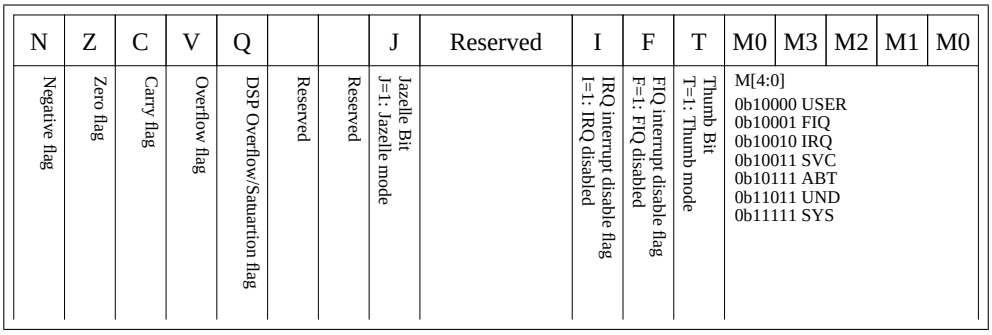

Figure 3.2: program status register format

(SVC), Abort (ABT), Undefined Instruction (UND), and System (SYS). All but the User mode are so called privileged modes, with full access to the hardware. Depending on the current mode, only a subset of the 31 general purpose registers and 6 status registers may be accessed.

- The current processor state is either ARM, Thumb, or Jazelle (on cores with Java support).

- A flat address space of 2^{32} 8-bit bytes.

Cores with a memory management unit (MMU) and caches require additional properties defining the current execution context. The MMU translates virtual addresses (VA) into physical addresses (PA) and may have a translation lookaside buffer (TLB) to store recently used or explicitly stored translations. The current content of the TLB should be considered part of the execution context, as additional page table walks, caused by evicted TLB entries, could have an impact on application critical timings. The same is true for caches that keep recently used memory blocks in high-speed memory tightly coupled to the processor. When accessing memory in a cached system, a debugger has to make sure that as much of the cache state as possible is preserved.

ARM7TDMI Implementation

The ARM7TDMI features a 3-stage pipeline with Fetch, Decode and Execute stages (see Figure 3.3), and a von-Neumann architecture memory system, where instructions and data are fetched from a unified bus. Implementation defined store instructions, that read the program counter [DDI0100E, p. A2-7] (STR, STRT, and STM), store the address of the current instruction plus 12 bytes. On a data abort, instructions with addressing modes that update a base register will have their base register updated ("base updated" data abort model) [DDI0180A, p. 2.2].
An instruction is fetched from the memory system during the Fetch stage, decoded (and possibly decompressed in case of Thumb instructions) in the Decode stage, and executed during one or more cycles in the Execute stage. Required registers are read during the first Execute cycle, and data memory is accessed during one or more subsequent Execute cycles.

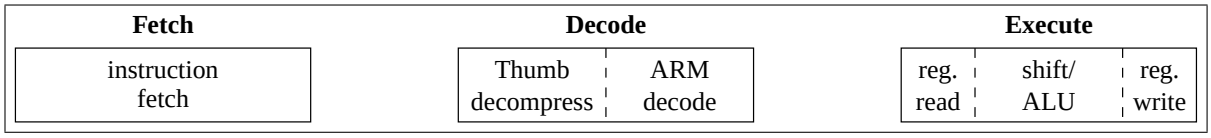

Figure 3.3: ARM7TDMI 3-stage pipeline

ARM9TDMI Implementation

The ARM9TDMI features a 5-stage pipeline that splits the ARM7TDMI's Execute stage into separate Execute, Memory and Write stages (see Figure 3.4). It has a modified Harvard architecture with two separate internal busses for instruction and data that connect externally to unified memory. Like the ARM7TDMI, implementation defined store instructions, that read the program counter [DDI0100E, p. A2-7] (STR, STRT, and STM), store the address of the current instruction plus 12 bytes. The ARM9TDMI implements a "base restored" data abort model, where the base register will always be restored to the value before the aborted instruction was executed [DDI0180A, p. 2.2].
Like the ARM7TDMI, an ARM9TDMI fetches an instruction in the Fetch stage, and decodes it in the Decode stage. Registers are read during the Decode stage, and additional logic ensures a behavior similar to older ARM cores where registers were read one stage later. During the Execute stage, shift and ALU operations are executed, generating results for data instructions, or addresses used in load/store instructions. Data memory is either read or written in the Memory stage, and registers are written in the Write stage. Because of the pipelined architecture, instructions may have to be stalled, if source operands of an instruction were written by an immediately preceding instruction which isn't yet finished. This case is called an interlock, and the core stops fetching new instructions until the results from the preceding instruction are available [DDI0180A, p. 7.5].

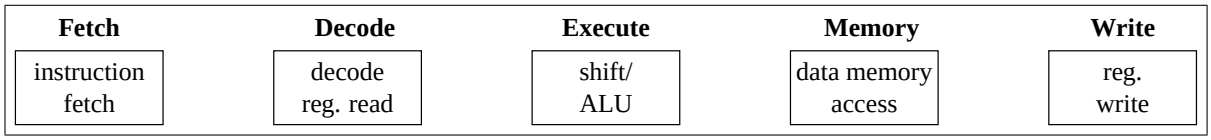

Figure 3.4: ARM9TDMI 5-stage pipeline

3.2 Core Debugging

All ARM7 and ARM9 cores have halt-mode debugging support, that allows the core to be completely stopped. During this debug state, a debugger may capture and modify core signals, allowing the core and system state to be examined and changed. While in debug state, the core is no longer clocked from its main clock (memory clock (MCLK) on ARM7TDMI, fast clock (FCLK) or bus clock (BCLK) on ARM9TDMI) but from a debug clock (DCLK) that's generated by the debug logic.
The ARM core macrocell is deeply embedded inside an ARM based SOC and core signals are not available on external pins. To still be able to debug these systems, ARM7 and ARM9 cores implement a JTAG compatible TAP controller with boundary-scan chains around the core signals. There are two scan chains available on hard macrocells (ARM7TDMI and ARM9TDMI based), one consisting of almost all

core signals, primarily intended for device testing, and another one that consists of a subset of the first, with signals especially important for debug. Figures 3.5 and 3.6 show the order of signals on the debug scan chains. When shifting data in or out of the device, the signal closest to TDO is the least significant bit. It is important to note that D[0:31] (ARM7TDMI) and I[0:31] (ARM9TDMI) are in reversed bit order. Systems based on the synthesizable ARM7TDMI-S and ARM9TDMI-S core lack the first scan chain but are otherwise similar to the hard macrocell implementations.

For the purpose of a debugger it's sufficient to use scan chain 1, which shall from now on be called the debug scan chain. This scan chain may be used in INTEST (see table 2.2) mode, allowing core signals to be captured, and known values to be scanned into the core, or in EXTEST mode, allowing signals from outside of the core to be captured, and known values to be driven to the outside of the core. During debug, the debug scan chain is used in conjunction with the INTEST instruction. The *scan path select register*, a special test data register used to select between several boundary-scan paths, is accessible by the ARM specific SCAN_N JTAG instruction. When SCAN_N is selected, a fixed value, with the most significant bit set to one and all others set to zero, is loaded into the scan path select register during Capture-DR, making it possible to recognize serial communication problems. After scanning a new value into the scan path select register, the new scan chain is made the currently active scan chain during Update-DR. From that point on, JTAG instructions accessing the boundary-scan register (INTEST, EXTEST) apply to the new scan chain.

ARM7TDMI Debug Scan Chain

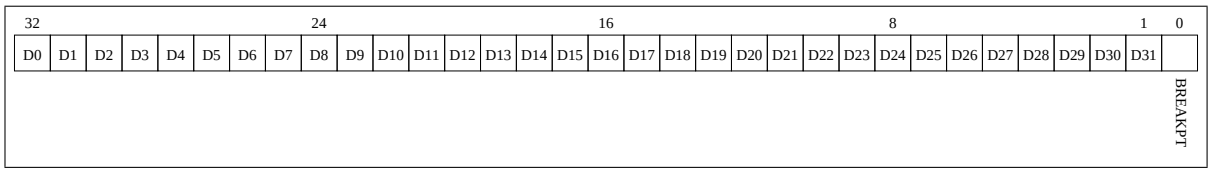

Figure 3.5: ARM7TDMI scan chain 1 (Debug)

Signals D[0:31] of the ARM7TDMI are connected to the core's data bus, and are used to fetch instructions or read/write data. The BREAKPT signal is used to mark instructions that have to be executed at system speed (clocked from MCLK, rather than DCLK), like instructions accessing memory or instructions that make the core return from debug state back to its normal state. The first time BREAKPT is scanned out of the core, it contains information about whether the core entered debug state due to a breakpoint (BREAKPT low) or because of a watchpoint (BREAKPT high).

ARM9TDMI Debug Scan Chain

Signals ID[0:31] of the ARM9TDMI are connected to the core's instruction bus, and are used to fetch instructions. The data lines DD[31:0] connected to the bi-directional data bus are used to read or write data. SYSSPEED is similar to the ARM7TDMI's BREAKPT signal, serving both as a flag for system speed instructions and as an indicator for the reason for debug entry. The WPTANDBKPT signal allows a debugger to determine if an instruction that triggered a watchpoint was immediately followed by a

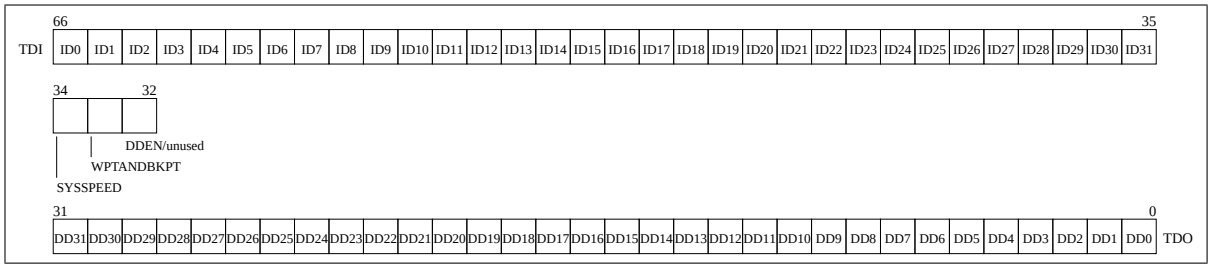

Figure 3.6: ARM9TDMI scan chain 1 (Debug)

breakpointed instruction. In that case, the breakpointed instruction wouldn't have been executed, and would be the next instruction after debug state is left. The DDEN signal is available only on hard macrocell cores, its bit position is unused on synthesizable cores. When DDEN is high, the core is driving data out on DD[31:0] which may be captured by a debugger.

Debug Instruction Execution

Once in debug state, a debugger may serially shift data into the debug scan chain by selecting scan chain 1 (via SCAN_N) and INTEST. While the debug scan chain is selected and INTEST is the current instruction, a DCLK cycle is pulsed on the rising edge of TCK when the TAP controller is in Run-Test/Idle state, making the core act upon the values currently contained in the debug scan chain. A debugger has to take the processor pipeline into account, that is the pipeline stages in which values appear on the databus or have to be written to the bus by the debugger, and possible interlocks in case of ARM9TDMI based cores.

3.3 Embedded-ICE

The Embedded-ICE (formerly known as "ICEBreaker") macrocell available on all ARM7 and ARM9 cores provides on-chip debug functionality similar to an ICE (see §1.1). The Embedded-ICE unit is accessed through JTAG scan chain 2, which is selected through SCAN_N similarly to the debug scan chain (scan chain 1) and may only be used with the INTEST instruction. The Embedded-ICE scan chain (see Figure 3.7) is the same for ARM7 and ARM9 targets, and consists of 32 data bits, 5 address bits and a flag to distinguish between read (nRW low) and write (nRW high) accesses. Embedded-ICE features are accessible through registers, whose number is placed in the address field. The data field contains register data read or to be written, and is aligned to the least significant bit for registers with less than 32 bits. For register reads, the Embedded-ICE scan chain has to be accessed twice, once to program the nRW field for reading and the address of the register to be read, and once to capture the data of the selected register. Register writes are accomplished with a single access programming nRW for writing, the address, and the new register data. Register reads and writes are executed during the Update-DR state.

Every Embedded-ICE implementation provides a common set of supported features, with extensions or restrictions specific to certain families, cores or revisions. Embedded-ICE units contained in ARM7 and ARM9 family cores have two comparators that can be used to break on instruction fetches or data

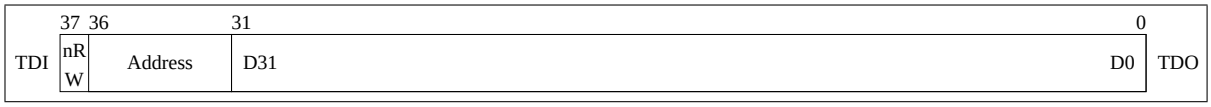

Figure 3.7: Embedded-ICE scan chain (Scan chain 2)

accesses. Each comparator consists of an address register, a data register, a control register, and a mask register for each of the three registers with a layout similar to the value register that may be used to make the comparator ignore masked bits in the comparison. The value/mask register combination allows three possible requirements to be set:

- A positive match. The respective bit is required to be 1. This is achieved by programming the value register bit to 1, and the mask register bit to 0.
- A negative match. The respective bit is required to be 0. This is achieved by programming the value register bit to 0, and the mask register bit to 0.
- An ignored bit. A match should occur whether the bit is 1 or 0. This is achieved by programming the mask register to 1, irrespective of the value bit.

The layout of the comparator registers is almost the same for all ARM7/ARM9 cores, with the exception of a slightly different layout of the control register found on ARM9TDMI based cores, which is due to the modified Harvard architecture of these cores.
The debug control register and the debug status register give access to debug signals that allow a core to be put into halt-mode debug state using an external request, and information about the core state to be examined by a debugger. The signals available through these registers depend on the exact core being used, but a common subset is provided by all implementations.
The Embedded-ICE debug communications channel allows a debugger to communicate with software running inside the core without using additional system resources like a RS232 port. It is accessible from a debugger via a control and a data register, and can be accessed from the core using coprocessor instructions. The control register is used to manage the communication between a debugger and the running core, and contains information about the Embedded-ICE version implemented.
Table 3.2 shows the available registers and their addresses. A detailed layout of the registers is given in figure 3.8, without registers of a flat 32 bit layout like the watchpoint data and address registers or the debug comms data register. The watchpoint control mask register has a layout similar to its control value register but is one bit shorter, as the Enable bit can not be masked.

Embedded-ICE Usage

Debug Request

Entering debug mode on the debugger's request works the same for all ARM7 and ARM9 targets. The debugger asserts DBGRQ by programming the debug control register with DBGRQ set to 1, and polls the debug status register until it reads a 1 in DBGACK. On ARM9 based cores DBGRQ could be left asserted, but ARM7 based cores require it to be deasserted in order to execute instructions at debug speed. The core is then in debug state and may be examined by the debugger.

Table 3.2: Embedded-ICE register map

Address	Register name	Availability restrictions
0x0	Debug control register	
0x1	Debug status register	
0x2	Abort status register	only ARM7 cores with monitor mode debug (ARM7TDMI Rev 4, ARM7TDMI-S Rev4, and ARM720t Rev4)
	Vector catch register	all ARM9 cores
0x4	Debug comms control register	
0x5	Debug comms data register	
0x8	Watchpoint 0 address value	
0x9	Watchpoint 0 address mask	
0xa	Watchpoint 0 data value	
0xb	Watchpoint 0 data mask	
0xc	Watchpoint 0 control value	
0xd	Watchpoint 0 control mask	
0x10	Watchpoint 1 address value	
0x11	Watchpoint 1 address mask	
0x12	Watchpoint 1 data value	
0x13	Watchpoint 1 data mask	
0x14	Watchpoint 1 control value	
0x15	Watchpoint 1 control mask	

Hardware Breakpoints

Hardware breakpoints are realized using one of the two comparators. The address value and mask register should be programmed to match the desired address, with the least significant bit masked for Thumb breakpoints (16 bit instructions) or the two least significant bits masked for ARM breakpoints. This ensures that the breakpoint triggers even with undefined signal levels on unused address lines.
A variant of HW breakpoints would be the use of a data dependent breakpoint. By programming the watchpoint's data register to match a certain instruction value, the breakpoint would only trigger if that instruction is fetched. Together with the address value and mask registers this could be used to set a breakpoint on the execution of a certain instruction in a specified part of the address space.
On ARM7 targets, a comparator may be programmed to match on instruction and data accesses by masking the nOPC field in the watchpoint control register. That's not possible on ARM9 targets due to their modified Harvard architecture, as the comparator can only watch a single bus. A generic breakpoint layout that works for ARM7 and ARM9 targets programs nOPC to a negative match.
If breakpoint matches should be restricted to Thumb instruction fetches, MAS[0] (ITBIT on ARM9) may be programmed to require a positive match. MAS[1:0] is used to determine the size of a memory access, with b00 meaning byte accesses, b01 being a half word access (16 bit) and b10 being a word access (32 bit). An instruction fetch with MAS[1:0] set to b01 is therefor a 16-bit Thumb instruction fetch.
Using the Range, Chain and Extern fields in the watchpoint control register allows more complex breakpoints to be defined. The technical reference manuals of each ARM core give information on the possibilities and some usage guidelines.

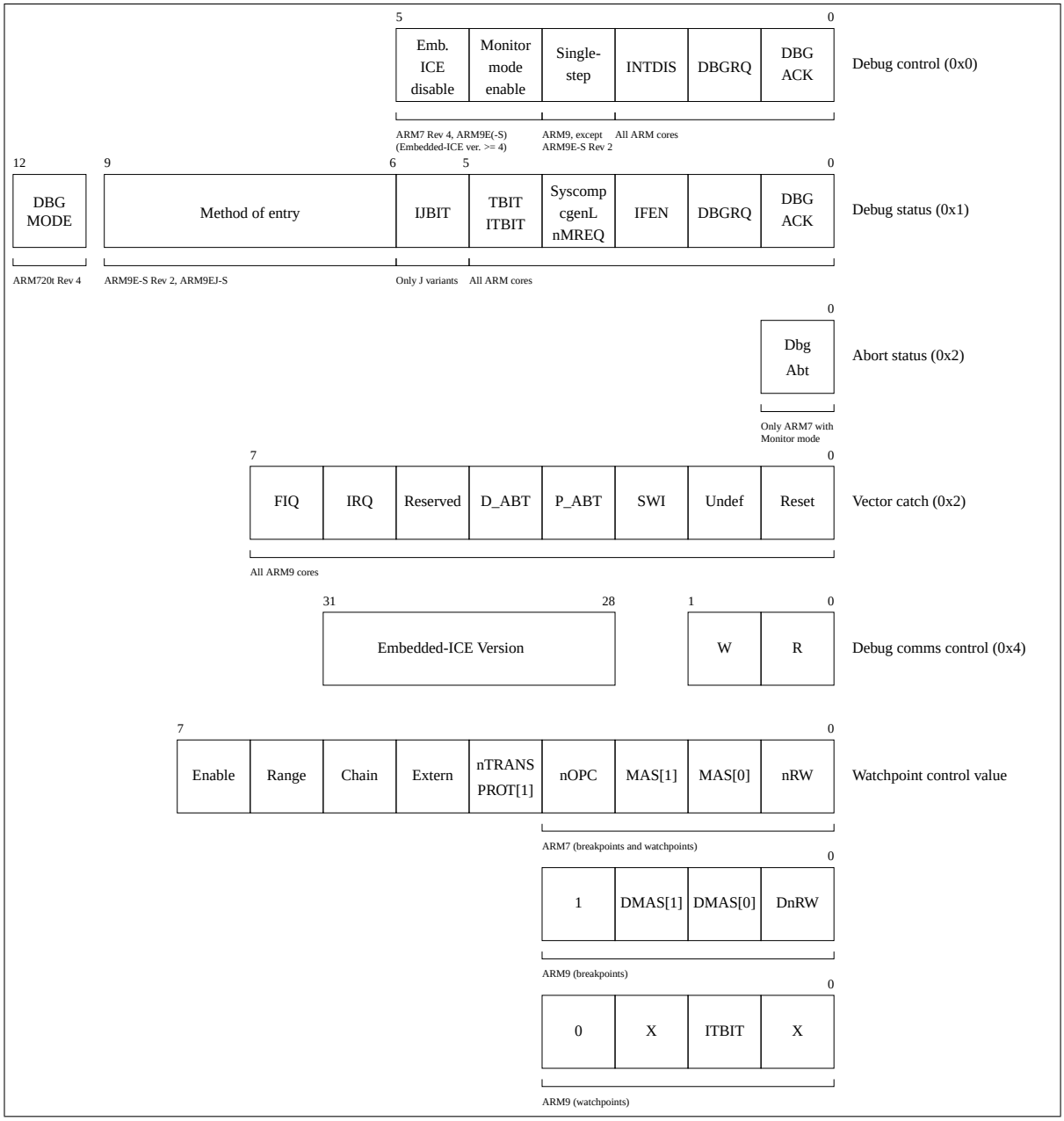

Figure 3.8: Embedded-ICE register layout

Software Breakpoints

Software breakpoints work by replacing an instruction in the target memory with a special instruction that forces the core to enter debug state. Cores implementing the ARMv5TE(J) architecture may use the dedicated BKPT instruction for this purpose, while older cores use a value defined in a data dependent instruction breakpoint. The watchpoint address mask register is programmed to ignore the address (all bits 1), and the data register is set to match on a certain instruction value. To be able to use the same instruction value for ARM and Thumb state breakpoints, a symmetric pattern with the same value in the upper and lower 16 bit of a 32 bit word should be chosen. 0xDEEEDEEE (32 bit ARM breakpoint) and 0xDEEE (16 bit Thumb breakpoint) is a possible implementation that fits to the Thumb state breakpoint instruction available on ARMv5TE(J) BKPT (0xDExx) [DDI0100E]. The control register should be programmed to require a negative match on nOPC and ignore all other bits.

Watchpoints

Hardware watchpoints are similar to HW breakpoints, but monitor the data bus. This is achieved by programming the nOPC field in the watchpoint control register to a positive match. Using the nRW field, a watchpoint can be limited to reads (negative match), writes (positive match) or any access (ignore).

Vector Catching

It may be important for a debugger to catch all or certain exceptions generated in the target. On ARM7 targets, this has to be achieved using a HW or SW breakpoint, requiring the use of one of the two comparators. ARM9 targets contain a dedicated vector catch register that allows breakpoints to be set on all or selected exceptions. The vector catch register only triggers on exception mode entry, not on a regular fetch caused by a branch to an exception address.

Single-Stepping

Single-stepping is implemented in hardware on most ARM9 cores with the exception of ARM9E-S Rev 2 and ARM9EJ-S based designs, where this capability can't be found. The *Single Step* bit contained in the debug control register of cores supporting single-stepping forces the core to re-enter debug state after a single instruction has been fetched and executed.
Cores without hardware single-stepping capability have to simulate this behavior using a breakpoint combined out of two comparators. The two comparators form an inverse breakpoint, that breaks on everything but the current address:

- Both watchpoint units are programmed for HW breakpoint usage, requiring a negative match on nOPC and ignoring the data value (data mask registers set to all ones).
- Watchpoint 1 matches the address of the current instruction (the one to be executed), but isn't enabled. Inside the ARM7/ARM9 Embedded-ICE unit, the Range output of watchpoint 1 is derived from its address comparator and connected to watchpoint 0's Range input. An address match on watchpoint 1 appears as a positive value on watchpoint 0's Range field.
- Watchpoint 0 is enabled and set to match on any address, but is required to have a negative match on its Range field.

- The core resumes execution from debug state. On the first instruction fetch, watchpoint 1 matches the address but doesn't trigger as it's not enabled. Watchpoint 0 matches the address, but its Range input is high (from watchpoint 1), preventing it from triggering.
- On the second instruction fetch, watchpoint 1 no longer matches the address. Watchpoint 0 still matches the address, this time with a low Range input, making it trigger. The core enters debug mode after executing one instruction.

This method doesn't work for instructions that branch back to themself, a combination that's probably rarely seen in reality. In that case, watchpoint 1's address comparator would match forever, preventing the core from re-entering debug state. A debugger should take care of that possibility by implementing a timeout by which the core should have reentered debug state.

3.4 Debug State Entry

Debug state may be entered as a result of the following conditions:

- Debug request. Either an external debug request (EDBGRQ) or as a result of programming the Embedded-ICE control register with DBGRQ set to 1. The core is forced to enter debug state after it finished executing the current instruction. On ARM7 cores, the program counter (PC) contains the address of the instruction to be executed next plus two addresses (8 byte in ARM state, 4 byte in Thumb state), whereas on ARM9 systems it contains the address plus three addresses (12/6 bytes).
- Breakpoint. A breakpoint can be triggered by an Embedded-ICE watchpoint, a software breakpoint instruction on ARMv5TE(J) targets or an external breakpoint signal (IEBKPT). If an instruction fetch causes a breakpoint to trigger, the instruction is still fetched into the pipeline and marked as breakpointed. If the instruction reaches the execute stage (i.e. it's not flushed due to a branch or exception entry), the core enters debug state without executing the breakpointed instruction. On both ARM7 and ARM9 systems, the PC contains the address of the breakpointed instruction plus three addresses. BREAKPT on ARM7 cores or SYSSPEED on ARM9 cores are low the first time they're scanned out of the debug scan chain after a breakpoint occurred.
- Watchpoint. Either an external watchpoint (DEWPT) or an Embedded-ICE watchpoint (nOPC positive match). The instruction causing the memory access and the immediately following instruction have been executed after a watchpoint triggered. Just as it's the case for a breakpoint, the PC contains the value of the next instruction plus three addresses on both ARM7 and ARM9 systems. A watchpoint is signaled by high values of BREAKPT and SYSSPEED the first time these bits are scanned out.
- Watchpoint + Breakpoint. The instruction immediately following a watchpoint may be breakpointed, in which case it's not going to be executed. This can't be detected on an ARM7 system, but on an ARM9 WPTANDBKPT may be examined to detect such a situation. The PC contains the address of the instruction to be executed next plus three addresses.

In addition to the debug reason detection via BREAKPT/SYSSPEED, newer ARM9 cores like the ARM9E-S Rev 2 and the ARM9EJ-S offer a Method of entry field in the debug status register (see Figure 3.8) with detailed information about the condition that caused debug entry [DDI0240A].

3.5 Core State

Once in debug state, a debugger may start to examine the core state using instructions scanned into the debug scan chain (see §3.2). The core registers of the current processor mode can be read using a "store multiple" (STM) instruction. The debugger puts the `STM` instruction in the processor pipeline, clocks the core by moving through Run-Test/Idle, and loads two additional "no operations" (`NOP`) into the pipeline. During the 4th cycle, the values of the registers referenced by the `STM` instruction start to appear on D[0:31] (ARM7) and DD[31:0] (ARM9), and can be capture by the debugger [DDI0100E, p. A4-84].
The current program status register (CPSR) may be read using a "move PSR to general purpose register" (`MRS`) instruction that moves the CPSR into one of the general purpose registers followed by a "store register" instruction that makes the value of that register appear on the data bus (D[0:31]/DD[31:0]). Saved program status registers of exception modes are handled similarly using the R bit of the `MRS` instruction that moves the SPSR instead of the CPSR [DDI0100E, p. A4-60].
Registers of other modes require a change to that mode which can be done using a "move to status register from core register" (`MSR`) instruction. While during normal execution, the current processore mode may only be changed when in privileged modes or on exception entry, there's no such restriction while the core is in debug state. A debugger can put a `MSR` instruction with an immediate operand in the processor pipeline, and may start examining registers of the new mode once the `MSR` instruction is completed [DDI0100E, p. A4-64].

3.6 System State

It's not possible to access system memory while the core is in debug state and clocked from DCLK, so the core must resynchronize to its main clock (BCLK/FCLK/MCLK, see §3.2). ARM cores define a JTAG instruction RESTART that's used to restart the core from debug state. The core resynchronizes to the memory system once the TAP controller reaches the Run-Test/Idle state with RESTART as the current instruction. Using load multiple (`LDM`) instructions executed at system speed to read system memory and store multiple (`STM`) instructions executed at debug speed to capture the read values a debugger may examine the system state. If system memory is to be modified, the operations may occur in the opposite order, using `LDM` at debug speed to load the new values into core registers and using `STM` to write them.
On ARM7 based cores, the instruction prior to the one that is to be executed at system speed has to be scanned into the core with the BREAKPT bit set high. ARM9 based cores require the instruction that should be executed at system speed to be scanned in with the SYSSPEED bit low, followed by a `NOP` with SYSSPEED high.
After the necessary instructions have been put into the processor pipeline, RESTART is loaded into the TAP controller, and the state machine is moved to Run-Test/Idle state. The core resynchronizes to the memory clock, executes the system speed access, and reenters debug state. A debugger should poll the debug status register to determine when the operation completed.

3.7 Exit from Debug State

Exit from debug state is similar to system state accesses, but instead of a load/store instruction a branch is loaded into the processor pipeline. A debugger has to restore the execution context (see list 3.1) before it may exit from debug state. This may be achieved using `MSR` instructions to modify the CPSR and

Table 3.3: LDR operation cycle timing (ARM7) with PC as destination

Pipeline stage	Cycle number	Action
Fetch	1	LDR instruction is fetched
Decode	1	instruction fetched, LDR is decoded
Execute	1	instruction fetched, LDR source address is calculated
Execute	2	nothing fetched, new PC is loaded from memory
Execute	3	nothing fetched, PC register is written
Execute	4	instruction fetched from new PC
Execute	5	instruction fetched from new PC+4

`LDM` instructions that load the core registers. Finally, the PC has to be reloaded, as it got incremented on every instruction executed during debug. `LDR` instructions that load the PC are similar to a branch, as they require the core pipeline to be flushed, and new instructions have to be fetched from memory. Table 3.3 shows the cycles executed on ARM7 cores [DDI0029G, p. 6-13], but the order of operations is the same for ARM9 systems. During Execute cycles 4 and 5, new instructions are fetched, and the PC is incremented. ARM9 systems may fetch the branch with SYSSPEED set and the following `NOP` during these cycles, requiring a branch back to the current instruction (-2 addresses), but ARM7 systems need an additional `NOP` during the 4th Execute cycle. This results in a `NOP` fetched during `LDR's` execute cycle 4, a `NOP` with BREAKPT set during `LDR's` execute cycle 5, and the final branch back to the last but two instruction (-4 addresses).
After the branch has been clocked into the pipeline, RESTART is selected as the current JTAG instruction. Once the TAP controller reaches Run-Test/Idle, the processor starts executing from the restored PC.

