Application Note

First Time Chip Bring-up Success

Non-Confidential

First Time Chip Bring-up Success

Release Information

The following changes have been made to this Application Note.

Document History

Date	Issue	Confidentiality	Change
06/06/2016	**A**	Non-Confidential	First release

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

http://www.arm.com

Contents

First Time Chip Bring-up Success

1 Conventions and Feedback

The following describes the typographical conventions and how to give feedback:

Typographical conventions

The following typographical conventions are used:

monospace
: denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

monospace
: denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

monospace italic
: denotes arguments to commands and functions where the argument is to be replaced by a specific value.

monospace bold
: denotes language keywords when used outside example code.

italic
: highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold
: highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM® processor signal names.

Feedback on this document

If you have comments on this document, e-mail errata@arm.com. Give:

- The title, First Time Chip Bring-up Success.
- The number, ARM-ECM-0524950, A.
- If viewing online, the topic names to which your comments apply.
- If viewing a PDF version of a document, the page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM periodically provides updates and corrections to its documentation on the ARM Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

- ARM Information Center, http://infocenter.arm.com/help/index.jsp.
- ARM Technical Support Knowledge Articles, http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html.

- ARM Support and Maintenance, http://www.arm.com/support/services/support-maintenance.php.
- ARM Glossary, http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html.

2 Preface

This Application Note is intended for partners developing an ARM processor who are interested in connecting a debugger to the processor for the first time.

It contains the following chapters:

- *Introduction* on page 10.
- *SoC Design Considerations* on page 11.
- *System Design Considerations* on page 18.

2.1 References

- *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile* (ARM DDI 0487).
- *ARM CoreSight Architecture Specification v2.0* (ARM IHI 0029).
- *ARM CoreSight SoC-400 Technical Reference Manual* (ARM DDI 0480).
- *ARM Debug Interface Architecture Specification ADI v5.0 to ADI v5.2* (ARM IHI 0031).
- *CoreSight Connectors*, http://www2.keil.com/coresight/coresight-connectors.

2.2 Terms and abbreviations

AHB-AP
Advanced High-performance Bus Access Port.

AP
Access Port.

APB-AP
Advanced Peripheral Bus Access Port.

AXI-AP
Advanced eXtensible Interface Access Port.

DAP
Debug Access Port.

DP
Debug Port.

ETM
Embedded Trace Macrocell.

FPGA
Field Programmable Gate Array.

JTAG
Joint Test Action Group.

JTAG-AP
JTAG Access Port.

MEM-AP
Memory Access Port.

PADDR31
APB-AP signal used to distinguish debugger accesses.

PMU
Performance Monitor Unit.

SWD
Serial Wire Debug.

TRM
Technical Reference Manual.

3 Introduction

This application note discusses the common issues associated with the bring-up of new ARM-based designs in emulation, FPGA, and hardened silicon. A smooth chip bring-up is important for meeting project schedule demands. It also provides a stable target platform for the ensuing validation and software development efforts that are required for the launch of a new SoC. This application note describes how to connect a debugger to an ARM processor in a bare metal environment.

Many pitfalls stand in the way of first time bring-up success, especially for partners that are new to ARM technology. This application note does not guarantee first time success, but it will help make it possible.

Note

This application note is written from the perspective of bring-up of an ARMv8-A class of processor. However the topics are relevant for any ARM processor. When processor registers are referenced, the v8-A register name is used, but usually there is a register with equivalent functionality for the other ARM architectures.

4 SoC Design Considerations

Chip design is the most important factor to achieve first time bring-up success. Even though there are only a few signals that directly impact a target connection to a debugger, it is critical that they are correct. An improper tie-off or floating signal can result in a target that can't be properly identified by debug tools or even debugged.

This chapter covers elements of the SoC design that impact the ability of a debugger to establish a bare metal connection to the processor. It contains the following sections:

4.1 Authentication signals

The ARMv8-A architecture provides two security states: Secure state and Non-secure state. Each security state has authentication signals that are used to restrict invasive debug and non-invasive access to an ARM processor. The term “invasive debug” refers to the ability to affect the behavior of the system, such as writing internal registers or system memory. Conversely, “non-invasive debug” refers to real-time trace, the Performance Monitor Unit (PMU) and sample based profiling.

These features do not affect the behavior of the system but still allow you to gather pertinent information about program execution. Table 4-1 shows the authentication signals.

