Technical Article

JTAG Implementation in Arm Core Devices

December 14, 2020 by Sam Gallagher

This article will teach you about the intersection between JTAG and Arm core devices, with special attention paid to the Arm Debug Interface or ADI.

Thus far in our series on JTAG, we've looked at the IEEE 1149.1 standard, including the test access port (TAP) controller and the TAP state machine. Then we reviewed the different physical interfaces available for working with JTAG, including common pinouts for connectors, and the JTAG interfaces and debug probes available on the market.

In this article, we're going to depart slightly from the JTAG standard and instead look at how JTAG is implemented in the ubiquitous ARM core devices.

What Is Arm?

Arm refers to a processor architecture, along with a large amount of intellectual property relating to microprocessor and microcontroller interfaces. Where consumer PCs tend to use x86-derived processors, or PowerPC, or MIPS, embedded electronics are most often implemented with Arm-core processors.

The "core" of the processors are distributed to manufacturers such as ST Microelectronics or NXP, and these manufacturers then add additional peripheral features, such as I2C and SPI interfaces, ADCs and DACs, USB interfaces, and so on.

Arm architectures are versioned as ARMv, examples being ARMv2 (dating from 1987), ARMv6 (processors produced 2002-2005) and so on, and the processor cores which utilize those architectures are branded as ARMx series (ARM1, ARM6, ARM7), and more recently as ARM Cortex-A/R/M series for high-performance (Cortex-A), real-time (Cortex-R), and microcontroller (Cortex-M) applications. The architecture versioning follows the Cortex suffix naming, becoming such versions as ARMv6-M, ARMv7-R, ARMv7-A.

Arm's debugging interface falls under the name of the Arm CoreSight Architecture; this includes the debug interface (Arm Debug Interface, ADI), embedded trace macrocells (ETM), high-speed serial trace ports (HSSTP), and CoreSight program flow trace architecture. The ADI forms the base for debugging operations with Arm-core processors, and part of this standard includes a JTAG interface as well as the Serial Wire Debug (SWD) alternative. The topic of this article is the ADI, and particularly the hardware interface layers.

Introduction to the Arm Debug Interface (ADI)

The Arm Debug Interface (ADI) is a specification of both the hardware interface and the logical interface for debugging between a host and one or more devices. Currently, most processors are implementing ADIv5 (specified in Arm IHI0031E), while the newer ADIv6 (see Arm IHI0074C) is being slowly phased in. Because it is more popular, we'll be focusing here on the ADIv5 standard.

The ADI defines a debug access port (DAP), made up of a debug port (DP) and access ports (APs). The DAP is the primary "unit" of the debug interface. A given device will have one debug port, which handles the physical connection with a debugger, as well as a number of access ports that provide access to system resources such as debug registers, trace port registers, a ROM table, or system memory. Thus the DP includes the physical connection (JTAG, SWD) as well as some built-in registers, and each AP has its connections to the system, and a number of its own registers.

In ADIv5, there are two types of debug ports, and (broadly speaking) three types of access ports. The DPs can be either JTAG-DPs, or SWD-DPs, named for the interface used in connecting to the DAP from outside the device. The APs can be MEM-APs, providing access to resources by addressing (analogous to memory mapping, hence the name), JTAG-APs, allowing JTAG scan chains to be connected to the whole debug unit (the DAP), and vendor-specific APs, which are not specified by Arm. MEM-APs are by far the most common and useful, so we won't be covering the other types here.

In the context of JTAG, we would expect the Debug Interface to provide JTAG instruction codes, as well as vendor-specific JTAG features. In fact, the ADIv5 standard provides the following instructions:

- EXTEST (0b00000000)
- SAMPLE (0b00000001)
- PRELOAD (0b00000010)
- INTEST (0b00000100)
- CLAMP (0b00000101)
- HIGHZ (0b00000110)
- ABORT (0b11111000)
- DPACC (0b11111010)
- APACC (0b11111011)
- IDCODE (0b11111110)
- BYPASS (0b11111111)

As well, the JTAG data registers include:

- ABORT (35 bits), register to force an Access Port abort
- DPACC (35 bits), Debug Port read/write access register
- APACC (35 bits), Access Port read/write access register
- IDCODE (32 bits)
- BYPASS (1 bit)

What should stand out here are the instructions DPACC and APACC, and the associated data registers. DPACC (Debug Port Access) and APACC (Access Port Access) are the instructions/registers used to pass commands through to the associated DP and APs of a device. By setting different values in the DPACC or APACC data registers, the debugger can execute different operations, generally interacting with the registers of the DP and APs themselves. Note the difference between these DPACC and APACC registers (JTAG registers) and the registers built into the DPs and APs.