4 ARM MMU/Cache Handling

ARM cores with a MMU or Caches require special treatment. The debugger has to ensure coherency between the caches and main memory, while as much as possible of the cache and MMU state has to be preserved. This chapter is going to look at the implications for ARM720t cores (MMU and unified cache) and ARM920t cores (MMU and separate instruction and data caches). The same considerations are true for other cores, but the support provided by the on-chip debug facilities may be different, requiring different actions by a debugger. A basic understanding of MMU/Cache implementations in general are required, detailed information is given in [DDI0100E] and [Sf00].

4.1 Unified versus Separate Cache Implementations

Unified Cache (e.g. ARM720t)

Figure 4.1 shows the basic organization of a unified cache system like the ARM720t. The ARM7TDMI processor core issues virtual addresses to the MMU and the unified cache. While the MMU is turned off, the core still issues its addresses to the MMU, which passes them through unaltered, giving a flat 32 bit address space.
ARM7TDMI data reads from addresses that are already contained in the cache are satisfied from there. If the data isn't in the cache, the MMU checks its TLB to see if there's already a translation for the required virtual address. In cases where the TLB contains the required translation, and the virtual address is in a cacheable memory area, a line fetch is executed to fill the cache with data from the bus interface, which is then transfered to the ARM7TDMI. Uncacheable memory is transfered directly from the bus interface to the core. On a TLB miss, the MMU executes a page table walk to find a translation for the virtual address, which is written to the TLB. The MMU generates a translation abort, if no valid translation is found, and the core enters the data abort exception handler. Instruction fetches are similar, but an instruction abort is generated instead of a data abort.
The ARM720t has a write-through cache and operates on read-miss allocation [DDI0192A, p. 4.2]. Memory writes update the cache on a hit, but will always be written to main memory, too. Cache lines are only loaded or replaced on read operations - write operations that don't generate a hit inside the cache wont alter it.
Coherency isn't an issue for ARM720t based systems due to the unified write-through cache. It's sufficient to disable the caches during memory reads to ensure that cache content is preserved. Read hits are still served from the cache if it's disabled, [DDI0192A, p. 4.2], and cache misses lead to system memory accesses.

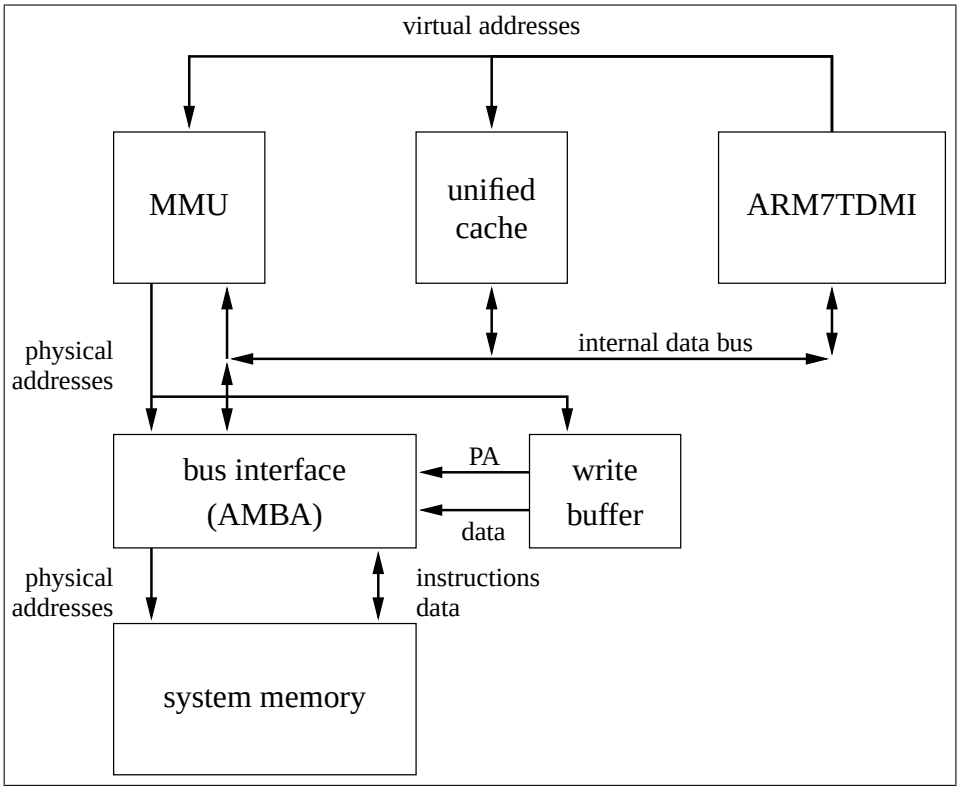

Figure 4.1: ARM unified cache organization (ARM720t)

Separate Caches (e.g. ARM920t)

Figure 4.2 shows the organization of an ARM920t processor with its separate data and instruction caches. Virtual/physical address translation is handled similar to the ARM720t, but two independent MMUs take care of instruction and data virtual addresses.
The ARM920t data cache can operate on a write-through or a write-back policy, depending on the configuration of the particular memory location, while naturally the instruction cache doesn't write any memory. Writes to write-through memory regions update the cache and are sent to the write buffer, too, while writes to write-back memory only update the data cache entry and mark it as dirty. Dirty cache lines are written to main memory if the cache line is to be replaced or if an explicit data cache flush was initiated. Both caches implement a read-miss allocation like the ARM720t.
Keeping the instruction and data cache in a coherent state is an important task of a debugger designed for ARM920t based systems, as accesses to the data caches and main memory wont affect the instruction cache. When the debugger modifies program code, for example by setting a software breakpoint, a write could go to the data cache only, in case of a write-back region. As soon as that instruction is to be fetched, the instruction cache is queried, and might return an instruction it fetched before the debugger modified the code. On an instruction cache miss, a line fill is executed, loading instructions from system memory, which may have outdated code, too. To ensure coherency and preserve as much of the cache state as possible in every possible case, it is important to carry out the following steps in case of a memory write while caches are enabled:

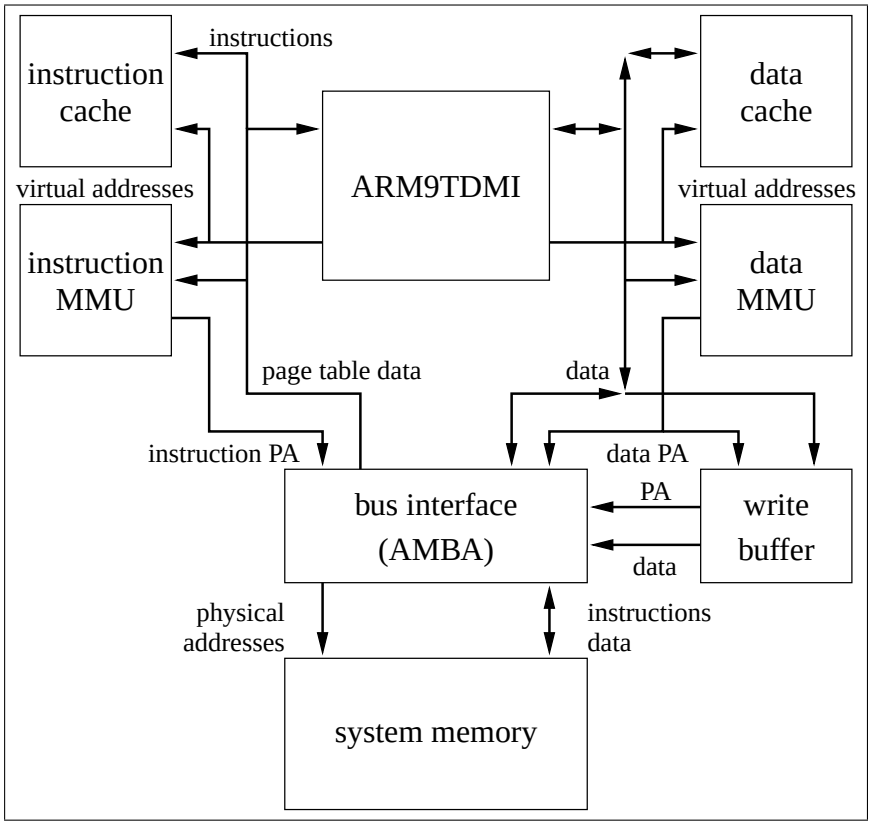

Figure 4.2: ARM separate instruction and data caches (ARM920t)

- Disable line fills for instruction and data caches on debug entry. This ensures that no cache lines are replaced during debugging.

- Examine the cacheable/bufferable bits for a memory location that is to be written. If the region is marked as write-back cacheable, execute either a cache flush, change the memory region temporarily to write-through, or write the memory twice, once using the virtual address while MMU and caches are enabled, and once using the physical address while MMU and caches are disabled. This guarantees that the data cache and system memory are in a coherent state.

- Invalidate the instruction cache for every address that was written. The core is going to execute a line fetch if it has to execute an instruction from an address that was invalidated before, fetching the modified code from system memory.

These steps are time consuming, if larger memory blocks have to be written, so a debugger might apply them only to small modifications, like writing of half-words and words. This ensures that breakpoints always affect instructions possibly contained in the instruction cache, while keeping the overhead for other operations to a minimum. Larger transfers typically affect either modified data, where coherency isn't an issue, or code download, in which case the user can explicitly specify that coherency is to be ensured, if this is desired.

4.2 System Control Coprocessor

The MMU, caches and other system features available only on cached systems are controlled by coprocessor 15 (CP15). It has a common programmer's model available on all implementations, while special features or restrictions only apply to selected cores. [DDI0100E, p. B2-1 ff.] gives detailed information about the system control coprocessor's programmer's model. This section is going to explain the use of functionality necessary or useful for a debugger.

- Main ID register. Accessible as register 0 with opcode2 set to 0. Contains information about the processor core like architecture version, part number and core revision number. Useful to determine if revision dependent features are available.
- Cache Type register. If available (like on ARM9 cores) this register gives detailed information about the cache type (write-back/through, supported functions), whether it's a unified cache or separate I/D caches, and the size and organization of the caches (line length, associativity, number of cache sets). This is especially important on cores with configurable cache sizes like the ARM926EJ-S. The Cache Type register is accessed as register 0 with opcode2 set to 1.
- Control register, CP15 Register 1. Used to control system features like the MMU and caches. See [DDI0100E, p. B2-13] and a particular core's technical reference manual for a list of available configuration options. This register should be made user-accessible through a debugger.
- MMU translation table base register, CP15 Register 2. Used as the offset to the first-level page table for a first-level descriptor fetch. Only bits 31 to 14 are used, the remaining 14 bits should be zero. The first-level page table is therefor aligned to a 16 kB boundary.
- Fault status register, CP15 Register 3. Contains information about the abort reason. On ARM9 cores there are two registers available, one for data aborts and one for prefetch aborts (instruction fetch abort). The instruction fault registers are only accessible from a debugger.
- Fault address register, CP15 Register 4. The address that caused an abort. ARM9 cores have two registers, one for data aborts and one for prefetch aborts.
- MMU, cache and write buffer control registers. The use of these registers is mostly implementation defined.
- FCSE ID register, CP15 Register 13. This register contains the fast context switch extension process id (PID) of the current process in its top seven bits. FCSE allows process memory to be relocated by replacing the top seven bits of a virtual address with the PID. This gives 128 process memory blocks of 32 MB size that can be switched without having to modify the virtual-to-physical address translation. Virtual addresses (VA) are first translated using the FCSE, producing a modified virtual address (MVA), which is then fed to the caches and MMU.

ARM720t CP15 Accesses

Coprocessor 15 registers may be accessed through a debugger using the JTAG boundary-scan chain 15 together with the INTEST instruction. Figure 4.3 shows the layout of scan chain 15, with data bits CPDATA[0:31] and a flag indicating whether the value represents data or an instruction. Coprocessor

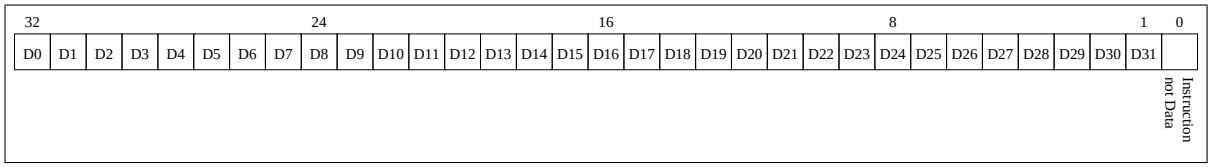

Figure 4.3: ARM720t scan chain 15 (CP15)

Table 4.1: ARM720t CP15 read operations

CPDATA31..0 (in)	CPDATA31..0 (out)	Instruction bit	Clock
coprocessor instruction	ignored	1	yes
NOP instruction	ignored	1	yes
NOP instruction	ignored	1	no
0x0	ignored	0	yes
0x0	read value	0	yes
NOP instruction	ignored	1	yes

instructions are executed by serially shifting them into scan chain, and moving the TAP controller to Run-Test/Idle where the coprocessor is clocked. [ARMFAQ2] gives an example on how to access coprocessor 15 using JTAG accesses. The CP15 follows the ARM7TDMI pipeline with its Fetch, Decode and Execute stages. An instruction that should be executed has to be scanned into the pipeline with the instruction bit high, followed by two `NOP` instructions. The last access to scan chain 15 before the value is read has to have the instruction bit low, indicating a data access. The coprocessor instruction is executed in the second Execute cycle, during which the debugger can capture the data. Coprocessor register writes are similar and require the new value to be scanned into scan chain 15 during the second Execute cycle. Table 4.1 shows the process of executing a coprocessor 15 instruction that reads a coprocessor register. The coprocessor instruction has to be built according to the register that should be accessed. The final `NOP` was necessary during all tests to ensure that CP15 operations worked properly. The only public documentation about ARM720t CP15 accesses is the FAQ entry [ARMFAQ2].

ARM920t CP15 Accesses

There are two access types for CP15 registers on ARM920t cores, physical access mode and interpreted access mode. Both use the JTAG boundary-scan chain 15, which may only be used together with the INTEST instruction. [DDI0151C, p. 9-32 ff.] lists the registers accessible by each of the two methods. The layout of scan chain 15 is shown in figure 4.4. The mode of operation is selected by bit 0 (next to TDO), with a 0 indicating an interpreted access and 1 a physical access.

Physical Access Mode

Scan chain 15 behaves similar to the Embedded-ICE scan chain when used in physical access mode with bit 0 set high. The data, address and nRW bits are serially shifted into the scan chain. During Update-DR the register is read or written, requiring an additional pass for register reads, where the value of the selected register is shifted out of the boundary-scan register.

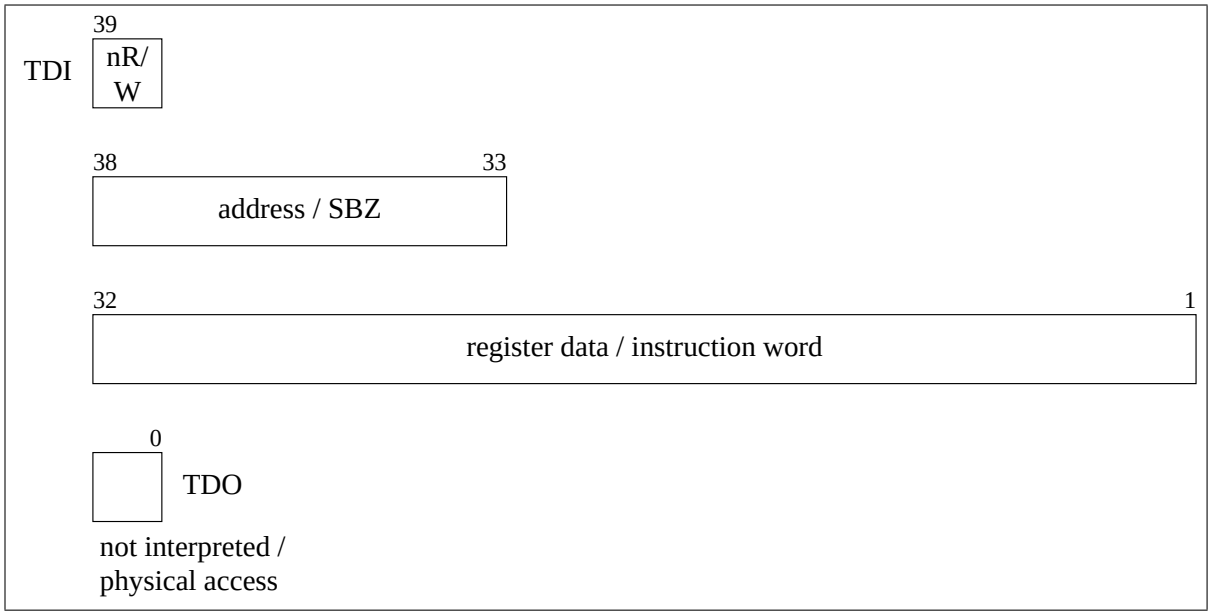

Figure 4.4: ARM920t scan chain 15 (CP15)

Interpreted Access Mode

Before an interpreted access may be executed, the CP15 test state register (register 15) has to be modified using a physical access to set the CP15 interpret mode bit [DDI0151C, p. B-4]. The desired coprocessor instruction is then scanned into scan chain 15 with bit 0 low to select interpreted access mode. The ARM9TDMI is used to execute a system speed load (CP15 register read) or store (CP15 register write) operation, which executes the coprocessor instruction, reading or writing the core register specified in the system speed load/store. After the coprocessor accesses are finished, the CP15 test state register has to be restored in order to disable interpreted access mode.

5 Requirements Specification

This chapter is going to analyze the requirements for the debugger, providing a basis for the architectural design and the following implementation. The Open On-Chip Debugger is going to run as a daemon process on a host PC, making use of a JTAG compliant hardware interface that connects to the target system. The target system consists of a SOC with an ARM7 or ARM9 core, memory in form of RAM and ROM, and peripherals like Ethernet, RS232, and CAN. The debugger should be able to load code into target memory, control code execution on the target, and examine the target state. In case there's no code on the target that takes care of required initial setup, the debugger has to carry out these setup steps. User interaction with the debugger ought to be possible via a command line interface that allows users from remote systems to log in and make use of the debug functionality, as well as through a GDB (The GNU Debugger) remote protocol server that lets a user take advantage of the sophisticated debug support available in GDB.
Figure 5.1 shows how the parts of the debugging environment are connected together. The debug host connects to the target microcontroller through some kind of JTAG hardware interface. Access to target's memory has to go through the target's microcontroller, as the debugger has no direct connection to the memory chips. During development, peripherals like Ethernet and RS232 are connected to the debug host, too, allowing target functions to be accessed.

The debugger will have to continuously evolve to support new or modified cores, to provide support for additional flash devices, and to allow additional JTAG hardware interfaces to be integrated. Debug statements should be left in the code, and may be enabled using a configuration option. It should be possible to change the amount of debug information presented to the user during runtime, to allow a developer to examine the debugger's behavior during selected operations.
Both a configuration file and command line arguments may be used to configure the debugger, and command line arguments should take precedence over configuration file options. It should be possible to select a configuration file via a commandline argument, to allow a developer to debug multiple targets or target configurations, without having to change or replace the configuration file every time.

5.1 JTAG

To allow easy adoption to different JTAG hardware devices, a common interface that abstracts the underlying hardware has to be defined. JTAG operations move the state machine, scan bitstreams into either a data- or the instruction register, and allow the Test Reset (TRST) line to be manipulated. ARM defines an additional System Reset (SRST), that allows core logic to be reset without affecting the test state and should be considered to be part of the JTAG interface, too. The Test clock (TCK) may be free running,

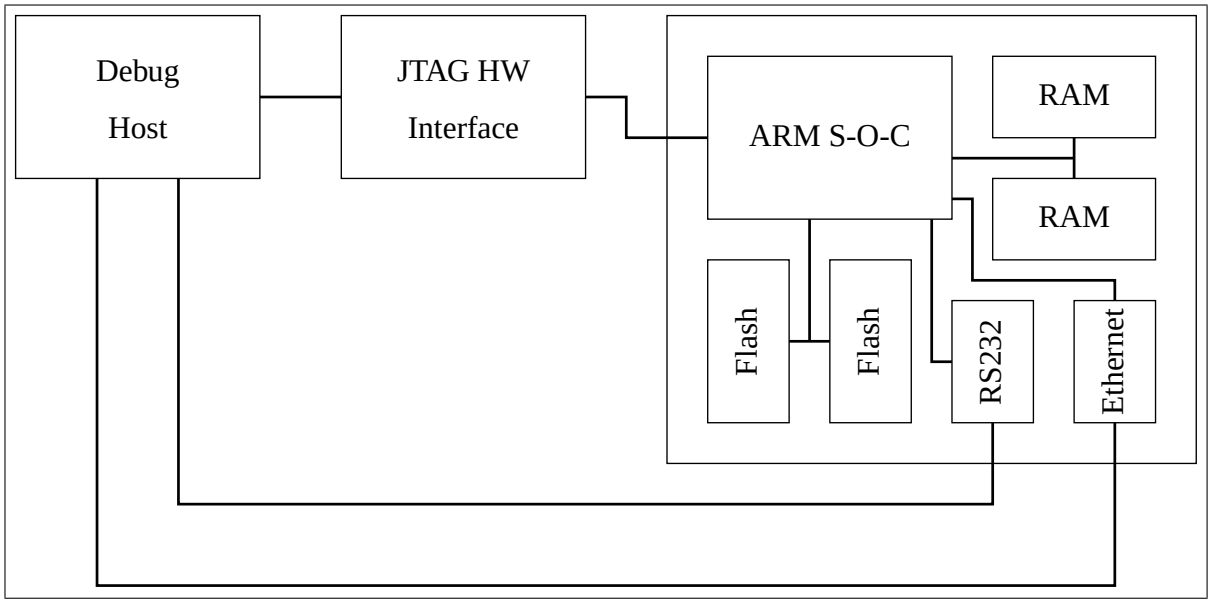

Figure 5.1: Debug environment

i.e. it runs even when there's no JTAG operation to be executed, or may be stopped during its low phase. To account for interfaces that don't allow TCK to be stopped, operations may only finish in stable states, where one of the two possible transitions leads back to the state itself. Test-Logic Reset (TLR), Run-Test/Idle (RTI), Shift-DR (SD), Pause-DR (PD), Shift-IR (SI) and Pause-IR (PI) fulfill this requirement. The two shift states are not suitable to finish an operation, as they modify the register content. Table 5.1 lists the possible JTAG operations.

The operations listed in 5.1 are of a very low level, and a large number of them is required to execute an operation in the target. Depending on the JTAG hardware and how it's connected to the debug host, a considerable amount of time may pass while a command is sent to the interface and the host waits for its completion. Devices connected via USB 1.1 for example send data in timeframes of 1 ms. If the host sends individual commands to the device, latency may limit the interface to a few hundred operations per second. To allow high throughput despite high latency, it should be possible to send many operations to the JTAG hardware before the host has to wait for their completion.
The JTAG interface is defined by the hardware being used, the current state inside the state machine, the endstate in which JTAG operations should be finished, and the currently selected instruction.

5.2 Target

The debugger's goal is to support ARM7 and ARM9 based targets. While these share some common debug functionality, there are differences between the two core families, and between the members of each core family. An abstract interface should encapsulate as much debug functionality as possible, while target specific extension give access to non-standard functions. Table 5.2 lists the operations that have to be implemented by each target.

Table 5.1: JTAG operations

Operation	Description
Endstate	Changes the currently selected endstate, affecting following operations but not causing any immediate action.
Reset	Reset the JTAG state machine, either by pulling nTRST low, or via five TCK cycles during which Test Mode Select (TMS) is held high. This moves the state machine from every possible state to Test-Logic-Reset.
IR Scan	Move the state machine from its current state to Shift-IR, serially shift a binary pattern into the instruction register, and capture the shiftregister output. When the scan is finished, move the state machine to the currently selected endstate.
DR Scan	Similar to an IR Scan, but operates on the currently selected data register instead of the instruction register.
Runtest	Move the state machine to Run-Test/Idle, and execute a given number of TCK cycles, before moving to the selected endstate.
Statemove	Moves from the current state to the selected endstate.
TRST	Change the TRST line as requested
SRST	Change the SRST line as requested

Table 5.2: Common target operations

Operation	Description
Poll target state	Retrieve information about the current target state. If the target entered debug state since the last poll, the current execution context should be captured and saved, to allow the target to be resumed later.
Architecture state	Retrieve architecture specific information about the current target state. This should give a quick overview about target dependent state information, like the state of the MMU or caches (if available), or important target registers like the program counter (PC) or the current program status register (CPSR)
Halt	Forces the target into debug state. No code should be executing on the target, and the debugger should be able to examine and modify the target's state.
Resume	Makes the target leave debug state. The execution context has to be restored, and the target should start executing instructions from where it was stopped, or at a new address, if this is desired.
Step	The target leaves debug state, executes exactly one instruction, and reenters debug state. Before leaving debug state, the execution context has to be restored. How stepping is implemented depends on the actual core version.

Reset	A warm reset of the target is executed. This resets all system functionality, but leaves the debugger in control of the target. It should be possible to halt the target immediately after coming out of reset, and target specific initialization may be carried out if desired. If neither halt nor initialization is required, the target may start executing instructions from the reset vector.
Get/Set GDB register(s)	Because a GDB remote protocol server is a major requirement for the debugger, every target has to implement access to registers in accordance with the GDB remote protocol. This allows a single GDB stub to be used for all targets, as the target specific ordering of registers is done by the target. The ARM GDB remote protocol expects registers to be transfered in a bytestream of the target's endianess, starting with the core registers *r0* to *r15* (each 32 bit), the floating point registers *f0* to *f7* (each 96 bit), the floating point status register *fps* and the current program status register *CPSR*. If a single register is requested, a number, starting with zero and ordered like the full register list, is given to each register.
Read/Write memory	Access to target memory using load/store instructions of a specified size should be possible. ARM cores are 32 bit cores that support 8, 16, and 32 bit accesses. The debugger may specify any number of items of the selected size to be read or written. The target should take care of memory consistency between caches and main memory. Memory accesses may produce a data abort, in which case the debugger should restore the target state and inform the user about problems this may have caused.
Add/Remove breakpoints	It should be possible to set breakpoints of a selected size on addresses aligned to that size. This allows both ARM and Thumb state breakpoints to be specified. If desired, the breakpoint may be specified as a hardware breakpoint, in which case no memory modifications are necessary. Otherwise, the breakpoint should be implemented as a software breakpoint that replaces the original instruction in the target memory with a special pattern that forces the core into debug state once it is executed.
Add/Remove watchpoints	Watchpoints, that monitor a given memory region for reads, writes, or both, may be set using an address and a mask that specifies which bits of the accessed address should be ignored when comparing it to the monitored memory region. Watchpoints should only be implemented using dedicated HW resources, because monitoring every instruction that is executed for the memory it accesses (SW watchpoint) would be too time consuming.

A target may be either little- or big endian, and the debugger may be running on a little- or big endian host. The GNU Debugger (GDB) expects all data to be transfered in target endianess, so this should be the endianess used while the data is handled by the debugger, too. If data is to be presented to the user, it should be evaluated according to the target's endianess.

On startup, the target is in an unknown state, as the debugger can not know if the target is halted or running. After the debugger examined the target state, the target may be in running state, where it is executing instructions at full system speed. If the target was already halted when the debugger examined it for the first time, the target remains in an unknown state, as the debugger can not know if the core is in a state where it's safe to be resumed later. When a user requests a target reset, or if the debugger is configured to reset the target on startup, the target is in reset state until it's either resumed or put into debug mode. A configuration option should allow a user to specify what the debugger does when it is started and connects to the target.

Core Differences

While an abstract target interface allows different cores to be controlled using a common set of operations, there are many similarities between ARM7 and ARM9 cores that may be handled by the same code, making an additional interface necessary that gives common code parts access to target specific operations.
All ARM cores use a scan path select register and the SCAN_N JTAG instruction to determine which boundary-scan register is being accessed, but this register is four bits wide on ARM7 cores, and five bits wide on ARM9 cores. This information should be available to common code that selects the current boundary-scan register.
The Embedded-ICE unit has a debug comms control register that contains the version of the Embedded-ICE interface implemented by the core. Depending on that version, the debugger may decide what debug functionality is available. The Embedded-ICE registers provided by a particular core and their size may also vary, and a target should make the layout of its Embedded-ICE registers available to common code that handles access to these registers.
Common code should be able to handle the core's current state, and needs access to the target specific operations listed in table 5.3.

ARM7/ARM9 MMU and Cache Support

Every target should provide access to as much information as possible. Targets with a system control coprocessor (CP15) should allow the user to read and write coprocessor registers. The format of the coprocessor register accesses depends on the core family. On ARM720t cores, CP15 is accessed using coprocessor instructions as if they were executed in the program code. ARM920t cores provide two access methods (see §4.2), a user should refer to the technical reference manual for a list of registers accessible with each method [DDI0151C].
Cores with a MMU should provide a command that allows a user to translate virtual addresses to physical addresses. All ARM7 and ARM9 cores implement similar address translation, making it possible to support both with a single solution. To allow common address translation for all cores, the debugger needs to be able to access memory using physical addresses to simulate the page table walking. It further has to turn off the MMU and needs to read the translation table base. While ARM9 cores support tiny pages of 1 kB size, the smallest pages usable in ARM7 systems are 4 kB small pages, and a page descriptor that indicates a tiny page is an error on these cores. A common interface has to be defined that encapsulates target specific access to the required functions.