Table 4-1 External debug authentication signals

Signal	Purpose
DBGEN	Enable debug in Non-secure state
SPIDEN	Enable debug in Secure state
NIDEN	Enable non-invasive debug in Non-secure state
SPNIDEN	Enable non-invasive debug in Secure state

Note

Not all signal combinations are valid. Consult with *ARM CoreSight Architecture Specification v2.0.*

All authentication signals are active high. For example, if a processor is configured with **DBGEN** low and **NIDEN** high, then the core cannot enter debug state but it can still generate trace when in Non-secure state.

For a device that supports security, the **SPIDEN** and **SPNIDEN** signals are provided to restrict access when the core is in Secure state. In early development, **SPIDEN** and **SPNIDEN** are customarily tied high to allow complete access to the device regardless of security state. However, in production systems, **SPIDEN** and **SPNIDEN** should be tied low to restrict debugger access to Secure state, secure memory and tracing secure software. The signals may also be driven by integrated authentication logic so that the signals may be changed dynamically, or by one-time programmable fuses which enable secure debug and trace only until the fuse is blown.

Application software and a debugger can read the Debug Authentication Status Register (DBGAUTHSTATUS_EL1) to determine the state of the authentication signals. A debugger can also read the External Debug Status and Control Register (EDSCR) to determine the inverse of the state of **SPIDEN**.

4.2 Reset signals

The processor reset logic is usually custom to the SoC, but an ARM processor has several different reset inputs to reset different components of the processor.

Typical reset signals for an ARMv8-A processor are shown in Table 4-2.

Table 4-2 Typical reset signals

Signal	Purpose
nCPUPORESET	Initializes the entire core logic, including debug, ETM, breakpoint and watchpoint logic in the processor CLK domain. Each core has one **nCPUPORESET** reset input.
nCORERESET	Initializes the entire core but excludes the debug, ETM, breakpoint and watchpoint logic. Each core has one **nCORERESET** reset input.
nPRESETDBG	Initializes the shared Debug APB, cross trigger interface, and cross trigger matrix logic in the PCLKDBG domain.
nL2RESET	Initializes the shared L2 memory system, generic interrupt controller, and timer logic.

Note

All reset signals are active low.

The reset signals are an important feature and are addressed in each processor's Technical Reference Manual (TRM). From a debug and chip bring-up perspective, there is value in providing the capability to assert **nCORERESET** in isolation from the other reset signals. Asserting only **nCORERESET** is considered a warm reset as it will reset the core, but keep the debug logic intact. This capability is particularly useful when trying to debug a core that starts execution from uninitialized memory. This approach is frequently the case with initial board bring-up.

A common method to gain control of a "runaway target" is to establish a debug connection to the DAP of the processor (while allowing the core to continue to execute), enable reset vector catch from the debugger, and then assert **nCORERESET**. This warm reset brings the processor back to the reset vector and when execution begins, the processor enters debug state due to vector catch. You can then initialize memory with the target halted.

On many platforms, the **nCORERESET** signal is not accessible at the system level and one is only left with the ability to assert **nCPUPORESET**. This approach not only resets the processor, but it also resets the debug logic and prevents the use of vector catch or hardware breakpoints to enter debug state.

4.3 Debug Access Port (DAP)

The Debug Access Port (DAP) provides an interface for the debugger to communicate with the processor. The DAP is consists of a Debug Port (DP) which supports a JTAG and/or Serial Wire interface to the debugger, and an Access Port (AP) which provides an interface to the processor debug logic and the implemented CoreSight components. Many CoreSight components are typically implemented in a processor design to support trace and cross triggering of cores in a processor. These components are accessible via the AP.

A DAP can also include additional APs to access system memory and debug legacy ARM cores that are not CoreSight based. APs are classified into two groups:

- MEM-APs access a memory subsystem
- JTAG-APs access legacy ARM cores

The DAP is an integral part of the processor's debug infrastructure and is frequently the source of many errors when trying to establish a debugger connection to a processor. This section will highlight the most common pitfalls when implementing the DAP.

4.3.1 Power domain handshaking

The DAP provides the ability to work with processors that have multiple power domains, including even a power domain for the debug logic. To do this, the DAP has a small, always-on power domain that can be used to request that the processor's debug and system logic are powered on for debugging purposes. The DAP utilizes the handshake signals shown in Table 4-3 to perform this task.

