The Advanced JTAG Bridge

Nathan Yawn

nathan.yawn@opencores.org

05/12/09

History

Rev	Date	Author	Comments
1.0	28/7/2008	Nathan Yawn	Initial version
1.1	14/6/2009	NY	Legacy dbg unit support, improved BSDL section, Apx. B

Contents

1. Function

1.1. Overview

The adv_jtag_bridge program is part of the toolchain used to debug an OpenRisc system. It acts as a bridge between GDB (the GNU debugger) and an OpenRisc-based System on Chip (SoC). The adv_jtag_bridge program accepts commands from GDB via a network port, then translates those commands into a format understandable by the Advanced Debug Interface hardware core (which must be included in the hardware system). The translated commands are sent to the hardware system via a JTAG cable, by using one of the cable drivers included in adv_jtag_bridge. See the *Cable Drivers* section for a complete listing of supported hardware cables. The program also has "cable" drivers to connect to an HDL simulator such as ModelSim. Note that GDB connects to the or1ksim architectural simulator directly; adv_jtag_bridge is not needed in this case.

The adv_jtag_bridge program is designed to work with the "Advanced Debug Interface" hardware core (adv_dbg_if). However, it may also be compiled to word with the legacy "dbg_interface" core. In order to support the legacy core, adv_jtag_bridge must be recompiled with the __LEGACY__ compiler macro #defined for all files. Support for the legacy debug unit is mutually exclusive with support for the advanced debug interface – if legacy support is enabled, support for the advanced unit will be deactivated.

The advanced debug interface may be connected to one of several supported JTAG TAP hardware cores: the standalone TAP ("jtag" core), a Xilinx internal BSCAN device ("xilinx_internal_jtag" core), or an Altera sld_virtual_jtag megafunction ("altera_virtual_jtag" core). The adv_jtag_bridge program includes drivers for all three of these TAPs. See the *TAP Drivers* section for details on each of these TAP devices.

At startup, adv_jtag_bridge probes for the cable specified on the command line, and attempts to identify all devices in the JTAG chain. The device chain information is printed to the console. In order to operate, adv_jtag_bridge needs to know which device in the chain is the target device (the OpenRisc system to be debugged), the length of the Instruction Register (IR) for every device in the chain, and the IR command which selects the advanced debug interface in the target device.

In order to operate correctly, adv_jtag_bridge needs to know the length of the IR in every device on the JTAG chain, and the command for the target device's IR which will make the debug hardware module active. The easiest way to provide this information is using BSDL files. Most chip manufacturers provide a BSDL file for each of their chips, specifically for use with JTAG driver programs. A BSDL file contains all the information for each chip that adv_jtag_bridge needs. To use BSDL files, simply place a copy of the BSDL file for each chip in the JTAG chain into the directory where the adv_jtag_bridge binary resides (or one of the directories adv_jtag_bridge searches – see the program's individual documentation for details). Adv_jtag_bridge will automatically parse the BSDL files and use the information.

If any device in the chain does not support the IDCODE instruction, or if a BSDL file for the device is not available, then the length of that device's IR must be supplied on the command line. If the target device does not support IDCODE, or its BSDL file does not specify a DEBUG instruction (USER1 for Xilinx devices), then the debug command for the target device must also be specified on the command line.

Once the enumeration of the JTAG chain is complete, an optional self-test may be run on the hardware, which tests debugger access to RAM, the OR1000 CPU, and SDRAM attached to an OpenCores SDRAM controller ("mem_ctrl" core). The exact tests run are a compile-time option for adv_jtag_bridge, and may be changed in or32_selftest.c. In particular, SDRAM controller initialization is system-specific, and is not done by default. After the optional self-test, adv_jtag_bridge sets up a network server port for GDB to connect to.

The adv_jtag_bridge program includes drivers for both the CPU debug interface and the Wishbone interface present in the advanced debug core. The CPU driver can be used to stall and restart the processor, to take control at breakpoints, and to read and set the values of CPU registers (future versions may add support for hardware breakpoints and watchpoints, and instruction trace). The Wishbone driver can be used to load programs, examine memory, and set software breakpoints. Note that while adv_jtag_bridge includes drivers for a second CPU, GDB does not support a second CPU; these drivers are therefore currently unused.

1.2. TAP Drivers

Three different JTAG Test Access Port (TAP) devices are supported by adv_jtag_bridge. Each requires different behavior by the program. In general, the type of TAP used by the target device is automatically detected, and the appropriate driver is selected automatically. This section gives some details on each type of supported TAP.