Table 5.3: ARM core specific target operations

Operation	Description
Examine debug reason	While all ARM cores may enter debug state because of the same reasons, it depends on the debug functionality provided by each core how this information is accessed.
Save execution context	Once the core entered debug state, its current execution context has to be saved. This includes the core registers of the current processor mode on all cores, but may extend to state information specific to a particular core, like the instruction/data fault status and address registers on ARM9 based cores with a MMU.
Save full context	Registers that don't belong to the processor's current mode only have to be read if they are explicitly requested, as they're safe from getting modified during normal debug operation.
Restore execution context	Before the core may be restarted (for resume or single-stepping), the execution context has to be restored. Every register that has been modified by debug operations and every register explicitly changed by the user has to be written back to the core.
Step	Execute a single step. The context has already been restored.
Resume	Resume the core. The context has already been restored.

5.3 Flash

On ARM7/ARM9 based systems, flash memory is handled by writing commands to addresses assigned to the flash chip. Status information for example is obtained by writing a status command, followed by reading from an address located within the flash. The actual procedure depends on the flash chip in question and varies among vendors and product families. Flash writing thus requires working memory accesses for a given core.
Flash chips on a target should be erasable and programmable through the debugger. If a flash chip provides a hardware protection mechanism, it should be possible to set and remove this protection. Because additional flash chips should be easy to include, an interface that describes the operations on a flash chip is required. The code necessary for a new flash chip is then limited to the implementation of that interface without having to touch other parts of the debugger.
A flash chip may be 8, 16, or 32 bits wide, and is connected to the microcontroller using a bus of 8, 16, or 32 bits. It's possible to combine multiple chips of a uniform size to form a bus wider than the width of each chip. One or more chips form a flash bank that starts on a certain memory address, and the width of the bus and the chips determines how many chips form a single flash bank. The flash interface should provide operations to probe if flash memory is located at a given address, report information about the blocks of a flash bank located at a given address, set or remove the protection of selected blocks, erase blocks, and write binary data at a selected offset on a flash bank.

5.4 User Interaction

A command line interface should be realized using a telnet server embedded in the debugger. A user connects to the server process using a telnet client, allowing a single debug system to be used by different users at possibly remote locations. The use of a remote interface further allows easy integration of the software into a standalone debugging solution without the need for a host PC with JTAG hardware. The telnet interface should be conformant to [RFC854], with support for advanced features described in later telnet specifications. The TCP port on which the server listens for incoming connections should be user configurable, and should use 4444 as a default if no other port is specified.

The GNU Debugger (GDB) implements a remote protocol for use over serial lines that's also being used on TCP/IP network connections [GDB01]. The protocol specifies packets that are used by GDB to control the target's operation. Packets are introduced by a `$`, followed by the packet data, and finished by a `#` and a two hex-digit (8 bit) checksum that is calculated as the sum of all characters between `$` and `#` modulo 256. The target replies to commands with packets of the same format. After commands that resume or single-step the target, the reply is delayed, until the target reenters debug state. While the target is running, a GDB session can be interrupted by a user, usually using '^C'. This event is transfered as a binary 0x3 to the target, and not enclosed in a normal GDB packet. The GDB remote serial protocol server should use a user configurable TCP port, or 3333 as a default if no other port is specified.

5.5 Quality and Performance

It is important that a debugger never displays wrong or inaccurate information. In case of a problem, the debugger has to inform the user that something might have gone wrong, and it has to abort execution if an unrecoverable error occurred. The daemon process should provide four levels of information:

- Error messages that are fatal for the program's further execution.
- Warnings that indicate a problem, but allow the program to continue execution.
- Informational messages, that are generated during normal program execution. They give a user additional information about the debuggers operation which may be helpful when diagnosing a problem.
- Debug messages, which may occur at a high rate. These should be used to identify problems during further development of the debugger, to allow easy adoption to new cores, and to assist in adding support for new flash devices and JTAG hardware interfaces.

Debuggers for embedded systems are often measured by the speed at which it's possible to download code into the target's memory. It is important to optimize the debugger for high download speeds, as the size of the data that has to be transfered may be large compared to the speed that is typically achieved. A Linux kernel image and a ramdisk that should be tested may require several megabytes to be transfered to a target, and simple JTAG interfaces only allow speeds up to a few kilobytes per second.

Single-step operation is another aspect that's sensitive to performance. A user might want to trace through the code executing on the target. This makes it necessary to reenter debug state, save the core's execution context, restore register modified by the debug entry, and resume the core after every instruction executed.

6 Design and Architecture

This chapter is going to describe the design and architecture of openocd, the Open On-Chip Debugger, based on the requirements specified in the previous chapter, laying a foundation for the implementation to build upon.

6.1 Software Modules

The software should be modularized, to allow subsystems to be easily extended or exchanged, without having to worry about dependencies in other parts of the code. Figure 6.1 shows the modules and their interaction. The daemon is the first part that is initialized, and controls the remaining components. It manages the program configuration by evaluating the command line arguments, and uses the CLI module to parse the configuration file. Every module may register commands with the CLI module that are used as configuration statements, user commands, or both, in case a configuration option may be changed later. The JTAG module is the only part that accesses the hardware being debugged, and provides an interface for other modules to communicate with the target. The Target module encapsulates common debug functions and registers target specific commands with the CLI module. It is accessed by the daemon to keep track of the target state while the daemon waits for new connections, and offers target debug functionality to the GDB and CLI modules. The GDB module is invoked after the daemon

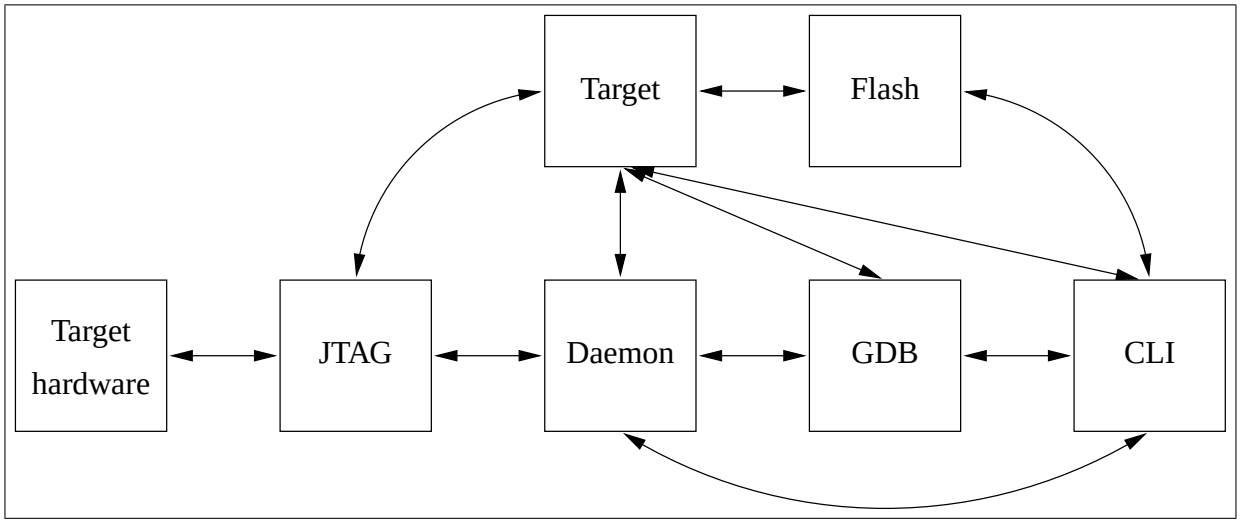

Figure 6.1: Openocd modules

accepted a new GDB connection, and maps the GDB remote serial protocol to the interface provided by the target module. A monitor packet is specified by the GDB remote protocol, that allows commands to be sent which should be interpreted by the GDB server. The GDB module hands these command strings to the CLI module for evaluation, and returns any replies to the remote GDB, extending the functions accessible from within a GDB session to all commands provided by the various modules. The CLI module implements the telnet server, handles configuration parsing and allows other modules to register commands. The Flash module is only accessed via the command interface, as the GDB protocol doesn't specify flash operations, and uses memory access functions provided by the Target module to program the flash chips.

6.2 Configuration Management and CLI Module

The debugger configuration is determined by command line arguments, a configuration file, and any changes a user makes at runtime using configuration commands. The configuration file should be used both to configure the debugger and to describe necessary target initialization. Because the commands used to initialize the target are the same as those that will be used later on the command line interface, a configuration file format that allows arbitrary commands to be specified is required. Simple formats that use key-value pairs aren't sufficient, and complex formats based on the Extensible Markup Language (XML) are inconvenient to be used for files that will be manually edited.
The LGPL licensed libcli (`http://sourceforge.net/projects/libcli/`) implements a telnet server with command line editing capabilities, and allows command strings and corresponding call-back functions to be registered. In addition to the telnet interface, the library provides a function to interpret a text file as if the lines were entered at the command line interface. When the library encounters one of the registered commands, it invokes the call-back function, and supplies that function with an array of all the arguments given to the command. The libcli library defines two modes in which it evaluates commands, an execution mode and a configuration mode. Commands may be specified to be available in one or both modes. It's possible to limit a command's scope by parsing the configuration file in configuration mode, and setting execution mode for all interactively entered commands. This ideally fits all requirements, and provides configuration commands and a telnet based command line interface using a single library.
The number of configuration options that may be changed using command line arguments should be kept small, to avoid a large number of arguments a user would have to remember. Options that may be specified on the command line have to be initialized to an invalid value. This allows the configuration command handler to determine if the option was set on the command line (which takes precedence), or if it contains an invalid value, in which case the value from the configuration file may be used. If a module is called and finds one of its options still uninitialized, it may choose a default value, turn off a feature that would be configured by this option, or signal an error that forces the debugger to quit, if the option is essential for the debugger's operation.
Every module registers its configuration commands with the CLI module. When the configuration file is parsed, these functions get called, and allow further configuration commands to be registered. This allows different module implementations (like targets or JTAG interfaces) to use the same command name for different implementation specific functions, because only commands that belong to the currently configured implementation are registered with the CLI module. The following example shows an excerpt from an openocd configuration file that configures the debugger for use with an arm920t target, and sets the arm7/arm9 specific option 'breakmode' to the value 'hw'.

```
#target
target arm920t
...
#arm7/9 specific
breakmode hw
```

The configuration command 'target' was registered by the target module. When it's called, it searches a list of implemented targets for the name specified. If a matching target is found, this is asked to register commands specific to it. The ARM920t is handled by code common to all ARM7 and ARM9 targets, and registers commands specific to these targets, like the breakmode command that enables software breakpoints. The breakmode command has already been registered when it is encountered, and the corresponding call-back function is called.

The configuration file is also used as a script of initialization commands that should be executed on the target, for example to setup an SDRAM controller, or to configure the buswidth of connected flash chips. When the debugger is set to perform target initialization after the target has been reset, the configuration file is parsed in libcli's execution mode, ignoring all commands that are registered for configuration mode.

6.3 JTAG Module

The JTAG module provides access to JTAG operations. Different JTAG hardware interfaces define their own implementation of an abstract interface, and may use common code that's suitable for many different devices. The JTAG interface defines two levels of commands: High-level commands, that read or write test data registers and control the TAP state machine, and low-level commands, that directly modify the JTAG signals (so-called bit-bang operation). Interfaces that implement the high-level command set don't have to provide the low-level commands, while simpler devices (like Wiggler compatibles) that only offer direct control of the JTAG signals use the high-level bitbang operations defined by the JTAG module. The JTAG module keeps track of the currently selected instruction, and provides CLI commands to adjust the JTAG device speed and to read the IDCODE register of a connected target.
To support JTAG hardware interfaces with high latency, all performance critical code queues JTAG commands until their results are required. Queued data register scan commands specify the new value that should be written into the register, and may specify a memory location where data scanned out of the register should be stored. A JTAG data register is described by multiple fields of up to 32 bits length, which is sufficient for the 32 bit architecture of ARM7 and ARM9 based systems.

6.4 Target Module

The Target module encapsulates the debug functionality, and defines CLI commands useful for all target implementations:

- Configure the startup mode. The debugger may just attach to the target, reset the target and let it run, reset the target and put it into halt mode, or reset the target and initialize it.
- Configure the target's endianess.

Table 6.1: ARM7 and ARM9 target modules

<table>
<tr><td colspan="2">ARM7</td><td colspan="2">ARM9</td></tr>
<tr><td>ARM720t
- Memory access
- MMU support</td><td>ARM740t
- Memory access
- MPU support</td><td>ARM92xt
- Memory access
- MMU support
- CP15 access</td><td>ARM940t
- Memory access
- MPU support
- CP15 access</td></tr>
<tr><td colspan="2">- CP15 access</td><td colspan="2"></td></tr>
<tr><td colspan="2">ARM7TDMI
- Debug instruction execution
- Step and Resume</td><td colspan="2">ARM9TDMI
- Debug instruction execution
- Step and Resume</td></tr>
<tr><td colspan="4">ARM7 / ARM9 common
- Core registers
- Embedded-ICE access
- Debug mode request
- JTAG instructions
- Breakpoint and Watchpoint handling</td></tr>
</table>

- Control the target execution. A user may halt, single-step, resume, or reset the target.
- Poll the target state. This calls the target interface, and provides the user with information about the current target state.
- Read memory. Memory may be read using any number of accesses of 8, 16, or 32 bits size.
- Write memory. A single memory location may be written using an access of 8, 16, or 32 bits size.
- Load binary file. This downloads a binary file from the host PC to the target at a selected memory address. To achieve high speed, memory is accessed using 32 bit accesses.
- Dump binary file. A selected part of target memory is read using 32 bit accesses and written to a file on the host PC.

ARM7 and ARM9 Common Code

Table 6.1 shows how the ARM cores of the ARM7 and ARM9 family (without ARM9E members) build upon common functionality. There are various dependencies between these parts: Target specific parts, like memory accesses for example require access to the Embedded-ICE registers, which can be shared between all implementations. Common parts on the other hand, like handling software breakpoints, require access to the target specific implementation of memory accesses. These dependencies are solved using interfaces that give common parts access to the required target specific implementations, and by making common parts known to all cores that build upon this common code.

Target State Management

The Target module keeps track of the current target state. Once the debugger is launched, its view of the current target state (unknown, running, halted, or reset) is initialized to report an unknown state. The

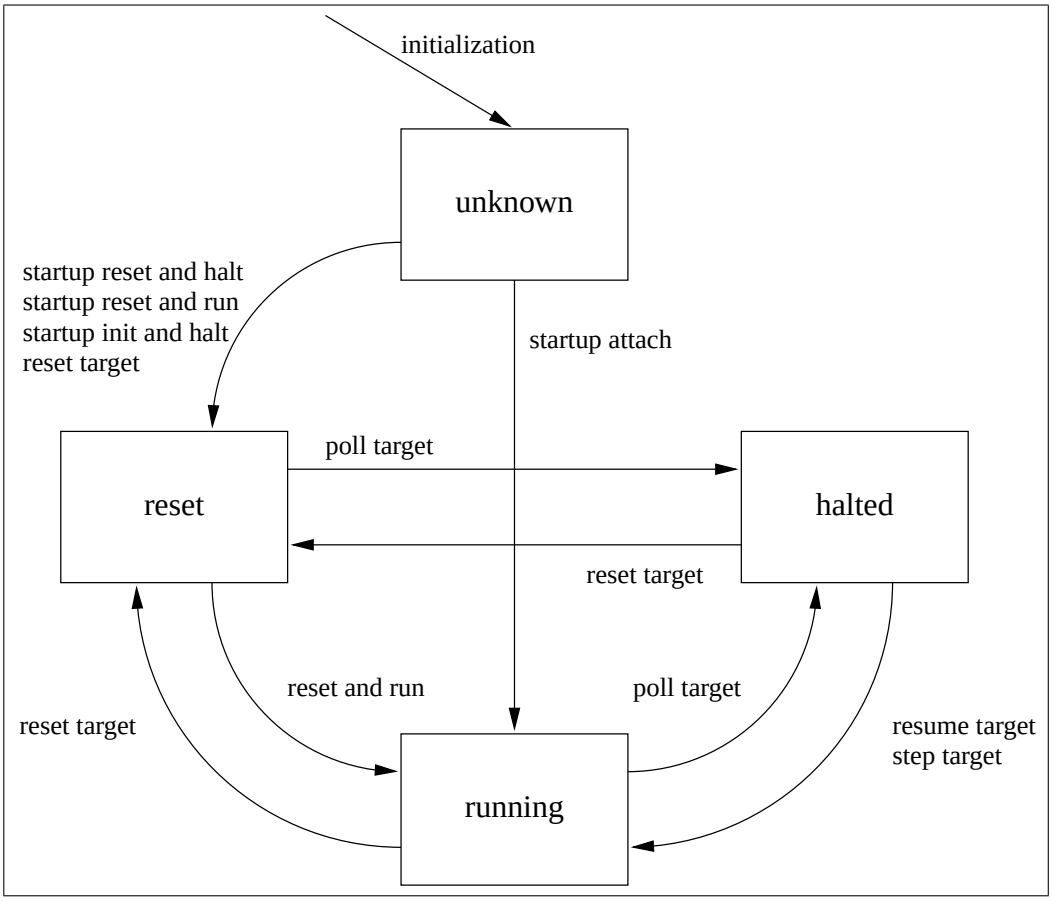

Figure 6.2: Target state

target state is polled by the currently active server module, which is either the Daemon, the CLI, or the GDB server, using calls to the target module. If the target module detects an entry into debug state, it handles that state change, and sets a flag signaling debug entry to other modules. The server modules use this flag to inform the user about state changes, and may request additional information from the target module, like the current architecture state or the current register content (see table 5.2).

The possible transitions of the target state are shown in figure 6.2. From unknown state, the target enters either the reset or the running state, depending on the selected startup mode. It's not possible to know if the target is in a sane state when the debugger detects a halted target when it first polls the target, so the target remains in unknown state in that case, until the user explicitly requests a target reset. Reset is only a temporary state, which is left by moving into halted or running state, depending on the reason for entry into reset state (selected startup mode or reset command). While in halted stated, the execution context is preserved, and debugging functions may be executed, like examining registers and memory read and write accesses. Halt state is left when the user requested the target to be resumed, on single-step requests, and if a target reset should be executed. Before moving from halt state to the running state, the execution context has to be restored.

During running state, the target is polled to detect entry into debug state. The running state is left when the target module detects a debug entry or if the user requested a target reset.

The target state is expected not to change on other occasions than those mentioned above. Especially, asynchronously resetting the target hardware should be avoided, as this changes the target's state without giving a debugger a chance to detect this change. The SRST line is designed to be bi-directional, so it should possible to detect such an event, but on many targets, the SRST line is only connected uni-directional.

Breakpoint Handling

Breakpoints are kept in a list, with information about the address they affect, if it's an ARM or a Thumb state breakpoint, whether it's a software or a hardware breakpoint, if it's currently set, which comparator it uses in case of a hardware breakpoint, and the original instruction in case of a software breakpoint. When a breakpoint is to be added, the list of breakpoints is searched for a previous breakpoint set on that address, in which case no further actions are taken. Otherwise the breakpoint is added to the list, and the debugger sets the newly added breakpoint.

A set hardware breakpoint is associated with one of the two Embedded-ICE comparators, and forces the target into debug state if an instruction from that address is fetched and executed. In case of a software breakpoint, a target dependent instruction code has been written into the target's memory, and the original content of that address is stored in the breakpoint list.

When the target resumes execution, the list of breakpoints has to be searched for the address at which the target should be resume. If there's a breakpoint set on this address, it has to be disabled, or the debugger would reenter debug state without executing that instruction. Disabling a breakpoint means either disabling the comparator with which it is associated, or restoring the original instruction in case of a software breakpoint. The debugger single-steps over that address, and reenables the breakpoint. The debugger is then resumed, no matter if there's a breakpoint set or not. This makes it trigger on breakpoints set on the next but one instruction, but not on breakpoints set on the next address. Single-stepping is similar, as a breakpoint set on the current address has to be disabled, too, or the target wouldn't execute the instruction.

If a breakpoint should be removed, its address is searched in the list. When the address is found, the breakpoint is disabled and then removed from the list.

Watchpoint Handling

Like breakpoints, watchpoints are kept in a list, with information about the memory region they affect, whether they should trigger on reads, write, or both, and on which of the two comparators they are set. Watchpoints don't have to be disabled, as any instruction that causes a watchpoint to trigger is executed before debug state is entered.

Before a watchpoint is removed, the associated comparator is disabled and marked as free, then the watchpoint is deleted from the list.

6.5 Flash Module

The Flash module uses the target module to access memory. The flash interface has to be implemented for every Flash type that should be supported. The module registers CLI commands that configure the flash used on the target, and commands that allow a user to access flash functionality:

- Configure the size of one flash chip.
- Configure the width of a single flash chip. Flash chips like the Intel Strata Flash (28FxxxJ3) can be used in byte (8 bits) or in half-word (16 bits) mode, depending on how they're wired to the microcontroller.
- Configure the width of the bus that connects a flash bank to the microcontroller.
- Probe a given address for the presence of a flash bank that matches the configuration.
- Print information about the blocks of a flash bank at a given location. If the flash supports a protection mechanism, information about the protection state should be printed, too.
- Erase a range of blocks on a flash bank at a given location.
- Write a binary file at a selected offset on a flash bank at a given location.

6.6 GDB Module

The GDB module accepts packets from a remote gdb client, validates the checksum, and calls the appropriate target functions. It registers a call-back function with libcli that takes any output generated by commands processed by the CLI module, and sends them as a reply to a connected gdb client. When a monitor command is submitted to the GDB server, it hands the command unaltered to the CLI module.

The gdb dynamically handles breakpoints and watchpoints. Once a target is resumed, all currently enabled breakpoints and watchpoints are sent to the debugger. When the target reenters debug state, they are removed again. If a breakpoint is set on the current address, that breakpoint isn't sent to the debugger. This means that the debugger must not disable any breakpoints while a gdb client is connected, and any encountered breakpoint should trigger, even if it's set on the current address. This scenario may occur with multiple successive breakpoints, in which case the gdb steps over the first instruction, sets all breakpoints, and resumes the target on an address with a breakpoint set.

7 Implementation

This chapter is going to show how the design and architecture described in the previous chapter is implemented. During this chapter, the term "the software" shall refer to the Open On-Chip Debugger (openocd), version 0.3, as released on July, 17th 2005. Excerpts from the code use line numbers from this version of the program code. The implementation of one JTAG hardware interface (USBJTAG-1/ftd2xx) and one target (ARM920t) will be explained in detail, as the other interfaces and targets implemented are similar enough to be documented by the source code alone. All code is written in C, and tries to use only a minimum of external functionality, to allow easy porting of the code to different host platforms.

The software's code is located in several subdirectories below the `./src` directory, with only the code for the daemon residing in the top level directory. Every subsystem has a directory of its own, and will be linked into a static library. These libraries, together with external dependencies and the code of the daemon, are linked into the executable.

```
./openocd.c
./jtag/*.[ch]   -> libjtag.a
./target/*.[ch] -> libtarget.a
./flash/*.[ch]  -> libflash.a
./gdb/*.[ch]    -> libgdb.a
./helper/*.[ch] -> libhelper.a
```

7.1 Program Subsystems

daemon

The daemon is responsible for setting up the other subsystems and for accepting new connections to the telnet and gdb ports. It also manages the program shutdown, by checking the target state and resuming it in case it was left halted, and calls deinitialization functions of the other subsystems to give them an chance to free used resources.

Listing 7.1: ./openocd.c

```
65 /* shutdown_openocd == 1: exit the main event loop,
           and quit the debugger */
66 int shutdown_openocd = 0;
```

The `shutdown_openocd` flag is used to tell the daemon that a shutdown has been requested. It may be set by the CLI handler for the `shutdown` command or by the `sigint_handler()`:

Listing 7.2: ./openocd.c

```
74 /* allow shutdown with SIGINT to the daemon */
75 void sigint_handler(int sig)
76 {
77     signal(SIGINT, sigint_handler);
78     shutdown_openocd = 1;
79 }
```

The handler reinstalls itself as the signal handler for a SIGINT signal (otherwise it would catch the signal only once), in case the possibility to abort a shutdown is added later, and sets the `shutdown_openocd` flag to 1. When this flag is evaluated next time, the daemon will initiate the shutdown procedure.

Listing 7.3: ./openocd.c

```
74 /* daemon initialization and main loop */
75 int main(int argc, char *argv[])
```

`main()` performs various initialization tasks, creates the sockets for telnet and gdb connections, waits for incoming connections, and supervises the program shutdown. It initializes the CLI, calls `jtag_register_commands()`, `target_register_commands()`, and `flash_register_commands()`, invokes `parse_cmdline_args()` and `parse_config_file`, and calls the `xxx_init()` functions of the jtag, target, and flash subsystems. It implements a simple TCP server that waits for new connections, accepts those, and gives the corresponding file descriptor to the appropriate server module (CLI or gdb).

libcli

The libcli library (`http://sourceforge.net/projects/libcli/`) is used to implement the command line interface (CLI), the telnet server through which the CLI is accessible, and the configuration command handling. It was written to mimic the configuration interface found on Cisco networking products, and is licensed under the terms of the GNU Lesser General Public License (LGPL).

The library is initialized by calling `struct cli_def *cli_init()`, which returns a value of type `struct cli_def*`, that has to be passed to all other libcli functions. Several functions allow the appearance of the CLI to be adjusted, like setting the hostname, defining a command line prompt, or setting a banner that is shown when a client connects.

```
cli_register_command(struct cli_def *cli, struct cli_command *parent,
 char *command, int (*callback)(struct cli_def *, char *, char **, int),
 int privilege, int mode, char *help)
```

`cli_register_command()` adds a new command to the CLI. The `command` parameter is the command's name, the `callback` is a pointer to a function that's called to handle the command. The `privilege` would allow multiple levels of access to be set up, but that's unnecessary for the purposes of this software. `mode` may be `MODE_EXEC`, which specifies that the command may only be used while in execute mode, `MODE_CONFIG`, specifying commands that are only executed during configuration mode, or `MODE_ANY`, indicating that the command may be executed at any time. The distinction between configuration and execute mode comes from the Cisco interface this library tries to simulate. On these systems, the normal mode of operation is "execute", and only after entering a special statement, "configuration" commands may be executed. For this software, the two modes serve perfectly to distinguish between configuration commands and CLI commands. While the command line options and the configuration file are being

parsed, the libcli is driven in MODE_CONFIG state, and is then switched to MODE_EXEC, when the configuration parsing is done. The help strings are displayed on the CLI when a user entered the help command.

```
cli_regular(struct cli_def *cli, int(*callback)(struct cli_def *))
```

A timeout mechanism is implemented in libcli to give other parts of the software a chance to run code on a regular basis. The callback function specified by cli_regular() is called every second while a client is connected to the telnet server. A future version of the library may allow to select the timeout, which would allow the debugger to react faster on changes of the target state.

./helper

The helper directory contains functionality that doesn't belong to any of the modules, but is used by those, like centralized logging functionality, commonly used user-defined data types, and the program configuration management.
types.h defines three unsigned data types of 8, 16, and 32 bits size, that are used to access binary data transfered from the target. These definitions are probably not portable to systems with a native integer size of more than 32 bits, and would have to be adapted to such systems.

Centralized logging is implemented in error.[ch], and was inspired by the logging code found in the Input Abstraction Layer (IAL), by Timo Hönig. The logging code defines four levels of log information, with an increasing volume of messages. LOG_ERROR defines messages of the highest priority, which indicate fatal errors that cause the program to abort execution. LOG_WARNING messages inform about problems, that can likely be dealt with. LOG_INFO is for informational messages, that may help when a problem with the software arises. LOG_DEBUG is used for debug statements and trace marks, that show which parts of the code were executed.

Listing 7.4: ./helper/error.h

```
31 enum log_levels
32 {
33     LOG_ERROR = 0,
34     LOG_WARNING = 1,
35     LOG_INFO = 2,
36     LOG_DEBUG = 3
37 };
38
39 void log_printf(enum log_levels level, const char *file, int line,
40     const char *function, const char *format, ...);
41
42 extern int debug_level;
43
44 #define DEBUG(expr ...) \
45     do { \
46         log_printf (LOG_DEBUG, __FILE__, __LINE__, __FUNCTION__, expr); \
47     } while(0)
```

The global variable debug_level, that may be changed using a command line option or a configuration command suppresses messages of a higher level. Inside log_printf(), a buffer is filled with the supplied format string and its arguments, and then printed to stderr. A macro is defined for each message type, that prepends a printf-style format string and its arguments with information about the

level of that message, and the source file, line number, and function that printed the message. The above code shows the definition of the `DEBUG` macro, the other levels are defined similarly. The **`do ... while`** loop around the call to `log_printf()` allows the macro to be used like a function call, with a finishing semicolon.

```
if (expr)
    DEBUG("expr was true");
else
    DEBUG("expr was false");
```

Without the enclosing **`do ... while`**, the above example would expand to

```
if (expr)
    log_printf (LOG_DEBUG, __FILE__, __LINE__, __FUNCTION__,
        "expr was true");;
else
    log_printf (LOG_DEBUG, __FILE__, __LINE__, __FUNCTION__,
        "expr was false");;
```

which is an error, as the second semicolon, while being harmless in a different context, would be an additional statement before the **`else`** and thus invalid C code.

The command line and the configuration file are parsed in `configuration.[ch]`, which uses libcli to call the appropriate configuration command handlers. Command line arguments are processed by `parse_cmdline_args()`. GNU `getopt_long()` parses the argument array, and returns a character identifying the encountered option after each call, which is then used in a **`switch`** statement to handle the option and its arguments. In case of an option with a corresponding configuration command, a command string is built and fed into `cli_run_command()`, which then calls the command handler. This allows the same validity checks to be applied to command line arguments and configuration file options.

