Cortex™-M0

Revision: r0p0

Integration and Implementation Manual

Confidential

ARM®

ARM DII 0238A (ID081009)

Cortex-M0
Integration and Implementation Manual

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
27 July 2009	A	Confidential	First release for r0p0

Proprietary Notice

Confidentiality Status

This document is Confidential. This document may only be used and distributed 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
Cortex-M0 Integration and Implementation Manual

List of Tables
Cortex-M0 Integration and Implementation Manual

List of Figures
Cortex-M0 Integration and Implementation Manual

Preface

This preface introduces the *Cortex-M0 Integration and Implementation Manual* (IIM). It contains the following sections:

About this guide

This guide is the IIM for the Cortex-M0 processor.

Note

Information from your EDA tool vendor might supersede some of the information in this manual. You must read this manual because it describes how to integrate and implement your device, and the steps you must take to sign your design off.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this guide, where:

r*n* Identifies the major revision of the product.

p*n* Identifies the minor revision or modification status of the product.

Intended audience

This guide is written for experienced hardware and *System-on-Chip* (SoC) engineers who might or might not have experience with ARM products. Such engineers typically have experience of writing Verilog and of performing synthesis, but might have limited experience of integrating and implementing ARM products.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this for a description of the processor design platforms and tools, including the supported design flow, directory structure, and design hierarchy.

Chapter 2 *Configuration Guidelines*

Read this for a description of the configuration options that you must be aware of before integrating and implementing the processor into your design.

Chapter 3 *Key Integration Points*

Read this for a description of the key points that you must consider when you integrate the Cortex-M0 processor with your SoC design.

Chapter 4 ***Functional Integration Guidelines***

Read this for guidelines on how to perform functional integration of the processor.

Chapter 5 ***Integration Kit***

Read this for a description of the Cortex-M0 integration kit.

Chapter 6 ***Key Implementation Points***

Read this for a description of the key points that you must consider when you implement the processor into your design.

Chapter 7 ***Floorplan Guidelines***

Read this for a description of floorplanning guidelines.

Chapter 8 ***Design For Test***

Read this for information about the design for test features.

Chapter 9 ***Netlist Dynamic Verification***

Read this for information about how to perform netlist dynamic verification on the processor.

Chapter 10 ***Sign-off***

Read this for a description of the ARM verification criteria, and how to sign off your design.

Appendix A ***GPIO Programmers Model***

Read this for a description of the *General Purpose Input/Output* (GPIO) programmers model.

Appendix B ***CM0IKMCU GPIO Integration***

Read this for a description of the example *Microcontroller Unit* (MCU) system supplied with the integration kit.

Appendix C ***SysTick Examples***

Read this for examples of how to set up the SysTick timer during integration.

Appendix D ***Debug Driver***

Read this for a description of the debug driver supplied with the integration kit.

Appendix E *Revisions*

Read this for a list of the technical changes between released issues of this book.

Glossary

Read this for definitions of terms used in this guide.

Conventions

Conventions that this book can use are described in:

- *Typographical*.

Typographical

The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.

Additional reading

This section lists publications by ARM and by third parties.

See `http://infocenter.arm.com` for access to ARM documentation. See `http://onarm.com` for other resources for embedded software developers, including the *Cortex Microcontroller Software Interface Standard* (CMSIS).

See the release note for details of EDA tool vendor documents.

ARM publications

This guide contains information that is specific to this product. See the following documents for other relevant information:

- *ARM AMBA® 3 AHB-Lite Protocol* (ARM IHI 0033)
- *ARM AMBA Design Kit Technical Reference Manual* (ARM DDI 0243)
- *ARM CoreSight™ Components Technical Reference Manual* (ARM DDI 0314)

- *ARMv6-M Architecture Reference Manual* (ARM DDI 0419)
- *Cortex-M0™ Technical Reference Manual* (ARM DDI 0432)
- *Cortex-M0 User Guide Reference Material* (ARM DUI 0467A).

Other publications

This section lists relevant documents published by third parties:

- *IEEE 1149.1-2001 IEEE Standard Test Access Port and Boundary Scan Architecture (JTAG)*.

Feedback

ARM welcomes feedback both on this product and its documentation.

Feedback on the product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms if appropriate.

Feedback on this document

If you have any comments about this guide, send an email to `errata@arm.com`. Give:

- the title
- the number, ARM DII 0238A
- the page number(s) to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter gives an overview of the process of integrating and implementing the Cortex-M0 processor. It contains the following sections:

1.1 About the processor

The Cortex-M0 processor is an energy-efficient processor with a very low gate count. It is intended to be used for microcontroller and deeply embedded applications that require an area-optimized processor.

The Cortex-M0 processor supports two levels of hierarchy for integration or implementation:

- processor component level, `CORTEXM0`
- integration level, `CORTEXM0INTEGRATION`.

Figure 1-1 shows the two levels and the main interfaces of the processor.

Note

You can modify the `CORTEXM0INTEGRATION` level for your own requirements.

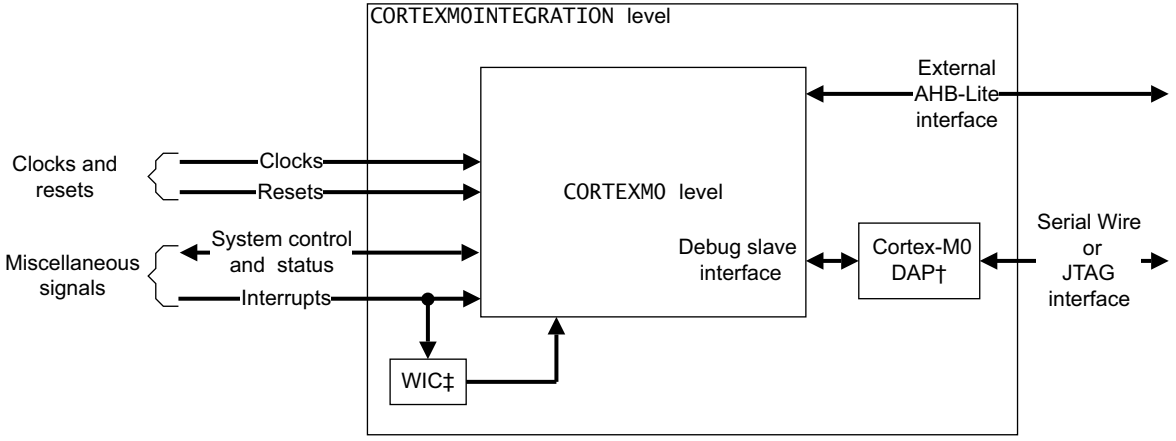


Figure 1-1 Module hierarchy and the main interfaces

CORTEXM0 component level

Contains the basic processor. This level is not modifiable.

