Open On-Chip Debugger
– OpenOCD –

Hubert Högl, Dominic Rath
Hubert.Hoegl@fh-augsburg.de, Dominic.Rath@gmx.de
Fachhochschule Augsburg
Fakultät für Informatik

December 15, 2006

Abstract

This paper describes OpenOCD, a free open-source JTAG debugger for microprocessors with ARM7, ARM9 and related cores. OpenOCD is an ideal complement for the GNU GCC toolchain for ARM processors. It can be controlled from the GNU debugger GDB and from a Telnet commandline interface. OpenOCD is capable of driving a variety of open JTAG hardware interfaces and can easily be adapted to new interfaces. It is licensed under the General Public License (GPL).

1 Introduction

Due to free and open-source software it has never been easier to get high-quality cross development software for a broad range of microcontrollers. This application field is mainly dominated by the GNU Compiler Collection GCC [4], consisting of a C/C++ compiler, assembler, linker and utilities. Considering especially the ARM architecture [5], a well-established "arm-gcc" port is available. When it comes to debugging the situation gets worse. There is GDB, the GNU Debugger [7]. For most architectures GDB comes with a simulator, so that simulating programs written with GCC is in general not a problem. The difficulties start when programs shall run on the target processor under control of GDB when restricting oneself to only free software.

Many modern controllers like those from ARM have "Embedded ICE" (EICE) facilities on chip to get full control within a debugger over the target program. The EICE is driven by a "JTAG" port [6], which is a four-wire synchronous serial port accessible at some package pins. The JTAG port is often called "TAP" (Test Access Port).

In contrast to the target hardware a plain GDB has no concepts of EICE, JTAG and similar details at the hardware level. GDBs concept of debugging is expressed in a *Remote Serial Protocol* (RSP) which is a simple ASCII high-level protocol to control the debug process. For example for a memory modify operation GDB will emit the string `M4015cc,2:c320#6d` (address `0x4015cc`, size 2 bytes, data `0xc320`; the value `6d` after `#` is a checksum). The RSP specification is contained in the GDB documentation. The necessary piece of software between toolchain and target is in principle a protocol converter from RSP to JTAG bitstream commands (fig. 1).

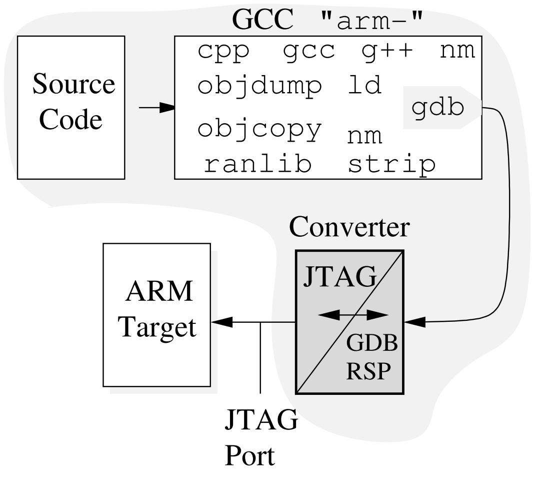

Figure 1: The GNU Toolchain, enhanced with the GPL'ed OpenOCD converter between GDB/RSP and JTAG bitstream commands. Everything on the gray background runs on the development (host) computer.

Back in 2004 there were some zero-cost debug solutions for ARM consisting of JTAG control software and JTAG adaptor. However all of these solutions seem to have one or more of the following deficiencies: not open-source / distribution only as binary, fixed to MS-Windows platform, poor Linux support, unstable, support only for a fixed JTAG adaptor, lack of support for the most recent controllers, support only for a specific (outdated) GDB. However, beside the free solutions there exist a number of commercial products which work very well.

The OpenOCD project [1] started in 2004 to fill this gap with an open-source software agent sitting between a GDB compiled for ARM and a JTAG adaptor. It began as a diploma thesis with the goal in mind to write an open-source JTAG-debugger restricted to ARM7 controllers which can easily be interfaced to different JTAG adaptors. Until now the feature set and maturity level has very much improved to nearly that of commercial debuggers.

In more detail, debugging target code in GDB via JTAG consists of the interfaces shown in fig. 2. OpenOCD opens two network connctions for GDB/RSP and Telnet. The debugger and the OpenOCD server can thus be run on different machines. New JTAG adaptors can be easily added to OpenOCD.