Listing 7.5: ./helper/configuration.c

```
82 case 'd':   /* --debug | -d */
83     if (optarg)
84         snprintf(command_buffer, 128, "debuglevel %s", optarg);
85     else
86         snprintf(command_buffer, 128, "debuglevel 3");
87     cli_run_command(cli, command_buffer);
88     break;
89 case 'i':   /* --interface | -i */
90     snprintf(command_buffer, 128, "interface %s", optarg);
91     cli_run_command(cli, command_buffer);
```

Listing 7.5 illustrates the handling of two configuration options. The debug option `(--debug | -d)` sets the `debug_level` that is used to limit the number of log messages printed to the daemon's error stream (`stderr`). The debug option is insofar a special case, as the semantics of the command line argument differs from the configuration command, which may be given in a configuration file or on the CLI. If no argument is given to the command line option, debug level 3 (`LOG_DEBUG`) is assumed, while on the CLI, the absence of an argument is used to display the current debug level. This is handled in the **`switch`** statement, as the command handler can't know about these semantic differences. The interface option `(--interface | -i)` doesn't require such special treatment, as the semantics are the same for command line arguments and the CLI.

`parse_config_file()` opens `openocd.cfg` by default, or the config file specified on the command line. It passes the `FILE*` pointer to the config file to `cli_file()` in `MODE_CONFIG`, which interprets the file as if every line was entered on the CLI, executing the command handler of every configuration command encountered, while commands reserved for execution mode are ignored.

`binarybuffer.[ch]` provides handling of little-endian bit streams like they're required when dealing with JTAG scan chains. Data is shifted in on TDI of a connected component, and out on TDO. The least significant bit of the least significant byte has to be scanned in first, making a little-endian byte stream the natural endianess for this application.

Listing 7.6: ./helper/binary_buffer.c

```
53 int buf_set_u32(u8* buffer, unsigned int first, unsigned int num,
           u32 value);
```

sets `num` bits (up to 32) starting at the `first` bit in the appropriate bytes of `buffer`,

Listing 7.7: ./helper/binary_buffer.c

```
71 u32 buf_get_u32(u8* buffer, unsigned int first, unsigned int num)
```

does the opposite, reading up to 32 bits at an arbitrary position inside the buffer. `u32 flip_u32(u32 value)` provides efficient flipping of 32-bit words by using a table with 256 entries to look-up eight bit at a time. This is required for the debug boundary-scan registers of ARM7 and ARM9 controllers, where some of the bits are in reversed order (see §3.2).

./jtag

Generic JTAG code and drivers for JTAG hardware devices are found in the jtag subdirectory. The header file `jtag.h` has to be included by all parts of the software that require access to jtag functions.

Listing 7.8: ./jtag/jtag.h

```
38 enum tap_state
39 {
40     TAP_TLR = 0x0, TAP_RTI = 0x8,
41     TAP_SDS = 0x1, TAP_CD = 0x2, TAP_SD = 0x3, TAP_E1D = 0x4,
42     TAP_PD = 0x5, TAP_E2D = 0x6, TAP_UD = 0x7,
43     TAP_SIS = 0x9, TAP_CI = 0xa, TAP_SI = 0xb, TAP_E1I = 0xc,
44     TAP_PI = 0xd, TAP_E2I = 0xe, TAP_UI = 0xf
45 };
46
47 extern int tap_move_map[16];
```

Listing 7.9: ./jtag/jtag.c

```
415 /* maps the 16 states of the TAP state machine to one of the six
           stable states
416  * unstable states map to -1
417  * that way the tap_move array can be just 6 items, instead of 16
418  */
419 int tap_move_map[16] = {
420     0, -1, -1,  2, -1,  3, -1, -1,
421     1, -1, -1,  4, -1,  5, -1, -1
422 };
```

The `tap_state` enumeration defines the 16 possible TAP controller states, which are mapped to six stable states by the `tap_move_map` array. Because the implemented JTAG system is designed with provision for systems were the test clock (TCK) is free running, the JTAG functions only move from one stable state to another stable state. A driver may look up the required TMS sequence to move from one stable state to another using a two dimensional table. Using the mapping of 16 TAP states to 6 stable states via `tap_move_map`, the size of that table can be reduced from a 16x16 table to a 6x6 table. `TAP_TLR` maps to 0, `TAP_RTI` to 1, `TAP_SD` to 2, `TAP_PD` to 3, `TAP_SI` to 4, and `TAP_PI` maps to 5. Unstable states map to -1, which could be used to detect errors.

Listing 7.10: ./jtag/jtag.h

```
123 typedef struct jtag_interface_s
124 {
125     char* name;
126
127     /* high level command set
128      * reset() - reset tap controller to TLR, set endstate to TLR
129      * xx_scan(...) - scan write_buf into DR or IR scanpath, finish in
           selected end_state
130      * runtest(...) - go to R-T/I, and execute num_cycles in there,
          then advance to end_state
131      * state_move() - advance to selected end_state
132      * endstate(...) - set desired end_state - this throws an exception
          for unstable states
133      */
134     int (*reset)(void);
135     int (*ir_scan)(int num_bits, const u8 *write_buf, u8 *read_buf);
136     int (*dr_scan)(int num_bits, const u8 *write_buf, u8 *read_buf);
137     int (*runtest)(int num_cycles);
138     int (*state_move)(void);
139     int (*endstate)(enum tap_state state);
140
141     /* queued command execution
142      */
143     int (*queue_command)(jtag_command_t command);
144     int (*execute_queue)(void);
145
146     /* reset functions
147      */
148     int (*trst)(int val);
149     int (*srst)(int val);
150
151     /* low level command set
152      */
153     int (*read)(void);
154     void (*write)(int tck, int tms, int tdi);
155
156     /* interface initialization
157      */
158     int (*speed)(int speed);
159     int (*register_commands)(struct cli_def *cli);
160     int (*init)(void);
```

```
161     int (*quit)(void);
162
163     int ir_length;
164     u32 expected_instruction;
165     u32 idcode_instruction;
166
167 } jtag_interface_t;
```

The user defined `jtag_interface_t` of type **`struct`** `jtag_interface_s` is the interface that has to be implemented for every JTAG hardware interface. A JTAG hardware interface is referenced by its `name` field, which is matched against the argument to a `interface` configuration command or command line option. The high-level command set has to be provided by all implementations, but `jtag.c` defines generic functions for use with bitbang devices, that are able to toggle every line individually. The low-level commands are only required for devices that make use of the bitbang functions.

Device setup may be performed in `init()`, which is called before the first jtag access is initiated. Clean-up code can be executed in `quit()`, that is called after the last jtag access has finished, if this is necessary. Device specific configuration commands can be registered in `register_commands()`, which is called immediately after a `interface` configuration command or command line option has been found. The meaning of the **`int`** `speed` argument required by the `speed()` function depends on the current interface. Slow interfaces, like the bitbang interface implemented in `parport.c` don't necessarily have to interpret `speed` calls at all. `ir_length` is necessary to determine how many bits have to be scanned into the instruction register. The `expected_instruction` is compared against the value captured when a new instruction is scanned into the target, and allows JTAG communication problems to be detected.
The high-level command set directly reflects the requirements set for the JTAG subsystem, and should be self explanatory. The queued command execution will be explained in some detail, as it's the way JTAG is used most often throughout the remaining code. `queue_command()` takes one argument of type `jtag_command_t`, which is a user defined **`struct`** `jtag_command_s` type. The command is added to a queue inside the JTAG driver, and will be executed by `execute_queue()`, but hardware interfaces, that don't take a performance penalty when executing commands immediately, don't have to implement the queueing.

Listing 7.11: ./jtag/jtag.h

```
 98 typedef union jtag_command_container_u {
 99     ir_scan_command_t ir_scan;
100     dr_scan_command_t dr_scan;
101     state_move_command_t state_move;
102     runtest_command_t runtest;
103     reset_command_t reset;
104     endstate_command_t endstate;
105 } jtag_command_container_t;
106
107 enum jtag_command_type
108 {
109     JTAG_IR_SCAN = 0, JTAG_DR_SCAN = 1,
110     JTAG_STATE_MOVE = 2, JTAG_RUNTEST = 3,
111     JTAG_RESET = 4, JTAG_ENDSTATE = 5,
112 };
113
```

```
114 typedef struct jtag_command_s
115 {
116     jtag_command_container_t cmd;
117     enum jtag_command_type type;
118 } jtag_command_t;
```

`jtag_command_t` implements polymorphism by declaring a structure with a user defined **`union`** type `jtag_command_container_t`, and a `type` field that specifies the actual type of the command. The command types offer functionality similar to the high-level command set, with the exception of `dr_scan_command_t`, which works on data fields of up to 32 bits length, instead of a linear buffer. This gives the JTAG subsystem information about the relation between bits on the scan chain and their meaning to the code that queued the scan command, allowing it to write data captured from the scan chain to the appropriate buffers without further help from the calling code. Additionally, each command type allows a new JTAG endstate to be specified, reducing the number of commands that have to be queued. If the endstate doesn't have to be changed, an invalid value of -1 may be assigned to the `new_endstate` field.

Listing 7.12: ./jtag/jtag.h

```
51 typedef struct dr_scan_field_s
52 {
53     int num_bits;
54     int do_flip;
55     u32 in_value;
56     u32 *out_value;
57     int do_check;
58     u32 check_value;
59 } dr_scan_field_t;
```

Listing 7.13: ./jtag/jtag.h

```
67 typedef struct dr_scan_command_s
68 {
69     int num_fields;
70     int num_bits;
71     dr_scan_field_t *fields;
72     enum tap_state new_endstate;
73 } dr_scan_command_t;
```

The `fields` member of **`struct`** `dr_scan_command_s` expects a pointer to an array of `dr_scan_field_t` items, that define the layout of the scan chain this command should work on. The items start at the least significant bit, and later items define parts of the scan chain of higher significance. Figure 7.1 shows how the ARM9TDMI debug scan chain (scan chain 1) maps to a `dr_scan_command_t` and its associated `dr_scan_field_t`. `num_fields` defines the length of the array pointed to by `*fields`, `num_bits` is just a convenience field to help debugging and to facilitate buffer calculations without having to parse the `fields` array to get the total number of bits.
The first 32 bits (those closest to TDO) of the debug scan chain connect to the databus of the ARM9TDMI. They're written to the value specified as 'D', and their previous content may be stored to the location pointed to by 'd', if there's data to be read. The next three bits define the values of DDEN, WPTANDBKPT, and SYSSPEED, but when clocking instructions into the debug scan chain, only SYSSPEED may be set, to indicate an instruction that should be executed at system speed. The last 32 bits are in reversed

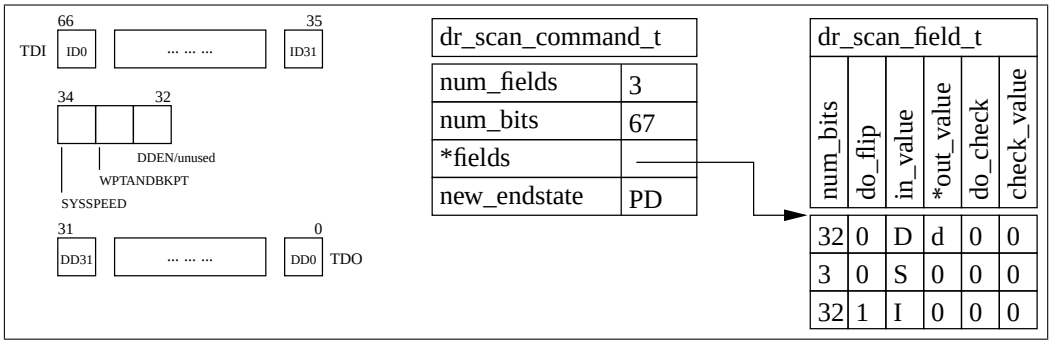

Figure 7.1: ARM9TDMI dr_scan_command_t and dr_scan_field_t[]

bit order, and have thus the `do_flip` flag set to 1. The value scanned in is the new instruction, and data scanned out can be discarded, as the instruction previously written to the instruction bus isn't relevant. `do_check` can be used to tell the JTAG subsystem to verify that the data scanned out of the scan chain into this field matches the `check_value`. This allows test patterns to be defined and validated without having to store every value. If a check failed, the JTAG driver has to inform the caller of `execute_queue()` about that error.

`jtag.c` provides two helper functions for building and processing the buffers used in data register scan operations. `jtag_build_dr_buffer()` uses the buffer handling functions defined in `binarybuffer.c` to set the bits of a linear buffer according to the `fields` array associated with a data register scan operation. `jtag_read_dr_buffer` operates on a linear buffer, and evaluates its contents according to the `fields` description of the current command. If a field has its `do_check` flag set, the captured value is compared against the `check_value`, and an error `ERROR_JTAG_CHECK_FAILED` is returned in case the comparison failed.

Listing 7.14: ./jtag/jtag.c

```
41  jtag_interface_t* jtag_interfaces[] = {
42      &parport_interface,
43      &ftdi2232_interface,
44      &ftd2xx_interface,
45      NULL,
46  };
```

`jtag.c` defines `jtag_interfaces`, an array of pointers to the JTAG drivers. `jtag_init()` is called after the configuration has been parsed. At that point, `jtag_interface` should have been set by an `interface` configuration command, and the `jtag_interfaces` array is searched for an entry that matches the configured interface. If a match is found, the driver's `init()` function is called, and the driver's `jtag_t` interface is assigned to the global `jtag` variable, that gives other subsystems access to the currently configured JTAG hardware interface.

ftd2xx.c and ftdi2232.c

The ftd2xx driver implements the `jtag_interface_t` for the USBJTAG-1, a USB 1.1 Full-Speed device (up to 11 Mbit/s) based on the FT2232C chip from FTDI (see §1.2). The driver relies on the closed-

source libftd2xx from FTDI, but an alternative driver is available as `ftdi2232.c`, which uses the GPL licensed libftdi from Intra2net AG (`http://www.intra2net.com/opensource/ftdi/`). The reason for including both drivers is the superior performance of libftd2xx. Using libftdi, an average download speed of 5 kB/s from the host PC to an ARM920 based SOC could be achieved, while the libftd2xx based solution reached around 25 kB/s. Low-level documentation for the FT2232C's USB communication is available only under an NDA, making an in-depth analysis of libftdi's performance problems difficult.

Listing 7.15: ./jtag/ftd2xx.c

```
66 enum { FTD2XX_TRST = 0x10, FTD2XX_SRST = 0x40 };
67 static u8 discrete_output = 0x0 | FTD2XX_TRST | FTD2XX_SRST;
68 static FT_HANDLE ftdih;
```

The enumeration defining `FTD2XX_TRST` and `FTD2XX_SRST` is used to control the nTRST and nSRST lines, which are connected to GPIOL0 and GPIOL2 on the USBJTAG-1, but could be connected to any of the eight available GPIO lines of the FTDI2232C channel A [DS2232C]. If similar designs were to be implemented, using a different reset signal assignment, this definition would have to be extended to allow multiple variants. `u8 discrete_output` saves the current state of the eight GPIO lines, to allow a single line to be toggled while retaining the state of the other lines. `FT_HANDLE ftdih` is used by libftd2xx to identify an open connection to a FT2232C device, and is an required argument of almost every `FT_xxx()` function.

Listing 7.16: ./jtag/ftd2xx.c

```
70 /* queued command support
71  */
72 typedef struct ftd2xx_command_queue_s
73 {
74     jtag_command_t cmd;
75     struct ftd2xx_command_queue_s *next_cmd;
76 } ftd2xx_command_queue_t;
77
78 static ftd2xx_command_queue_t *ftd2xx_command_queue = NULL;
79 static u8 *ftd2xx_buffer = NULL;
80 static int ftd2xx_buffer_size = 0;
81 static int ftd2xx_queued_size = 1;
82 static int ftd2xx_read_pointer = 0;
83 static enum tap_state ftd2xx_queue_endstate = TAP_TLR;
84 static int ftd2xx_queued_error = ERROR_OK;
85 #define FTD2XX_BUFFER_SIZE  131072
86 #define BUFFER_ADD ftd2xx_buffer[ftd2xx_buffer_size++]
87 #define BUFFER_READ ftd2xx_buffer[ftd2xx_read_pointer++]
```

`ftd2xx_command_queue` implements a linked list of `ftd2xx_command_queue_t` objects which encapsulate the generic `jtag_command_t` type. `ftd2xx_queue_command()` adds a new command to the queue, which is executed once `ftd2xx_execute_queue` is called. `ftd2xx_buffer` points to a linear buffer of `FTD2XX_BUFFER_SIZE` bytes size. The number of bytes currently in the buffer is saved in `ftd2xx_buffer_size`, while `ftd2xx_queued_size` is used to calculate the number of bytes that's required for all queued commands. If `ftd2xx_queued_size` grows beyond `FTD2XX_BUFFER_SIZE`, the last command is temporarily removed from the queue which is then executed. After that, the temporarily removed command is put into the queue as the first item, and will be executed when `ftd2xx_execute_queue()` is called.

ftd2xx_read_pointer is used when the buffer read from the device is processed. The BUFFER_ADD and BUFFER_READ macros provide convenient access to the buffer and make the code more readable.

Listing 7.17: ./jtag/ftd2xx.c

```
 89 /* tap_array[i][j]: tap movement command to go from state i to state j
 90  * 0: Test-Logic-Reset
 91  * 1: Run-Test/Idle
 92  * 2: Shift-DR
 93  * 3: Pause-DR
 94  * 4: Shift-IR
 95  * 5: Pause-IR
 96  */
 97 static u8 tap_move[6][6] =
 98 {
 99     {0x7f, 0x00, 0x17, 0x0a, 0x1b, 0x16},
100     {0x7f, 0x00, 0x25, 0x05, 0x2b, 0x0b},
101     {0x7f, 0x31, 0x00, 0x01, 0x0f, 0x2f},
102     {0x7f, 0x30, 0x20, 0x17, 0x1e, 0x2f},
103     {0x7f, 0x31, 0x07, 0x17, 0x00, 0x01},
104     {0x7f, 0x30, 0x1c, 0x17, 0x20, 0x2f}
105 };
```

When clocking data to TMS, the FT2232C expects a single byte of which up to seven bits will be scanned, starting with the least significant bit. The value of the eighth bit will be written to TDO, which is then held static for the duration of the TMS clocking [AN2232C-01]. The tap_move array lists the necessary TMS sequences when moving from one stable state to another stable state. The mapping from one of the sixteen TAP controller states to one of the six stable states is achieved using the tap_move_map from jtag.c.

ftd2xx_init() configures the FT2232C for JTAG communication, sets the direction of the GPIO lines used for TRST and SRST to output, and allocates the buffer used to communicate with the device. ftd2xx_quit closes the device, and frees the allocated buffer. The high-level command set is implemented similar to the queued commands, but executes every command immediately. Its use is limited because of the performance problems that results from the high latency caused by the USB communication, and is included only for compatibility with code that doesn't use queued commands.

Listing 7.18: ./jtag/ftd2xx.c

```
525 int ftd2xx_queue_command(jtag_command_t command)
526 {
527     ftd2xx_command_queue_t **last_cmd = ftd2xx_get_last_command_p();
528
529     if ((command.type < JTAG_IR_SCAN) || (command.type > JTAG_ENDSTATE))
530         return ERROR_INVALID_ARGUMENTS;
531
532     /* allocate memory for a local copy of command */
533     *last_cmd = malloc(sizeof(ftd2xx_command_queue_t));
534     (*last_cmd)->cmd = command;
535     (*last_cmd)->next_cmd = NULL;
```

Commands are added to the queue in ftd2xx_queue_command(). The **if** statement ensures that only valid command types are processed. A pointer to memory allocated for the new command is assigned to

the last pointer to a `ftd2xx_command_queue_t` object, and the content of the supplied `command` argument is copied to the newly allocated memory.

Listing 7.19: ./jtag/ftd2xx.c

```
525     if (command.type == JTAG_DR_SCAN)
526     {
527         /* allocate memory and copy scan field descriptions */
528         (*last_cmd)->cmd.cmd.dr_scan.fields =
529             malloc(sizeof(dr_scan_field_t) *
    (*last_cmd)->cmd.cmd.dr_scan.num_fields);
530         memcpy((*last_cmd)->cmd.cmd.dr_scan.fields,
    command.cmd.dr_scan.fields,
531             (sizeof(dr_scan_field_t) *
    (*last_cmd)->cmd.cmd.dr_scan.num_fields));
532     }
```

In case of a data register scan command, the scan field descriptions are copied to allocated memory. This is necessary to allow the calling code to destroy all data structures associated with the scan command. `ftd2xx_queue_command()` further calculates the number of FT2232 command bytes required to execute the scan command, and adjusts `ftd2xx_queued_size` accordingly. Listing 7.20 shows how the last command is temporarily removed from the queue, if the required buffer size grew beyond the size of the buffer allocated for communication with the device.

Listing 7.20: ./jtag/ftd2xx.c

```
583     if (ftd2xx_queued_size > FTD2XX_BUFFER_SIZE)
584     {
585         int retval;
586         ftd2xx_command_queue_t *tmp_cmd = *last_cmd;
587
588         *last_cmd = NULL;
589         if ((retval = ftd2xx_execute_queue()) != ERROR_OK)
590             ftd2xx_queued_error = retval;
591         ftd2xx_command_queue = tmp_cmd;
592     }
```

An error that occurred while executing the queue is saved in `ftd2xx_queued_error`, and will be delivered when `ftd2xx_execute_queue()` is called.

`ftd2xx_execute_queue()` iterates through the linked list of commands, fills the command buffer `ftd2xx_buffer` accordingly, and calculates the number of bytes that will be read back in `ftd2xx_expect_read`. This is necessary because the libftd2xx function `FT_Read()`, that's used to read back scan results from the device, has to know the amount of bytes that should be read. Instruction and data register scans are handled by `ftd2xx_add_[id]r_scan()`, while simpler commands are handled inside `ftd2xx_execute_queue()`.

A `JTAG_STATE_MOVE` command adds the MPSSE command byte for a TMS scan, the number of bits to be scanned, and the TMS sequence required to move from the current TAP state to the currently defined end state. `JTAG_RUNTEST` saves the current endstate and adds the commands to move to Run-Test/Idle (similar to `JTAG_STATE_MOVE`) for the desired number of cycles, and to move back to the saved endstate,

or a new endstate if one has been specified. `JTAG_RESET` adds the command to change nTRST and nSRST as specified, and `JTAG_ENDSTATE` changes the endstate without adding anything to the command buffer.

Listing 7.21: ./jtag/ftd2xx.c

```
692     jtag_build_dr_buffer(command, buffer);
693
694     ftd2xx_endstate(TAP_SD);
695
696     /* move from current state to Shift-DR */
697     BUFFER_ADD = 0x4b;
698     BUFFER_ADD = 0x6;
699     BUFFER_ADD = tap_move[tap_move_map[cur_state]]
          [tap_move_map[end_state]];
700     cur_state = end_state;
701
702     if (command->new_endstate == -1)
703         ftd2xx_endstate(saved_endstate);
704     else
705         ftd2xx_endstate(command->new_endstate);
```

Listing 7.21 shows how `ftd2xx_add_dr_scan()` calls `jtag_build_dr_buffer` to build a linear buffer from the `fields` description, and adds the command bytes necessary for moving from the current TAP controller state to Shift-DR. `ftd2xx_endstate()` checks if its argument is a valid endstate (i.e. if it's a stable state), and changes the `end_state` variable. The MPSSE command byte for executing a TMS scan without writing or reading TDI/TDO is 0x4b. The next byte is the amount of bits to be scanned minus one (one bit is always scanned), followed by a byte defining the TMS sequence. This shows how `tap_move` and `tap_move_map` are used to look up the necessary value. If a new endstate was specified for the scan command, that state is set using `ftd2xx_endstate()`, otherwise the previous endstate, which was saved on entry to `ftd2xx_add_dr_scan`, is restored.

Listing 7.22: ./jtag/ftd2xx.c

```
707     /* add command for complete bytes */
708     if (num_bytes > 1)
709     {
710         BUFFER_ADD = 0x39;
711         BUFFER_ADD = (num_bytes-2) & 0xff;
712         BUFFER_ADD = (num_bytes >> 8) & 0xff;
713     }
714
715     /* add complete bytes */
716     while(num_bytes-- > 1)
717     {
718         BUFFER_ADD = buffer[cur_byte];
719         cur_byte++;
720         bits_left -= 8;
721     }
```

After the TAP state movement command has been added, a command to scan complete bytes is added. The command byte 0x39 executes a TDI/TDO scan, and expects a little-endian 16 bit value specifying the number of bytes that have to be scanned. `num_bytes` is the result of dividing the total number of bits by 8, rounded towards positive infinity (always rounded up). The integer division implemented

in C rounds towards zero by cutting any decimals, so rounding towards positive infinity is achieved by calculating (`num_bits` + 7) / 8. Because the last bit of a scan operation has to be scanned while moving out of Shift-DR, the last byte has to be treated separately.

Listing 7.23: ./jtag/ftd2xx.c

```
723     /* the most significant bit is scanned during TAP movement */
724     last_bit = (buffer[cur_byte] >> (bits_left - 1)) & 0x1;
725
726     /* process remaining bits but the last one */
727     if (bits_left > 1)
728     {
729         BUFFER_ADD = 0x3b;
730         BUFFER_ADD = bits_left - 2;
731         BUFFER_ADD = buffer[cur_byte];
732     }
733
734     /* move from Shift-DR to end_state */
735     BUFFER_ADD = 0x6b;
736     BUFFER_ADD = 0x6;
737     BUFFER_ADD = tap_move[tap_move_map[cur_state]]
          [tap_move_map[end_state]] | (last_bit << 7);
738     cur_state = end_state;
739
740     free(buffer);
741
742     return ERROR_OK;
743 }
```

The most significant bit is stored in `last_bit`. If more than one bit is left to be scanned, a command to scan between one and seven bits is added to the buffer, followed by the number of bits and a byte defining their value. Shift-DR is left using a command that executes a TMS scan while writing TDO and reading TDI. After the command byte 0x6b, the number of bits minus one and the required TMS sequence, bit-wise ORed with the most significant data bit, are added to the command buffer.

`ftd2xx_add_ir_scan()` is similar to `ftd2xx_add_dr_scan()`, but moves to Shift-IR, and simply assigns the instruction value to a local variable, instead of using a helper function to build a linear buffer. If the software should to be ported to a big-endian host, this has to be handled differently, as the currently used assignment would result in a wrong byte order on a big-endian system.

Listing 7.24: ./jtag/ftd2xx.c

```
605     u32 instr = command->new_instruction;
```

After all commands have been processed, the "Send Immediate" command byte is added to the buffer, to inform the device that all captured data should be transmitted immediately. The buffer is sent to the device using `FT_Write()`, and results are read back via `FT_Read()`. `ftd2xx_execute_queue` once again iterates through all commands, and reads the scan results for data and instruction register scan commands. In case of a `JTAG_DR_SCAN` command, the `fields` array and the command itself are freed, otherwise only the command is freed.

`ftd2xx_read_dr_scan()` extracts the scan results and adds them to a linear buffer, which is then supplied to `jtag_read_dr_buffer()` to execute necessary validity checks and to save scan results to the appropriate location, where it can then be read by the code that originally queued the scan command. Processing the buffer returned from the device is similar to creating the command buffer. Complete bytes are handled first, followed by the remaining bits minus one, which is saved in a byte of its own, resulting from the TMS scan.

./target

The target directory contains the specification of the abstract target interface (`target.h`), target independent functions in `target.c`, the ARM7/ARM9 common code, and the target specific code and interfaces for ARM7TDMI, ARM9TDMI, ARM720t and ARM920t cores. The header file `target.h` has to be included by all parts of the code that require access to the target.

Listing 7.25: ./target/target.h

```
26 enum target_state
27 {
28     TARGET_UNKNOWN = 0, TARGET_RUNNING = 1,
         TARGET_HALTED = 2, TARGET_RESET = 3
29 };
30
31 enum target_startup_mode
32 {
33     TARGET_RESET_AND_HALT = 0, TARGET_RESET_AND_RUN = 1,
           TARGET_ATTACH = 2, TARGET_INIT_HALT = 3,
34 };
35
36 extern char *target_state_strings[];
37
38 enum target_endianess
39 {
40     TARGET_BIG_ENDIAN = 0, TARGET_LITTLE_ENDIAN = 1
41 };
42
43 extern char *target_endianess_strings[];
```

`target.h` defines enumerations that describe the possible states of a target (**enum** `target_state`), its endianess (**enum** `target_endianess`) and the desired mode of startup (**enum** `target_startup_mode`). The **char*** arrays `target_state_strings` and `target_endianess_strings` provide text strings for the enumeration values, and are used to print status information.