1.2.1. Standalone JTAG TAP

This type is exemplified by the “jtag” hardware core. It is a complete JTAG TAP with its own Instruction Register and decoding. It requires four external IO pins on the device in use, for the standard JTAG signals (TCK, TMS, TDI, TDO). This device requires no special logic to drive; the DEBUG command must simply be shifted into the IR; the advanced debug interface is connected as an ordinary Data Register. The value of the DEBUG command may be changed in the hardware at synthesis time.

1.2.2. Altera Virtual JTAG

This device is used in Altera FPGAs to give internal devices / cores access to the chip's JTAG lines (the same lines used to upload the bitstream to the FPGA). As such, it requires no separate, dedicated external IO. This TAP is used by the “altera_virtual_jtag” core, which this driver is designed to work with.

The virtual JTAG device represents a second TAP, controlled by the chip's main TAP. Both the IR and the DR of the virtual TAP are connected as Data Registers of the main TAP, and the advanced debug interface is connected as the virtual TAP's Data Register. To enable the advanced debug interface, the virtual IR must first be selected in the real IR. Then the DEBUG command is placed into the virtual IR by performing a real DR shift. Finally, the virtual DR (the advanced debug interface) must be selected in the real IR.

The virtual IR (VIR) and virtual DR (VDR) select commands are the same for all known Altera devices; these are hardcoded into the program. They may be overridden on the command line if it ever becomes necessary.

The virtual IR in the “altera_virtual_jtag” core is 4 bits, and the virtual DEBUG command is 0x8, the same values used in the standalone “jtag” core TAP. These values may be changed in the hardware core at synthesis time. If these values are changed in hardware, then the hardcoded values in this program should be changed to match. These values can be found in the file opencores_tap.h.

The driver for the Altera Virtual JTAG TAP is automatically selected if the IDCODE of the target device has an Altera manufacturer ID. The virtual JTAG driver may be explicitly enabled or disabled on the command line.

1.2.3. Xilinx internal BSCAN

This device is used in Xilinx FPGAs to give internal devices / cores access to the chip's JTAG lines (the same lines used to upload the bitstream to the FPGA). As such, it requires no dedicated external IO. The Xilinx BSCAN module is used in the “xilinx_internal_jtag” core.

The BSCAN device contains the IR and decoding logic, but the advanced debug interface is connected as an ordinary Data Register. For these devices, the USER1 command must be used instead of the DEBUG command; the USER1 command cannot be changed. No other special logic is required.

Xilinx internal BSCAN mode is enabled automatically if the IDCODE of the target device includes a Xilinx manufacturer code. Because the only change required is the use of the USER1 command instead of DEBUG, internal BSCAN mode can be “disabled” on the command line by explicitly specifying the DEBUG command; the specified command will be used in place of USER1.

1.3. Cable Drivers

Several different communication drivers are supported by adv_jtag_bridge. These are listed and described below.

1.3.1. XPC3

This is a driver for the Xilinx Parallel Cable, version III. This is a bit-banging parallel-to-JTAG adapter. This may also work with version IV cables in low-speed 'compatibility mode'. The command line name is “xpc3”.

1.3.2. XESS

This is another bit-banging parallel interface. It is compatible with some hardware made by XESS Corp., in particular the XSV-800. Note that the JTAG interface to the FPGA on XESS boards is routed through a CPLD; this CPLD must be programmed correctly to allow access to the FPGA. The command line name is “xess”.

1.3.3. USB-Blaster

This driver supports the Altera USB-Blaster USB-to-JTAG cable. High-speed transfers are fully supported. Clones of this cable, such as those which use the usbjtag project software, should also work with this driver. This driver requires that the “libusb” library be installed. The command-line name is “usbblaster”.

1.3.4. Xilinx Platform Cable USB (DLC9)

This driver supports the Xilinx Platform Cable USB, model DLC9. This driver is experimental, and currently does not support any of the cable's high-speed modes. This driver requires that the “libusb” library be installed. The command-line name is “xpc_usb”.

1.3.5. VPI Simulation

This driver interfaces with an HDL simulation program such as ModelSim or Icarus via the verilog VPI mechanism. A C library is required to interface to the simulation. Communication between adv_jtag_bridge and the C library is done using network sockets. The simulator-side C library is distributed along with the adv_jtag_bridge source as jp-io-vpi.c. This feature also requires a special verilog module in the simulation, which calls the C library functions. This core, called dbg_comm_vpi.v, is distributed with the adv_jtag_bridge source code (in the rtl_sim/ subdirectory). The command-line name is "vpi".