The general methodology here is that the debugger uses the JTAG or SWD interface to execute instructions by going through the TAP state machine, then the instructions take the data and load it into the DP or an AP, and depending on the data, different registers within the DP or AP are accessed, providing the desired link to the system.

So, what are the DP and AP registers? The DP registers available include:

- CTRL/STAT, used to control and obtain status information about the DP
- DLCR, Data Link Control register, controls the operating mode of the Data Link
- DLPIDR, Data Link Protocol Identification register, protocol version information
- DPIDR, Debug Port Identification register, Debug Port information
- EVENTSTAT, Event Status register, used by the system to signal an event to the external debugger
- RDBUFF, Read Buffer register, provides a read operation; dependent on DP (JTAG or SWD)
- SELECT, AP Select register, selects an Access Port and the active register banks with that AP; selects the DP address bank
- TARGETID, provides information about the target when the host is connected to a single device

While MEM-AP registers are:

- Control/Status Word register (CSW, 0x00), holds control and status information
- Transfer Address Register (TAR, 0x04), holds the address for the next access to the memory system or system resource
- Data Read/Write register (DRW, 0x0C), sets reading or writing of the address in TAR – if you read DRW, the access is set to read; if you write to DRW, the access is set to write
- Banked Data registers (BD0 to BD3, 0x10, 0x14, 0x18, 0x1C), provide direct read or write access to four 32-bit blocks of memory, starting at the address in TAR
- Configuration register (CFG, 0xF4), information about MEM-AP configuration

- Debase Base Address register (BASE, 0xF8), pointer to memory system, either the start of a set of debug registers or the start of a ROM table
- Identification Register (IDR, 0xFC), identifies the MEM-AP.

The connections are shown schematically in Figure 1, below.