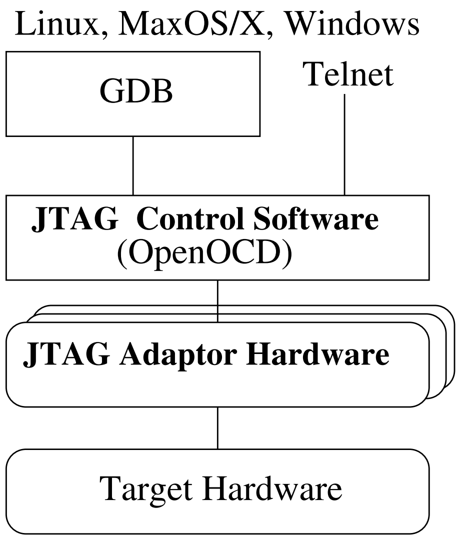

Figure 2: More detailed interfaces when debugging with GDB and OpenOCD.

2 Overview of ARM Debugging

The ARM core is the CPU of an ARM-based system-on-chip (SOC). ARM released a series of architectures of their cores, currently the most important are v4, v4T and v5T. The architecture corresponds with the instruction set. The cores can be grouped into core families, e.g. ARM7T, ARM9T and ARM9E. Besides the CPU there are a number of other macrocells e.g. for debugging (D), in-circuit emulation (I), memory management, caches, DSP extensions (E), 64-bit multiplication (M) and other purposes.

A *system core* is a CPU core plus a memory system unit. For example the ARM720T core as a member of the ARM7TDMI-S core family has a mixed 8kb instruction and data cache, a write buffer and a MMU. Another example is the ARM920T which is a member of the ARM9TDMI family with separate instruction and data chaches, each 16kb and a MMU. See the following table for a few architectures, core families, system cores, and concrete chips.

Arch.	Core Family	System Core	Chip
v4	StrongARM		
v4T	ARM7TDMI		(a)
		ARM720T	(h)
	ARM7TDMI-S		(d)
		ARM720T	
	ARM9TDMI	ARM920T	(b)
		ARM922T	(c)
		ARM940T	
v5TE	ARM9E	ARM946E-S	TBD
		ARM966E	(e)
	ARM9EJ	ARM926EJ-S	TBD
	XScale	PXA2xx	(f)
		IXP4xx	(f)
		IOP3xx	(f)
v7m	Cortex	Cortex M3	(g)

Currently OpenOCD successfully has been used with Atmel AT91R40008 (a), AT91SAM7 (a), AT91RM9200 (b), Analog Devices ADuC70xx (a), Cirrus EP93xx (b), Hynix/Hyundai HMS30C7202 (h), Intel XScale (f), Luminarymicro LM3S811 (g), Micrel KS8695PX (c), NetSilicon NET+50 (a), NS7520 (a), NXP (Philips) LPC2xxx (d), Samsung S3C44B0 (a), S3C2410 (b), Sharp LH7A404 (c), ST Microelectronics STR7x (a), STR9x (e), TI MSP470 (a). Probably many more devices including those from other manufacturers will immediately work. Devices with ARM926E system core like the Atmel AT91RM9261, Samsung S3C2412, Philips LPC3xxx and Hilscher NETX 500 will soon be supported.

ARM7 cores have a 3-stage (fetch – decode – execute) and ARM9 cores have a 5-stage (fetch – decode – execute – memory – writeback) pipeline. The ARM7 has a von-Neumann architecture. The ARM9 has a Harvard architecture, i.e. it internally uses two separate busses for instructions and data. Note that this text ignores most of the differences in debugging the core families ARM7 and ARM9.

Like most other modern microcontrollers the ARM system-on-chip uses JTAG as the main interface for controlling the debugging process. JTAG was originally invented for board level tests (*boundary scan*) and is now used for additional tasks like CPLD programming, FPGA configuration (in general *in-system programming* features) and core debugging. JTAG ist described in the IEEE standard 1149.1 [6]. Chip manufacturers add chip-specific functionality e.g. for debugging. Section 3.2 gives an overview of the JTAG port.

The basic idea behind JTAG is a state machine which is navigated by the TMS (Test Mode Select) signal, see fig. 3.