CORTEXM0INTEGRATION level

Instantiates the CORTEXM0 component level and provides example integration with the Cortex-M0 *Debug Access Port* (DAP) and *Wakeup Interrupt Controller* (WIC). This level is provided as a working example of a single-processor sub-system. You can modify it to suit your requirements.

The Cortex-M0 processor is highly configurable. Although the top-level signals at both the CORTEXM0INTEGRATION and CORTEXM0 component levels are independent of configuration, the functionality of these interfaces is dependent on configuration. For example, the debug interfaces at either level are non-functional if your configuration does not include debug. For more details about configuring the CORTEXM0INTEGRATION or CORTEXM0 component levels, see Chapter 2 *Configuration Guidelines*.

The Cortex-M0 processor is supplied with two example implementation wrappers:

- CORTEXM0IMP for the CORTEXM0 level
- CORTEXM0INTEGRATIONIMP for the CORTEXM0INTEGRATION level.

These enable you to configure the Cortex-M0 processor to your requirements and to change the pinout to match the *State Retention Power Gating* (SRPG) requirements of your cell library.

1.2 About integration and implementation

The flows that you can use to integrate a Cortex-M0 processor into your SoC can vary depending on the capacity of your tools.

Figure 1-2 shows the implementation and integration flow.

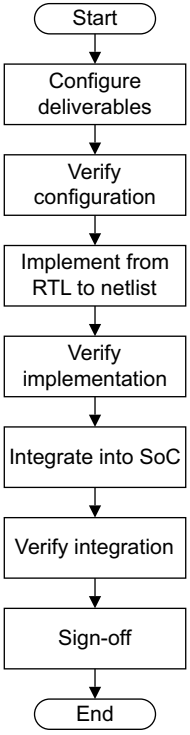


Figure 1-2 Implementation and integration flow

Figure 1-3 on page 1-5 shows the Cortex-M0 integration and implementation flow.

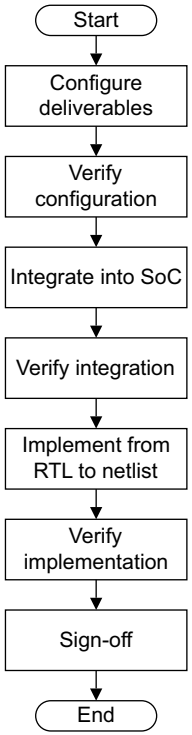


Figure 1-3 Integration and implementation flow

1.2.1 Integration

Integration is the first step in this process of including the processor in your SoC design. Integration involves:

- connecting the required clocks and resets to the processor
- connecting the processor to all the necessary peripherals and buses
- implementing the chosen test strategies for the processor
- verifying the processor within the SoC design.

Various implications arise for your design, depending on the elements that you include. You must consider:

- interfaces, especially the treatment of unused signals
- verification.

Although you might have additional elements in your design, the main connection is the AHB-Lite bus that conforms to the AMBA® 3 AHB-Lite Protocol.

1.2.2 Implementation

Figure 1-4 shows the implementation process, including the top-level inputs, resources, outputs, and controls and constraints for implementation.

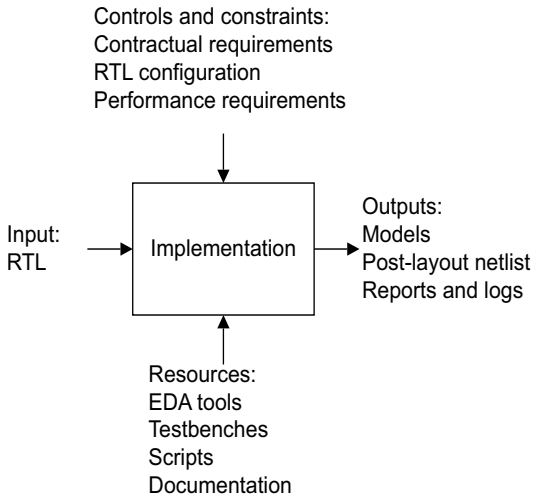


Figure 1-4 Implementation process

For an overview of the implementation process, see:

- *Implementation resources*
- *Implementation controls and constraints* on page 1-7
- *Implementation inputs* on page 1-7
- *Implementation flow* on page 1-8
- *Implementation outputs* on page 1-8

For information on the deliverables see:

- *Reference data* on page 1-11.

Implementation resources

This guide assumes that you have suitable EDA tools and compute resources for implementation. See the release note for a list of deliverables and any specific tool revisions required for implementation.

Note

The release note describes any special requirements that might affect the flow, such as details of any special tool requirements that enable optional flows within the implementation.

Implementation controls and constraints

Figure 1-4 on page 1-6 shows the general controls and constraints that apply to the implementation. You must implement the device in accordance with your contract, see *Implementation obligations* for more information.

Implementation obligations

The details set out in this guide are designed to help you implement the RTL into products. The extent to which the deliverables may be modified or disclosed is governed by the contract between ARM and Licensee. There may be validation requirements, which if applicable will be described in the contract between ARM and Licensee and which if present must be complied with prior to the distribution of any silicon devices incorporating the technology described in this document. Reproduction of this document is only permitted in accordance with the licences granted to Licensee.

ARM assumes no liability for your overall system design and performance. The verification procedures defined by ARM are only intended to verify the correct implementation of the licensed technology by ARM, and are not intended to test the functionality or performance of the overall system. You or the Licensee will be responsible for performing any system level tests.

You are responsible for any applications used in conjunction with the ARM technology described in this document, and to minimize risks adequate design and operating safeguards should be provided for by you. ARM's publication of any information in this document of information regarding any third party products or services is not an express or implied approval, or endorsement of the use thereof.