Table 4-3 DAP handshake signals

Signal	Purpose
CDBGPWRUPREQ	Requests that the debug power domain receives clocks and power.
CDBGPWRUPACK	Acknowledge for **CDBGPWRUPREQ**.
CSYSPWRUPREQ	Requests that the system power domain receives clocks and power.
CSYSPWRUPACK	Acknowledge for **CSYSPWRUPREQ**.

Note

All reset signals are active high.

The request signals are asserted by the debugger from the CTRL/STAT register in the DAP. The debugger expects to see the corresponding **ACK** signals become asserted after the request is made. The system power controller is responsible for responding to each **REQ** signal, enabling the appropriate clocks and power domain and only then asserting the respective **ACK** signal to indicate that the domain is receiving clocks and power. If there are no separate debug or system power domains, then the **REQ** signal can simply be routed back to generate the **ACK**.

Failure to acknowledge **CDBGPWRUREQ** properly results in an immediate fault response for any transfer requests made to an AP.

Failure to acknowledge **CSYSPWRUPREQ** properly might result in the debugger not establishing a connection to the processor as it might assume that the processor's primary power domain is off. The **ACK** signals should also remain asserted as long as the associated **REQ** signal is asserted.

As a processor can support many power domains (including a separate domain for individual cores in a MP processor), the definition of the DAP system power domain is implementation defined. Therefore, even though the debugger requests that the system be powered, the system power controller can still power down cores on request by another entity (such as an operating system) as long as it retains system power.

4.3.2 Topology detection and ROM tables

Since each SoC contains a custom and unique debug topology, topology detection is an essential step of the chip bring-up process. Each MEM-AP in the DAP contains a ROM table which is used to identify the AP's memory system connections. The ROM table is simply a list of offset addresses located sequentially in memory. The last entry in the table will be 0x0, which signifies the end of the table. The location of the ROM table is fixed and is set by the BASE register in the MEM-AP. It is the job of the development tools to determine what exists at each address specified in the table.

Each valid entry in the table points to a 4KB block of memory. This provides the base register interface for a CoreSight component for identification purposes or the entry could point to another ROM table. From the perspective of topology detection, CoreSight components are classified into different categories such as trace source or trace sink. A processor core is also a class of CoreSight component. To simplify matters in this section, the term CoreSight component includes the processor core.

To determine what components resides there, the topology detection inspects the identification registers in the specified block of memory. The ROM table is critical for proper topology detection of the processor. The CoreSight components in a processor must be properly identified such that a connection to the processor can be established for debugging. If the ROM table is incorrect, then the topology detection process fails. If the topology detection process fails, then the processor can still be debugged as long as the debugger provides a way of manually defining the debug topology.

A more complex part of topology detection is determining how the various trace components are connected so the processor can be traced. Validating a processor's trace functionality is often a secondary step in the bring-up process, but still a vital one. If the component connections cannot be detected by the tools, as with the identification process from the ROM table, then a manual configuration of the trace topology is required.

Usually an APB-AP provides the processor debug interface for the processor and an AHB-AP or AXI-AP is added to support access to system memory. For MEM-APs that only provide access to system memory, the AP's BASE register must be set correctly to indicate that it contains no debug components. Otherwise, the auto-discovery process fails as it attempts to read a non-existent ROM Table for the MEM-AP.

4.3.3 PADDR31 of the APB-AP

As mentioned in section 4.3.2 Topology Detection and ROM Tables, the APB-AP interface is usually used for the debug interface. Commonly, a processor design will grant the processor privileged access to the debug components by mapping the APB-AP address space to the

system address space. This mapping allows the processor to perform self-hosted debug and enables the privileged software running on the processor to program the trace infrastructure without the need for a debugger.

Granting access to the CoreSight components to the system space opens the opportunity for application software to corrupt the debug environment, so a lock mechanism is used to protect the debug logic from errant system-side accesses. The ABP-AP only provides a 31-bit addressable interface and the most-significant address bit, PADDR31, is used to indicate if the access is coming from an external debugger (PADDR31=1) or from another on-chip master (PADDR31=0). If PADDR31 is set, then a CoreSight component can be accessed without needing to be unlocked. However, if the component access is made with PADDR31=0, then the component can only be programmed if its Lock Access Register is first programmed with 0xC5ACCE55.