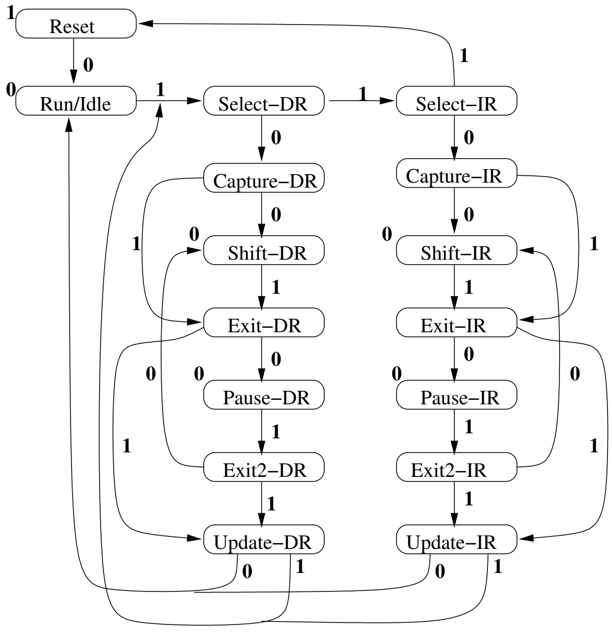

Figure 3: The JTAG state machine. The bold bits at the arcs indicate the TMS signal. A TMS bit at the upper left corner indicates a state transition to itself.

The above state diagram is used to transfer instructions and data into and out of the JTAG registers of the ARM system core. A quick overview of the JTAG registers is given in fig. 4. The mandatory JTAG registers are marked by a gray background, the others are extensions of the ARM architecture.

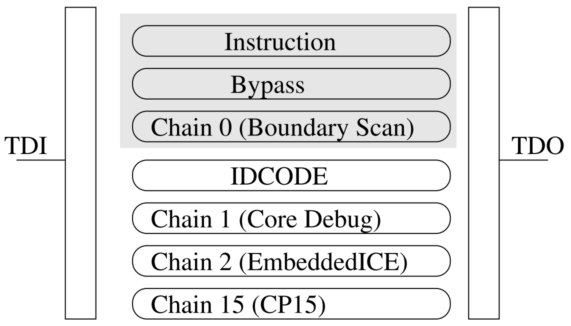

Figure 4: ARM JTAG scan chains.

The basic instructions which can be scanned into the JTAG instruction register are SCAN_N, INTEST, IDCODE, BYPASS and RESET. SCAN_N selects one of the chains **0** (Boundary Scan), **1** (Core Debug), **2** (EmbeddedICE), and **15** (CP15).

Debugging is fully controlled over the JTAG port. The core can be forced by JTAG operation (beside other methods, see below) into a *debug state* in which it is halted so that the full execution state can be examined and modified. Scan chain 1 is connected to the system bus to scan instructions and data into the core pipeline and read resulting data. While in debug state the main clock is replaced by a *debug clock* which is generated by a specific state transition in the JTAG state machine (see fig. 3). However there is an option to execute single instructions at system speed. After the debugging action has completed the core can be *restarted* for normal operation.

The crucial points when "manually" feeding code into the pipeline are to satisfy the dependencies for pipeline processing and to exactly save the previous execution state. This is even more important when cache maintenance and memory management is involved for more feature-rich system cores.

For doing in-circuit debugging another building block is necessary – the *EmbeddedICE* (EICE) macrocell, shown in fig. 5. EICE implements typical debugger functionality, most importantly *breakpoints*, *watchpoints* and *single stepping*. The EICE is controlled by 16 registers which are accessed by scan chain 2.

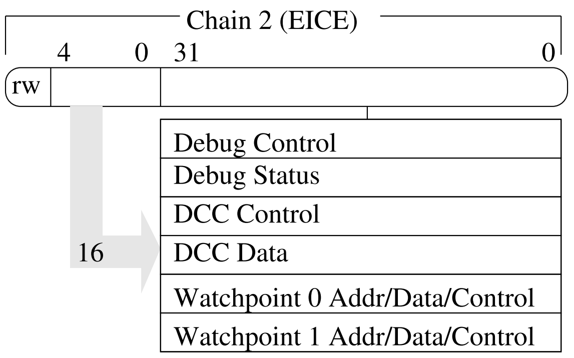

Figure 5: EmbeddedICE macrocell

The debug control register is responsible to enter debug state. Also both watchpoint units can be the source of events to enter debug state. Each of the watchpoints 0 and 1 consists of six registers (address value and mask, data value and mask, control value and mask). Due to their general design these registers can hold a breakpoint (stop at instruction), a watchpoint (stop at data access) or can be ignored. These are *hard* break- or watchpoints. Contrast this with *soft* breakpoints which are set by replacing an instruction in memory by a specific instruction which forces the core into debug state. It is clear that soft breakpoints only work for writeable memory (RAM). Programs in ROM can only use hard breakpoints. Single stepping is achieved on some cores by adequately configuring both watchpoint registers. Others, e.g. the ARM920T have a single step bit in the debug control register.