Scroll to continue with content

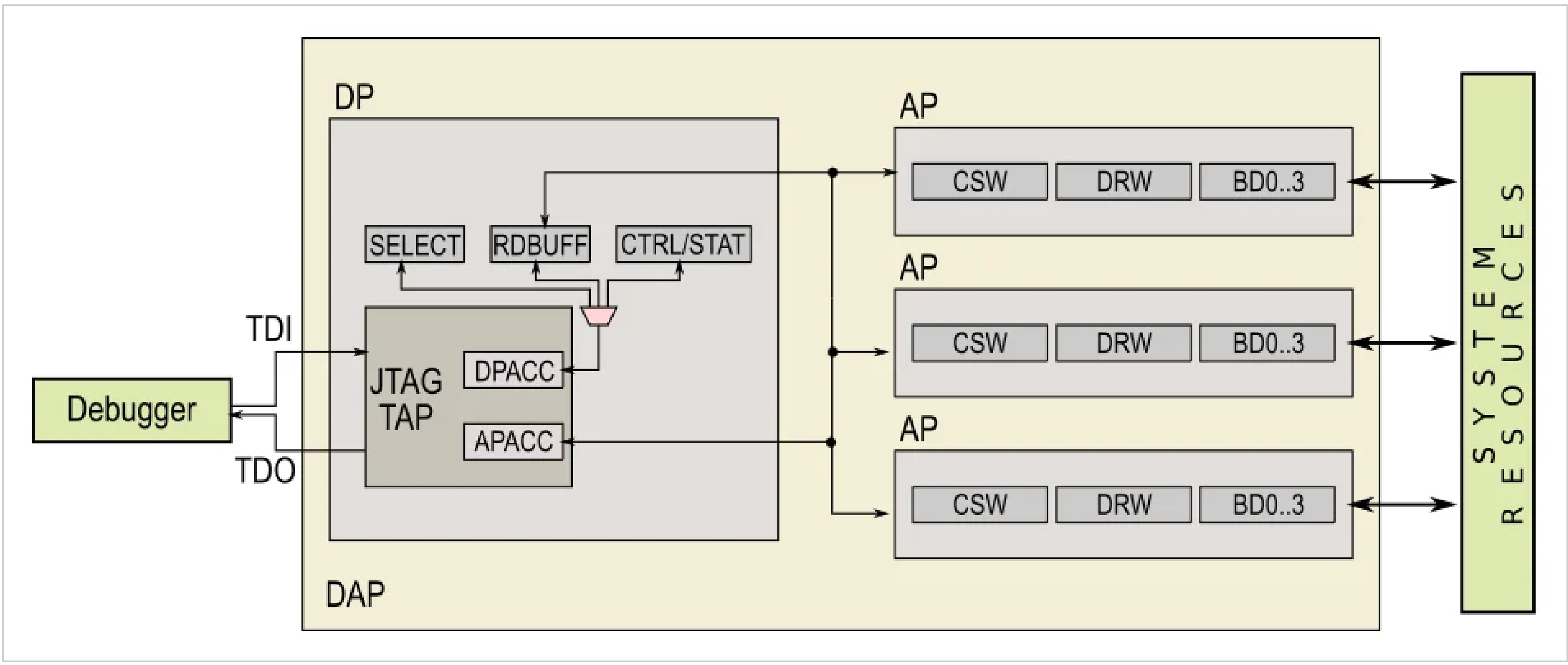

Figure 1. Arm Debug Interface diagram, summarizing connections. Note: not all registers are shown for DPs and APs.

Details of the DP/AP registers and the memory mapping can be found in the specification, IHI 0031E. Instead of going further into the details, I would like to focus on the other type of debug port, the SWD-DP, and how it implements JTAG using only two wires.

Serial Wire Debug (SWD)

While the JTAG-DP is a common and familiar example of a debugging interface, most relevant to our discussion is the JTAG alternative defined for Arm devices, the Arm Serial Wire Debug (SWD). SWD was developed as a two-wire interface for Arm-core devices with limited pin counts. As microcontrollers tend to be quite dense in peripherals, most Cortex-M devices will implement SWD in place of full JTAG to save pin real-estate. So how does this protocol work?

SWD is specified in the ADIv5 specification (chapter B4). The TDI and TDO pins from JTAG are replaced by a single bidirectional pin called SWDIO; the test mode select (TMS) pin is removed entirely; and the clock (TCK) remains the same (relabeled CLK or SWCLK). Thus SWD uses only two pins compared to the four pins used in JTAG. To make this work, SWD relies on the repetitive nature of JTAG operations: the state machine is manipulated, data is shifted in or out, and the process repeats. The difference with SWD is there is no state machine. Instead, commands are issued serially over SWDIO, and then that same pin is used for reading or writing data.

There are three phases to SWD communication: packet request, acknowledgment, and data transfer. During packet request, the host platform issues a request to the DP, and this must be followed by an acknowledge response. If the packet request was a data read or data write request, and there was a valid acknowledgement, the system enters the data transfer phase, where data is clocked in (write) or clocked out (read) through SWDIO. After a data transfer, the host is responsible for either starting a packet request, or driving the SWD interface with idle cycles (clocking SWDIO LOW). A parity check is applied to packet requests and data transfer phases.

The particulars of SWD are best understood by seeing a timing diagram, shown in Figure 2.