An external debugger assumes that PADDR31 will be properly managed in the system and that no unlock is required to access a component before it is accessed. Thus if PADDR31 is not high for accesses coming from the APB-AP, it's likely that debugger access to the component will fail.

This potential problem is further magnified on 64-bit address space ARMv8-A systems where you may elect to use an AXI-AP to interface with the debug logic. Here PADDR31 is obfuscated but the requirement still remains for the debugger to be able to make unlocked component accesses. One likely consequence of using the AXI-AP (or even AHB-AP) interface is that you could simply pull PADDR31 high at the component interface. This shortcut would appease the debugger, but provide no protection from errant system-side accesses.

4.3.4 DAP tie offs

The DAP can support up to 256 different APs. Any AP that is not in use should return 0x0 when its ID register is read during the processor discovery phase of the development tools. If an AP fails to return 0x0 for an ID (indicating no AP is present), the tools continue to attempt reads from the AP. As these reads won't be returned, this can lock up the DAP, leading to auto-discovery failure.

A similar problem existed on early CoreSight designs, where the DAP contained a dedicated Auxiliary Port which was used to interface with Cortex-M3/M4 processors (as they contain an embedded AHB-AP interface). Even if the design didn't contain a Cortex-M3/M4, the Auxiliary Port was still present and needed to be tied off. So that when it was read during discovery, it would return a value indicating that no Cortex-M3/M4 was present. As with the regular APs, failure to tie off the Auxiliary Port usually breaks the auto-discovery process as it likely results in an unrecoverable DAP bus error.

4.3.5 MEM-APs for system memory access

Having a MEM-AP available for system memory access is a useful feature during chip bring-up and when debugging general issues after the bring-up process completes. There are no extra dangers to avoid when implementing a MEM-AP that have not already been addressed in the previous sections. This section has been added to highlight the benefits of having a MEM-AP.

The MEM-AP offers the debugger a method to access system memory which doesn't utilize the usual register-based mechanism through the processor. Normally the debugger accesses memory when the processor is in debug state by issuing a series of load or store instructions which are executed on the halted core. This has two notable drawbacks:

1. The processor must be halted
2. Large accesses could potentially be slow as transactions through the DAP will run at the JTAG test clock frequency **TCK**.

Conversely, MEM-AP accesses can be made while the core is running. This is a distinct advantage as it allows you to inspect and modify system state while the processor is running. In the situation of a hung processor, accesses over the MEM-AP may provide valuable information about the state of the system. If MEM-AP accesses fail while the system is hung, this can at least suggest on which memory interface the problem may exist. A secondary benefit of MEM-AP accesses is that they can be considerably faster than accesses through the core. This can prove beneficial if downloading a large image to memory. Note that the MEM-AP accesses physical memory while debug accesses through the core will utilize the MMU translation regime in effect. This can lead to coherency issues as accesses made over the MEM-AP bypass the cache(s) of the processor.

5 System Design Considerations

Even with a properly designed SoC, there is still ample opportunity for chip bring-up to fail as the board design also plays a critical role in processor debugging. This section addresses system level issues that are frequently out of the hands of the SoC design team and the responsibility of the end users of the processor.

This chapter covers elements of system design that impact the ability of a debugger to establish a bare metal connection to the processor. It contains the following sections:

5.1 JTAG circuitry and debug connectors

The JTAG circuit is used to interface the debug adapter (sometimes called emulator or run control unit) to the host computer running the debugger software. The debug adapter connects to the processor board through a header which in turn drives the JTAG circuit with commands from the debugger. The circuit then interfaces with a standard TAP controller in the DP portion of the DAP.

The DAP supports one or both of the following physical interfaces. The communication signals are shown in parenthesis:

- Serial Wire Debug (**SWDIO**, **SWDCLK**)
- JTAG (**TDI**, **TDO**, **TCK**, **TMS**)

Debug adapter headers come in many different pin outs depending on the debugger vendor selected. It is imperative that the board designer always follows the guidelines, described in the debugger adapter user guide, for designing the JTAG circuit. The user guide should include detailed information on signal pull-ups/pull-downs, layout considerations and signal termination.

For standard third-party boards provided, it is common to find one of three connectors defined by ARM: a CoreSight 10, CoreSight 20, or JTAG 20. These connectors might have slightly different names from tool vendors, but pinouts for these headers are readily available online. Table 5-1 highlights the primary differences between these connectors.