When in debug state the debugger can examine the system state, i.e. the core registers, by scanning a “store multiple” instruction (STM) into scan chain 1, followed by two NOP instructions. At the 4th debug clock cycle the given registers are pushed out of the pipeline so that their value can be read with chain 1. For the status registers CPSR and SPSR there are other instructions to be pushed into the pipeline.

To run certain instructions at system speed about the following has to be done (this is different for ARM7 and ARM9): shift the “fast” instruction with other necessary instructions into the pipeline and mark a specific bit in scan chain 1 (BREAKPT or SYSSPEED). Then load RESTART into the JTAG TAP controller and enter the Run/Idle state. After that the core resynchronizes to the main clock and executes the fast instruction, followed by re-entering the debug state. A typical example for instructions which only run at system speed are memory accesses, e.g. LDM (load multiple) and STM (store multiple).

When control is passed back from the debugger to the target application a few steps have to be taken. First, the execution context has to be restored. The context comprises CPSR, all core registers and the program counter. They are restored with MSR, LDM and LDR with PC as destination respectively. Next a branch instruction with the desired continuation address must be clocked into the pipeline. Finally a RESTART instruction must be shifted into the TAP controller to run the branch at system speed (as described in the previous paragraph).

Until now we only talked about *halt-mode debugging*, which means the debugger can interact with the target only in halted mode. By way of a facility in the EICE called *Debug Communication Channel* (DCC) the debugger and the target can communicate even when the target is in the running state. In fig. 5 the two 32-bit DCC registers for data and control can be seen. The DCC control register contains bits to synchronize the asynchronous processor and the debugger. From the target the DCC registers can be accessed by coprocessor 14 transfer functions MRC (ARM register to coprocessor) and MCR (coprocessor to ARM register). By using the DCC certain debugger operations like downloading larger amounts of data is about a factor of 3 to 4 faster than in debug mode. When using DCC communication it is in general also advantageous to configure one or more *working areas* in RAM. This is an OpenOCD concept to reserve regions in target RAM to speedup certain debugger commands. It is selected by an entry in the configuration file (see section 3.4).

3 Architecture and Implementation

OpenOCD is a server program with the following tasks:

- It offers a GDB remote serial protocol (RSP) network socket on default port 3333.
- It offers a Telnet network socket on default port 4444.
- It reads a configuration file with target specific configuration commands.
- It accesses the target processor over one of several JTAG hardware interfaces.
- It writes log data.

Fig. 6 shows the main components of the OpenOCD environment.

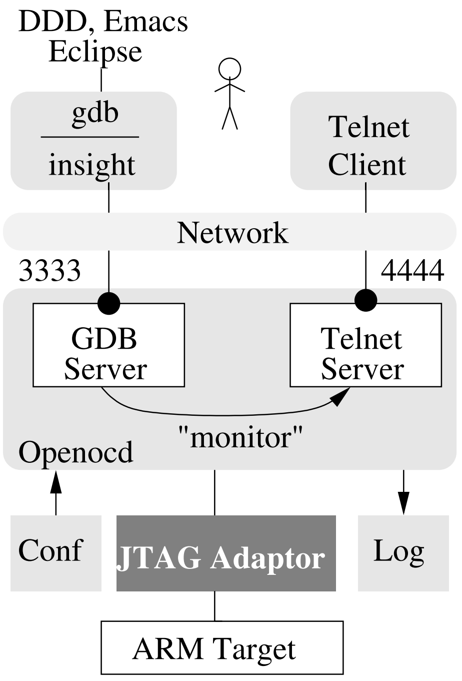

Figure 6: The OpenOCD environment.

A typical invocation specifies a debug level and a configuration file, e.g. `openocd -d2 -f mytarget.cfg`. The configuration file contains commands which adjust OpenOCD for a specific target (see section 3.4).

During startup and operation the server will write log data to standard output. The log levels ERROR (0), WARNING (1), INFO (2) and DEBUG (3) are defined.

After the server is running, the user may connect to the GDB part of the server with `arm-gdb` or with `arm-insight` ([20]). Now the user is able to interact with the target using all the debugger commands described in the GDB documentation [7]. The alternative OpenOCD Telnet interface is a commandline interface for a large number of commands which are not available within the debugger except for a few which are available both in the GDB interface and in the Telnet interface. Currently the following functionality needs extra user commands: Server handling (shutdown, exit), Target management (targets, reg, poll, wait_halt, halt, resume, reset, soft_reset_halt), JTAG (jtag_speed, scan_chain, endstate, jtag_reset, runtest, statemov, irscan, drscan) and Flash memory management (banks, info, probe, erase, write, protect and others).