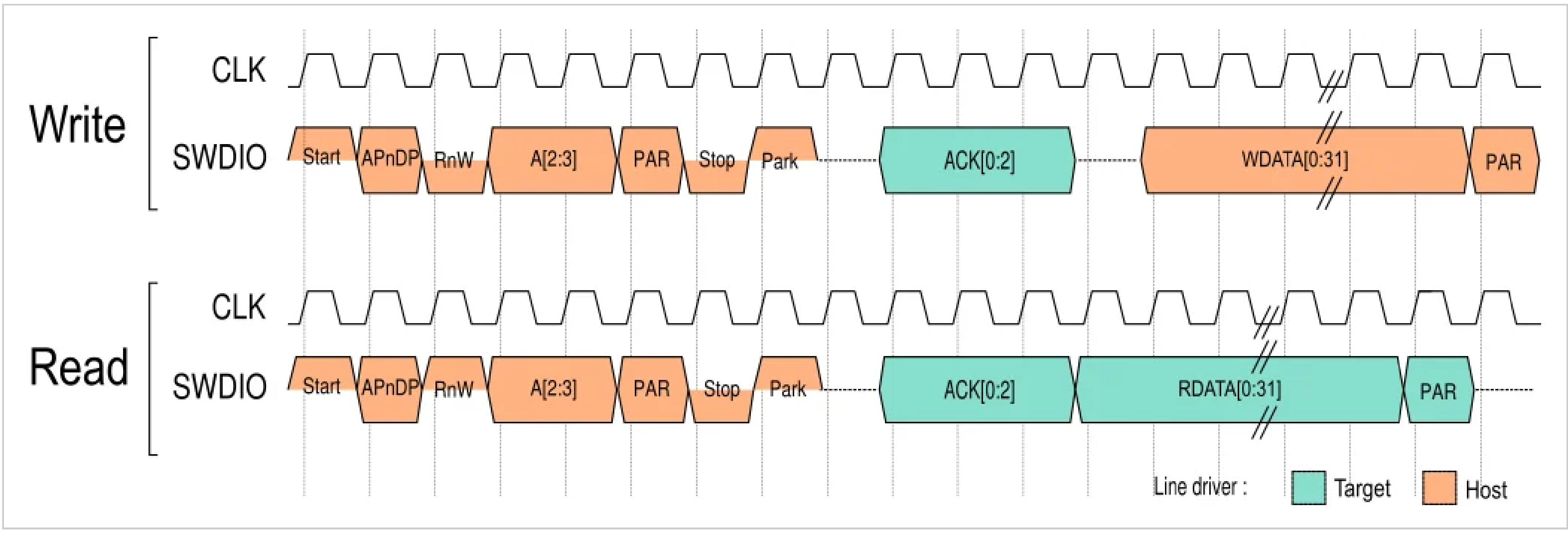

Figure 2. Timing diagrams showing read and write operations for Serial Wire Debug. [Click to enlarge]

The read and write operations both begin the same, starting with the host driving the SWDIO line to a Start bit, which is a HIGH signal. This is followed by the configuration, including the AP or DP access bit (APnDP), the read or write bit (RnW), the address (A[2:3]), a parity bit (PAR), and a Stop bit (a LOW signal), followed finally by a Park bit, which is when the host drives the line HIGH before the line goes into turnaround. During turnaround, neither the target nor the host is allowed to drive the line.

Depending on the value of RnW, a read or write operation is selected. If writing, the target provides a 3-bit ACK signal, then there is another turnaround period, followed by the 32-bit data to be written (WDATA), and a parity bit. If reading, the target provides an ACK, and then continues to drive the line with the 32-bit read data (RDATA), followed by a parity bit. If an error has occurred, the ACK bits will indicate the fault, and the host can attempt to restart the operation. Observe that

WDATA and RDATA are transmitted least-significant bit (LSb) first, indicated in Figure 2 by writing WDATA[0:31] instead of WDATA[31:0].

The Arm IHI0031E document provides further timing diagrams to clarify various cases in communication, but the above are the primary use-cases. It is worth noting that there are (as of the time of writing) two versions of SWD; SWDv1 supports only one connection between a host and a target (point-to-point), while SWDv2 implements single-host multiple-target communication (multidrop). Version 2 is nearly backwards compatible with version 1, apart from a few edge cases.

Other Variants of JTAG

SWD is not the only two-wire variant of the JTAG IEEE 1149.1 standard. Notably, the IEEE 1149.7 standard provides a reduced-pin-count (2-wire) JTAG interface. Other 1149.x standards such as IEEE 1149.4 (Standard for Mixed-Signal Test Bus), and IEEE 1149.6 (Boundary-Scan Standard of Advanced Digital Networks) have been developed in the last two decades, and provide additional functionality for more involved applications. This includes things like analog boundary-scan testing and improved capabilities for differential and AC-coupled lines.

The most up to date standards are available from the IEEE Standards Association website.

Conclusion

This concludes our coverage of JTAG and SWD. Throughout this series, we have learned where JTAG comes from, how it works, and how it's used to debug and program devices. We've taken a look at the physical connections for JTAG, including the connectors and interfaces available, both commercial and open-source. Finally, we concluded with an overview of the JTAG implementation for the popular Arm processor core technologies, including the SWD two-wire interface.

From here, we can go out and confidently use the debugging and programming features of our embedded devices, learning the particulars for different implementations, and making the best use of our time.

Content from NXP

Load more comments