Listing 7.26: ./target/target.h

```
45 typedef struct target_s
46 {
47     char *name;
48     enum target_endianess endianess;
49     enum target_state state;
50     int state_changed;
51
52     /* poll current target status */
```

```
53      enum target_state (*poll)();
54      /* architecture specific status reply */
55      int (*arch_state)(void);
56      /* gdb specific halt reason */
57      int (*gdb_last_signal)(void);
58
59      /* target execution control */
60      int (*halt)(void);
61      int (*resume)(int current, u32 address);
62      int (*step)(int current, u32 address);
63      int (*reset)(int halt);
64
65      /* target register access
66       * get/set_gdb_registers operates on a linear buffer in
          target-endianess,
67       * ordered as expected by the gdb remote protocol
68       * get/set_gdb_reg access a single target register, reg_num is
          defined by gdb
69       */
70      int (*get_gdb_registers)(u8 **buffer, int *size);
71      int (*set_gdb_registers)(u8 *buffer, int size);
72      int (*get_gdb_reg)(u32 *value, int reg_num);
73      int (*set_gdb_reg)(u32 value, int reg_num);
74
75      /* target memory access
76       * size: 1 = byte (8bit), 2 = half-word (16bit), 4 = word (32bit)
77       * count: number of items of <size>
78       */
79      int (*read_memory)(u32 address, u32 size, u32 count, u8 *buffer);
80      int (*write_memory)(u32 address, u32 size, u32 count, u8 *buffer);
81
82      /* target break-/watchpoint control
83       * hw: 0 = sw breakpoint, 1 = hw breakpoint
84       * rw: 0 = write, 1 = read, 2 = access
85       */
86      int (*add_breakpoint)(u32 address, u32 size, int hw);
87      int (*remove_breakpoint)(u32 address);
88      int (*add_hw_watchpoint)(u32 address, u32 mask, int rw);
89      int (*remove_hw_watchpoint)(u32 address);
90
91      int (*register_commands)(struct cli_def *cli);
92      int (*init)(struct cli_def *cli);
93      int (*quit)(void);
94
95  } target_t;
```

The interface defined by `target_t` has to be provided by all target implementations. The `name` field is used to reference a target, and is matched against the argument to a `target` configuration command. The `endianess` is set by the generic target code in `target.c`, and may be evaluated by the target specific code where necessary. The `state` is handled by target specific code (see §6.4), that uses `state_changed` to inform the user interface (CLI or gdb) about changes of the target state.

When a target configuration command is encountered and a matching target is found, the target's register_commands() function is called to give the target specific code a chance to register configuration commands and regular CLI commands. After the configuration is finished, the target initialization code in init() is called. Before the system is shutdown, the target's quit() function is called, to allow target specific code to free used resources.
poll() is used by generic target code to obtain information about the target's current state in a regular interval. If the target state changed, appropriate action should be taken, like examining the reason for debug entry and saving the basic execution context. arch_state() may display more detailed information on the CLI about the current target state. gdb_last_signal() is called to inform a gdb client about the reason for debug entry using POSIX signal definitions like SIGINT or SIGTRAP.
The target state is controlled using halt(), resume(), step(), and reset(). Resume and single-step take two arguments, specifying whether the target should execute from the current or a different address. After coming out of reset, an immediate halt may be requested.
The get_gdb_registers(), set_gdb_registers(), get_gdb_reg(), and set_gdb_reg() functions are called to access core registers on behalf of a connected gdb client. See table 5.2 in the requirements specification for a description of the expected format. get_gdb_registers() has to allocate a buffer large enough to hold the full register list, and returns a pointer to the buffer of **int** *size bytes in u8 **buffer, which has to be freed by the calling code. gdb_set_registers() takes a buffer supplied by the calling code, validates the buffer's size, and sets the registers to the values from the buffer. get_gdb_reg() and set_gdb_reg() operate on the register selected by the reg_num argument.
Target memory is accessed in items of 8, 16, or 32 bits. A buffer of an appropriate size is given to read_memory() or write_memory(). A target may require accesses to be aligned to the item size.
Breakpoints are set using add_breakpoint(). The size field specifies the size of the instruction that should be breakpointed, and is only relevant for software breakpoints. A size of 2 requests a Thumb state breakpoint, while a size of 4 requests an ARM state breakpoint. The hw flag indicates a hardware breakpoint. Breakpoints are removed with remove_breakpoint(), which should remove any breakpoint set on a given address.
Watchpoints are defined by an address, a mask, and a flag indicating the type of access on which the watchpoint should trigger. The rw argument to add_hw_watchpoint() is 0 for reads, 1 for writes, and 2 to catch both reads and writes. remove_hw_watchpoint() is called to remove all watchpoints set on an address.

Listing 7.27: ./target/target.c

```
180 int target_init(struct cli_def *cli)
```

target_init() is called after the configuration has been parsed and the JTAG subsystem has been successfully initialized. The targets array is searched for an entry that matches the target_name configuration variable, and the init() function of a matching target_t implementation is called to initialize the configured target. After the initialization has been passed, the global target pointer is set to the configured target.
If a valid target was found, target_process_startup() is called to evaluate the target_startup configuration. If the debugger is configured to attach to the target, nothing is done in target_process_startup(). In case of reset, or reset_halt, target->reset() with the halt argument set accordingly is used to reset the target. If the startup mode is set to init_halt, target->reset() is called with halt set to 1. Once the target entered debug state, target_process_init() is invoked to open the configuration file and to run it through cli_file() with the libcli mode set to MODE_EXEC. This doesn't execute any configuration commands, but runs the commands registered as MODE_EXEC or MODE_ANY instead.

Listing 7.28: ./target/target.c

```
219 int handle_target(struct cli_def *cli)
```

`handle_target()` is called in regular intervals by the daemon (while waiting for new connections), the CLI, and the gdb server to monitor the target state. While the target isn't in halt state, `target->poll()` is called to have target specific code determine the current target state. The `target->state_changed` flag is evaluated, and in case of a state change, a non-zero value is returned to inform the user interface about it.

arm7_9_common.[ch]

Listing 7.29: ./target/arm7_9_common.h

```
27 enum arm7_9_mode
```

Listing 7.30: ./target/arm7_9_common.h

```
49 enum arm7_9_state
```

The `arm7_9_mode` enumeration defines symbolic names for the seven modes an ARM processor may be in, and assigns them values that match the M[4:0] bits in the current program status register (CPSR). The `arm7_9_state` currently defines two states, ARM and Thumb, which could be extended to include Jazelle, once support for the ARM9EJ-S cores is added.

Listing 7.31: ./target/arm7_9_common.h

```
55 typedef struct arm7_9_core_reg_s
56 {
57     char *name;
58     int num;
59     enum arm7_9_mode mode;
60     u32 value;
61     int dirty;
62 } arm7_9_core_reg_t;
63
64 typedef struct arm7_9_ice_reg_s
65 {
66     char *name;
67     int size;
68     int addr;
69     u32 value;
70 } arm7_9_ice_reg_t;
```

Userdefined types are used for core registers (`arm7_9_core_reg_t`) and Embedded-ICE registers (`arm7_9_ice_reg_t`). A core register is defined by its `name` (r0-r15, and a mode suffix), a number `num` (0-15, or a negative value to indicate special registers), the processor (`mode`) to which the register belongs, the register's current `value`, and a `dirty` flag indicating that the register has to be restored before debug state may be left. An ICE register has a `name`, a `size` field that determines how many of the 32 Embedded-ICE data bits are valid, an `address`, and a current `value`.

Listing 7.32: ./target/arm7_9_common.h

```
73 enum arm7_9_bkpt_type
```

```
74  {
75      ARM7_9_SW_BKPT,
76      ARM7_9_HW_BKPT,
77  };
78
79  typedef struct arm7_9_breakpoint_s
80  {
81      u32 addr;
82      enum arm7_9_state state;
83      enum arm7_9_bkpt_type type;
84      int set;
85      u32 orig_instr;
86      struct arm7_9_breakpoint_s *next;
87  } arm7_9_breakpoint_t;
88
89  typedef struct arm7_9_watchpoint_s
90  {
91      u32 addr;
92      u32 mask;
93      int rw;
94      int set;
95      struct arm7_9_watchpoint_s *next;
96  } arm7_9_watchpoint_t;
```

ARM7 and ARM9 based cores may use software or hardware breakpoints, for which the enumeration data type `arm7_9_bkpt_type` defines symbolic names. `arm7_9_breakpoint_t` defines attributes for both software and hardware breakpoints. The address `addr` specifies the code location on which the breakpoint should trigger. The `state` distinguishes between ARM and Thumb state breakpoints, and `type` is either `ARM7_9_SW_BKPT` or `ARM7_9_HW_BKPT`. The `set` flag determines if the breakpoint is currently disabled (set == 0) or active, and the watchpoint unit used in case of a hardware breakpoint (1 for watchpoint unit 0, 2 for unit 1). In case of a software breakpoint, the original instruction is saved in `orig_instr`. Breakpoints are managed using a linked list, with `next` pointing to the next breakpoint, or `NULL`, in case of the last breakpoint in the list.
A watchpoint is defined by its address, the `mask` that selects which bits will be ignored in a comparison, a read/write/access flag, and a field holding the number of the watchpoint unit used to implement the watchpoint (`set`). Like breakpoints, watchpoints are contained in a linked list, with `next` pointing to the next watchpoint in the list.

Listing 7.33: ./target/arm7_9_common.h

```
191 typedef struct arm7_9_debug_s
192 {
193     int scan_n_size;
194     u32 embedded_ice_ver;
195     arm7_9_ice_reg_t *ice_regs;
196
197     u32 arm_bkpt;
198     u16 thumb_bkpt;
199     int sw_bkpts_use_wp;
200
```

```
201     int (*examine_debug_reason)(void);
202     int (*minimum_context)(void);
203     int (*full_context)(void);
204     int (*restore_context)(void);
205
206     int (*execute_resume)(void);
207     int (*execute_step)(void);
208
209     int (*read_memory)(u32 address, u32 size, u32 count, u8 *buffer);
210     int (*write_memory)(u32 address, u32 size, u32 count, u8 *buffer);
211 } arm7_9_debug_t;
```

To account for differences between the ARM7 and ARM9 families, the actual cores, and core revisions, the `arm7_9_debug_t` interface has been defined. The `scan_n_size` field determines the size of the scan path select register (SCAN_N), to allow common scan chain selection code for all ARM7 and ARM9 cores to be used. `embedded_ice_ver` holds the Embedded-ICE version implemented by the core being debugged. This would allow ARM7TDMI(-S) Rev4 cores to be distinguished from older ARM7TDMI cores, which is necessary to handle the monitor mode debug features available on Rev4 cores correctly. `arm_bkpt` and `thumb_bkpt` specify the instruction value that should be used for software breakpoints in ARM and Thumb state. `sw_bkpts_use_wp` is a flag that decides whether a watchpoint unit has to be used to implement software breakpoints or if the core supports the breakpoint instruction (BKPT) introduced with ARM9E cores.

`examine_debug_reason()` is called immediately after debug entry was detected, and must not modify the core state. `minimum_context()` has to save the basic execution context, including core specific registers like fault status and address registers (FAR, FSR). Only registers of the current mode have to be saved, as the core shouldn't change the processor mode while it is in debug state. If a user requests a register that doesn't belong to the current mode, `full_context()` is called to query all remaining registers. Before debug state is left with `execute_resume()` or `execute_step()`, `restore_context()` has to restore every register that has been modified during debug. `read_memory()` and `write_memory()` are required by the ARM7 and ARM9 common code to support software breakpoints, where the original instruction is saved and replaced by a special debug instruction.

The `arm7_9_common.h` header also defines enumerations for the JTAG instructions (**enum** `arm7_9_jtag_instr`), the reasons for debug entry (`arm7_9_debug_reason`), and Embedded-ICE constants. It further provides macros for ARM opcodes used during debug, that allow an opcode to be built using the instruction mnemonic and a list of parameters. Listing 7.34 shows an example macro that is used for store multiple instructions:

Listing 7.34: ./target/arm7_9_common.h

```
261 /* Store multiple increment after
262  * Rn: base register
263  * List: for each bit in list: store register
264  * S: in privileged mode: store user-mode registers
265  * W=1: update the base register. W=0: leave the base register untouched
266  */
267 #define ARM7_9_STMIA(Rn, List, S, W)    (0xe8800000 | (S << 22) |
            (W << 21) | (Rn << 16) | (List))
```

The STMIA instruction saves multiple registers to increasing memory locations. The base register contains the address at which the registers should be stored. The list is a field of 16 bits, where each bit determines if the corresponding register (r0-r15) should be stored. With the S flag set, the STMIA instruction would store the user mode registers, instead of the registers that belong to the current mode. The W flag determines if the base register is updated after STMIA finishes (plus 4 bytes for every register stored), or if the base is left untouched.
The other instruction macros are implemented similarly, and [DDI0100E] may be consulted for further information on the format of each instruction.

arm7_9_common.c contains code relevant for all ARM7 and ARM9 based systems, and defines important data structures. The arm7_9_core_regs array holds all core registers. The general purpose registers (r0 to r15) are initialized with their name, mode and number, while negative numbers are assigned to the status registers. The current processor status register (CPSR) has the number -1, and the saved status registers have -2. These are used to identify the status registers when the context is saved or restored.
The two dimensional arm7_core_reg_map[mode][number] array maps the two indices *mode* (an ordinal number, ranging from 0 to 6) and *number* to one of the register's from the linear arm7_9_core_regs array, simulating the banked registers implemented in ARM processor cores. The register numbers 0 to 8 for example always map to the corresponding user mode registers, while register 13 and 14 map to a different register for each mode (except system mode, which shares all registers with the user mode). Two functions, arm7_9_mode_to_number and arm7_9_number_to_mode, translate between the mode values defined by **enum** arm7_9_mode and the ordinal number used by the arm7_9_core_reg_map array. If the mode values were used as indices into the array, there would have been many unused entries, as only seven of the 32 possible mode encodings are valid.

Listing 7.35: ./target/arm7_9_common.h

```
179 #define ARM7_9_CORE_REG_MODE(mode, num)
      (arm7_9_core_reg_map[arm7_9_mode_to_number(mode)][num])
```

The convenience macro ARM7_9_CORE_REG_MODE takes a mode value (**enum** arm7_9_mode type) and a number, and resolves to a pointer to the appropriate register. This allows the list of registers to be accessed using the linear array, the two dimensional mode-number array, or the semantically more convenient arm7_9_mode-number array.

There are three arrays of strings, providing textual representations of the ordinal processor mode (arm7_9_mode_strings), the processor state (arm7_9_state_strings), and the reason for debug entry (arm7_9_debug_reason_strings).

Listing 7.36: ./target/arm7_9_common.c

```
150 struct cli_def *arm7_9_cli;
151 extern int is_gdb_session;
152
153 /* implementation specific, supplied by actual core code */
154 arm7_9_debug_t *arm7_9_debug = NULL;
155
156 /* global state vars */
157 enum arm7_9_state arm7_9_saved_state;
158 enum arm7_9_mode arm7_9_saved_mode;
159 enum arm7_9_debug_reason arm7_9_saved_debug_reason;
```

```
160 int arm7_9_cur_scanchain;
161
162 /* linked list of breakpoints */
163 struct arm7_9_breakpoint_s *arm7_9_breakpoints;
164 struct arm7_9_watchpoint_s *arm7_9_watchpoints;
165
166 /* embedded ice comparators in use */
167 static int wp0_used = 0, wp1_used = 0;
168 static int wp_available = 2;
169
170 /* watchpoint unit used for implementing sw breakpoints */
171 static int sw_bkpt_wp = -1;
172
173 /* ARM7_9_HW_BKPT or _SW_BKPT, -1 signals uninitialized value */
174 enum arm7_9_bkpt_type arm7_9_breakmode = -1;
```

Listing 7.36 shows global variables introduced by `arm7_9_common.c`. `arm7_9_cli` is used as a hack to decouple the target interface defined in `target_t` from the command line interface library (libcli). It is initialized when `arm7_9_init()` is called, and is only used by `arm7_9_arch_state()`, which prints verbose information about the target state to the CLI. `is_gdb_session` tells the target implementation that a gdb client is currently connected, which has implications for the handling of breakpoints (see §6.6). If additional debug frontends are added in a later version, probably by supporting the ARM remote debug interface (RDI), a common interface would have to be defined, that allows the debugger to specify how debugger-dependent features should be handled. `arm7_9_debug` has to be set to the appropriate values by the code implementing the target interface for a particular core.
The target state is described by `arm7_9_saved_state`, `arm7_9_saved_mode` (both set by `arm7_9_debug->minimum_context()`), `arm7_9_saved_debug_reason` (set by `arm7_9_debug->examine_debug_reason()`), and `arm7_9_cur_scanchain` (modified by `arm7_9_set_scanchain()`).
`wp[01]_used` show if the corresponding Embedded-ICE watchpoint unit is still available, if it's already used for implementing a hardware breakpoint or watchpoint (`wp[01]_used` == 1), or if it's locked, either to support software breakpoints or due to a user request. If the current core doesn't support a breakpoint instruction to implement software breakpoints, `sw_bkpt_wp` is set to the number of the watchpoint unit used. `wp_available` holds the number of currently available watchpoint units. The configuration command `breakmode` controls `arm7_9_breakmode`, setting it to either `ARM7_9_HW_BKPT` (only hardware breakpoints may be set) or `ARM7_9_SW_BKPT` (software breakpoints are enabled).
`arm7_9_lock_watchpoint()` and `arm7_9_unlock_watchpoint()` may be used to lock/unlock a watchpoint unit outside of `arm7_9_common.c`. If the requested watchpoint is available, it is marked as used, and a non-zero value is returned, otherwise zero is returned. While it is locked, the watchpoint wont be used by the breakpoint handling code.
`arm7_9_get_breakpoint()` searches the linked list of breakpoints for a given address, and returns a pointer to a matching `arm7_9_breakpoint_t`, or NULL, if no match is found.

Listing 7.37: ./target/arm7_9_common.c

```
293 /* write SCAN_N register */
294 int arm7_9_set_scanchain(jtag_scan_chain_t* sc_new)
```

`arm7_9_set_scanchain()` changes the current JTAG instruction to SCAN_N, selects the requested scan chain, and verifies the value scanned out of the scan path select register, which should have the

most significant bit set to one, and all other bits set to zero. It makes use of the JTAG queued command system to be able to work with JTAG hardware interfaces that have a high latency.

Listing 7.38: ./target/arm7_9_common.c

```
331 /* mark all core regs as clean (don't have to be restored at
           step/resume) */
332 int arm7_9_clean_core_regs(void)
```

`arm7_9_clean_core_regs()` iterates through the linear list of core registers, and marks every register as clean. It may be called by core-specific code in `minimum_context()` to mark all registers as clean, after they have been read from the target.

Listing 7.39: ./target/arm7_9_common.c

```
345 int arm7_9_guard_ice_status(u32 value)
```

`arm7_9_guard_ice_status()` allows JTAG commands to be queued even if the content of an Embedded-ICE register would have to be examined before additional commands can be queued. This is the case when the debugger has to poll the Embedded-ICE status register, to determine if it returned from an instruction executed at system speed during memory read or write operations. With typical JTAG speeds, it is unlikely that a core requires more time to reenter debug state than it takes the debugger to poll the Embedded-ICE status register, so the debugger could continue after the first poll. `arm7_9_guard_ice_status()` reads the status register and requests the queued command support code to compare the register with an expected value. If the core was able to execute its system speed instruction fast enough, the status register matches the expected value, and the debugger may continue, otherwise it detects a fatal error, and has to reset the system.

Listing 7.40: ./target/arm7_9_common.c

```
400 int arm7_9_read_ice_reg(arm7_9_ice_reg_t* reg)
```

Listing 7.41: ./target/arm7_9_common.c

```
454 int arm7_9_write_ice_reg(arm7_9_ice_reg_t* reg)
```

`arm7_9_read_ice_reg()` queues JTAG commands to change the current scan chain to the Embedded-ICE scan chain (scan chain 2), selects INTEST as the current instruction, and executes two data register scans, one to shift the desired Embedded-ICE register address and the read bit into the scan chain, and one to capture the register value. To write an Embedded-ICE register, `arm7_9_write_ice_reg()` may be called, which queues JTAG commands similar to the read function, but adds only one data register scan that shifts the Embedded-ICE register address, the write bit, and the new register value into the scan chain.

Listing 7.42: ./target/arm7_9_common.c

```
496 int arm7_9_setup_sw_bkpt(void)
```

Listing 7.43: ./target/arm7_9_common.c

```
544 int arm7_9_dismantle_sw_bkpt(void)
```

On cores that don't support a software breakpoint instruction (BKPT), `arm7_9_setup_sw_bkpt()` may be called to setup the Embedded-ICE unit to trigger a breakpoint on a debug instruction value. It tries to lock one of the watchpoint units, and initializes the unit as a data dependent breakpoint with the value

specified in `arm7_9_debug->arm_bkpt` and the address mask register set to ignore the address from which an instruction is fetched. `arm7_9_dismantle_sw_bkpt()` is provided to allow the software breakpoint support to be turned off and to free the watchpoint unit, in case a user requires an additional hardware breakpoint or watchpoint.

`arm7_9_debug_entry()` is called after the core entered debug state, and uses `arm7_9_debug->examine_debug_reason()` and `arm7_9_debug->minimum_context()` to query the target state.

`arm7_9_register_commands()` registers configuration commands and CLI commands that apply to all ARM7 and ARM9 targets. `arm7_9_init()` configures the JTAG interface for use with ARM7 and ARM9 systems, initializes some state variables, and configures one of the Embedded-ICE units for software breakpoints, if these are enabled via the `breakmode` configuration option and the core doesn't support the software breakpoint instruction. `arm7_9_quit()` disables both watchpoint units, to make sure that no breakpoint or watchpoint triggers while the core isn't monitored by the debugger.

Listing 7.44: ./target/arm7_9_common.c

```
643 enum target_state arm7_9_poll()
```

`arm7_9_poll()` queues JTAG commands to read the Embedded-ICE status register, and executes the JTAG command queue. If the DBGACK bit is set and the target was previously in running state, the entry into debug state is handled, and the user interface is informed about the state change by setting the `target->state_changed` flag.

`arm7_9_arch_state` prints information about the current target state, including the core state and mode, the address of the instruction that will be executed next, and the current processor status register.

`arm7_9_gdb_last_signal()` returns a POSIX signal that reflects the reason for debug entry. If debug state was entered because of a debug request, SIGINT is delivered. If the reason was a breakpoint or a watchpoint, SIGTRAP is delivered.

Listing 7.45: ./target/arm7_9_common.c

```
727 int arm7_9_halt(void)
```

`arm7_9_halt()` ensures that the processor is currently in running- or in reset state, otherwise it returns an error, to ensure that the target state isn't corrupted. It programs the Embedded-ICE control register with DBGRQ set high, deasserts a possibly asserted reset line, and waits for the core to enter debug state by continuously reading the Embedded-ICE status register. If DBGACK doesn't go high within five seconds after debug entry was requested, a timeout is signaled to the calling code. No matter if debug entry was successful, the DBGRQ signal is set low again, to allow the core to continue operation. On ARM9 targets, this wouldn't be necessary until a system speed instruction is executed, but ARM7 systems require this step to be able to execute instructions at debug speed. If DBGACK went high, `arm7_9_poll()` is called, which handles the entry into debug state (see §6.4).

Listing 7.46: ./target/arm7_9_common.c

```
788 int arm7_9_resume(int current, u32 address)
```

`arm7_9_resume()` checks if the target is in debug state, and changes the program counter (r15), if resume was requested for a different address than that of the current instruction. In case of a CLI session,

the list of breakpoints is searched for the address on which execution should be resumed. Should the address be found, that breakpoint is disabled, and a single-step is executed by calling core specific code to restore the context, execute the step, and save the new context. After that, the breakpoint is enabled again. If a gdb client is connected, these steps are unnecessary, as gdb handles breakpoints differently (see §6.6). Core specific code is then called to restore the context and execute the resume sequence.

Listing 7.47: ./target/arm7_9_common.c

```
836 void arm7_9_embeddedice_step()
```

`arm7_9_embeddedice_step()` is used to program the Embedded-ICE watchpoint units to support single-step on cores without hardware single-step support, according to the procedure described at §3.3. Code that calls this function should backup the registers of both Embedded-ICE units to be able to restore them after the step has been executed.

Listing 7.48: ./target/arm7_9_common.c

```
870 int arm7_9_step(int current, u32 address)
```

`arm7_9_step()` checks the target state, changes the program counter if desired, and disables a breakpoint set on the address at which the single-step should be executed, like `arm7_9_resume()`. It sets the `arm7_9_saved_debug_reason` to indicate a single-step, and calls core specific code to restore the context and execute the step. Similar to `arm7_9_resume()`, the breakpoint is enabled again, if there was one set.

Listing 7.49: ./target/arm7_9_common.c

```
910 int arm7_9_reset(int halt)
```

If the target was halted, it is resumed, before `arm7_9_reset()` asserts the SRST line. The software waits 500ms to give the target time to reset itself, and changes the `target->state` to reflect the reset. If the `halt` flag is set, `arm7_9_halt()` is called to enter debug state immediately after coming out of reset, otherwise SRST is desasserted, to allow the target to start executing instructions.

Listing 7.50: ./target/arm7_9_common.c

```
950 int arm7_9_get_gdb_registers(u8 **buffer, int *size)
```

`arm7_9_get_gdb_registers()` uses `calloc()` to allocate 168 bytes of zero-initialized memory. GDB expects a buffer with sixteen core registers of 4 bytes size, eight floating point registers of 12 bytes size, the floating point status register, and the current program status register. The core registers of the current processor mode are stored in the allocated memory by treating it as an array of unsigned integers. To support big-endian targets or hosts, this would have to be changed. Floating point registers aren't supported, so the space reserved for these is left at zero. At the end of the buffer, the CPSR is stored. The pointer to the buffer and its size are returned to the caller, which has to free the memory when it's done using it.

Listing 7.51: ./target/arm7_9_common.c

```
981 int arm7_9_set_gdb_registers(u8 *buffer, int size)
```

`arm7_9_set_gdb_registers()` works similar to `arm7_9_get_gdb_registers()`. It validates the buffer size, and treats the buffer as an array of unsigned integers, that are written to the core registers. Every register written is marked as dirty, and will be transfered to the core when the target is resumed.

`arm7_9_get_gdb_reg()` and `arm7_9_set_gdb_reg()` simply return or set the requested core register value. Register between 0 and 15 refer to the general purpose registers. Registers 16 to 24 select floating point register, and result in an error. Register 25 selects the current program status register.

Breakpoints are handled by `arm7_9_set_breakpoint()`, `arm7_9_unset_breakpoint()`, `arm7_9_add_breakpoint()`, and `arm7_9_remove_breakpoint()`. When a breakpoint should be added, `arm7_9_add_breakpoint` is called. The list of breakpoints is searched for a breakpoint with the same address, and the function aborts with a positive return code, if an existing breakpoint is found. In case of a hardware breakpoint, `wp_available` is examined to see if one of the comparators is available, and an error is returned if no watchpoint unit is free to be used. If a software breakpoint has to be added, `arm7_9_breakmode` is checked if software breakpoints are enabled. In both cases, memory for an `arm7_9_breakpoint_t` object is allocated, and the fields are filled according to the desired breakpoint. The breakpoint size is used to determine if an ARM or an Thumb state instruction should be breakpointed. If all checks were passed, `arm7_9_set_breakpoint()` is called. This function ensures that the target is halted, and that the breakpoint isn't already set. In case of a hardware breakpoint, it checks which of the comparators is unused, and programs it as a data independent instruction breakpoint that triggers on the desired address. In case of a software breakpoint, the target memory at the desired address is read, and its content is saved in the breakpoint's `orig_instr` field. Depending on the size, either the `arm7_9_debug->arm_bkpt` or the `arm7_9_debug->thumb_bkpt` instruction is written to replace the original instruction. The `set` field is changed to indicate that the breakpoint has been set, and reflects the watchpoint unit used in case of a hardware breakpoint.

`arm7_9_unset_breakpoint()` is called when a breakpoint should be disabled. For hardware breakpoints, the watchpoint unit on which the breakpoint is set is disabled, and if a software breakpoint should be disabled, the saved instruction is written back to the target memory. The `set` field is then cleared, to show that the breakpoint has been unset. When `arm7_9_remove_breakpoint()` is called, the list of breakpoints is searched for the address, and `arm7_9_unset_breakpoint()` is called if the breakpoint is currently enabled. The linked list is modified to allow the breakpoint to be removed, and the memory used for the breakpoint is freed.

Watchpoints are handled by `arm7_9_add_hw_watchpoint()` and `arm7_9_remove_hw_watchpoint()`. Set/unset functions aren't required, because watchpoints don't have to be disabled, as an instruction that triggers a watchpoint is always executed. When a watchpoint should be added, the list of watchpoints is searched for the address of the new watchpoint, and the function is aborted if there's already a watchpoint set on that address. `wp_available` is checked to see if a watchpoint unit is available, and an error is returned if not. Depending on the `rw` flag, the watchpoint's control mask register is programmed to ignore the nRW signal (rw == 2, catch all accesses), or to evaluate that signal (rw == 0, catch reads, or rw == 1, catch writes). The data mask register is programmed to ignore the data value, and the address and mask are programmed as requested. The watchpoint's `set` flag is set to the number of the comparator used, and the corresponding `wp[01]_used` variable is set to 1 to show that it's not available. When `arm7_9_remove_hw_watchpoint()` is called, the list of watchpoints is searched for the address of the watchpoint that should be removed. When a match is found, the comparator used for the watchpoint it disabled, and the `wp[01]_used` variable is set to 0. The linked list is modified to allow the watchpoint to be removed, and the memory used for the watchpoint is freed.

Listing 7.52: ./target/arm7_9_common.c

```
1398 int arm7_9_handle_breakmode_command(struct cli_def *cli, char *cmd,
         char **args, int argc)
```

`arm7_9_handle_breakmode_command()` handles the breakmode configuration command. It evaluates the first argument given to the command, and enables or disables the use of software breakpoints accordingly.

Listing 7.53: ./target/arm7_9_common.c

```
1427 int arm7_9_handle_reg_command(struct cli_def *cli, char *cmd,
           char **args, int argc)
```

`arm7_9_handle_reg_command()` is called when a user entered the `reg` command on the CLI. `arm7_9_debug->full_context()` is invoked to read the contents of all core registers in all modes. The command's arguments are parsed, and depending on the number of arguments, either all registers are displayed on the CLI (no arguments), a single register is displayed (one argument), or one register is assigned a new value (two arguments).

Listing 7.54: ./target/arm7_9_common.c

```
1500 int arm7_9_handle_icereg_command(struct cli_def *cli, char *cmd,
           char **args, int argc)
```

`arm7_9_handle_icereg_command()` is similar to the `reg` command handler, but operates on Embedded-ICE registers instead of core registers. `arm7_9_read_ice_reg()` and `arm7_9_write_ice_reg()` are used to access the Embedded-ICE registers referenced by their address.