The GDB "monitor" command forwards commands entered on the GDB command interface to the Telnet server. GDB ignores these commands.

Implementation

The trunk revision of OpenOCD currently (rev 121) comprises 23,220 Ansi-C sourcelines, counted according to the "SLOCCount" method [25]. It compiles on Linux, Embedded Linux, MacOS/X and also natively on Windows.

The source code is divided into several modules as shown in fig. 7. An arrow from module A to module B indicates that module B calls functions from module A.

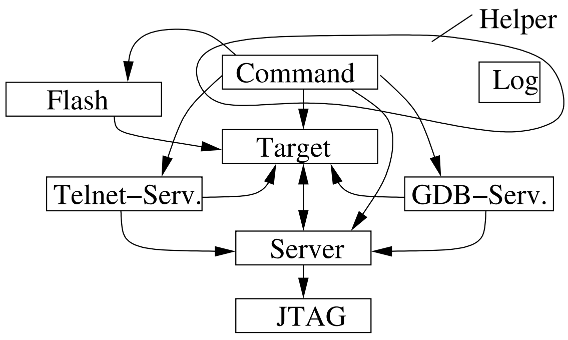

Figure 7: OpenOCD modules.

The `main()` function is located in file `trunk/src/openocd.c`. It initializes all other modules, starting with the Server. The server is divided in the part handling Telnet and in the part handling GDB RSP commands. A central position has the Command module because it allows to register commands offered by nearly all other modules. About 100 user- and configuration commands exist (see [3]).

A few minor helper modules are the interpreter (define variables and run scripts), the log module (used by nearly all other modules) and the commandline parser.

3.1 Target Module

The Target module in directory `target/` is probably the most complex of all. Each supported core family has its own source file, currently this is `arm7tdmi.c`, `arm720t.c`, `arm9tdmi.c`, `arm920t.c` and `arm966e.c`. Support for ARM926EJ-S and ARM946E-S is currently being implemented. Two additional OpenOCD branches provide support for Cortex-M3 (`cortex_m3.c`) and Xscale (`xscale.c`).

These files are complemented by code to handle the common operations defined in the ARM v4/v5 architecture and in the ARM7/ARM9 core family. The corresponding files are `armv4_5.c` and `arm7_9_common.c`. For systems with MMU and cache additional code is in armv4_5_cache.c and armv4_5_mmu.c.

An interface to access the EICE registers is defined in embeddedice.c. Normal core registers are accessed through register.c. Breakpoints and watchpoints are managed in a linked list in file breakpoint.c. Support functions to construct algorithms in target RAM used for Flash memory programming are located in algorithm.c.

A disassembler supporting ARM v4/v5 is provided to reconstruct the assembler code from ARM machine code (arm_disassembler.c).

The features of the JTAG interface for ARM system-on-chip are implemented in arm_jtag.c. It builds on the generic JTAG interface in directory jtag/.

3.2 JTAG Module

The OpenOCD software drives the target processors' JTAG port by a JTAG interface (or "adaptor"). Fig. 8 shows the main JTAG signals which must be provided by a JTAG adaptor. The main signals for data exchange are TDI (Test Data In) and TDO (Test Data Out). Data is shifted synchronously to TCK (Test ClocK), similar to the SPI standard. The data shifted into TMS leads to state transitions in the JTAG state machine shown in 3.

The maximum JTAG clock should be substantially lower than the main clock of the processor. TCK frequencies from 500 KHz to 25 MHz are typical.

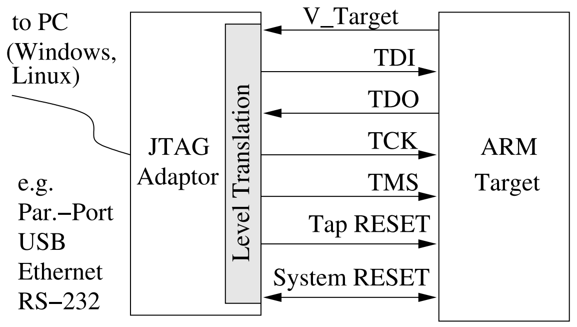

Figure 8: The most important JTAG signals.

At the rising edge of TCK the TMS, TDI and TDO signals are sampled. At the falling TCK edge new signals can be fed to TMS and TDI and a new output level appears on TDO.