Table 5-1 Debug connectors

Property	CoreSight 10	CoreSight 20	JTAG 20
Pin Spacing	0.05”	0.05”	0.1”
JTAG or SWD	Both	Both	Both
Reset Signal(s)	**nRESET**	**nRESET**	**nSRST** and **nTRST**
Serial Wire Output	Yes	Yes	Yes
External Trace Support	No	Yes (4-bit)	No

Recommendations can vary between tool vendors, but generally each JTAG communication signal should have a pull-up or pull-down resistor to keep the signals stable when no adapter is connected. In addition, reset signals must be pulled high.

The JTAG 20 header supports a **DBGREQ** and **DBGACK** signal. These are used to request and acknowledge entry into debug state, but these signals are rarely brought out to the chip level by the processor. They also may not be supported by the debugger. If they are not brought out, they may be left as no connects. Otherwise they must be pulled low.

5.2 JTAG reset signals

The topic of processor reset signals was addressed in Section 4.2 Reset signals. Advanced processors usually include an integrated power/reset management controller (such as a Cortex-M core) which provides cold and warm reset capability, while a basic design may only expose a single reset signal for the entire processor. The board designer must understand the reset signals exposed by a particular SoC and design an appropriate reset circuit which includes the available JTAG reset signal(s) as shown in Table 5-1.

It is typical for a board design to have the **nRESET** or **nSRST** signal perform a system level reset. This resets the processor and other critical board-level components (such as other processors, coprocessors and memory controllers). This allows you to conveniently reset the system from a debugger. Remember that this very well could result in the equivalent of a power-on reset which prevents the ability of halting the processor with a hardware breakpoint or vector catch event as described in Section 4.2 Reset signals.

The **nRESET** and **nSRST** signals are bidirectional; this has the benefit of the debugger detecting if the system was reset by some other logic. An asynchronous reset event will likely force the debugger connection to drop as it is possible a transaction through the DAP was in progress. However it is possible that the debugger could immediately re-establish the processor connection without any negative effect for the end user.

If the JTAG 20 connector is used in JTAG operating mode, **nTRST** should be connected to the **nTRST** signal of the DAP. This allows the debugger to reset the TAP controller in the DAP if required. The DAP's **nTRST** signal may not be exposed at the pin level of the SoC, in which case this would be a no-connect.

5.3 JTAG/SWD clock frequency

The clock used for debugging (**SWDCLK** or **TCK**) is driven by the debug adapter and set by the debugger. The theoretical maximum frequency that a debug adapter supports will be documented in the adapter's user guide. A typical maximum frequency is 20 MHz for JTAG interfaces and 50 MHz for SWD interfaces. There is no formal flow control in communications with the DAP. Ultimately, operations can swamp the processor and lead to communications failure, which results in debugger connection failure.

The JTAG circuit, board layout and processor implementation play a role in the selection of an appropriate clock frequency. For example, a design in emulation or FPGA may only tolerate a very slow clock, such as 20kHz, as the processor will be running at a very slow frequency. A good approach is to start with a high clock frequency and gradually reduce it until you observe reliable JTAG communications.

Note

Legacy ARM processors (pre-CoreSight) had a **RTCK** signal which allowed for an adaptive JTAG clock to be driven by the debug adapter.

5.4 System memory at boot time

The processor boot process is often overlooked at early stages of SoC development and this can lead to frustrating debugger connection failures for the bring-up team. If no mechanism exists to generate a proper warm reset (as described in Section 4.2 Reset signals), then it is imperative that the processor begins execution with an initialized memory system that contains a valid program. If the processor attempts to execute what is essentially random opcodes stored in memory, it could lead to a memory access that hangs the processor, and prevents the debugger from halting the core.

For initial board bring-up, this often poses a Catch-22 dilemma – how can the board memory be programmed without a debugger connection? This is compounded by the fact that the processor may have internal non-volatile memory at the boot location. There are several solutions to this problem. For example, a system can provide alternative boot options which remap valid and initialized memory to the boot location. Another master in the system can program the boot memory or the debugger could even program RAM via the AXI-AP (if it is available and the system bus is not hung). In emulation or FPGA systems, the boot location will likely be implemented as RAM and the boot memory can be pre-initialized in the design with a valid program.

Remember that the requirement is not for a meaningful program to reside in memory. A simple “branch to self” instruction at the boot location is adequate for the debugger to halt the processor.