Listing 7.55: ./target/arm7_9_common.c

```
1542 int arm7_9_handle_bp_command(struct cli_def *cli, char *cmd,
           char **args, int argc)
```

`arm7_9_handle_bp_command()` gives access to the list of breakpoints. If the `bp` command is entered without any arguments, a list of all breakpoints currently set is displayed. Otherwise, `arm7_9_add_breakpoint()` is called with the address of the breakpoint. The size is set to 4, if nothing else was specified, and a software breakpoint is requested, if there wasn't an argument `hw` requesting a hardware breakpoint.

Listing 7.56: ./target/arm7_9_common.c

```
1592 int arm7_9_handle_wp_command(struct cli_def *cli, char *cmd,
           char **args, int argc)
```

Like the `bp` handler, `arm7_9_handle_wp_command()` displays the list of watchpoints currently set. If three arguments were specified, `arm7_9_add_hw_watchpoint()` is called to set a read, write, or access watchpoint at the desired address with the chosen mask.

Listing 7.57: ./target/arm7_9_common.c

```
1616 int arm7_9_handle_rbp_command(struct cli_def *cli, char *cmd,
           char **args, int argc)
```

Listing 7.58: ./target/arm7_9_common.c

```
1624 int arm7_9_handle_rwp_command(struct cli_def *cli, char *cmd,
           char **args, int argc)
```

`arm7_9_handle_rbp_command()` and `arm7_9_handle_rwp_command()` call `arm7_9_remove_breakpoint()` and `arm7_9_remove_watchpoint()` to remove a breakpoint or watchpoint set on the address specified in the first argument to the `rbp` or `rwp` command.

arm9tdmi.[ch]

The header file for the code common to all ARM9 based cores declares functions that may be called by core specific code, and defines two additional Embedded-ICE enumeration values. `ARM9TDMI_VEC_CATCH` holds the address of the Embedded-ICE vector catch register found on all ARM9 based cores. `ARM9TDMI_DBG_CONTROL_SSTEP` is the bit mask for the debug control register used to enable hardware single-stepping, a feature available on ARM9TDMI and ARM9E-S Rev. 1 based cores.

The `arm9tdmi_target` interface may be used for ARM9 based cores while their MMU and caches are disabled. It doesn't handle cache consistency, and works with the addresses presented on the core's internal bus. If these are virtual addresses, displaying memory works, but modifications may fail to update the instruction cache, resulting in inconsistencies. The `arm9tdmi_ice_regs` array defines the list of Embedded-ICE registers available on ARM9TDMI based cores. Core specific code should change the size fields of the registers to match a particular core (e.g. ARM9E-S cores have a larger debug status register).
The `arm9tdmi_debug` structure provides ARM7/ARM9 common code with the size of the ARM9 SCAN_N register, the Embedded-ICE version, the list of Embedded-ICE registers, the instruction values required for software breakpoints, the flag indicating that software breakpoints require a watchpoint unit, and pointers to core specific debug functions.

Listing 7.59: ./target/arm9tdmi.c

```
151 static int arm9tdmi_has_singlestep = 0;
152 static int arm9tdmi_full_context_queried;
```

`arm9tdmi_has_singlestep` is a flag used by the ARM9TDMI code to determine if the core supports the hardware single-step feature. It is set in `arm9tdmi_init()` if an Embedded-ICE version 2 or 5 is found. `arm9tdmi_full_context_queried` shows whether the content of all core registers in all modes has already been fetched since the last debug entry.

Listing 7.60: ./target/arm9tdmi.c

```
154 /* put an instruction in the ARM9TDMI pipeline, and set the databus to
        "in"
155  * the databus content is then placed in "out" */
156 int arm9tdmi_put_instruction(u32 instruction, int sysspeed, u32 in,
          u32 *out)
```

`arm9tdmi_put_instruction` is used to place an instruction in the ARM9TDMI pipeline, and to set and read the ARM9TDMI data bus. The function queues JTAG commands, but doesn't execute them, to allow larger command queues to be built. The macro `_SLOW_DEBUG_` may be defined to make `arm9tdmi_put_instruction()` execute every instruction immediately. This allows more debug information to be displayed and helps in diagnosing problems with the debug code. The JTAG endstate is changed to `TAP_PD`, and the JTAG instruction is switched to INTEST.
A data register scan command with three scan register fields is built. The first field is 32 bits long, and is used to read and write the data bus. The second field is 3 bits long, and sets the value of the SYSSPEED signal used to flag instructions that should be executed at system speed. The third field contains the

instruction that should be executed and is 32 bits long, but has a reversed bit order. The `do_flip` flag of the scan field description is used to indicate that the JTAG code should reverse this field before scanning it into the target and before writing the results to the debugger memory.
Following the data register scan command, a runtest command is queued, that moves the TAP state machine from TAP_PD to TAP_RTI and back to TAP_PD, pulsing the debug clock (DCLK) to execute one cycle.

Listing 7.61: ./target/arm9tdmi.c

```
218 int arm9tdmi_register_commands(struct cli_def *cli)
```

The ARM9TDMI code doesn't register commands of its own, but calls `arm7_9_register_commands()` to make use of commands that are relevant to all ARM7 and ARM9 targets.

Listing 7.62: ./target/arm9tdmi.c

```
228 int arm9tdmi_init(struct cli_def *cli)
```

The initialization code for the ARM9TDMI assigns its `arm9tdmi_debug` structure to the `arm7_9_debug` variable that's used by ARM7 and ARM9 common code to access core specific debug functionality. `arm7_9_init()` is called to give shared code a chance to initialize itself. The Embedded-ICE debug communications control register is read, and its upper four bits are evaluated to determine the Embedded-ICE version implemented by the current core. On cores that support hardware single-stepping, the `arm9tdmi_has_singlestep` flag is set accordingly. The `arm9tdmi_quit()` code disables the vector catching feature to ensure that debug state isn't entered without a debugger monitoring the target, and calls `arm7_9_quit()`.

Listing 7.63: ./target/arm9tdmi.c

```
267 int arm9tdmi_examine_debug_reason(void)
```

`arm9tdmi_examine_debug_reason()` is called after the ARM7/ARM9 shared code detected an entry into debug state. On ARM9TDMI cores, the reason for debug entry is encoded in two control bits of the debug scan chain. A data register scan command is built to read the scan chain and is executed immediately. Another scan command is built to write the values scanned out with the first command back to the target, to ensure that its state isn't modified. Because `arm9tdmi_examine_debug_reason()` doesn't move through the TAP_RTI state, no debug cycles are executed.

Listing 7.64: ./target/arm9tdmi.c

```
329 if (debug_reason & 0x4)
330     if (debug_reason & 0x2)
331         arm7_9_saved_debug_reason = ARM7_9_REASON_WPTANDBKPT;
332     else
333         arm7_9_saved_debug_reason = ARM7_9_REASON_WATCHPOINT;
334 else
335     arm7_9_saved_debug_reason = ARM7_9_REASON_BREAKPOINT;
```

The `debug_reason` is taken from the three control bits of the debug scan chain. The most significant bit is the BREAKPT signal, and indicates a breakpoint condition when low. If BREAKPT is high, the second bit, WPTANDBKPT has to be examined. When it's high, a watchpoint and breakpoint occurred simultaneously, otherwise the reason for debug entry was a watchpoint. The least significant bit is the DDEN signal (Data Data Bus Output Enabled, indicates a write on DD[31:0]), which is irrelevant for the debug entry reason.

Listing 7.65: ./target/arm9tdmi.c

```
342 int arm9tdmi_read_cpsr(u32 *cpsr)
343 {
344     /* MRS r0, cpsr */
345     arm9tdmi_put_instruction(ARM7_9_MRS(0, 0), 0, 0, NULL);
346
347     /* STR r0, [r15] */
348     arm9tdmi_put_instruction(ARM7_9_STR(0, 15), 0, 0, NULL);
349     arm9tdmi_put_instruction(ARM7_9_NOP, 0, 0, NULL);
350     arm9tdmi_put_instruction(ARM7_9_NOP, 0, 0, NULL);
351     arm9tdmi_put_instruction(ARM7_9_NOP, 0, 0, cpsr);
352
353     return ERROR_OK;
354 }
```

To read the current program status register, a `MRS r0, CPSR` instruction is inserted into the ARM9TDMI pipeline, that moves the content of the CPSR to the general purpose register r0. The `ARM7_9_MRS` macro takes two arguments, one to specify the destination register, and one to set the S bit that selects between the CPSR (S == 0) and the SPSR (S == 1). After that, a `STR r0, [r15]` instruction is put into the pipeline, that stores the content of register r0 to the address located in r15. Three `NOPs` are inserted to move the store register instruction into the Memory stage, where the value can be captured on the data bus. The code doesn't execute the JTAG scan chain to allow larger command queues to be built.

Listing 7.66: ./target/arm9tdmi.c

```
356 int arm9tdmi_minimum_context(void)
```

`arm9tdmi_minimum_context()` is called after the core entered debug state and the reason for debug entry has been identified. It reads the Embedded-ICE status register to check the current value of the ITBIT bit that shows if the core is in ARM state (ITBIT low) or in Thumb state (ITBIT high). Thumb state debugging isn't supported yet, so the debugger quits with an error message if the target is found to be in Thumb state. A `STMIA r0, {r0-r15}` is put into the processor pipeline to save the contents of all registers that belong to the current processor mode. Two `NOPs` move the `STMIA` to the Memory stage, where the register content is captured. The `arm9tdmi_put_instruction()` calls that capture the register values don't actually put instructions into the pipeline, because no new instructions are being fetched while the `STMIA` stays in the Memory stage. `arm9tdmi_read_cpsr()` is called to get the CPSR, and all JTAG commands queued up to this point are executed. This ensures that the `context` array and the CPSR value have been read from the target before they're used.
`arm7_9_mode_to_number()` is called to check that the content of the CPSR holds a valid processor mode. If this check fails a serious problem has occurred, and the software quits. The values from the `context` array are transfered to the register list using the `ARM7_9_CORE_REG_MODE` macro, and all of these values are marked as clean, to indicate that the values held by the debugger are the same as those in the core and don't have to be restored before debug state is left.
The value of the program counter stored by the `STMIA` instruction is fixed by subtracting 0x18, because the `STMIA` itself added 0xc to the PC, and debug entry added another 0xc. If the core was in one of the exception modes (not User or System mode), the saved program status register (SPSR) of that mode is saved similar to `arm9tdmi_read_cpsr()`.
All core regs are marked as clean, `arm9tdmi_full_context_queried` is set to 0, as only the registers of one mode have been queried yet, and registers r0 and r15 are marked as dirty, because register r0 was

modified when the CPSR was read, and register r15 was changed with each instruction executed in debug state.

Listing 7.67: ./target/arm9tdmi.c

```
457 int arm9tdmi_full_context(void)
```

`arm9tdmi_full_context()` goes through all modes except the current processor mode (has already been stored) and the System mode (shares registers with the User mode) using ordinal mode numbers, and stores the banked registers of each mode. The processor mode is changed using a MSR instruction with an immediate operand:

Listing 7.68: ./target/arm9tdmi.c

```
482 /* change processor mode */
483 tmp_cpsr = arm7_9_core_regs[ARM7_9_CPSR].value & 0xE0;
484 tmp_cpsr |= arm7_9_number_to_mode(i);
485
486 /* MSR CPSR_c, #imm8 */
487 arm9tdmi_put_instruction(ARM7_9_MSR_IM(tmp_cpsr, 0, 1, 0), 0, 0, NULL);
```

The processor mode is stored in bits 0 to 4 of the program status register, but using an immediate operand with `MSR`, eight bits of the CPSR are changed at a time. The upper three bits of the CPSR's least significant byte are bitwise ORed with the mode value of the mode currently being handled to form the operand. Registers r13 and r14 are banked in each mode, and are therefor always saved using two `STR rN, [r0]` instructions. In case of the FIQ mode, additionally r8 to r12 are stored using `STR` instructions. If the mode being handled is an exception mode (not User or System), the SPSR of that mode is saved, too, using an `MRS` instruction macro with the S bit (second macro argument) set high. The memory locations to which the register values should be stored are calculated using the `arm7_9_core_reg_map[mode][num]` array. After all registers have been handled, the original core mode is restored using an `MSR` instruction.

Listing 7.69: ./target/arm9tdmi.c

```
550 int arm9tdmi_write_psr(u32 value, int r)
```

`arm9tdmi_write_psr()` may be called to change all 32 bits of the CPSR (r == 0) or the SPSR (r == 1) at the same time. Using a `MSR` instruction with an immediate operand, 8 bits are written at the same time. The additional `NOPs` that are inserted into the pipeline are necessary to adhere to the instruction cycle timing of the `MSR` instruction, which incurs two additional I-cycles (internal cycles) after the instruction reached the Execute stage. The number of cycles could be reduced by using a `MSR` instruction with a register operand, but that would require the use of a general purpose register, which would have to be restored afterwards. Using the immediate instruction variant, no register's have to be modified, making `arm9tdmi_write_psr()` more flexible to use.

Listing 7.70: ./target/arm9tdmi.c

```
572 int arm9tdmi_restore_context(void)
```

Before the core may leave debug state, the execution context has to be restored using `arm9tdmi_restore_context`, which is called by ARM7/ARM9 common code. It uses the linear register array (`arm7_9_core_regs`) to iterate through each register. If a register is marked as dirty and doesn't belong to the current processor mode, the CPSR is modified to bring the processor into that mode using a `MSR` instruction with an immediate operand. In case of a regular register (r0-r14), the register is restored

using a LDR instruction with r15 as the base address. It is important to use a base register that contains a 4-byte aligned address, otherwise the loaded value would be rotated to reflect endianess effects on unaligned loads [DDI0100E, p. A4-37]. If the current register is one of the saved program status registers (SPSR), arm9tdmi_write_psr() is called with the *r* argument set to 1 (see above).
After all registers have been restored, the CPSR is examined. If it's marked as not dirty, but the current core mode doesn't match the saved core mode, a MSR instruction with an immediate operand is used to restore the processor mode. If the CPSR is marked as dirty, arm9tdmi_write_psr() is called with the *r* argument set to 0. The program counter (r15) is restored last, as it will be modified again by each instruction executed. When loading the PC, the instruction cycle timing has to be carefully followed, to ensure that the instruction fetched next comes from the desired address. The data bus is read during the second Execute cycle. During the third Execute cycle, the value read is written to the register file, and in the fourth Execute cycle, the first instruction is fetched from the new address. arm9tdmi_restore_context() therefor executes Execute cycles one to three of the LDR instruction, to ensure a following instruction is fetched from the correct address.

Listing 7.71: ./target/arm9tdmi.c

```
656 int arm9tdmi_execute_resume(void)
```

arm9tdmi_execute_resume() is called by the ARM7/ARM9 common code after the core context has been restored. It scans a branch instruction B -2 into the pipeline that causes a branch back to the instruction itself. After the branch, a NOP with the SYSSPEED bit set high is scanned into the pipeline. The JTAG instruction is changed to RESTART, and a runtest command is built and put into the queue with the new_endstate field set to TAP_RTI. The JTAG command queue is executed, which results in the branch instruction being executed at system speed, bringing the core back to normal operation.

Listing 7.72: ./target/arm9tdmi.c

```
689 int arm9tdmi_execute_step(void)
```

Single-stepping is similar to resuming the core, but depending on the arm9tdmi_has_singlestep flag, either the ARM9TDMI_DBG_CONTROL_SSTEP bit has to be set in the Embedded-ICE debug control register, or the two Embedded-ICE watchpoint units have to be programmed for an inverse-breakpoint (see §3.3). The watchpoint unit registers are backed up, before arm7_9_embeddedice_step() is called to program the inverse breakpoint. The core is then resumed with the same steps as in arm9tdmi_resume(). arm7_9_poll() is called for up to five seconds to wait for the core to reenter debug state. The ARM9TDMI_DBG_CONTROL_SSTEP is then cleared again, in case the core supports this feature, otherwise the Embedded-ICE registers are restored. If the target timed out while the debugger waited for it to reenter debug state, an error is returned, to inform calling code that the core isn't in halt mode.

Listing 7.73: ./target/arm9tdmi.c

```
773 int arm9tdmi_read_memory(u32 address, u32 size, u32 count, u8 *buffer)
```

Only memory reads of 8, 16, or 32 bit items aligned to the access size are supported, and the arguments given to arm9tdmi_read_memory() are checked against these requirements. The address is loaded with a LDR r0, [r15] instruction into register r0, which serves as the base register for the following system memory accesses.
Memory is most efficiently accessed using a LDM load multiple instruction, that may be used to load up to 14 values from system memory into registers r1 to r14. Registers r0 and r15 can not be used, because one register has to contain the base address for the accesses (r0), and the PC (r15) is too limited in its

use. Only full word accesses (32 bit) are possible using load multiple instructions, therefor LDRH (load register halfword) and LDRB (load register byte) have to be used for smaller accesses.

Listing 7.74: ./target/arm9tdmi.c

```
812 case 4:
813     buf32 = (u32*)buffer;
814     while (num_accesses < count)
815     {
816         u32 reg_list;
817         thisrun_accesses = ((count - num_accesses) >= 14) ? 14 :
             (count - num_accesses);
818         reg_list = (0xffff >> (15 - thisrun_accesses)) & 0xfffe;
819         arm9tdmi_put_instruction(ARM7_9_LDMIA(0, reg_list, 0, 1),
            0, 0, NULL);
820         arm9tdmi_put_instruction(ARM7_9_NOP, 1, 0, NULL);
821
822         jtag_set_instruction(ARM7_9_RESTART);
823         jtag->queue_command(state_move);
824
825         for (timeout=0; timeout<5; timeout++)
826         {
827             arm7_9_read_ice_reg(&arm9tdmi_ice_regs[ARM7_9_DBG_STAT]);
828             if ((retval = jtag->execute_queue()) != ERROR_OK)
829                 return retval;
830
831             if (arm9tdmi_ice_regs[ARM7_9_DBG_STAT].value &
          ARM7_9_DBG_STATUS_SYSCOMP)
832                 break;
833             sleep(1);
834         }
```

If size is set to 4, a pointer variable buf32 is set to the value of buffer, casted to a U32* pointer. num_accesses counts the number of items accessed so far, and as long as this is below the number of requested accesses (count), memory is read from the target system in blocks of thisrun_accesses, and stored using the buf32 pointer. thisrun_accesses is either the number of items remaining (count - num_accesses) or 14, whichever is smaller. The reg_list for the ARM7_9_LDMIA macro is calculated depending on the number of accesses required. The LDMIA instruction is put into the pipeline, followed by a NOP with the SYSSPEED bit set high. The JTAG instruction is changed to RESTART, a state move command is queued that moves the TAP controller state machine to TAP_RTI, and the Embedded-ICE status register is scheduled to be read. Executing the JTAG command queue makes the core execute the LDMIA at system speed and reads the status register, which is checked for the ARM7_9_DBG_STATUS_SYSCOMP bit to determine if the core reentered debug state. This is done for up to five seconds, after which the target is considered to be stalled, and a timeout error is returned.

Listing 7.75: ./target/arm9tdmi.c

```
841         arm9tdmi_put_instruction(ARM7_9_STMIA(0, reg_list, 0, 0), 0, 0,
          NULL);
842         arm9tdmi_put_instruction(ARM7_9_NOP, 0, 0, NULL);
843         arm9tdmi_put_instruction(ARM7_9_NOP, 0, 0, NULL);
```

```
844
845             for (i=1; i<=thisrun_accesses; i++)
846             {
847                 arm9tdmi_put_instruction(ARM7_9_NOP, 0, 0, buf32);
848                 buf32++;
849             }
850             num_accesses += thisrun_accesses;
851             if ((retval = jtag->execute_queue()) != ERROR_OK)
852                 return retval;
```

If the accesses completed successfully, a `STMIA` instruction executed at debug speed is used to scan the data out of the registers into the debugger. The values are captured once `STMIA` reached the Memory stage, and are written to the memory pointed to by the `buf32` pointer, which is advanced after each word. If the software is to be ported to a big-endian host, or if support for big-endian targets is to be added, this would have to be changed, to take endianess effects into account.

Accesses of 2 and 1 bytes size are handled similarly, but instead of a single `LDMIA` instruction, `LDRH` or `LDRB` are executed `thisrun_accesses` times. An additional array `thisrun_buffer`, large enough to take 14 words, is used to capture the up to 14 words saved by `STMIA`. If the data should have been stored directly to the buffer, as it's done for 32 bit accesses, the debug scan chain field description would have to be changed, to be able to access the lower 8 or 16 bit of the data bus independently from the others, and the JTAG queued command system would have to support half-word or byte destinations for the values captured from a data register scan. After storing the data in the `thisrun_buffer` using a single run, the data is copied into the `buffer` using the `buf8` or `buf16` pointer respectively.

After all requested memory has been read, the used registers are marked as dirty, to make sure they're restored before the core is resumed. `arm9tdmi_read_cpsr()` is called to see if an access caused a data abort, in which case the CPSR would be restored using a `MSR` instruction with an immediate operand.

Listing 7.76: ./target/arm9tdmi.c

```
841 int arm9tdmi_write_memory(u32 address, u32 size, u32 count, u8 *buffer)
```

Like memory reads, memory writes are only possible with items of 8, 16, or 32 bits, and have to be aligned to the access size. `arm9tdmi_write_memory()` checks the parameters against these restriction, and loads the `address` into register r0, which is used as the base register. Memory is written similarly to how it's read, but instead of executing system speed load instructions followed by debug speed store multiples, load multiple instructions are executed at debug speed to fill the registers with the data that should be written, and store instructions are executed at system speed to write the register content to the target memory. The code to write memory doesn't have to use temporary arrays as the read code does (`thisrun_buffer`), because the JTAG scan chain fields for the data bus are always 32 bits large, no matter what size the data being accessed is.

Using 14 registers of 4 bytes size in a single `STM` instruction, it's possible to write 56 bytes at a time to the target memory. After the `STM` has executed, the debugger has to wait for the DBGACK signal to go high in the debug status register. That means that on systems with high latency, the time, that passes between sending the queued commands to the target, and waiting for a reply with the content of the Embedded-ICE status register, limits the maximum download speed that can be achieved, no matter how fast the JTAG clock actually is. To be able to bypass this restriction, possibly unsafe code can be enabled by defining the `UNSAFE_MEMORY_WRITE` macro inside `arm9tdmi.c`. This replaces the status regis-

ter checking with a call to `arm7_9_guard_ice_status()` that checks if the Embedded-ICE status register contains the value 0xd to indicate a completed system speed access. If everything went well, and the core was able to execute every access in the time it took the debugger to initiate the ICE register read, the Embedded-ICE status register value matches, and the target memory was written successfully. If during all the system speed accesses, the status register didn't match only once, the target is in an possibly unsafe state, as the debugger may have tried to execute instructions at debug speed while the core was still performing the system speed access. This error can be detected, and the procedure may be tried again, after resetting the target and slowing the JTAG clock down.

Like `arm9tdmi_read_memory()`, `arm9tdmi_write_memory()` checks the CPSR after all data is written, and restores it if necessary.

armv4mmu.[ch]

The memory management units implemented in ARM7 and ARM9 cores offer the same programmer's model with only small differences, so both core families are supported by common code located in `armv4mmu.[ch]`.

Listing 7.77: ./target/armv4mmu.h

```
27 typedef struct armv4mmu_debug_s
28 {
29     u32 (*get_ttb)(void);
30     int (*read_memory)(u32 address, u32 size, u32 count, u8 *buffer);
31     int (*write_memory)(u32 address, u32 size, u32 count, u8 *buffer);
32     void (*disable_mmu_caches)(int disable_mmu, int disable_caches);
33     void (*restore_mmu_caches)(void);
34     int has_tiny_pages;
35 } armv4mmu_debug_t;
```

The common MMU code requires access to core-specific functions, which is realized through the `armv4mmu_debug_t` interface defined in `armv4mmu.h`. `get_ttb()` is called to determine the translation table base address, the physical address where the first-level page table is located. `read_memory()` and `write_memory()` are used to access target memory. They shouldn't do any translations or cache/MMU operations, but simply read or write memory at the address specified (e.g. like the `arm9tdmi_xxx_memory()` functions). `disable_mmu_caches()` is called to disable the MMU, the caches, or both, while `restore_mmu_caches()` has to restore both to their previous state, i.e. to what they were set on debug entry. `has_tiny_pages` is a flag indicating whether the current core supports tiny pages (ARM9) or not. This is used to catch page table errors on ARM7 systems, where information indiciating a tiny page is an error.

`armv4mmu.h` further defines an enumeration of possible memory regions (`ARMV4_SECTION`, `ARMV4_xxx_PAGE`), and declares functions that may be used by cores with MMU and caches to handle these:

Listing 7.78: ./target/armv4mmu.h

```
41 extern u32 armv4mmu_translate_va(u32 va, int *type, u32 *cb,
           int *domain, u32 *ap);
42 extern int armv4mmu_read_physical(u32 address, u32 size, u32 count,
         u8 *buffer);
```

```
43 extern int armv4mmu_write_physical(u32 address, u32 size, u32 count,
           u8 *buffer);
44
45 extern int armv4mmu_handle_virt2phys_command(struct cli_def *cli,
          char *cmd, char **args, int argc);
46 extern int armv4mmu_handle_md_phys_command(struct cli_def *cli,
         char *cmd, char **args, int argc);
47 extern int armv4mmu_handle_mw_phys_command(struct cli_def *cli,
         char *cmd, char **args, int argc);
```

Textual representations of the memory region enumeration types are defined in `armv4mmu.c` by the `armv4mmu_page_type_names` array of **`char`** pointers.

Listing 7.79: ./target/armv4mmu.c

```
39 u32 armv4mmu_translate_va(u32 va, int *type, u32 *cb, int *domain,
          u32 *ap)
```

`armv4mmu_translate_va()` takes a virtual address in `va`, and returns the corresponding physical address. The type of memory region is returned using the `*type` pointer, the cacheable and bufferable bits via `*cb`, the domain bits through `*domain`, and the access permission via `*ap`. The translation process is fully described in [DDI0100E, p. B3-6].
The translation table base (TTB) is queried using `armv4mmu_debug->get_ttb()`, and the first-level descriptor is fetched using a phyiscal memory access. The location of the first-level descriptor is calculated by taking the most significant 18 bits from the TTB, and adding the most significant 12 bits from the virtual address shifted to the right by 18. The two least significant bits of the first level descriptor determine its type. The value b00 is reserved, and results in a translation error. The value b11 is used for a fine page table, and is only valid for systems that support tiny pages, so `armv4mmu_debug->has_tiny_pages` is checked, and a translation error is returned if the core doesn't support these. The domain is always encoded in bits 5 to 8 of the first-level descriptor, and is assigned to the memory pointed at by `*domain`. A value of b10 indicates a section descriptor, in which case no second-level descriptor is required. The remaining fields are filled using the values from the descriptor, and the return value is calculated by taking bits 24 to 31 from the first-level descriptor ORed with bits 0 to 23 of the virtual address.
The address of the second-level descriptor is calculated by taking the most significant 22 bits of a coarse page table first-level descriptor (type b01) ORed with bits 12 to 19 of the virtual address shifted 10 bits to the right, or the most significant 20 bit of a fine page table first-level descriptor (type b11) ORed with bits 10 to 19 of the virtual address shifted 8 bits to the right.
The type of a second-level descriptor is determined by bits 0 and 1, too, like for a first-level descriptor. The value b00 is reserved, and results in a translation error. The cacheable and bufferable bits are always encoded in bits 2 and 3, and are returned via the `*cb` pointer. The other fields are filled using the bits from the second-level descriptor, and the physical address of the requested page is calculated and returned.
If a translation error occurred within `armv4mmu_translate_va()`, the `*type` is set to -1 to indicate an error, and the return value can be interpreted as an error code.

The `armv4mmu_read_physical()` and `armv4mmu_write_physical()` functions use `armv4mmu_debug->disable_mmu_caches()` to disable both the MMU and caches, call the read or write memory function, and use `armv4mmu_debug->restore_mmu_caches()` to restore the MMU and caches to their previous state.

`armv4mmu_handle_virt2phys_command()` is a simple wrapper around the `armv4mmu_translate_va()` function. It takes one argument that is interpreted as a virtual address, calls `armv4mmu_translate_va()`,

and displays the results.

armv4mmu_handle_md_phys_command() and armv4mmu_handle_mw_phys_command() are wrappers around the armv4mmu_xxx_physical() functions. They take an address and a count (md), or an address and a value (mw), and call the corresponding physical memory access function.

arm9cache.[ch]

ARM9 based targets may have caches of a variable size. The code in arm9cache.[ch] can be used to identify these caches and provides a command handler for a CLI command that prints information about the caches to the user interface.

Listing 7.80: ./target/arm9cache.h

```
24 typedef struct arm9_cachesize_s
25 {
26     int linelen;
27     int associativity;
28     int nsets;
29     int cachesize;
30 } arm9_cachesize_t;
31
32 typedef struct arm9_cache_s
33 {
34     int ctype;  /* specify supported cache operations */
35     int separate;   /* separate caches or unified cache */
36     arm9_cachesize_t Dsize; /* data cache */
37     arm9_cachesize_t Isize; /* instruction cache */
38 } arm9_cache_t;
39
40 arm9_cache_t arm9_cache;
41
42 extern int arm9_identify_cache(u32 cache_type_reg, arm9_cache_t *cache);
43 extern int arm9_handle_cacheinfo_command(struct cli_def *cli, char *cmd,
            char **args, int argc);
```

The arm9_cachesize_t type holds information about a data or an instruction cache, describing its internal structure. The arm9_cache_t type specifies the caches implemented on a core: the operations they support, whether it's a unified cache or separate data and instruction caches, and two arm9_cachesize_t fields that describe the two parts of the cache.