Implementation inputs

The release note describes deliverables that are inputs to the implementation flow. These deliverables include:

- *Register Transfer Level* (RTL) code
- implementation scripts
- documentation.

Implementation outputs

The outputs from the implementation flow are:

- Logs and reports:
  - synthesis logs and reports
  - post-layout *Static Timing Analysis* (STA) logs and reports
  - logs and reports showing logical equivalence of post-layout netlist with implemented RTL.
- Components:
  - post-layout netlist
  - GDSII
  - *Standard Delay Format* (SDF).
- Test:
  - *Automatic Test Pattern Generation* (ATPG) vectors.

Implementation flow

Figure 1-5 on page 1-9 shows the implementation flow.

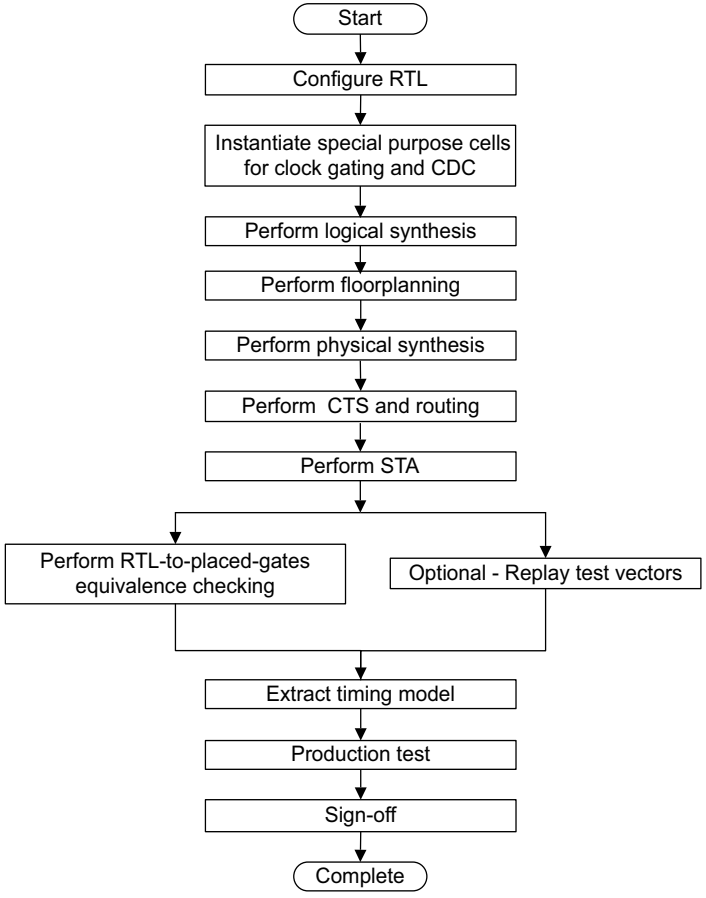


Figure 1-5 Implementation flow

Note

For information on contractual obligations to complete sign-off as part of the completed flow, see *Implementation controls and constraints* on page 1-7.

1.3 About sign-off

In addition to your normal sign-off checks, you must satisfy certain verification criteria before you sign off the design, see Chapter 10 *Sign-off* for more information.

1.4 Reference data

Before starting, you must ensure the unpacked deliverables are located in the correct directory structure. Figure 1-6 and Figure 1-7 on page 1-12 show the principal directory structure when you unpack the deliverables.