1.3.6. RTL Simulation

This driver is also designed to interface to an HDL simulation. Unlike VPI, this driver communicates directly with a verilog simulation using file IO. A special hardware core must be included in the hardware system in order to use this interface, called dbg_comm.v (distributed along with the adv_jtag_bridge source, in the rtl_sim/ subdirectory). The command-line name for this cable is "rtl_sim".

1.4. BSDL Parsing

The adv_jtag_bridge includes a simple BSDL parser, which extracts the minimum amount of required information from a BSDL file. This prevents the user from having to enter the IR length of every device in the JTAG chain on the command line or the DEBUG command of the target device.

After the IDCODES of the devices in the chain have been determined, BSDL information is sought for each device. Four directories are searched by default, in this order: "." (the current directory), "~/.bsdl", "/usr/share/bsdl", and "/opt/bsdl". Directories added on the command line are searched before the default directories. For each device on the chain, BSDL files are opened and parsed until a matching IDCODE is found (a 'lazy' algorithm). Note that multiple BSDL files may match the IDCODE sought; "X" ("don't care") is a valid bit value in a BSDL IDCODE, and some manufacturers provide BSDL files for both families of devices and for specific devices in that family. The first match found will be used.

All parsed data is retained in memory, meaning that a BSDL file will never be parsed more than once for any given execution of the program. It is suggested that if a large number of BSDL files are kept in a default directory, a minimum subset should be copied to a separate directory and specified on the command line – this will limit the number of BSDL files which may be parsed, improving program performance at startup.

1.5. RSP Server

The adv_jtag_bridge program communicates with GDB via network sockets, using the RSP protocol. Once the JTAG chain has been enumerated (and the self-test optionally performed), adv_jtag_bridge will wait for an RSP connection from GDB on port 9999 (or another port specified on the command line).

Because adv_jtag_bridge is designed to be used in a "bare metal" debugging system, many of the RSP commands are irrelevant (such as those relating to threads), and are therefore unsupported. However, all the basic commands needed for debugging are supported, such as read or write register (or all registers), read or write memory (binary or symbolic), insert or remove breakpoint, and asynchronous break. In general, all RSP commands supported by the or1ksim architectural simulator are also supported by adv_jtag_bridge, plus the asynchronous break.

2. Syntax

The adv_jtag_bridge command line has the following syntax:

```
adv_jtag_bridge  <options> [cable]  <cable options>
```

The global options are:

`-g [port number]` : Specifies the TCP port number where the RSP server for GDB will be started. If not specified, a default of 9999 is used.

`-x [index]` : Index of the target device on the JTAG chain. The device closest to the data input of the JTAG cable is index 0. The devices in the chain and their indexes are displayed at startup, this may be used to discover the device ordering. If not specified, a default of 0 is used (suitable for single-device chains).

`-l [<index>:<bits>]` : Specify the IR length for a particular device in the JTAG chain. This must be done for each device in the chain which does not support the IDCODE command, or for which a BSDL file is not available. This option may appear multiple times in the command line. Note that this option will override data found in BSDL files.

`-a [0 or 1]` : Force Altera virtual JTAG mode on or off. Normally, the program tests the value of a device's ID register, and switches to virtual JTAG mode if an Altera manufacturer code is found. However, there may be cases when auto-detection fails. In these cases, '`-a0`' will force virtual JTAG mode OFF, and '`-a1`' will force the use of virtual JTAG mode ON. A warning will be printed to the console if the auto-detected behavior is overridden by this command line option.

`-c [hex cmd]` : Specify the DEBUG command which will select the advanced debug unit in the IR of the target device TAP. Must be specified if the target device does not support the IDCODE command, or if a BSDL file is not available for it. This option will override data found in a BSDL file. (Ignored for Altera Virtual JTAG targets).

`-v [hex cmd]` : Specify the VIRTUAL_IR_SHIFT command, which will select the virtual IR in the (real) IR of the target device TAP. Used only for Altera Virtual JTAG targets. This is the same value for all current, known Altera devices, and this value is hard-coded into adv_jtag_bridge. If the value should change for future devices, this command may be used to override the hard-coded value.