Listing 7.81: ./target/arm9cache.c

```
24 int arm9_identify_cache(u32 cache_type_reg, arm9_cache_t *cache)
```

arm9_identify_cache() takes the value of the cache type register, a CP15 register on cores that support it, as an argument, and fills arm9_cache with information about the caches found on the current core. [DDI0100E, p. B2-9] explains the layout of the cache type register, and how this information may be used to calculate the properties of the caches.

arm920t.c

The `target_t` interface for arm920t cores is defined as `arm920t_target`, the `arm7_9_debug_t` is defined as `arm920t_debug`, and the `armv4mmu_debug_t` interface is defined as `arm920t_mmu_debug`. The ARM920t is based on an ARM9TDMI core, so their `target_t` and `arm7_9_debug_t` are very similiar, with a few exceptions where the ARM920t has to do additional steps to those the ARM9TDMI does. The ARM920t supports tiny pages, and has the `has_tiny_pages` flag set to 1.

Listing 7.82: ./target/arm920t.c

```
123 static int arm920t_mmu_enabled = 0;
124 static int arm920t_icache_enabled = 0;
125 static int arm920t_dcache_enabled = 0;
126 static int arm920t_caches_identified = 0;
127
128 u32 arm920t_saved_cp15_control_reg;
129 u32 arm920t_saved_dfsr;
130 u32 arm920t_saved_ifsr;
131 u32 arm920t_saved_dfar;
132 u32 arm920t_saved_ifar;
```

The ARM920t has additional attributes that describe its execution context. On debug entry, the MMU, instruction cache, and data cache may be enabled, and have to be in the same state when the core is resumed. `arm920t_caches_identified` is a flag that indicates whether `arm9_identify_cache()` has been called already to identify the caches. As the caches wont change during a debugging session, this information only has to be calculated once. `arm920t_saved_cp15_control_reg` holds the value of the coprocessor 15 register 1, the control register, which has to be restored before the core may be resumed. `arm920t_saved_xfsr` and `arm920t_saved_xfar` contain the value of the data and instruction fault status and address register. These are set by the core when a data or instruction fetch abort occured, and give detailed information about the reason for the abort and the address on which it occured. If a system speed memory access is aborted while the debugger is connected, the fault registers can get modified, and have to be restored to their previous state.

Listing 7.83: ./target/arm920t.c

```
134 int arm920t_arch_state()
```

`arm920t_arch_state()` calls `arm7_9_arch_state()` to display information about the current state, and additionally displays the state of the MMU, the data cache and the instruction cache, using the information from the `arm920t_xxx_enabled` flags.
The system control coprocessor (CP15) of an ARM920t core is accessed using a combination of physical and interpreted accesses (see §4.2).

Listing 7.84: ./target/arm920t.c

```
153 int arm920t_read_cp15_physical(int reg_addr, u32 *value)
```

Physical read access to CP15 registers is implemented in `arm920t_read_cp15_physical()`. The function takes the register address that should be read, and a pointer to the memory location where the result should be stored. The CP15 scan chain is selected, and the current JTAG instruction is changed to INTEST. A data register scan command is built, and the JTAG scan chain field description is created to match the layout of scan chain 15. The least-significant bit of the scan chain is set to 1 to indicate a physical access. Like Embedded-ICE registers, CP15 registers have to be read using two JTAG scans, one

that programs the read/write bit and the address, and the second to capture the data. The endstate for the scan operation is changed to TAP_RTI to ensure that the TAP state machine moves through Update-DR between the two scans.

Listing 7.85: ./target/arm920t.c

```
211 int arm920t_write_cp15_physical(int reg_addr, u32 value)
```

`arm920t_write_cp15_physical()` is called to write a CP15 register using a physical access. The procedure is similar to a CP15 register read, but only one access is required, programming the read/write field, the address, and the data to be written.

Listing 7.86: ./target/arm920t.c

```
265 int arm920t_read_cp15_interpreted(u32 opcode, u32 *value)
```

The CP15 instruction opcodes that may be used to access CP15 registers in interpret mode are listed in [DDI0151C, p. 9-35]. `arm920t_read_cp15_interpreted()` sets the CP15 interpret bit (bit 0) in the test state register using a read-modify-write access. The test state register is read through `arm920t_read_cp15_physical()`, its value is changed, and the new value is written back using `arm920t_write_cp15_physical()`. The CP15 scan chain is selected, and the current JTAG instruction is changed to INTEST. A data register scan command is built, similar to a physical access, but the least-significant bit is set to 0, to indicate an interpreted access. The opcode for the desired operation is scanned into bits 1 to 32 (instruction word), bits 33 to 39 (address and read/write) are set to zero. The scan command is put into the JTAG queue, and a `LDR r0, [r15]` instruction, followed by a `NOP` with the SYSSPEED bit set high, is put into the ARM9TDMI pipeline via `arm9tdmi_put_instruction()`. The system speed load instruction is executed by changing the JTAG instruction to RESTART and moving the state machine to Run-Test/Idle. The Embedded-ICE status register is polled until the SYSCOMP bit goes high, showing that the system speed access completed, or until five seconds have passed, which is treated as a timeout error. If the access completed successfully, the CP15 test state register is restored using a read-modify-write access that clears the CP15 interpret bit (bit 0). The debugger inserts a `STR r0, [r15]` instruction using `arm9tdmi_put_instruction()` into the core pipeline, and clocks it to the Memory stage with three `NOPs`. The content of the CP15 register that was transfered to r0 is captured and written to the memory location specified by the `*value` pointer.

Listing 7.87: ./target/arm920t.c

```
367 int arm920t_write_cp15_interpreted(u32 opcode, u32 value, u32 address)
```

`arm920t_write_cp15_interpreted()` loads the value that should be written into register r0, and the address at which the value should be written into r1. Specifying an address is necessary, because several CP15 operations, like flushing the instruction cache, depend on the address value. Using `arm9tdmi_put_instruction()`, two `LDR` instructions that load registers r0 and r1, are put into the processor pipeline, and clocked into the Memory stage, where `value` and `address` are written to the registers. The CP15 interpret bit (bit 0) of the CP15 test state register is set using a read-modify-write access similar to `arm920t_read_cp15_interpreted()`, the CP15 scan chain is selected, and the current JTAG instruction is set to INTEST. A data register scan command identical to the one from `arm920t_read_cp15_interpreted`() is built and put into the JTAG command queue. A system speed `STR r0, [r1]` is executed, and after the core completed the instruction (SYSCOMP high), the CP15 test state register is restored.

Listing 7.88: ./target/arm920t.c

```
468 void arm920t_disable_mmu_caches(int disable_mmu, int disable_caches)
```

To disable the MMU or the Caches, `arm920t_disable_mmu_cache()` calls `arm920t_read_cp15_physical()` to read the current value of the CP15 control register, clears the MMU enable bit (if `disable_mmu is set`) and the ICache and DCache enable bits (if `disable_caches` is set), and writes the modified value back to the control register.

`arm920_restore_mmu_caches()` restores the CP15 control register using a call to `arm920t_write_cp15_physical()` with the value stored in `arm920t_saved_cp15_control_reg`.

Listing 7.89: ./target/arm920t.c

```
493 int arm920t_register_commands(struct cli_def *cli)
```

`arm920t_register_commands()` registers commands from `arm9cache.c`, `armv4mmu.c`, and its own commands, that give a user access to the CP15 registers using physical accesses (`CP15`), and interpreted accesses (`CP15i`). After these have been registered, `arm9tdmi_register_commands()` is called to register commands that apply to all ARM9 cores, which in turn calls `arm7_9_register_commands()` for commands that are available on all ARM7 and ARM9 cores.

Listing 7.90: ./target/arm920t.c

```
515 int arm920t_init(struct cli_def *cli)
```

`arm920t_init()` assigns the arm920t implementations of the `arm7_9_debug_t` and `armv4mmu_debug_t` interfaces to the global `arm7_9_debug` and `armv4mmu_debug` pointers, and calls `arm9tdmi_init()` to initialize the common ARM9 code. `arm920t_quit()` doesn't execute any code itself, but calls `arm9tdmi_quit()` to give underlying code a chance to quit.

Listing 7.91: ./target/arm920t.c

```
538 int arm920t_minimum_context(void)
```

`arm920t_minimum_context()` has to save the additional attributes that define the ARM920t's execution context. `arm9tdmi_minimum_context()` is called to ensure that all registers have been saved before they're used otherwise. The CP15 control register is read using a physical access, and saved to `arm920t_saved_cp15_control_reg`. If `arm920t_caches_identified` isn't already set, the CP15 cache type register is read using a call to `arm920t_read_cp15_physical()`. `arm9_identify_caches()` is invoked with the value of the cache type register, and the `arm920t_caches_identified` flag is set to 1. The state of the MMU and the caches is saved in `arm920t_mmu_enabled`, `arm920t_dcache_enabled`, and `arm920t_icache_enabled` by examining the bits from `arm920t_saved_cp15_control_reg`. The instruction and data fault status and address register are read using calls to `arm920t_read_cp15_interpreted()`, and linefills on the instruction and data cache are disabled by setting the *disable DCache linefill* and *disable ICache linefill* bits using a read-modify-write access to the CP15 test state register.

Listing 7.92: ./target/arm920t.c

```
583 int arm920t_restore_context(void)
```

`arm920t_restore_context()` restores the content of the instruction and data fault status and address registers (IFSR, IFAR, DFSR, DFAR) using calls to `arm920t_write_cp15_interpreted()` with the values stored when the core entered debug state (`arm920t_saved_xfxr`). The CP15 test state register is restored using a read-modify-write access that clears the *disable DCache linefills* and *disable ICache linefills* bits.

Finally, `arm9tdmi_restore_context()` is called to prepare the core for a resume or single-step operation.

Because linefills are disabled for the data and instruction caches, `arm920t_read_memory()` doesn't have to take any extra measures when reading memory, and can rely on `arm9tdmi_read_memory()` for this purpose. If an address being accessed is contained in the data cache, the request is served from the cache, otherwise main memory is read and the cache isn't updated.

Listing 7.93: ./target/arm920t.c

```
614 int arm920t_write_memory(u32 address, u32 size, u32 count, u8 *buffer)
```

Because of its separate caches and the support for write-back (only the cache is updated on a hit, and modified cache lines are marked as dirty) memory regions, the ARM920t has to ensure consistency between the data cache, the main memory, and the instruction cache (see §4). `arm920t_write_memory()` calls `arm9tdmi_write_memory()` to write all data, without taking caches into account. The memory writes most likely to cause coherency problems are writes that modify a single item of two or four bytes, used to set software breakpoints in ARM and Thumb state. If the data cache is enabled during such a write operation, `arm920t_write_memory()` calls `armv4mmu_translate_va()` to retrieve information about the memory region affected by the write. If the cacheable and bufferable bits are set to b11, indicating a write-back memory region, `armv4mmu_write_physical()` is called to update the main memory, too. If the instruction cache is enabled, an interpreted write access to CP15 register 7 is initiated to clean an instruction cache line selected by an address.

`arm920t_get_ttb()` calls `arm920t_read_cp15_interpreted()` to read the translation table base and returns the captured value.

`arm920t_handle_cp15_command()` and `arm920t_handle_cp15i_command()` are simple wrappers around the CP15 access functions.

./flash

flash.[ch]

Listing 7.94: ./flash/flash.h

```
27 typedef struct flash_s
28 {
29     char *name;
30     int (*info)(u32 base, struct cli_def *cli);
31     int (*erase)(u32 base, int first, int last);
32     int (*protect)(u32 base, int set, int first, int last);
33     int (*write)(u32 base, u8 *buffer, u32 offset, u32 count);
34     int (*probe)(u32 base);
35 } flash_t;
```

The `flash_t` interface defined in `flash.h` has to be implemented for every supported flash chip or chip family. The `name` field is used to reference a flash driver, and is matched against the argument to a `flash` configuration command. The `base` argument has to be given to every flash function, and identifies a flash bank. This allows multiple flash banks of a similar configuration to be accessed individually.

`info()` is called when the user requested information about a flash bank. `erase()` is used to erase the blocks from `first` to `last`. If a flash implements hardware protection, `protect()` is used to set or clear the protection bits of the selected blocks, otherwise a flash may return a positive result and ignore the call. `write()` should write `count` bytes from the memory pointed at by `buffer` to the flash starting at `offset` bytes from the base of a flash bank. `probe()` is called to check if a flash bank that matches the configuration is located at the address `base`.

`flash.c` defines the `flashes` array that contains pointers to all available flash drivers, the flash initialization function `flash_init()`, that is called after the JTAG and target subsystems have been initialized, and several CLI handler functions that allow a user to access the `flash_t` interface. `flash_init()` checks the flash configuration variables to ensure that the chip and bus widths are multiples of 8 and that the flash size isn't zero. If a flash is configured, the array of `flashes` is searched for a matching driver. If a driver is found, the driver's `flash_t` interface implementation is assigned to the global `flash` variable to allow other parts of the code to access the flash, otherwise an error is returned, indicating an invalid flash configuration. The debugger may continue operation in that case, but the CLI flash commands are not available.

The handler functions defined in `flash.c` are simple wrappers around the interface functions. They parse the command arguments and call the appropriate interface functions.

intel28fxxxj3.c

Listing 7.95: ./flash/flash.c

```
36  flash_t intel28fxxxj3_flash =
37  {
38      .name = "intel28fxxxj3",
39      .info = intel28fxxxj3_info,
40      .erase = intel28fxxxj3_erase,
41      .protect = intel28fxxxj3_protect,
42      .write = intel28fxxxj3_write,
43      .probe = intel28fxxxj3_probe
44  };
45
46  static u32 num_blocks = 0x0;
47  static u32 block_size = 0x0;
48  #define SINGLE_BLOCK (128 * 1024)
```

`intel28fxxxj3.c` implements the `flash_t` interface for Intel 28FxxxJ3 flash chips [I290667]. `intel28fxxxj3_flash` contains the `name` of the flash driver ("intel28fxxxj3"), and pointers to the functions required by the interface. `num_blocks` holds the number of blocks of one flash chip, and `block_size` contains the size of one flash block. A single 28FxxxJ3 flash block has 128 kB or 64 kW, but up to four chips may form a single flash bank, in which case the blocks are twice or four times the size of a `SINGLE_BLOCK`.

Listing 7.96: ./flash/flash.c

```
50  u8 intel28fxxxj3_read_status_register(u32 base)
```

The command set supported by StrataFlash chips features a status register, that's accessible after writing 0x70 to any address located on the chip. `intel28fxxxj3_read_status_register()` uses

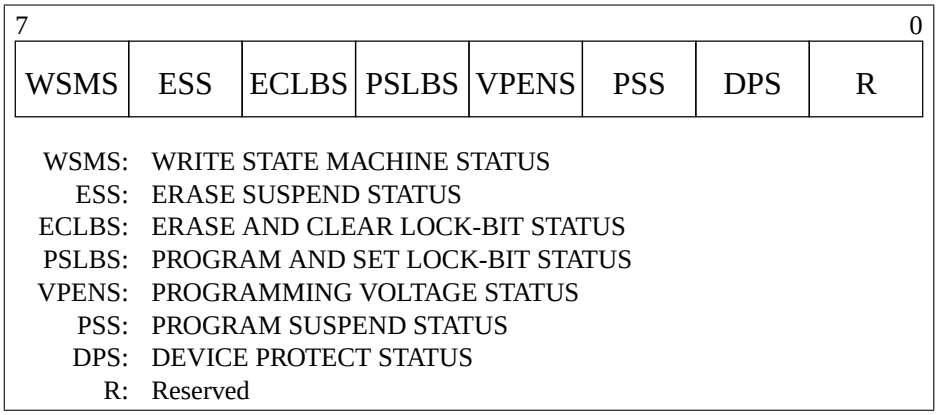

Figure 7.2: Intel 28FxxxJ3 status register layout

`target->write_memory()` to write the command to the base of the flash bank, reads the current status using `target->read_memory()`, and switches the flash chip back to array mode (the default mode of operation, that allows random read accesses to the device) by writing 0xff to the flash. Figure 7.2 shows the layout of the status register.

`intel28fxxxj3_quick_status()` may be used when the device is already in the *read status register* mode, as it's the case after a erase or write command was sent to the flash.

If an error occured during a flash operation, the corresponding bit in the status register is set, and remains set until the status register is cleared by writing 0x50 to an address on the flash chip. `intel28fxxxj3_clear_status_register()` clears the status register by calling `target->write_memory()` to write 0x50 to the base of the flash bank.

`intel28fxxxj3_array_mode()` switches the device back to array mode by writing 0xff to the base of the flash bank.

`intel28fxxxj3_info()` calls `intel28fxxxj3_probe()` to check for the presence of a flash bank at the given base, and displays information about its size, the number of blocks, and the starting address of each block to the CLI.

Listing 7.97: ./flash/flash.c

```
115 int intel28fxxxj3_probe(u32 base)
```

`intel28fxxxj3_probe()` adjusts the `block_size`, if a flash bank consists of more than one chip. It calls `target->write_memory()` to write 0x90, the *Read Identifier Code* command, to the base of the flash bank. This allows the manufacturer code to be read from address 0x0 of the flash device, and the device code from address 0x1. Because address bit A0 isn't used when reading the identifier codes [I290667, p. 39], the offset to the base address has to be shifted one bit to the left, to align the address with address bit A1. After that, the flash is switched back to array mode by writing 0xff to the base of the flash bank. The manufacturer code has to be 0x89 (Intel), and the device code has to be 0x16, 0x17, 0x18, or 0x1d. `num_blocks` is set according to the size of the flash chip that is determined by the device code.

Listing 7.98: ./flash/flash.c

```
186 int intel28fxxxj3_erase(u32 base, int first, int last)
```

`intel28fxxxj3_erase()` verifies that `first` is smaller than `last`, and that both fall into a range valid for the flash device. `intel28fxxxj3()` is called to probe the supplied base address for a flash bank that matches the configuration. Before the erase operation begins, the status register is cleared using `intel28fxxxj3_clear_status_register()`. To erase a flash block, 0x20, the *block erase setup* command, has to be written to an address that falls on the block that should be erased, followed by 0xd0, the *block erase confirm* command. `intel28fxxxj3_quick_status()` is then called to read the status register until the WSMS bit (see Figure 7.2) goes high, indicating that the operation is finished. The status register is then checked for the ECLBS bit, which, if high, indicates an error that happened during the erase operation. If the operation completed successfully, the `block_address` is adjusted, and the operation is repeated until all requested blocks have been erased, and the flash is switched back to array mode using `intel28fxxxj3_array_mode()`.

Hardware protection isn't supported yet by the intel28fxxxj3 driver, so the `intel28fxxxj3_protect()` function simply returns `ERROR_OK`.

Because Intel StrataFlash memory may be used in 8 bit and 16 bit configurations, and flash banks may be formed of up to 32 bits width, flash has to be written using accesses of up to 32 bits at a time. This isn't a problem, because flashes have all bits set to 1 when they are erased, and are written by setting the required bits to 0. If only a part of a 32 bits wide flash bank has to be written, the remaining bytes of the flash word are written to the value they contained before, leaving them in their previous state.

Listing 7.99: ./flash/flash.c

```
231 void intel28fxxxj3_add_byte(u32 *word, u8 byte)
```

`intel28fxxxj3_add_byte()` adds single bytes to a word buffer that is later written to the flash. Depending on the `bus_width`, the previous value of `word` is shifted, and the new `byte` is put in the right place. If the software should be ported to big-endian systems, this code would have to be enhanced to handle the endianess correctly.

Listing 7.100: ./flash/flash.c

```
255 int intel28fxxxj3_write_word(u32 base, u32 address, u32 word)
```

`intel28fxxxj3_write_word()` changes the flash to word/byte program mode by writing 0x40 to the base of the flash bank. After that, the data with a size of the full `flash_buswidth` is written, and the status register is polled until the WSMS bit goes high, indicating that the flash is ready to accept a new command. The status register is checked for an error by examining the PSLBS bit, which is high if an error occured.

Listing 7.101: ./flash/flash.c

```
275 int intel28fxxxj3_write(u32 base, u8 *buffer, u32 offset, u32 count)
```

`intel28fxxxj3_write()` is called to write an arbitrary number of bytes without having to align the data to the access size. The supplied base address is probed for a flash device that matches the configuration, and the arguments are checked for obvious errors, like writing beyond the end of the flash or a count of zero bytes. If there's data to be written that isn't aligned to the `flash_buswidth`, a word is built, using calls to `intel28fxxxj3_add_byte()`, that contains the bytes from addresses before the new data,

the new data, and possibly bytes after the last address that should be written, if the `flash_buswidth` is 32 bits, and one of the middle bytes should be written. After those, data of the full `flash_buswidth` size is written, possibly followed by an incomplete word at the end. That last word is built using `intel28fxxxj3_add_byte()` to add remaining bytes from the `buffer` and any bytes after the last address that should be written. At the end of `intel28fxxxj3_write()`, or in case an error occured while writing, the flash is switched back to array mode by calling `intel28fxxxj3_array_mode()`.

./gdb

`gdb.c` implements the server for the GDB remote protocol.

Listing 7.102: ./gdb/gdb.c

```
39 static int (*regular)(struct cli_def *cli);
40
41 static int gdb_fd = 0;
42 int is_gdb_session = 0;
43
44 void gdb_regular(struct cli_def *cli, int (*callback)(struct cli_def *cli))
45 {
46     regular = callback;
47 }
```

A callback, that should be invoked in regular intervals, may be assigned to the `regular()` function pointer using `gdb_regular()`. This gives a behavior similar to the CLI implemented by libcli. `gdb_fd` is a global variable local to `gdb.c` (therefor **`static`**), that allows the file descriptor of the current gdb connection to be accessed from within several functions without having to carry it in every function's argument list. `is_gdb_session` is a globally visible flag that's used by the GDB module to inform other subsystems that the target is currently being controlled by a remote gdb client.

Listing 7.103: ./gdb/gdb.c

```
49 int get_char()
```

`get_char()` manages a **`static char`** `read_buffer[256]`, that is filled by reading 256 characters from the `gdb_fd` when the buffer is empty. One character at a time is returned from `read_buffer` to the calling function until an attempt to fill the buffer fails, because the remote connection has been closed.

Listing 7.104: ./gdb/gdb.c

```
72 void put_packet(char *buffer, int len)
```

`put_packet()` is used to send packets to the gdb client using the `gdb_fd`. A '$' character is sent to start a new packet, followed by the payload data of `len` bytes length supplied in `buffer`. The data is followed by a '#' character and the checksum transmitted as a two hex-digit byte. The checksum is calculated by continuously adding each byte from `buffer` to an **`unsigned char`** variable `my_checksum`. This results in the sum of all the bytes in `buffer` modulo 256, because `my_checksum` overflows whenever its value grows beyond 255. The result is written to the three character array `checksum` using `snprintf()` with a format string that prints an integer formatted as two hex digits. The resulting string is sent to the remote gdb client. To verify that the packet was received correctly, the reply is read from `gdb_fd`. If the reply was a '+' character, `put_packet()` can be finished, otherwise the packet has to be retransmitted.

Listing 7.105: ./gdb/gdb.c

```
101 void cli_print_gdb(struct cli_def *cli, char *line)
```

`cli_print_gdb()` is a call-back function for libcli that is used to route output from CLI commands through the GDB server. The call-back function registered with `cli_print_callback()` is called once for every line that should be printed, and doesn't contain any newlines. The gdb remote serial protocol specifies the 'O' packet for arbitrary text strings that should be output to the user. The text has to be transmitted hex-encoded, that is, every character has to be encoded using two hex digits. `cli_print_gdb()` allocates a buffer twice the size of the string that should be transmitted plus four additional bytes. The first character of the output buffer is set to 'O' to indicate a hex-encoded ASCII data packet. The text string is added to the buffer using a loop that prints every character as a two hex digit value. The last three characters are '0', 'a', and 0x0, "0a" for a newline, and 0x0 to terminate the string. The complete string is sent to the client using `put_packet()`.

Listing 7.106: ./gdb/gdb.c

```
122 int get_packet(char *buffer, int *len)
```

`get_packet()` reads a packet from the gdb client, and returns it in the `buffer` supplied by the calling function with `len` set to the number of valid characters. Characters from the client are requested using `get_char()`, until a '$' character that starts a new packet or '^C', the SIGINT signal is found. Any other characters are dropped with a warning printed to the daemon's error stream. The packet is read until a '#' is encountered that finishes the packet. Every character read is used to calculate `my_checksum`, similar to `put_packet()`. The checksum is then read using two calls to `get_char()`, and `strtoul()` is used to convert the two hex digits into an unsigned integer. If the two checksums match, a '+' character is sent to the client, acknowledging the packet, otherwise a '-' is sent to request a retransmission.

Listing 7.107: ./gdb/gdb.c

```
565 void gdb_loop(struct cli_def *cli, int fd)
```

`gdb_loop()` is called by the daemon when a new gdb connection has been accepted. The file descriptor argument `fd` is assigned to `gdb_fd` to able to access the connection from all functions inside `gdb.c`. The target is halted via `target->halt()`, because a gdb client expects the target to be stopped when the connection is initiated. After sleeping for a second to give the target time to enter halt state, the `regular()` callback is called to handle the target. Using calls to `poll()`, the remote connection is monitored for activity. If `poll()` returned an `EINTR` error, a user interrupted the system call using `<ctrl><c>` to shut down the daemon, and the `gdb_loop()` is left, giving control back to the daemon. If the `poll()` timed out, the `regular()` callback is invoked to handle the target. If a non-zero return value is returned, indicating entry into debug state, a 'S' packet, that delivers the `target->gdb_last_signal()`, is sent to the client to inform it about the changed state.
If the polling showed incoming data (`POLLIN` set), the data is requested by calling `get_packet()`, and the character after the last valid one is set to 0x0 to terminate the string. The first character of the read data determines the type of the packet received, and is used in a **`switch`** statement to determine the action that should be taken. The packets are handled by `xxx_packet()` functions defined in `gdb.c`. These functions are wrappers that call the appropriate functions of the `target_t` interface after parsing the received packet, see [GDB01] for a description of the various packets.
Before `gdb_loop()` is left, `target->resume()` is called to resume the target, and `is_gdb_session` is cleared.

8 Verification

This chapter is going to verify that the goals set for this diploma thesis have been achieved, by comparing the implemented software with the requirements. The software in its current state supports ARM7TDMI, ARM720t, ARM9TDMI (generic), and ARM920t cores, and allows two different JTAG hardware interfaces to be used. It is completely open, relying only on free and open source software, with the exception of the library required by the ftd2xx driver. Where this is a problem, the ftdi2232 driver may be used, that works with the GPL licensed libftdi, instead of the proprietary ftd2xx library.
The target subsystem is modularized and layered, allowing additional cores to be added while using parts of the existing code. All required target functionality is implemented in the `target_t` interface and the CLI commands that allow core specific functionality to be added. Debugging of cores with a MMU and caches is supported by the generic code in `armv4mmu.[ch]` and `arm9caches.[ch]`, together with the core specific CP15 support available for ARM720t and ARM920t cores.
The required `jtag_t` interface is defined in `jtag.h`, and supports all requested operations, including the queuing of large command lists to facilitate the use of JTAG interfaces that incur high latency. Support for additional JTAG interfaces can be easily added, possibly reusing large parts of the code, in case of a bit-bang device.
Flash writing is supported for Intel StrataFlash chips, but the `flash_t` interface is generic enough to support various different flash devices.
User interaction is implemented using libcli to provide a command line interface accessible via telnet, and through the code in `gdb.c`, that allows a remote gdb client to connect using the gdb serial remote protocol.
Error reporting is consistent throughout the code, printing messages of a user selectable priority to the daemon's console. To offer a maximum of performance, the ftd2xx driver for the USBJTAG-1 interface has been created, giving download speeds to target memory of up to 25 kB/s.

8.1 Functional Verification

The software has been successfully tested on a Cogent CSB337 single board computer (SBC) that contains an ARM920t based Atmel AT91RM9200 microcontroller, SDRAM and 8 MB Intel StrataFlash memory (28F640J3, 64 Mbit). The debugger was connected to the target through an Amontec Chameleon POD in its Wiggler configuration using the `parport` driver, as well as through an USBJTAG-1 interface using the `ftdi2232c` and the `ftd2xx` driver. The u-boot bootloader running on the core has been debugged using both the telnet and the gdb interface. An up-to-date Linux kernel, 2.6.12-rc1, has been used to verify the MMU and cache support. Breakpoints have been set on code executed in a loop, ensuring that the code was contained in the instruction cache, to test the handling of cache coherency. Random binary data was written to the flash, read back to the debugger, and compared, showing no differences.

The flash blocks used for testing have been successfully erased.
To test the ARM7 support, a Kurz FutureUnit SBC that contains an ARM720t based Hynix HMS30C7202 microcontroller, SDRAM and 16 MB Intel StrataFlash Memory (28F128J3, 128Mbit). Again, both a Wiggler-compatible interface and the USBJTAG-1 interface have been used to connect to the target. The LDBoot bootloader running on the FutureUnit works with an enabled MMU, so this has been used to test both the basic functionality as well as the MMU and cache handling.

The tests carried out so far were not formally specified, and don't cover every possible aspect, thus there's no guarantee that the code is actually correct in every way. Yet, there were no crashes of the running target with the latest version of the code, and a stopped Linux kernel or bootloader could always be resumed without affecting its operation.

Debug Entry

Because the ARM technical reference manuals are vague on the topic of debug entry, test code has been written to ensure correctness of the program counter (PC) that is reported to the user. The code listing below has been assembled, converted into a flat binary, and was transfered to the CSB337 at address 0x20000100 (start of SDRAM + 0x100) and the Kurz FutureUnit at address 0x7f000100 (start of internal SRAM + 0x100). The core was resumed at the beginning of the code, and the behavior on a debug request (`halt` command), a breakpoint, a watchpoint, and a watchpoint followed by a breakpointed instruction, has been tested.

```
0x20000100  mov r0, #0x20000000
0x20000104  mov r1, 1
0x20000108  str r1, [r0]
0x2000010c  mov r1, 2
0x20000110  str r1, [r0]
0x20000114  mov r1, 3
0x20000118  str r1, [r0]
0x2000011c  mov r1, 4
0x20000120  str r1, [r0]
0x20000124  b 0x20000100
```

This code loads register r0 with the address of a free memory location (start of SDRAM or SRAM), loads register r1 with four different values, and stores the content of r1 to the memory pointed at by r0. When the target is stopped, the instruction that has been executed last may be determined by looking at the PC, r1 and the current value at 0x20000000. If r1 has been changed, but the memory still contains the previous value, the `mov` instruction was executed, but the `str` not yet. If the content of r1 and the memory match, the `str` has been executed, but the following instruction not yet. The only ambiguity is at the end, where it's not completely clear if the branch has already been executed. This may be solved by looking at the PC: if it's beyond 0x20000124, the branch definitely hasn't been executed, as entry into debug state never added more than 0x10 to the address of the instruction that has to be executed next.