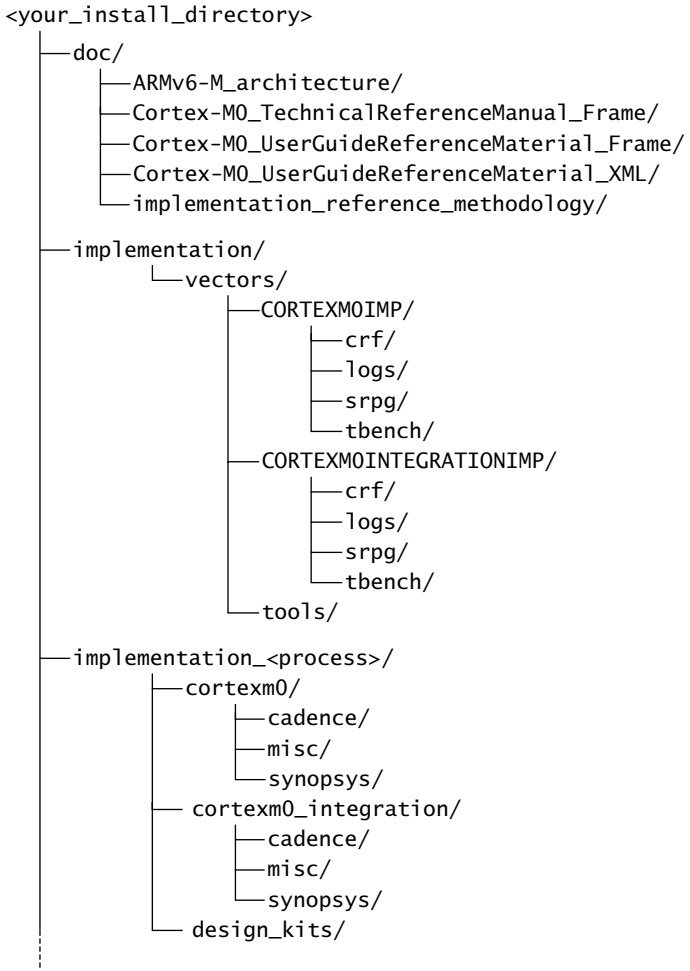


Figure 1-6 Directory structure, part one

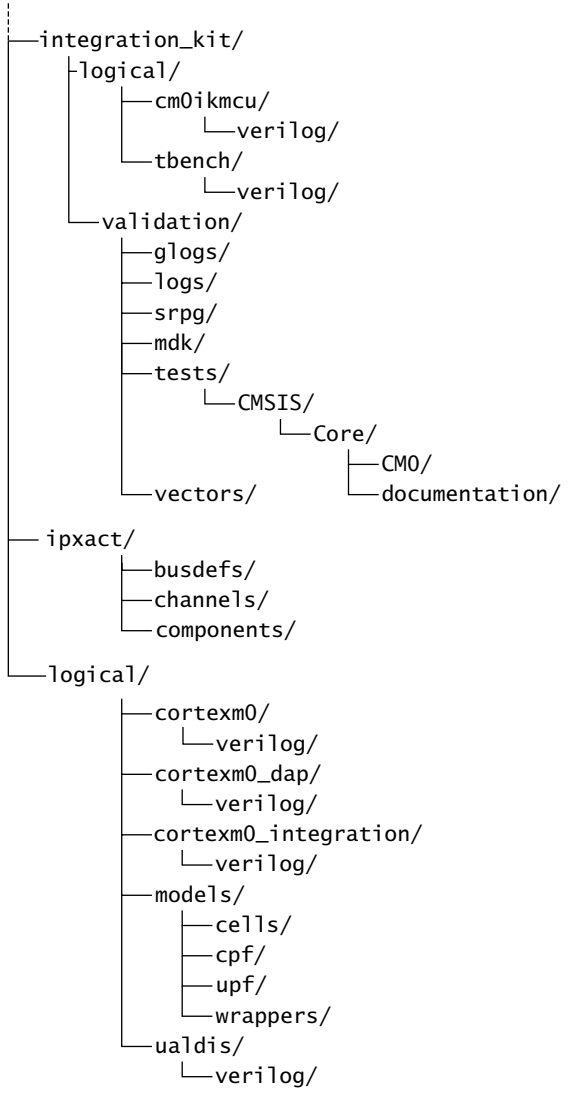


Figure 1-7 Directory structure, part two

Chapter 2
Configuration Guidelines

This chapter describes the guidelines for RTL configuration. These enable you to tailor the RTL to the specific requirements of the target application. It contains the following sections:

2.1 About configuration guidelines

Caution

To perform successful configuration of the RTL, you must correctly setup any configurable options.

Failure to complete the necessary configuration can result in malfunction.

The Cortex-M0 RTL includes two possible levels of RTL configuration:

- The `CORTEXM0` level implements the Cortex-M0 processor, the *Nested Vectored Interrupt Controller* (NVIC), breakpoint unit, watchpoint unit and associated debug control.
- The example `CORTEXM0INTEGRATION` level instantiates a `CORTEXM0` level pre-integrated with the WIC, and the Cortex-M0 Serial Wire or JTAG DAP.

The configurable options available depend on the level of hierarchy you want to implement.

At both the `CORTEXM0INTEGRATION` level and `CORTEXM0` level, the configurable options are controlled by parameters.

You can control the parameter values by defining them when instantiating the level of hierarchy you want to implement. Example implementation wrappers that instantiate the `CORTEXM0` and `CORTEXM0INTEGRATION` levels are provided in the files `CORTEXM0IMP.v` and `CORTEXM0INTEGATIONIMP.v` respectively. These files are located in the `logical/models/wrappers` directory. You can also use these implementation wrappers to specify additional inputs and outputs on the processor for the purposes of power gating or test.

The Reference Methodology deliverables include a copy of the implementation wrappers that might have implementation flow specific modifications. See the Reference Methodology release notes for the location of the implementation wrappers used for implementation. Use these files in place of the files in the `logical/models/wrappers` directory.

2.2 Configuration options

The following sections describe the Cortex-M0 processor configuration options:

- *CORTEXM0 configuration options*
- *CORTEXM0INTEGRATION configuration options* on page 2-6.

2.2.1 CORTEXM0 configuration options

Table 2-1 shows the configuration options summary for `CORTEXM0`.

Table 2-1 CORTEXM0 options summary

Parameter	Default value	Supported values	Description
`ACG`	1	0, 1	Specifies if internal architectural clock gates are included to minimize dynamic power dissipation: • `0` means architectural clock gates are excluded • `1` means architectural clock gates are included. **Note** If you use architectural clock gating, you must replace the provided model with a direct instantiation of a clock gating cell from your target cell library.
`AHBSLV`	1	0, 1	Specifies the bus protocol implemented on the SLV port. This is a debug port. See Chapter 4 *Functional Integration Guidelines* for additional information: • `0` means the SLV port implements a Cortex-M0 DAP specific protocol • `1` means the SLV port implements a subset of AHB-Lite. ARM recommends that you set this parameter to `1` if you implement the Cortex-M0 level for integration with any debug infrastructure other than the Cortex-M0 DAP, for example, the generic CoreSight DAP.
`BE`	0	0, 1	Specifies the endianness for processor data transfers: • `0` means little-endian • `1` means byte-invariant big-endian.
`BKPT`	4	0, 1, 2, 3, 4	Specifies the number of breakpoint unit comparators implemented. **Note** If debug is excluded, this parameter has no effect. See the `DBG` parameter description in this table.

Table 2-1 CORTEXM0 options summary (continued)

Parameter	Default value	Supported values	Description
DBG	1	0, 1	Specifies whether or not the debug extensions are implemented: • 0 means debug is not functional • 1 means debug is functional. The configuration of debug is controlled additionally by the BKPT and WPT parameters. **Note** Setting the DBG parameter to 0 can result in the **DCLK** input becoming unloaded.
NUMIRQ	32	1, 2, 4, 8, 16, 24, 32	Specifies the number of external interrupts implemented: • 1 means only **IRQ[0]** is functional . . . • 32 means **IRQ[31:0]** are all functional.
RAR	0	0, 1	Specifies whether all synchronous state or only architecturally required state is reset: • 0 means that only architecturally required state is reset • 1 means all synchronous state is reset. **Note** When RAR is 1 all registers in the design can be reset, incurring an area penalty. When RAR is 0 the registers in the design that do not require to be reset have no reset.
SMUL	0	0, 1	Specifies the implemented multiplier: • 0 includes the fast, single-cycle multiplier • 1 includes the small, 32-cycle multiplier.
SYST	1	0, 1	Specifies whether or not the SysTick timer functionality is included: • 0 means the SysTick timer is excluded • 1 means the SysTick timer is included.