In general two different reset signals for the TAP (TRST) and for the target system (SRST) are provided. The system reset signal can be read back to detect target resets. Correctly resetting the target and starting debugging from reset state is prohibited by some processors. In this case the processor must run for a while and then be reset (option run_and_halt). Debugging a target starting from the first instruction is only possible when the two reset signals are independently available and when the core has no protection for doing this.

For interfacing the PC to the external world a number of possibilities exist. The PC Parallel Port can easily be interfaced to external hardware. The *Wiggler* [8] is a very much used open-documented parallel port JTAG adaptor with a maximum JTAG clock speed of about 400 kHz (identical with the maximum port access speed). The *Chameleon POD* product by Amontec [10] is a universal parallel port adaptor which can be configured as *Wiggler* or as *JTAG Accelerator*, beside many other configurations. The Accelerator achieves a faster JTAG clock by shifting the parallel port data with hardware support. The Accelerator is also free and was suggested by the OpenOCD author and implemented by Amontec. One of the disadvantages of the parallel port is that it has been dropped in many new notebooks and desktop PCs.

Universal Serial Bus (USB) is todays technology to interface to peripherals according to the "plug-and-play" slogan. We early have built an open USB-to-JTAG interface [9] using the FTDIChip device FT2232C [14]. This USB device controller is currently the only one with a *Multi-Protocol Synchronous Serial Engine* (MPSSE) capable to directly drive a JTAG port. This feature makes building a JTAG interface with up to 6 MHz JTAG clock a trivial task – except for the optional level translation part of the schematic. Please note that new non-commercial JTAG interfaces should also be licensed under a free license, i.e. schematic and PCB should be available to the public.

A few commercially available interfaces have been derived from this USB device, e.g. [11], [12], [13] and [15].

The typical download speed to RAM ranges between 30 kbyte/sec and 120 kbyte/sec.

An elegant way to avoid a separate JTAG adaptor is to integrate the adaptor on the target board. Recently the Cortex-M3 evaluation kits by Luminarymicro [16] have taken this approach. By adding a FT2232L to the boards only a single USB connection is needed for power supply, JTAG debugging and serial port access.

There are also other approaches to build a JTAG interface. With an external CPLD or FPGA for the JTAG data ser-/deserialization nearly every microcontroller, e.g. a ARM7 or ARM9, can drive a JTAG port at a speed up to 25 MHz. Some of these controllers have even smart synchronous engines configurable to drive JTAG ports, e.g. the two Atmel ARM9 devices AT91RM9200 and AT91SAM9261. A typical network-enabled embedded system built with these controllers is even able to run the whole OpenOCD software, thus building an intelligent Ethernet enabled debugger box is a relatively straightforward task.

Note that we are restricted when commercial JTAG adaptors are to be used with OpenOCD. Since most commercial adaptors have a proprietary (closed) interface specification they can not be driven by open-source software.

The code in directory `jtag/` is organized as follows: The main JTAG support functions e.g. for state-machine navigation are contained in `jtag.c`. JTAG commands can be put into a linked listed. The selection of a specific JTAG interface is controlled during compile-time, not at runtime. Each hardware interface has its own file `ft2232.c`, `amt_jtagaccel.c` and `parport.c`. All interfaces which do "bit banging" on digital I/O lines, e.g. parport, use `bitbang.c`.

3.3 Flash Memory

Many system-on-chip devices contain some amount of Flash memory ranging in size from about 32K to 512K. Moreover Flash memory devices ranging from about 1M to 16M can be added to the external address- and databus of the microcontroller (see fig. 9). Flash programming is a task which is strongly related to the debugging process, therefore OpenOCD has support for it.

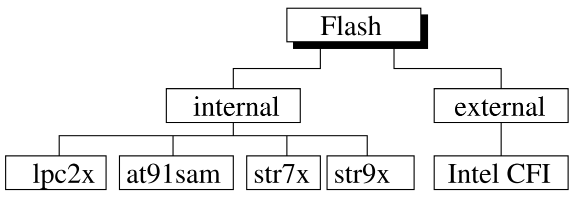

Figure 9: Flash memory support

Programming of the internal Flash memory can be implemented in different ways. The best write performance is usually achieved when Flash writing and verification is handled by short routines which OpenOCD transfers to *working areas*, which are reserved regions in target RAM. These routines write blocks of the Flash memory which have previously been transfered to other working areas. This kind of Flash programming is currently implemented for the targets LPC2xxx, STR7x/STR9x and external CFI devices. For example the Flash write and erase-check routines for CFI Flash devices are located in working areas on the target.