`-r [hex cmd]` : Specify the VIRTUAL_DR_SHIFT command, which will select the virtual DR in the (real) IR of the target device TAP. Used only for Altera Virtual JTAG targets. This is the same value for all current, known Altera devices, and this value is hard-coded into adv_jtag_bridge. If the value should change for future devices, this command may be used to override the hard-coded value.

`-b [dirname]` : Add a directory to the list of directories to search for BSDL files. By default, this list includes the current directory ".", "~/.bsdl", "/usr/share/bsdl", and "/opt/bsdl" (searched in this order). Any directories added using this command-line parameter are searched before the default directories, in the reverse order specified (the last directory specified on the command line is searched first). This option may appear on the command line more than once.

`-t` : Run self-test before starting GDB server. This will test CPU function and the first 1024 bytes of memory starting at address 0x0.

`-h` : Print help, with a summary of command line options.

The `[cable]` argument specifies which JTAG cable driver (or simulation driver) to use. This argument is mandatory. The `<cable options>` depend on the cable chosen – some cables have options which must be specified, others have none. The list below shows the supported cables and the options associated with each.

xpc3

Options:

`-p [port]` : The IO address of the parallel port which connects to the XPC III cable. Default is 0x378.

xess

Options:

`-p [port]` : The IO address of the parallel port which connects to the XESS board. Default is 0x378.

usbblaster

Options:

No options.

xpc_usb

Options:

No options.

vpi

Options:

`-s [server]` : Name of the server on which the VPI module is running. May be IP address or host name. Default is “localhost”.

`-p [port]` : Port number on which the VPI module is listening. Default is 4567.

rtl_sim

Options:

`-d [directory]` : Directory where the gdb_in.dat and gdb_out.dat files will be created.

2.1. Usage examples

```
> adv_jtag_bridge usbblaster
```

Starts adv_jtag_bridge with the USB-Blaster cable driver, target device of 0 in the JTAG scan chain, all devices support IDCODE, BSDL files available for all devices, no self-test, GDB server on port 9999.

```
> adv_jtag_bridge -g 1234 -x1 -l 0:5 -l 1:4 -c 0x8 -b "/bsdl" -t xpc3
-p 0x378
```

Starts adv_jtag_bridge with the xpc3 cable driver using the parallel port at 0x378. The JTAG scan chain has two devices which require command-line specification of the IR length (indexes 0 and 1, lengths 5 and 4 respectively); The target device is index 1; The debug command for the target device is specified as 0x8; The directory “/bsdl” will be searched first for BSDL files. Run self-test before starting GDB server on port 1234.

3. See also

- *Debugging System for OpenRisc 1000-based Systems*
- IEEE Std. 1149.1 (JTAG TAP and Boundary Scan Architecture)
- The Advanced Debug Interface (`adv_dbg_if`) core documentation
- OpenCores `jtag` core documentation
- `altera_virtual_jtag` core documentation
- `xilinx_internal_bscan` core documentation
- Altera's *sld_virtual_jtag Megafunction User Guide*

Appendix A: Simulator Connections

The adv_jtag_bridge program can connect to an HDL simulation by either of two different methods. The best method for you depends on your simulator and environment. Both methods require additional code to be added to the hardware system being simulated. The VPI method also requires an additional C shared library. Both methods are described below.

File IO

This method uses the workstation's file system to pass data between the adv_jtag_bridge program and an HDL simulation. The HDL simulator must support verilog file IO in order for this method to work.

A verilog module is added to the hardware system which uses data read from a file to set the states of the JTAG lines. The state of the serial data output is written to another file. These files are written and read, respectively, by adv_jtag_bridge.

The verilog module which performs the file IO and controls the JTAG lines is called dbg_comm.v, and is included with the adv_jtag_bridge source, in the rtl_sim/ subdirectory. To use it, the module must be instantiated in your HDL system, and its JTAG signals must be connected to the JTAG signals of the system's JTAG TAP controller. The dbg_comm module will then drive the inputs to the simulated TAP, just as a hardware cable would drive the inputs to a hardware TAP.

In order to use a file IO simulator connection, select the cable "rtl_sim" when starting adv_jtag_bridge. Be sure to specify the same directory for the communication files on the command line that is specified in the dbg_comm.v verilog file; the verilog file must be changed to match user preferences before compilation. Remember that your simulation must be actually running (simulation time must be passing) in order to communicate with the simulation.