Table 2-1 CORTEXM0 options summary (continued)

Parameter	Default value	Supported values	Description
WIC	1	0, 1	Specifies whether or not the WIC interface is implemented: • 0 means the WIC interface is excluded • 1 means the WIC interface is included.
WICLINES	34	2-34	Specifies the lines supported by the WIC interface: • 2 means only **NMI** and **RXEV** are supported • 3 means **NMI**, **RXEV** and **IRQ[0]** are supported • 4 means **NMI**, **RXEV** and I**RQ[1:0]** are supported . . . • 34 means **NMI**, **RXEV** and **IRQ[31:0]** are supported. **Note** If the WIC interface is excluded, this parameter has no effect. See the WIC parameter description in this table.
WPT	2	0, 1, 2	Specifies the number of watchpoint unit comparators implemented. **Note** If debug is excluded, this parameter has no effect. See the DBG parameter description in this table.

2.2.2 CORTEXM0INTEGRATION configuration options

Table 2-2 shows the configuration options summary for CORTEXM0INTEGRATION.

Table 2-2 CORTEXM0INTEGRATION options summary

Parameter	Default value	Supported values	Description
ACG	1	0, 1	Specifies if internal architectural clock gates are included to minimize dynamic power dissipation: • 0 means architectural clock gates are excluded • 1 means architectural clock gates are included. **Note** If you use architectural clock gating, you must replace the provided model with a direct instantiation of a clock gating cell from your target cell library.
BE	0	0, 1	Specifies the endianness for data transfers: • 0 means little-endian • 1 means byte-invariant big-endian.
BKPT	4	0, 1, 2, 3, 4	Specifies the number of breakpoint unit comparators implemented. **Note** If debug is excluded, this parameter has no effect. See the DBG parameter description in this table.
DBG	1	0, 1	Specifies whether or not debug is included: • 0 means debug is excluded • 1 means debug is included. The configuration of debug is controlled additionally by the BKPT and WPT parameters. **Note** Setting the DBG parameter to 0 can result in the **DCLK** and **SWCLKTCK** inputs becoming unloaded.

Table 2-2 CORTEXM0INTEGRATION options summary (continued)

Parameter	Default value	Supported values	Description
JTAGnSW [a]	0	0, 1	Specifies the external debug protocol implemented by the Cortex-M0 DAP: • 0 means Serial Wire is implemented • 1 means JTAG is implemented. **Note** If debug is excluded, this parameter has no effect. See the DBG parameter description in this table.
NUMIRQ	32	1, 2, 4, 8, 16, 24, 32	Specifies the number of external interrupts implemented.: • 1 means only **IRQ[0]** is functional ... • 32 means only **IRQ[31:0]** are all functional.
RAR	0	0, 1	Specifies whether all synchronous state or only architecturally required state is reset: • 0 means that only architecturally required state is reset • 1 means all synchronous state is reset. **Note** When RAR is 1, all registers in the design can be reset, incurring an area penalty. When RAR is 0, the registers in the design that do not require to be reset have no reset.
SYST	1	0,1	Specifies whether or not the SysTick timer functionality is included: • 0 means the SysTick timer is excluded • 1 means the SysTick timer is included.
SMUL	0	0, 1	Specifies the implemented multiplier: • 0 includes the fast, single-cycle multiplier • 1 includes the small, 32-cycle multiplier.

Table 2-2 CORTEXM0INTEGRATION options summary (continued)

Parameter	Default value	Supported values	Description
WIC	1	0, 1	Specifies whether or not the WIC is included: • 0 means the WIC is excluded • 1 means the WIC is included.
WICLINES	34	2, 3, 4, 34	Specifies the number of lines supported by the WIC: • 2 means only **NMI** and **RXEV** are supported • 3 means **NMI**, **RXEV** and **IRQ[0]** are supported • 4 means **NMI**, **RXEV** nd I**RQ[1:0]** are supported . . . • 34 means **NMI**, **RXEV** and **IRQ[31:0]** are supported. **Note** If the WIC is excluded, this parameter has no effect. See the WIC parameter description in this table.
WPT	2	0, 1, 2	Specifies the number of watchpoint unit comparators implemented. **Note** If debug is excluded, this parameter has no effect. See the DBG parameter description in this table.

a. The JTAGnSW parameter exists only for the Cortex-M0 DAP inside the example CORTEXM0INTEGRATION level. It can cease to exist or function if you change the CORTEXM0INTEGRATION level.

2.2.3 CoreSight ROM table base address

Note

This section applies only if you have configured the Cortex-M0 processor to include debug.

The Cortex-M0 processor includes a CoreSight-compliant ROM table that provides pointers to the memory-mapped NVIC and debug resources. See the *Cortex-M0 Technical Reference Manual* for more information on the Cortex-M0 ROM table and its functionality.

This ROM table is fixed in the processor memory map at address `0xE00FF000`. You must ensure that this ROM table is either set up as the single master ROM table in your system, or is referenced by another valid CoreSight ROM table in your system:

- If you are integrating the `CORTEXM0INTEGRATION` level, the single master ROM table in the system is preconfigured to be the processor ROM table at `0xE00FF000`. If you want to change this, you can configure the base address of the master ROM table in the system by driving the **BASEADDR** port of the Cortex-M0 DAP to the value:

  ```
  {ADDRESS[31:12], 12'h003}
  ```

  where `ADDRESS` is the 4KB aligned base address of your master ROM table and **BASEADDR** bits [1:0] indicate that the pointer is present and in 32-bit format.

- If you are integrating the `CORTEXM0` level, see the *CoreSight Components Technical Reference Manual* for information about how to configure your ROM tables to ensure that the Cortex-M0 ROM table is appropriately referenced.

The integration kit includes both an example system ROM table that correctly points to the Cortex-M0 ROM table and a test to check correct ROM table structure in the system. See Chapter 5 *Integration Kit* for more information.

Chapter 3
Key Integration Points

This chapter describes the key integration points you must consider when you integrate the processor into your SoC design. It contains the following sections:

3.1 About key integration points

This chapter contains a list of the main points to consider when you integrate the Cortex-M0 processor into your SoC design. You must read this chapter in conjunction with the rest of the information in this book, and the information referred to in *Additional reading* on page xv.