Another strategy for internal Flash programming is currently used for the AT91SAM7x devices. This device offers an Embedded Flash Controller (EFC) with a buffer for Flash memory pages. The EFC implements a set of commands to write, erase and status check of the internal Flash memory.

3.4 Configuration Files

The configuration file contains the following configuration groups:

- Server (daemon) configuration
- JTAG interface configuration
- JTAG scan chain configuration
- Target configuration
- Flash configuration

The target configuration comprises the selection of an ARM core family, endianness, reset mode, position in the JTAG chain and the variant of the ARM core family.

The following is a configuration file for the Philips LPC2294 from the ARM7TMDI-S core family. The server runs with default port numbers and speaks to a "JTAGkey" interface from Amontec. The JTAG speed is 2 MHz (6 MHz / (2+1)). The reset pins are configured with the hint that an active SRST also initiates an active TRST. The `jtag_device` command is for ARM chip identification. The target configuration says that the target will be reset on server startup. The ARM SOC is an ARM7TDMI-S (r4). After a hard reset of the target number 0 it will run for 30 msec, then the debug request will be triggered. For fast downloads a 16kb working area at the starting address of the internal SRAM will be used which is not backed up.

```
#daemon configuration
telnet_port 4444
gdb_port 3333

#interface
interface ft2232
ft2232_device_desc "Amontec JTAGkey A"
ft2232_layout jtagkey
ft2232_vid_pid 0x0403 0xcff8
```

```
jtag_speed 2

# reset
reset_config trst_and_srst \
   srst_pulls_trst

#jtag scan chain
jtag_device 4 0x1 0xf 0xe

#target configuration
daemon_startup reset
target arm7tdmi little \
   run_and_halt 0 arm7tdmi-s_r4
run_and_halt_time 0 30

working_area 0 0x40000000 0x4000 \
   nobackup

#flash configuration base, size, 0, 0
#  variant, target#, cclk(kHz), checksum
flash bank lpc2000 0x0 0x40000 0 0 \
   lpc2000_v1 0 14765 calc_checksum
```

A detailed description of all OpenOCD configuration file commands is far beyond this paper. A complete description of all configuration commands and other sample configuration files can be found at [1].

3.5 PLD Programming

Many Programmable Logic Devices (PLD) have a JTAG interface for programming. The OpenOCD software contains two modules that cope with PLDs. The PLD module can configure Xilinx Virtex2 FPGAs. The XSVF module is a player for Xilinx XSFV vector files.

4 Debugger Front Ends

4.1 The GNU Debugger

The plain GDB user interface is a text-only spartanic interface, which is nevertheless very functional (the authors of this paper like it!). After the `(gdb)` prompt the user may enter commands for the debugger. Note that your gdb executable may have a different filename. For example to connect to the OpenOCD server a `target remote` command specifying a computer and a portnumber must be entered:

```
hhoegl@egg:~$ arm-gdb
GNU gdb 6.4.50.20060226-cvs
...
(gdb) target remote localhost:3333
```

A slightly more comfortable text-based user interface can be invoked when starting gdb with the “text user interface” (tui):

```
hhoegl@egg:~$ arm-gdb --tui
```

4.2 Emacs

The Emacs Editor [17] can also control the GDB debugger. The Emacs command `M-x gdb` invokes the traditional GDB interface via `gdb-ui.el`, which is distributed with Emacs. An improved interface starts with `M-x gdbmi` which is based on `gdb-mi.el`, contained in the GDB distribution. At [18] is a small tutorial how to debug in Emacs.

4.3 DDD

The Data Display Debugger (DDD) [19] is a graphical front-end for GDB. It is invoked with the actual gdb debugger specified by the `--debugger` option:

```
ddd --debugger arm-gdb \
          --command=init.gdb
```

Some initial GDB commands can be written in the file given by the `--commmand` option. See [19] for more information about DDD.

4.4 Insight

The Insight debugger [20] is another attempt to realize a graphical user interface for GDB. It is different from DDD in that it also contains the full GDB source code. The user interface code is written in Tcl/Tk.

4.5 Eclipse

Eclipse [21] is a huge development environment which can also be used to write and debug C or C++ code. The C features are implemented in the C/C++ Development Tooling (CDT).

See the tutorial [22] about Eclipse and OpenOCD by James P. Lynch.

5 Development process

OpenOCD is an open-source project hosted on the open-source mediator Berlios (`http://www.berlios.de/`). The source code is managed with the Subversion (SVN) revision control system [24]. The SVN command to check out the current revision is

```
svn co http://svn.berlios.de/svnroot/
repos/openocd
```

After a local copy of the sources has been checked out the program can be compiled and installed. A short description about the installation process is in the text file `trunk/INSTALL`, also read `trunk/README`.