The adv_jtag_bridge program waits for an acknowledgment each time it writes a signal to the simulator. As such, adv_jtag_bridge and the simulation may be started in any order. If the simulation is started first, it will run without changing the state of the JTAG lines. If adv_jtag_bridge is started first, it will attempt to write the first bit to the simulator, then wait for the simulator to acknowledge. The simulation will wait until reset is complete before reading the shared files and reading the bit from the bridge program. Note however that the files are actually created by adv_jtag_bridge; if they do not exist when the simulator is running, warnings may occur in the simulator.

VPI IO

This method uses the Verilog Program Interface (VPI) to connect a verilog simulation to adv_jtag_bridge. VPI is an interface which allows arbitrary verilog system tasks to be written by a user in C. The code is compiled into a shared library, which is linked to the simulator at run-time. This allows the newly defined system tasks to be called by verilog code during simulation.

A C library has been implemented which performs communication to adv_jtag_bridge. The library uses network sockets for communication instead of filesystem IO, and may be faster than file IO. However, this method is more complex to use: your HDL simulator must support UDI / VPI in order for this method to work, and you must compile the shared library for your specific OS and simulator. Modelsim, NCsim, and Icarus are all known to support VPI. The source code for the C library, called jp-io-vpi.c, is included along with the source for the adv_jtag_bridge program, in the rtl_lib/src/ subdirectory.

Because the compiled library may be used with several different simulators and operating systems, the method for building the library may vary. Makefiles for some combinations are included in subdirectories of the rtl_lib/ directory, and some pre-built binaries are included as well. Examine the Makefile to find the valid make targets. If a Makefile is not included for your simulator / operating system, see the documentation for your simulator for instructions on how to build a VPI library for your system. The library source code is mostly generic; the network portions have been written for both the standard Berkeley socket API (used by default) and for the Win32 netsock2 API (used when WIN32 is defined at compile time). It should therefore be possible to compile the library on a wide variety of systems. Note that under Win32, the cygwin version of GCC cannot be used to compile the library; the MinGW version of GCC must be used (MSVC can also be used).

You will also need to find how to connect the library to your simulator. This step also differs for each simulator. For Modelsim, it is sufficient to place the compiled library in the base directory of the simulator project, and to indicate the library to be used by setting the simulator's PLIOBLS environment variable before starting a simulation (you may also specify VPI libraries in the modelsim.ini file, and on the vsim command line; see the Modelsim manual). For other simulators, different library locations and indications may be required. Check your simulator documentation for details.

Similar to the file IO method, a verilog module is added to the hardware system which interfaces to the C library. This module receives commands from adv_jtag_bridge, sets the JTAG outputs accordingly, and returns the state of the TDO line to adv_jtag_bridge via the C library. This verilog module (dbg_comm_vpi.v) is included with the adv_jtag_bridge source, in the rtl_sim/ subdirectory. To use it, the module must be instantiated in your HDL system, and its JTAG signals must be connected to the JTAG signals of the system's JTAG TAP controller. The dbg_comm_vpi module will then drive the inputs to the simulated TAP, just as a hardware cable would drive the inputs to a hardware TAP.

In order to use a VPI connection, select the cable “vpi” when starting adv_jtag_bridge. Remember that your simulation must be actually running (simulation time must be passing) in order to communicate with the simulation.

The VPI library acts as the network server, adv_jtag_bridge acts as a client. As such, the simulation must be started and some simulator time must have elapsed before adv_jtag_bridge can be started; starting adv_jtag_bridge first should result in a network connection error. Also note that the server socket is closed after a connection is made – this means that if adv_jtag_bridge is killed, the simulation must be restarted before it will accept another network connection from adv_jtag_bridge.

Debugging a simulation can be slow. Depending on the capabilities of your workstation and the complexity of the simulated system, the optional self-test may take 20 minutes or more to simulate.

Appendix B: Code Structure

The adv_jtag_bridge program includes multiple abstraction layers, in order to support multiple JTAG cables, as well as multiple debug hardware units, and potentially even multiple GDB interface protocols. This appendix is designed to give a high-level overview of the code structure, in order to assist those wishing to modify the code for their own purposes.