Although you can use this chapter to check that you have covered the integration steps described in the other chapters, you must complete all the integration steps described in the later chapters.

3.2 Key integration tasks

Table 3-1 lists the CORTEXM0 component level key integration tasks.

Table 3-1 CORTEXM0 component level key integration tasks

Key task		Description
1.	Connect the **SCLK**, **HCLK**, and **DCLK**[a] clocks correctly.	See *Clocks* on page 4-3
2.	Connect the **HRESETn** and **DBGRESETn**[a] resets correctly.	See *Resets* on page 4-7
3.	Tie off or connect the following interface inputs appropriately: • external AHB-Lite interface • AHB interface extensions • interrupt interface • debug slave interface • miscellaneous signals • SysTick signals • WIC interface.	See *Interfaces* on page 4-9
4.	Tie off the CoreSight ROM table base address.[b]	See Chapter 2 *Configuration Guidelines*
5.	Verify your design using the integration kit.	See Chapter 5 *Integration Kit*

a. Tie LOW if debug is not included.
b. This is only applicable if you have configured the Cortex-M0 processor to include debug.

Table 3-2 lists the CORTEXM0INTEGRATION level key integration tasks.

Table 3-2 CORTEXM0INTEGRATION level key integration tasks

Key task		Description
1.	Connect the **FCLK**, **SCLK**, **HCLK**, **DCLK**[a], and **SWCLKTCK**[a] clocks correctly.	See *Clocks* on page 4-3
2.	Connect the **PORESETn**, **HRESETn**, **DBGRESETn**[a], and **nTRST**[b] resets correctly.	See *Resets* on page 4-7
3.	Tie off or connect the following interface inputs appropriately: • external AHB-Lite interface • AHB interface extensions • interrupt interface • External debugger interface • miscellaneous signals • SysTick signals • *Power Management Unit* (PMU) interface.	See *Interfaces* on page 4-9
4.	Tie off the CoreSight ROM table base address.[c]	See Chapter 2 *Configuration Guidelines*
5.	Verify your design using the integration kit.	See Chapter 5 *Integration Kit*

a. Tie LOW if debug is not included.
b. Tie LOW if debug or Serial Wire is not included. Tie HIGH or tie to **PORESETn** if **nTRST** is not used and JTAG is included.
c. This is only applicable if you have configured the Cortex-M0 processor to include debug.

Chapter 4
Functional Integration Guidelines

This chapter describes the guidelines for functional integration of the processor into your SoC design. It contains the following sections:

4.1 About functional integration

This chapter contains information that you must consider before or during the integration of the processor into your SoC design.

4.2 Clocks

This section provides information on how to use the Cortex-M0 processor clocks in your SoC design in:

- *CORTEXM0 level clocks* on page 4-4
- *CORTEXM0INTEGRATION level clocks* on page 4-5
- *Clock gating combinations* on page 4-6.

Figure 4-1 shows the Cortex-M0 clock domains.

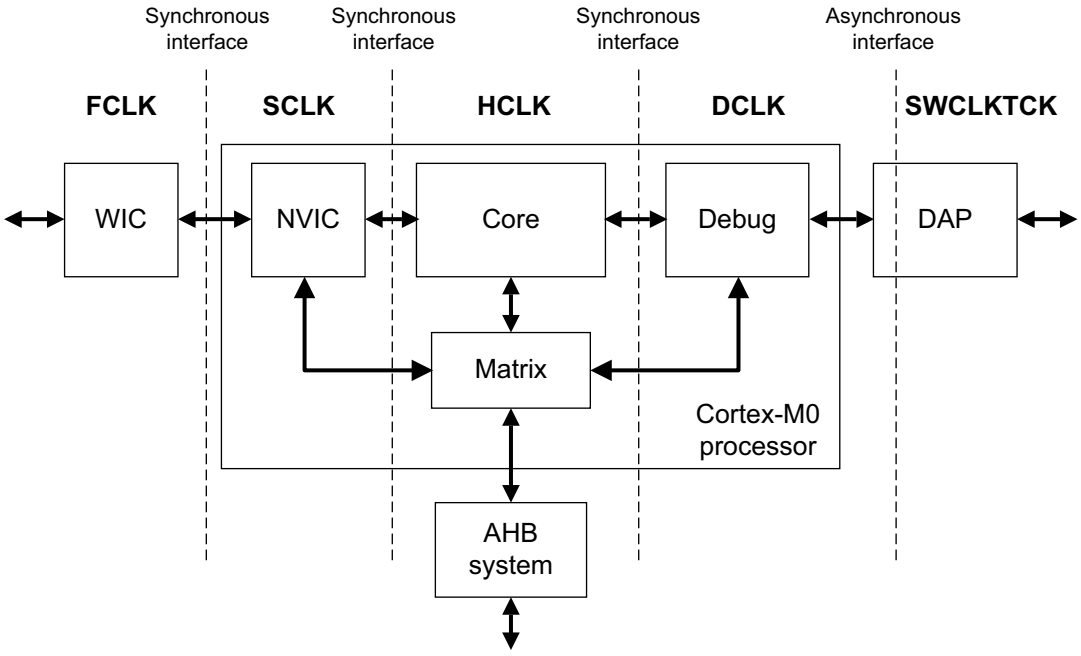


Figure 4-1 Cortex-M0 clock domains

4.2.1 CORTEXM0 level clocks

Table 4-1 shows the clocks at the CORTEXM0 level of hierarchy.

Table 4-1 CORTEXM0 level clocks

Name	Direction	Description	Connection information
SCLK	Input	Free running clock that clocks a small amount of logic in the processor system domain	**SCLK** must always be running unless the processor is in WIC-mode deep sleep and no debugger is connected. When the processor is in deep sleep, you can gate the other clocks and run **SCLK** at reduced frequency.
HCLK	Input	Clock for the majority of the non-debug logic in the processor system domain.	**HCLK** must be derived directly from **SCLK**. Connect **HCLK** to the AHB layer that the processor is connected to. **HCLK** can be gated when the processor is sleeping and no debugger is connected. You can use **SLEEPING** to determine when the processor is sleeping.
DCLK	Input	Clock for the processor debug domain	**DCLK** must be derived directly from **SCLK**. **DCLK** must always be driven while a debugger is connected. It can be gated when no debugger is connected. You can use the **CDBGPWRUPACK** output from your system PMU to detect the presence of an external debugger requesting a connection.