OpenOCD users which need some guidance are invited to visit the public forum at [2]. Be sure to first search through the archive to find answers to similar questions. If you would like to post a new question please add the OpenOCD version information (SVN revision) and add an excerpt of the log output to make it easier to locate possible bugs.

People interested in the development of OpenOCD should subscribe to the `openocd-development` mailing list at `http://developer.berlios.de/mail/?group_id=4148`

Patches should be submitted to `http://developer.berlios.de/patch/?group_id=4148`

6 Summary and Future Improvements

OpenOCD is probably the most stable and feature-rich open-source JTAG debugger for ARM controllers. The open sourcecode is an ideal basis for adding additional functionality like cores, flashes, boundary scan, programmable logic, and others. It is also a solid basis for doing experiments and research in the field of debug technology. The OpenOCD software is complemented by a number of open and also cheap closed/commercial JTAG adaptors. It is very easy to integrate new JTAG adaptors based on an arbitrary technology.

The author of the software, Dominic Rath, actively maintains the source code with the help of a few other contributors. To help the project you may give a donation to it as described at the homepage [1].

Room for future improvements

- High-speed JTAG interfaces based on Ethernet/Embedded Linux and high-speed USB
- Support for Coresight Serial Wire Debug
- Support for Boundary Scan testing
- Support for the Boundary Scan Description Language (BSDL)
- Support for IEEE 1532 in-system configuration
- More support for programmable logic
- Support for real-time trace data, collected by the Embedded Trace Macrocell (ETM)
- Get more test users
- Add targets from other manufacturers, e.g. PowerPC and MIPS.

7 Further Information

[1] Home of the OpenOCD
`http://openocd.berlios.de`

[2] OpenOCD Forum
`http://www.sparkfun.com/cgi-bin/phpbb/index.php`

[3] OpenOCD Quick Reference
`http://www.fh-augsburg.de/~hhoegl/proj/openocd/oocd-quickref.pdf`

[4] GNU Compiler Collection
`http://www.gnu.org/software/gcc`

[5] Advanced Risc Machines (ARM). Most of the ARM documentation can be downloaded here, e.g. the ARM Architecture Reference Manual and reference manuals for the core families.
`http://www.arm.com`.

[6] IEEE Standard 1149.1-2001 Test Access Port and Boundary-Scan Architecture. Available from
`http://ieeexplore.ieee.org`.

[7] Homepage of the GNU Debugger
`http://www.gnu.org/software/gdb/`

[8] The Wiggler JTAG adaptor
`http://andras.tantosonline.com/wiggler.htm`

[9] *USBJTAG* adaptor
`http://www.fh-augsburg.de/~hhoegl/proj/usbjtag/usbjtag.html`

[10] Amontec, supplier of various JTAG adaptors
`http://www.amontec.com`

[11] Amontec *JTAGkey* USB to JTAG adaptor
`http://www.amontec.com/jtagkey.shtml`

[12] Amontec *JTAGkey-Tiny* USB to JTAG adaptor
`http://www.amontec.com/jtagkey-tiny.shtml`

[13] Olimex *ARM-USB-OCD* USB to JTAG adaptor
`http://www.olimex.com`

[14] FTDIChip Homepage
`http://www.ftdichip.com`

[15] *Signalyzer* JTAG adaptor
`http://www.signalyzer.com`

[16] LuminaryMicro Homepage
`http://www.luminarymicro.com`

[17] Emacs Text Editor
`http://www.gnu.org/software/emacs/`

[18] Nick Roberts, Emacs Mode for GDB
`http://linuxjournal.com/article/7876`

[19] The *Data Display Debugger* (DDD)
`http://www.gnu.org/software/ddd/`

[20] The *Insight* debugger
`http://sources.redhat.com/insight/`

[21] Eclipse IDE for C programming
`http://www.eclipse.org/cdt/`.

[22] James P. Lynch, Using Open Source Tools for AT91SAM7S Cross Development, Rev. 2, Oct 8 2006, 145 pages (covers Eclipse and OpenOCD)
`http://www.atmel.com/dyn/resources/prod_documents/atmel_tutorial_source.zip`

[23] Michael Fischer, Yet Another GNU ARM Toolchain
`http://www.yagarto.de/howto/yagarto1/index.html`

[24] Subversion revision control
`http://subversion.tigris.org`

[25] D. A. Wheeler, SLOCCount User's Guide
`http://www.dwheeler.com/sloccount/sloccount.html`