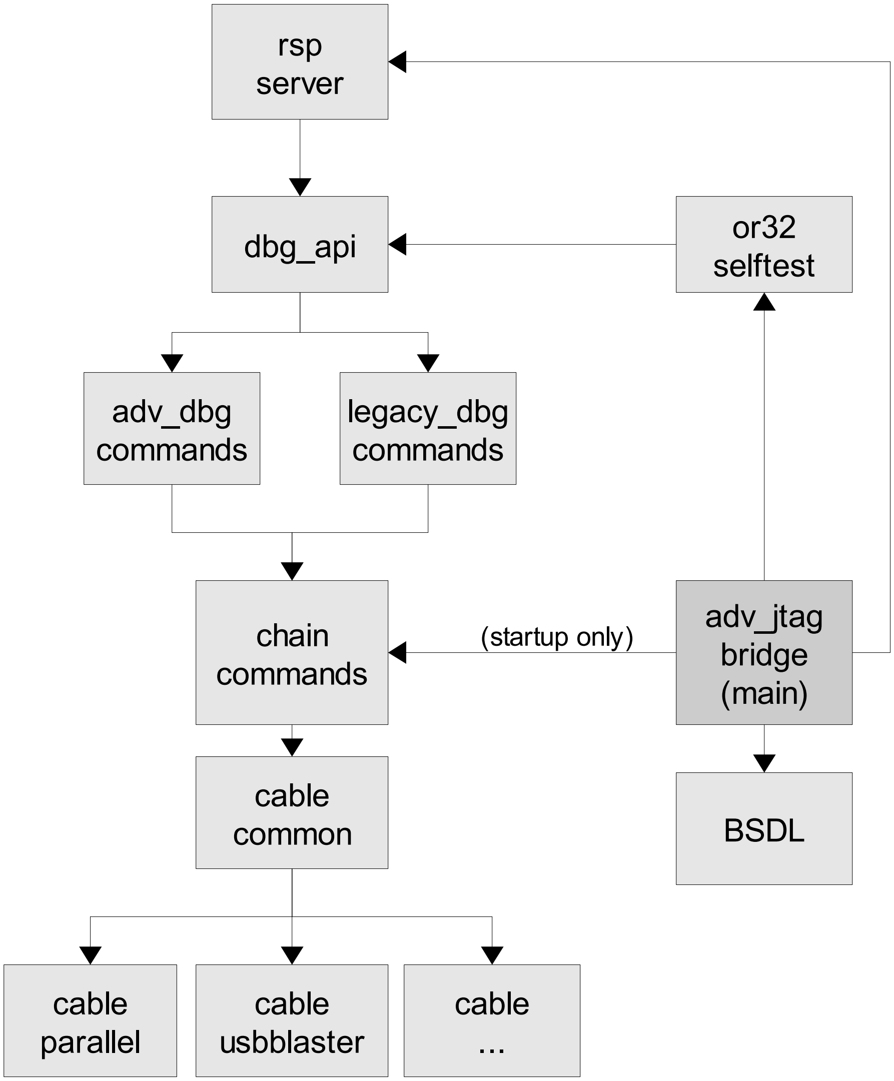

Figure 1: Block diagram of code structure

The main entry point of the program in in adv_jtag_bridge.c. At startup, the program calls functions in chain_commands.c to determine how many devices are on the JTAG chain, and the ID codes of as many of those devices as possible. Once the ID codes have been determined, the program calls functions in bsdl.c (which calls functions in bsdl_parser.c) to look up the IR length and, if necessary, the DEBUG command of all probed devices.

At this point in the program execution, the self-test is optionally run. The self-test calls the high-level API functions in dbg_api.c. Note that the self-test assumes that SRAM is present starting at address 0x00000000 on the target hardware's WishBone bus – this RAM is used for uploaded CPU instructions.

Once the (optional) self-test is finished, the program starts the RSP server (found in rsp-server.c), which runs until the program exits. The RSP server opens a network server socket, accepts a single connection from a client, and serves RSP requests until the client disconnects or the program exits. RSP requests are sent to the hardware by using the functions in dbg_api.c.

The functions in dbg_api.c are an abstraction layer for the two supported debug units. These functions will call the appropriate functions in adv_dbg_commands.c or legacy_dbg_commands.c, depending on which was enabled at compile time. These functions create the hardware-specific JTAG bitstreams which will be sent to the debug unit hardware. The bitstreams are sent by calling functions in chain_commands.c.

The functions in chain_commands.c are used to change the state of the TAP FSM, and to send or receive a JTAG bitstream. The functions at this layer may also adjust the bitstream in order to deal with multiple devices on the JTAG chain. Once a bitstream has been adjusted to take this into account, it is sent by calling functions in cable_common.c.

The cable_common.c layer is an abstraction layer for the various JTAG cable drivers. It will simply call the cable-specific driver function for the cable that was selected on the command line of the program.