Note

- Although under certain conditions, **HCLK** and **DCLK** can be gated versions of **SCLK** to minimize dynamic power dissipation, when active, **HCLK**, **DCLK** and **SCLK** must all be synchronous and balanced to a ratio of 1:1.
- When a debugger is connected, **HCLK**, **DCLK** and **SCLK** must all be running.
- **DCLK** must be tied LOW if the Cortex-M0 processor is configured without debug.

4.2.2 CORTEXM0INTEGRATION level clocks

Table 4-2 shows the clocks of the CORTEXM0INTEGRATION level of hierarchy.

Table 4-2 CORTEXM0INTEGRATION level clocks

Name	Direction	Description	Connection information
FCLK	Input	Free running clock that clocks the WIC.	**FCLK** must always be running.
SCLK	Input	System domain clock. This clocks a small amount of logic in the processor system domain.	**SCLK** must be derived directly from **FCLK**. It must always be running unless the processor is in WIC-mode deep sleep and no debugger is connected.
HCLK	Input	Clock for the majority of the non-debug logic in the processor system domain.	**HCLK** must be derived directly from **FCLK**. Connect **HCLK** to the AHB layer that the processor is connected to. **HCLK** can be gated when the processor is sleeping and no debugger is connected. You can use **GATEHCLK** to determine when you can gate **HCLK**.
DCLK	Input	Clock for the processor debug domain.	**DCLK** must be derived directly from **FCLK**. **DCLK** must always be driven while a debugger is connected. It can be gated when no debugger is connected. You can use the **CDBGPWRUPACK** input to the CORTEXM0INTEGRATION level to detect the presence of an external debugger requesting a connection.
SWCLKTCK	Input	Clock for the JTAG or Serial Wire interface.	**SWCLKTCK** is typically driven by an external debugger and is completely asynchronous to **FCLK**, **HCLK**, **SCLK**, and **DCLK**.

The example PMU provided meets the clocking requirements at the CORTEXM0INTEGRATION level.

Note

- Although under certain conditions, **HCLK**, **SCLK**, and **DCLK** can be gated versions of **FCLK** to minimize dynamic power dissipation, when active, **HCLK**, **SCLK**, **DCLK**, and **FCLK** must all be synchronous, and balanced to a ratio of 1:1.
- When a debugger is connected, **HCLK**, **DCLK** and **SCLK** must all be running.

- **DCLK** and **SWCLKTCK** must be tied LOW if the Cortex-M0 processor is configured without debug.

4.2.3 Clock gating combinations

Figure 4-2 shows the legal clock gating combinations.

Gated \ Ungated	FCLK	SCLK	HCLK	DCLK
FCLK	-	invalid	invalid	invalid
SCLK	WIC sleep†	-	invalid	invalid
HCLK	WIC sleep†	Core sleeping‡	-	invalid
DCLK	WIC sleep†	No debugger connected¶	No debugger connected¶	-

† **SLEEPDEEP** is HIGH, **WICDSACKn** is LOW, **WAKEUP** is LOW, and no debugger is connected.
‡ **SLEEPING** is HIGH and no debugger is connected.
¶ **CDBGPWRUPACK** is LOW.

Figure 4-2 Clock gating combinations

4.3 Resets

This section describes considerations for connecting and using the Cortex-M0 processor resets.

Before connecting resets in your system, you must consider the following:

- Assertion of **SYSRESETREQ** must only cause an assertion of **HRESETn** and not **DBGRESETn**. This permits a debugger to maintain a connection during a processor reset.
- Register **SYSRESETREQ** to ensure that **HRESETn** is driven for the minimum reset time for your SoC design. **SYSRESETREQ** is cleared asynchronously by **HRESETn**. Do not connect **SYSRESETREQ** directly to **HRESETn**.

4.3.1 CORTEXM0 level resets

Table 4-3 shows the resets at the `CORTEXM0` level of hierarchy.

Table 4-3 CORTEXM0 level resets

Name	Direction	Description	Connection information
HRESETn	Input	Reset for the processor system domain and the AHB system	Deassert **HRESETn** synchronously to **SCLK**. Assert **HRESETn** on power-on. Assert **HRESETn** for at least two **HCLK** cycles.
DBGRESETn	Input	Reset for the processor debug domain	Deassert **DBGRESETn** synchronously to **SCLK**. Assert **DBGRESETn** on power-on. Assert **DBGRESETn** for at least two **DCLK** cycles. Tie **DBGRESETn** LOW when no debugger is connected.

There are no reset synchronizers within the `CORTEXM0` level.

4.3.2 CORTEXM0INTEGRATION level resets

Table 4-4 shows the resets of the CORTEXM0INTEGRATION level of hierarchy.

Table 4-4 CORTEXM0INTEGRATION level resets

Name	Direction	Description	Connection information
HRESETn	Input	Reset for the processor system domain and the AHB system	Deassert **HRESETn** synchronously to **FCLK**. Assert **HRESETn** on power-on. Assert **HRESETn** for at least two **HCLK** cycles.
DBGRESETn	Input	Reset for the processor debug domain	Deassert **DBGRESETn** synchronously to **FCLK**. Assert **DBGRESETn** on power-on. Assert **DBGRESETn** for at least two **DCLK** cycles.
nTRST	Input	Reset for the JTAG TAP controller in the JTAG *Debug Port* (DP)	**nTRST** can be tied HIGH when a synchronous JTAG reset is provided through the **TMS** pin. If implemented, **nTRST** must not be synchronized to **SWCLKTCK**. Tie **nTRST** LOW if JTAG is not configured.
PORESETn	Input	Power-on reset for the JTAG or Serial Wire DP logic	Deassert **PORESETn** synchronously to **FCLK**. External debuggers cannot connect when **PORESETn** is asserted.

The example reset controller provided meets the reset requirements at the CM0IKMCU level. See *CM0IKMCU level* on page 5-23 for more information.