Table 8.1 shows the test scenarios (target system and reason for debug entry), the value that was stored by the *store multiple instruction* `STM` used to save core registers on debug entry, a corrected value, by subtracting 0xc that are added by `STM` instructions that store the PC, and the actual address of the instruction that would have to be executed next, derived from the current values of the PC, r1, and the

Table 8.1: Debug entry test results

System/Entry reason	Stored by STM	Corrected (-0xc)	Next instruction
ARM920t/debug request	0x2000013c	0x20000130	0x20000124
ARM920t/breakpoint	0x20000124	0x20000118	0x2000010c
ARM920t/watchpoint	0x20000118	0x2000010c	0x20000100
ARM920t/wpt. and bkpt.	0x20000124	0x20000118	0x2000010c
ARM720t/debug request	0x7f000128	0x7f00011c	0x7f000114
ARM720t/breakpoint	0x7f000134	0x7f000128	0x7f00011c
ARM720t/watchpoint	0x7f000140	0x7f000134	0x7f000128
ARM720t/wpt. and bkpt.	0x7f000124	0x20000118	0x2000010c

memory. The PC of ARM920t targets (and therefor all ARM9 targets, as these are compatible in that aspect) always contains the address of the instruction that has to be executed next plus three addresses (0xc bytes). On ARM720t targets, the PC contains the address of the instruction that has to be executed next plus two addresses (0x8 bytes) if it entered debug state because of a debug request, and plus three addresses (0xc bytes) if the reason for debug entry was a breakpoint, a watchpoint, or a watchpoint followed by a breakpoint. Single-stepping is similar to a breakpoint, so the values for breakpoints apply to single-step debug entry, too.

8.2 Performance Verification

The performance was measured by uploading a 128 kB file from the host PC to an ARM920t target's RAM using the `load_binary` command. Time was measured using a stopwatch, which is accurate enough, considering that the upload took between 5 seconds using the `ftd2xx` driver, and 25 seconds using the `parport` driver.
Memory is written in chunks of 560 bytes. At the beginning of a chunk, the base address is loaded, requiring 4 instructions to be inserted into the ARM9TDMI pipeline.
To load the data into target registers, a `LDMIA` instruction and two `NOPs`, that move the `LDMIA` to the Memory stage, are put into the pipeline. Loading the registers requires 14 instructions. A `STMIA` followed by a `NOP` with SYSSPEED high is used to execute the system-speed access. The RESTART instruction is selected, the JTAG state machine is moved to TAP_RTI, and the Embedded-ICE status register is read. This is repeated ten times to write a complete chunk (560 byte / (14 registers * 4 byte/register))

In order to evaluate the performance of the JTAG interfaces, the number of low level operations had to be calculated. Two measurements have been chosen: The number of TCK cycles required, and the number of MPSSE command bytes sent to the FT2232C chip.

Table 8.2 shows the number of TCK cycles and FT2232C command bytes required to put an instruction into the ARM9TDMI pipeline and to read an Embedded-ICE register. The first three operations of `arm9tdmi_put_instruction()` are only necessary when the debug scan chain and INTEST have to be selected (loading the base, and every `LDMIA` but the first, the number in brackets shows the TCK cycles / command bytes for successive instructions).

Table 8.3 shows the number of TCK cycles and FT2232C command bytes necessary to write 56 bytes to target memory. Writing a complete 560 byte chunk requires 20536 TCK cycles and 5991 FT2232C

Table 8.2: TCK Cycles and FT2232 Command Bytes

arm9tdmi_put_instruction()			arm7_9_read_ice_reg()		
JTAG operation	TCK	Cmd bytes	JTAG operation	TCK	Cmd bytes
IR Scan (4bit)	17	9	IR Scan (4bit)	17	9
DR Scan (5bit)	18	9	DR Scan (5bit)	18	9
IR Scan (4bit)	17	9	IR Scan (4bit)	17	9
DR Scan (67bit)	80	20	DR Scan (38bit)	51	13
Runtest	14	6	DR Scan (38bit)	51	13
Total	146 (94)	53 (26)	Total	154	53

Table 8.3: Costs for writing 56 byte

Operation	TCK cycles	Cmd bytes
`LDR r0, [r15]`	146	53
`NOP`	94	26
`NOP`	94	26
`NOP (base address)`	94	26
Load base address	428	131
`LDMIA r0, {r1-r14}`	146	53
`NOP`	94	26
`NOP`	94	26
`14x NOP (data)`	14 x 94	14 x 26
`STMIA r0!, {r1-r14}`	94	26
`NOP`	94	26
Select RESTART	17	9
Move to Run-Test/Idle	7	3
Read Embedded-ICE status	154	53
Write 56 byte	2016	586

command bytes, and writing 128 kB takes about 4.8 million TCK cycles or 1.4 MB FT2232 command bytes.

Parport Driver

The `parport` driver works by toggling bits on the PC's parallel port using port `in` and `out` commands. The TCK signal has to be switched to 1 and back to 0 using 2 `outs` for one TCK cycle, and whenever it's necessary to read TDO, an additional `in` has to be executed. Writing 128 kB took about 25 seconds, which equals 192,000 TCK cycles per second (128 kB / (560 Byte / 20536 TCK cycles))).
According to [Rs00], one port access should take 1μs, resulting in about 500,000 TCK cycles per second. Because 192,000 TCK cycles per second is less than half the theoretical maximum, the reason for the limited performance has been further investigated. To verify the information from [Rs00], a small C/assembler program has been written (see Listing D, Appendix D) that reads and stores the value of the time stamp counter (TSC, a 64-bit counter that's increased on every CPU cycle), executes two `out`

instructions, and reads the time stamp counter again. The difference of the two TSC values gives the number of cycles elapsed. This is repeated 100.000 times and an average is calculated, to smooth out effects like the current system load and context switches. On a 750 MHz host CPU an average of 3000 cycles, or 4μs (3000 cycles / 750 cycles/μs), were required to execute two `out` instructions.
In addition to the two `out` instructions, one `in` instruction is required in about 1/4 of all TCK cycles to read captured scan chain data. The performance is therefor obviously limited by the time required to toggle parallel port bits.

FTD2xx Driver

The `ftd2xx` driver sends 586 bytes to the FTDI chip to write 56 byte, until it has to wait for the Embedded-ICE status register to be transfered back to the system, therefor its performance is limited by the time that passes between sending the command buffer and receiving the captured data. Writing 128 kB took about 5 seconds, equaling 25 kB/s, or 2.1 ms per 56 byte. USB 1.1 data is transfered in frames of 1 ms length, so 2.1 ms for sending the data and reading back the results is close to the theoretical maximum.

9 Further Development

Although all major goals for the software haven been accomplished, there's room for enhancements and improvements. The software has been released under the terms of the GNU General Public License to allow other developers to build upon the code that has been written so far. The project is being hosted at BerliOS (`http://developer.berlios.de/projects/openocd/`), a platform that aims to provide services for developers and users of open source software.

- Endianess. The software currently supports only little-endian targets on little-endian hosts. Several parts of the code have to be converted to support big-endian systems.
- Thumb support. If the target is in Thumb state during debug entry, the debugger is shut down. The code identifying Thumb state is already there, but to complete Thumb support, code that handles the switch to ARM state on debug entry and back to Thumb state before the core is resumed has to be added and tested.
- Faster flash writing. If target RAM would be used while writing flashes, performance could be increased by several orders of magnitude. The data would be transfered to RAM, together with a small code fragment that programs the flash using the data contained in RAM.
- Debug communications channel. The Embedded-ICE debug communications channel allows a debugger to talk to code running on the target. This could be used together with the GDB File-I/O remote protocol extension to give code running on the target access to resources of the host system, like disk and terminal I/O.
- Additional cores. Currently, only ARM7TDMI, ARM720t, and ARM920t cores are supported. Support for more cores should be added, while using and possibly generalizing the existing parts.
- Additional JTAG devices. To further increase performance, USB 2.0 Hi-Speed interfaces or devices with more sophisticated on-board logic would have to be used. If the JTAG queued command support could be integrated in a JTAG hardware interface, performance could be increased to the limit set by the maximum JTAG clock frequency.
- Additional user interfaces. Adding support for other remote protocols, like the ARM remote debug interface (RDI), could increase the potential user base.

A Utilized Free and Open Source Software

This diploma thesis has been created only using free and open source software. The following chapter is going to list the tools utilized and what they've been used for.

A.1 Development Platform

A Debian 3.1 (Sarge) GNU/Linux system has been used for both developing the software and typesetting this document. The system ran Linux kernels up to 2.6.12-mm1, compiled from the sources available at `http://www.kernel.org`. The K Desktop Environment (KDE) (`http://www.kde.org`), Release 3.4.0, served as the desktop environment. KDevelop 3.2.0 (`http://www.kdevelop.org`), the KDE Development Environment, an integrated development environment supporting a wide variety of programming languages, has been used to write the source code and manage the build process.

A.2 Typesetting

This entire document was typeset using teTeX (`http://www.tug.org/teTeX/`), a TEX distribution for Unix systems, consisting only of free software. Instead of one of the document classes that come with teTeX, the Memoir class by Peter Wilson has been used. Memoir is a flexible class for typesetting general fiction, non-fiction and mathematical works as books, reports, articles or manuscripts. Memoir is available from the Comprehensive TeX Archive Network (CTAN) at `http://www.ctan.org/tex-archive/macros/latex/contrib/memoir/`. The glossary package (`http://www.ctan.org/tex-archive/macros/latex/contrib/glossary/?action=/tex-archive/macros/latex/contrib/`) was used to create the glossary and to maintain abbreviations used in the text. All listings have been typeset with the help of the listings package (part of teTeX).

A.3 Figures

All figures have been created using Xfig, a drawing program for the X Window System. Xfig offers built-in support for integrating figures within TEX documents by exporting them to combined Postscript/LaTeX and PDF/LaTeX formats. This combines TeX's typesetting flexibility with the drawing capabilities offered by Xfig.

B Source Code

The printed edition of the diploma thesis includes a CD-ROM with the source code of the Open On-Chip Debugger and the Port I/O Measurement program. Also included on the CD-ROM is the PDF and TEX source of this document. The source code is available as `openocd-0.3.tar.gz`, and in unpacked form in the `openocd` directory. The Port I/O Measurement program can be found in the `port_io` directory. The thesis PDF and source code are located in the `thesis` directory.

The latest version of the source code can be found at `http://developer.berlios.de/projects/openocd/`, either as a source archive or via CVS:

```
cvs -d:pserver:anonymous@cvs.berlios.de:/cvsroot/openocd login

cvs -z3 -d:pserver:anonymous@cvs.berlios.de:/cvsroot/openocd co
```

C Program Usage

C.1 Command Line Options

Table C.1 lists the options that may be given on the command line. These options take precedence over the configuration file.

Table C.1: Command line options

Short opt	Long opt	Description
-h	–help	Print command line argument description, and exit.
-f <file>	–file	Use <file> as configuration file, instead of the default openocd.cfg.
-d [0-3]	–debug	Set debug level. If the argument is omitted set debug level to 3 (all messages).
-i <interface>	–interface	Use JTAG hardware <interface>.
-c <cable>	–pp_cable	Use <cable> specification for the parport interface.
-p <port>	–pp_port	Use parallel port <port>.
-t <mode>	–startup	Select debugger action on startup. One of reset_run, reset_halt, init_halt, or attach.

C.2 Configuration Commands

Table C.2: Configuration commands

Command	Description
telnet_port <port>	Sets the port the daemon uses to listen for incoming telnet connections. Default is 4444.
gdb_port <port>	Sets the port the daemon uses to listen for incoming gdb connections. Default is 3333.
debuglevel <level>	Selects the maximum level of log messages that should be printed. 0 shows only Error messages, 1 includes Warnings, 2 Informational messages, and three print everything including Debug messages.
interface <name>	Use JTAG interface driver <name>. Currently supported drivers are `parport`, `ftdi2232`, and `ftd2xx`.

jtag_speed <speed>	Sets the JTAG clock frequency. The <speed> argument's actual meaning depends on the interface. The parport driver currently ignores this setting. For ftdi2232c devices, the resulting JTAG frequency is 6MHz / (1 + speed).
parport_port <port>	Use parallel port <port>.
parport_cable <cable>	Use <cable> specification for the parport interface.
target <name>	Debug a target with a <name> core. Currently supported cores are `arm7tdmi`, `arm720t`, `arm9tdmi`, and `arm920t`.
startup <mode>	Select debugger action on startup. <mode> should be one of reset_run, reset_halt, init_halt, or attach.
endianess <little|big>	Selects the endianess of the target.
breakmode <hw|sw>	Enable support for software breakpoints, or restrict to hardware breakpoints.
flash <name>	Select a flash driver. Currently only Intel Strata Flash is supported, using the `intel28fxxxj3` driver.
flash_size <size>	Sets the size of one flash chip. The intel28fxxxj3 driver uses this to ensure that the configured chip matches a found chip.
flash_chipwidth <width>	Sets the width of a single flash chip. Intel 28fxxj3 chips can be used in x8 or in x16 configuration.
flash_buswidth <width>	Sets the width of the flash bus. A flash bank may consist of more than one chip, in which case the bus is n-times the size of a single chip.

C.3 Command Line Interface

Table C.3: CLI commands

Command	Description
help	Show available commands
quit, logout, exit	Disconnect from the telnet server (doesn't touch target).
history	Show a list of previously run commands.
debuglevel <level>	Selects the maximum of log messages that should be printed. 0 shows only Error messages, 1 includes Warnings, 2 Informational messages, and three print everything including Debug messages.
shutdown	Shut the debugger debug (resumes target).
sleep <msec>	Sleeps for the specified amount of milliseconds. Useful to wait in startup scripts, for example to give a PLL time to lock.
idcode	Read and print the JTAG idcode.
jtag_speed <speed>	Sets the JTAG clock frequency. The <speed> argument's actual meaning depends on the interface. The parport driver currently ignores this setting. For ftdi2232c devices, the resulting JTAG frequency is 6 MHz / (1 + speed).
halt	Request debug entry. Once the target enters debug state, status information will be printed.

resume [addr]	Resumes the target at [address], or at the current position, if the argument is omitted.
poll	Get information about the current target state.
step [addr]	Single-step the target at [address], or at the current position, if the argument is omitted.
md[whb] <addr> [n]	Display [n] memory (w)ords, (h)alf-words, or (b)ytes at <addr>.
mw[whb] <addr> <value>	Write (w)ord, (h)alf-word, or (b)yte <value> at <addr>.
reset [halt|init]	Reset the target. If halt is specified, the target enters debug state immediately after coming out of reset. If init is specified, the configuration file is used to initialize the target after it entered debug state out of reset.
load_binary <file> <addr>	Download binary <file> to target memory at <addr>.
dump_binary <file> <addr> <size>	Dump <size> bytes starting at <addr> to <file>.
flash_probe <base>	Probe for the flash specified in the configuration file at address <base>.
flash_info <base>	Print information about the flash located at <base>.
flash_erase <base> <first> <last>	Erase blocks <first> to <last> of the flash bank located at <base>.
flash_binary <base> <addr> <file>	Write binary <file> to the flash bank located at <base>, starting at <addr>.
cp15 <opcode> [value]	ARM720t: Read or write a CP15 register using <opcode>. If [value] is specified, the register is written, otherwise a read is executed.
cp15 <num> [value]	ARM920t: Read or write CP15 register <num> using a physical access. If [value] is specified, the register is written, otherwise a read is executed. See the ARM920t TRM for the list of accessible registers.
cp15i <opcode> [value]	ARM920t: Read or write a CP15 register using an interpreted access with <opcode>. If [value] is specified, the register is written, otherwise a read is executed. See the ARM920t TRM for the list of accessible registers.
cacheinfo	ARM920t: Print information about identified caches.
virt2phys <va>	Execute a simulated page table walk, and print information about the translation found for the virtual address <va>.
md[whb]_phys <addr> [n]	Display [n] memory (w)ords, (h)alf-words, or (b)ytes at physical address <addr>. The MMU is temporarily switched off.
mw[whb]_phys <addr> <value>	Write (w)ord, (h)alf-word, or (b)yte <value> at physical address <addr>. The MMU is temporarily switched off.
breakmode <hw|sw>	Enable support for software breakpoints, or restrict to hardware breakpoints.
reg [num|name] [value]	Display/modify core registers. If no argument is given, all registers of all modes are displayed. If a number or a name is specified, that register is displayed. If a register is specified, and a value is given as a second argument, that register is set to [value].

icereg [num] [value]	Display/modify Embedded-ICE registers. If no argument is given, all ICE registers are listed. If a number is given, that register is read, or written, if a [value] is specified.
bp [addr] [hw]	List breakpoints, or set a breakpoint at [addr]. If hw is specified, a hardware breakpoint is set, otherwise software breakpoints are used.
rbp [addr]	Remove breakpoint set at <addr>.
wp <addr> <mask> <r/w>	Set a read, write, or access watchpoint on <addr> with the specified <mask>.
rwp <addr>	Remove watchpoint at <addr>.

D Port I/O Measurement Listing

```
 1 #include <sys/io.h>
 2 #include <stdio.h>
 3
 4 int main()
 5 {
 6         int i;
 7         int sum = 0;
 8         unsigned int upper, lower, upper2, lower2;
 9
10         if (ioperm (0x378, 3, 1) != 0)
11                 return -1;
12
13         for (i=0; i<100000; i++)
14         {
15                 asm (
16                         "rdtsc;"
17                         "pushl %%edx;"
18                         "pushl %%eax;"
19                         "movl $0x378, %%edx;"
20                         "movl $0x0, %%eax;"
21                         "outb %%al,%%dx;"
22                         "outb %%al,%%dx;"
23                         "rdtsc;"
24                         "popl %%ebx;"
25                         "popl %%ecx;"
26                         : "=c" (upper),
27                           "=b" (lower),
28                           "=a" (lower2),
29                           "=d" (upper2)
30                     );
31                 sum += lower2 - lower;
32         }
33
34         printf("average: %f\n", sum / 100000.0);
35 }
```

E GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file

format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use

technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications",

   Preserve the Title of the section, and preserve in the section all
   the substance and tone of each of the contributor acknowledgements
   and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document,
   unaltered in their text and in their titles. Section numbers
   or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section
   may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements"
   or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or
appendices that qualify as Secondary Sections and contain no material
copied from the Document, you may at your option designate some or all
of these sections as invariant. To do this, add their titles to the
list of Invariant Sections in the Modified Version's license notice.
These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains
nothing but endorsements of your Modified Version by various
parties--for example, statements of peer review or that the text has
been approved by an organization as the authoritative definition of a
standard.

You may add a passage of up to five words as a Front-Cover Text, and a
passage of up to 25 words as a Back-Cover Text, to the end of the list
of Cover Texts in the Modified Version. Only one passage of
Front-Cover Text and one of Back-Cover Text may be added by (or
through arrangements made by) any one entity. If the Document already
includes a cover text for the same cover, previously added by you or
by arrangement made by the same entity you are acting on behalf of,
you may not add another; but you may replace the old one, on explicit
permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License
give permission to use their names for publicity for or to assert or
imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this
License, under the terms defined in section 4 above for modified
versions, provided that you include in the combination all of the
Invariant Sections of all of the original documents, unmodified, and

list them all as Invariant Sections of your combined work in its
license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and
multiple identical Invariant Sections may be replaced with a single
copy. If there are multiple Invariant Sections with the same name but
different contents, make the title of each such section unique by
adding at the end of it, in parentheses, the name of the original
author or publisher of that section if known, or else a unique number.
Make the same adjustment to the section titles in the list of
Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History"
in the various original documents, forming one section Entitled
"History"; likewise combine any sections Entitled "Acknowledgements",
and any sections Entitled "Dedications". You must delete all sections
Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents
released under this License, and replace the individual copies of this
License in the various documents with a single copy that is included in
the collection, provided that you follow the rules of this License for
verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute
it individually under this License, provided you insert a copy of this
License into the extracted document, and follow this License in all
other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate
and independent documents or works, in or on a volume of a storage or
distribution medium, is called an "aggregate" if the copyright
resulting from the compilation is not used to limit the legal rights
of the compilation's users beyond what the individual works permit.
When the Document is included in an aggregate, this License does not
apply to the other works in the aggregate which are not themselves
derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these
copies of the Document, then if the Document is less than one half of

the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number.
If the Document specifies that a particular numbered version of this
License "or any later version" applies to it, you have the option of
following the terms and conditions either of that specified version or
of any later version that has been published (not as a draft) by the
Free Software Foundation. If the Document does not specify a version
number of this License, you may choose any version ever published (not
as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of
the License in the document and put the following copyright and
license notices just after the title page:

    Copyright (c) YEAR YOUR NAME.
    Permission is granted to copy, distribute and/or modify this document
    under the terms of the GNU Free Documentation License, Version 1.2
    or any later version published by the Free Software Foundation;
    with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
    A copy of the license is included in the section entitled "GNU
    Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts,
replace the "with...Texts." line with this:

    with the Invariant Sections being LIST THEIR TITLES, with the
    Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other
combination of the three, merge those two alternatives to suit the
situation.

If your document contains nontrivial examples of program code, we
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License,
to permit their use in free software.

Glossary

(modified) Harvard architecture	The Harvard architecture describes a memory system where instructions and data live in separate address spaces. Transfers from data memory to instruction memory are usually not possible, or only using special transfer functions. The modified Harvard architecture relaxes this restriction, and describes a system where two separate memory systems for data and instructions are connected to a unified memory containing data and instructions in a single address space.
BCLK	Bus clock. The clock by which an ARM9TDMI(-EJS) core is driven while it's running in FastBus mode. In synchronous or asynchronous mode, the core is clocked from BCLK during memory accesses that can't be satisfied by a cache or the write buffer.
Complex Programmable Logic Device	Logic device performing programmable functions in a circuit design. Macrocells containing the logic functions are interconnected using a central switching matrix (Global Routing Pool).
DCLK	Debug clock. The clock by which an ARM7/ARM9 core is driven while in halt-mode debug state.
Embedded System	A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. Contrast with general-purpose computer.

Embedded-ICE	On-chip debug circuit that is part of all current ARM7 and ARM9 family cores. It provides on-chip hardware debug capabilities through a JTAG compatible Test Access Port.
Enhanced Parallel Port	A parallel port standard available on many modern PC systems. Supports bi-directional communication and allows the use of DMA to accelerate communications.
FCLK	Fast clock. The clock by which an ARM9TDMI(-EJS) core is driven while it's running in synchronous or asynchronous mode and no memory access requiring a synchronization to BCLK is necessary.
Flash memory	A type of non-volatile memory segmented into blocks that can be individually erased and reprogrammed.
In-Circuit Emulator	Debug hardware that connects to a target system instead of the original microcontroller.
Jazelle	The Jazelle Java acceleration technology speeds up processing of Java bytecode by executing most Java instructions directly in hardware, without Emulation using a virtual machine.
JTAG	Joint Test Access Group, but commonly used to describe the IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE 1149.1 [IEEE1149].
MCLK	Memory clock. The clock by which an ARM7TDMI(-S) core is driven while it's running. It is the same clock used to access the memory system.
MMU	Memory management unit. Part of a processor that translates virtual address into physical address using page tables stored in system memory. Consits of the page table walking hardware and possibly a translation lookaside buffer (TLB).
nTRST	Test Reset is an optional asynchronous test logic reset signal on a JTAG compatible Test Access Port.

Random Access Memory	Memory which can be read and written without restrictions on the number of read and write operations or the order of successive operations.
Read Only Memory	Memory with fixed content, which can be read but not written.
Remote Debugging Interface	Proprietary protocol from ARM used for remote debugging
SBC	Single board computer. A complete computer system implemented on a single printed circuit board, consisting of a microprocessor together with memory, storage, communication interfaces and other peripherals.
Semihosting	Semihosting allows the target being debugged to access resources on the host PC, like disk and terminal I/O. The target code has to be linked against special libraries that use the semihosting facilities instead of normal target systems. The data is transfered through a communication channel like the Embedded-ICE debug comms channel available on ARM7 and ARM9 targets.
Software Interrupt	A software-generated interrupt, often used to call system functions from user space.
System On Chip	A highly integrated chip, containing a microcontroller together with peripherals like memory interfaces, Ethernet controllers, LCD controllers.
TCK	The Test Clock is a serial clock used to transfer data independent of component specific clocks on a JTAG compatible Test Access Port.
TDI	Test Data Input transmits serial data shifted from a TAP bus master on a JTAG compatible Test Access Port to connected components.
TDO	Test Data Output transmits serial data shifted from a connected component to the Test Access Port bus master on a JTAG compatible TAP.
Test Access Port	A general-purpose port that can give access to many test support functions built into a component [IEEE1149, p. 17]. Defined by the IEEE standard 1149.1. Proposed by the Joint Test Access Group as a way to test component functionality, component interconnections, and component interaction.

The GNU Project Debugger	A portable debugger available on many UNIX systems. Supported programming languages include C, C++, Pascal and Objective-C. Useful for debugging local applications as well as remote debugging.
Thumb	Thumb is a compressed 16-bit instruction set extension available on all current ARM7 and ARM9 family cores. It works with the full 32-bit length of ARM registers, but limits access to eight general purpose registers. The remaining registers may be accessed using special transfer instructions, but not with general data processing instructions.
TLB	Translation lookaside buffer. Stores recently used or explicitly stored translations between virtual addresses and physical addresses.
TMS	Test Mode Select determines the transition inside the JTAG state machine on the next rising edge of TCK.
Transistor-Transistor Logic	A circuit technology where the output is derived from two transistor. 0V indicate a logic zero, 5V a logic one.
Universal Serial Bus	A serial bus interface used to connect peripheral devices to a computer system. Widely adopted standard, increasingly common in computer-related areas such as digital home-entertainment
von-Neumann architecture	A processor design for universal computers that operates on a unified memory, where data and instructions live in the same address space.

Bibliography

[AN2232C-01] Future Technology Device International Ltd. *Application Note AN2232C-01, MPSSE* 2004 Available from http://www.ftdichip.com/Documents/AppNotes/AN2232C-01_MPSSE_Cmnd_11.pdf

[ARMFAQ2] ARM Ltd. *How can I access the 720T CP15 registers by JTAG debug sequences?* Available from http://www.arm.com/support/faqip/3700.html

[Asb01] Arnold S. Berger *Embedded Systems Design: An Introduction to Processes, Tools and Techniques* CMP Books 2001

[DDI0029G] ARM Ltd. *ARM7TDMI (Rev 3) Technical Reference Manual* 2001 Available from http://www.arm.com/pdfs/DDI0029G_7TDMI_R3_trm.pdf and on the ARM Technical Publications CD

[DDI0100E] ARM Ltd., David Seal *ARM Architecture Reference Manual* Addison-Wesley Professional 2000 2nd edition

[DDI0151C] ARM Ltd. *ARM920t (Rev 1) Technical Reference Manual* 2000 Available from http://www.arm.com/pdfs/DDI0151C_920T_TRM.pdf and on the ARM Technical Publications CD

[DDI0180A] ARM Ltd. *ARM9TDMI (Rev 3) Technical Reference Manual* 2000 Available from http://www.arm.com/pdfs/DDI0180A.zip and on the ARM Technical Publications CD

[DDI0192A] ARM Ltd. *ARM720T (Rev 3) Technical Reference Manual* 2000 Available from http://www.arm.com/pdfs/DDI0192A_720T_R3.pdf and on the ARM Technical Publications CD

[DDI0240A] ARM Ltd. *ARM9E-S (Rev 2) Technical Reference Manual* 2002 Available from http://www.arm.com/pdfs/DDI0240A_9ES_R2.pdf, and on the ARM Technical Publications CD (as DDI0240B)

[DS2232C] Future Technology Device International Ltd. *FT2232C Dual USB UART/FIFO IC Data Sheet Version 1.5* 2005 Available from http://www.ftdichip.com/Documents/DataSheets/ds2232c_15.pdf

[ELMS05] linuxdevices.com *Embedded Linux market snapshot* 2005 Available from http://linuxdevices.com/articles/AT4036830962.html

[GDB01] GDB developers *GDB Remote Serial Protocol* Available from `http://sources.redhat.com/gdb/current/onlinedocs/gdb_33.html`

[I290667] Intel corporation *Intel StrataFlash Memory (J3)* 2005 Available from `http://http://www.intel.com/design/flcomp/datashts/290667.htm`

[IEEE1149] IEEE *IEEE Standard 1149.1-2001 Test Access Port and Boundary-Scan Architecture* Available as Print (ISBN 0-7381-2944-5) or PDF (ISBN 0-7381-2945-3) edition, 2001

[IHI0014J] ARM Ltd. *Embedded Trace Macrocell Architecture Specification* Available from `http://www.arm.com/pdfs/IHI0014J_ETM_ArchSpec.pdf`, and on the ARM Technical Publications CD

[RFC854] J. Postel, J. Reynolds *RFC 854 - Telnet Protocol Specification* 1983 Available from `http://www.faqs.org/rfcs/rfc854.html`

[Rs00] Riku Saikkonen *Linux I/O port programming mini-HOWTO* 2000 Available from `http://www.faqs.org/docs/Linux-mini/IO-Port-Programming.html`

[Sf00] Steve Furber *ARM system-on-chip architecture* Addison-Wesley Professional 2000 2nd edition

Index