4.4 Interfaces

This section describes the interfaces of the Cortex-M0 processor in:

- *External AHB-Lite interface*
- *AHB interface extensions* on page 4-12
- *AHB memory considerations* on page 4-14
- *Interrupt interface* on page 4-14
- *External debugger interface* on page 4-15
- *Debug slave interface* on page 4-17
- *Miscellaneous signals* on page 4-17
- *SysTick signals* on page 4-20
- *WIC interface* on page 4-22
- *Power Management Unit interface* on page 4-24.

4.4.1 External AHB-Lite interface

This interface is present at both the `CORTEXM0` and `CORTEXM0INTEGRATION` levels.

Accesses on this interface can originate from processor-initiated transactions or transactions requested by an external debugger.

ARM recommends that you understand the AHB-Lite bus interface signals, described in the *AMBA 3 AHB-Lite Protocol Specification*, before connecting any of the signals described in this section.

Table 4-5 shows the External AHB-Lite interface.

Table 4-5 External AHB-Lite signals

Name	Direction	Connection information
HADDR[31:0]	Output	Connect to address decoders, arbiter, and slaves through the bus infrastructure
HBURST[2:0]	Output	Connect to the AHB arbiter and slaves through the bus infrastructure
HPROT[3:0]	Output	Connect to the slaves through the bus infrastructure
HSIZE[2:0]	Output	
HTRANS[1:0]	Output	Connect to the AHB arbiter and slaves through the bus infrastructure
HWDATA[31:0]	Output	Connect to the slaves through the bus infrastructure

Table 4-5 External AHB-Lite signals (continued)

Name	Direction	Connection information
HWRITE	Output	Connect to the slaves through the bus infrastructure
HMASTLOCK	Output	
HRDATA[31:0]	Input	Connect to the slaves through the bus infrastructure
HREADY	Input	
HRESP	Input	

See the *AMBA 3 AHB-Lite Protocol Specification* for more information on the External AHB-Lite interface and its signals.

AHB-Lite interface transaction types

Table 4-6 shows the Cortex-M0 processor generated AHB-Lite interface transaction types.

Table 4-6 AHB-Lite interface transaction types

Transaction	HTRANS[1:0]	HPROT[1:0]	HMASTER[a]	HSIZE[2:0]
Bus idle	00	1X	X	0XX
Core instruction fetch	10	10	0	010
Core word load/store	10	11	0	010
Core half-word load/store	10	11	0	001
Core byte load/store	10	11	0	000
Debug word read/write	10	11	1	010
Debug half-word read/write	10	11	1	001
Debug byte read/write	10	11	1	000

a. **HMASTER** can be used by peripherals to distinguish between accesses from the processor core and accesses from the debugger.

HPROT[3:2] derivation is based on the attributes defined by the memory map of the processor. They are applied to both processor and debugger transactions. See the *ARMv6-M Architecture Reference Manual* for more information.

Table 4-7 shows the **HPROT[3:2]** memory types.

Table 4-7 Memory types for HPROT[3:2]

Address	Memory type	HPROT[3:2]	Recommended usage
0xF0000000 - 0xFFFFFFFF	Device XN[a]	01	CoreSight ROM tables[b]
0xE0000000 - 0xEFFFFFFF	Reserved	-	System control space
0xA0000000 - 0xDFFFFFFF	Device XN	01	Peripherals
0x80000000 - 0x9FFFFFFF	Normal WT[c]	10	Off chip RAM
0x60000000 - 0x7FFFFFFF	Normal WBWA[d]	11	Off chip RAM
0x40000000 - 0x5FFFFFFF	Device XN	01	Peripherals
0x20000000 - 0x3FFFFFFF	Normal WBWA	11	On chip RAM
0x00000000 - 0x1FFFFFFF	Normal WT[e]	10	ROM or flash

a. XN means execute-never.
b. See *CoreSight ROM table base address* on page 2-8 for more information.
c. WT means write-through.
d. WBWA means write-back-write-allocate.
e. WT means write-through.

AHB-Lite byte lane definition

The Cortex-M0 processor AHB interface uses the byte lanes defined for a little-endian system in the *AMBA 3 AHB-Lite Protocol Specification*, regardless of the endianness configuration of the processor. Table 4-8 shows the byte lanes the interface uses during the data phase on **HRDATA** or **HWDATA**, and the mapping to bytes in the source or destination processor core register, Rd, for all transfer sizes and each endianness configuration.

Table 4-8 AHB-Lite byte lane assignments

Address phase			Corresponding data phase			
HSIZE [1:0]	HADDR [1:0]	Processor endianness	HxDATA [31:24]	HxDATA [23:16]	HxDATA [15:8]	HxDATA [7:0]
00	00	-	-	-	-	Rd[7:0]
00	01	-	-	-	Rd[7:0]	-
00	10	-	-	Rd[7:0]	-	-

Table 4-8 AHB-Lite byte lane assignments (continued)

Address phase			Corresponding data phase			
HSIZE [1:0]	**HADDR [1:0]**	**Processor endianness**	**HxDATA [31:24]**	**HxDATA [23:16]**	**HxDATA [15:8]**	**HxDATA [7:0]**
00	11	-	Rd[7:0]	-	-	-
01	00	Little-endian	-	-	Rd[15:8]	Rd[7:0]
01	00	Big-endian	-	-	Rd[7:0]	Rd[15:8]
01	10	Little-endian	Rd[15:8]	Rd[7:0]	-	-
01	10	Big-endian	Rd[7:0]	Rd[15:8]	-	-
10	00	Little-endian	Rd[31:24]	Rd[23:16]	Rd[15:8]	Rd[7:0]
10	00	Big-endian	Rd[7:0]	Rd[15:8]	Rd[23:16]	Rd[31:24]

Note

- **HBURST** is always `3'b000`.
- **HMASTLOCK** is always LOW because the Cortex-M0 processor does not perform any locked transactions.
- Instruction fetches are always performed as 32-bit aligned word accesses.

4.4.2 AHB interface extensions

This interface is present at both the `CORTEXM0` and `CORTEXM0INTEGRATION` levels.

You can use AHB interface extensions in conjunction with the AHB interface to provide efficient connection to memory controllers and other peripherals. If this interface is not in use, you can leave it unconnected.

Table 4-9 shows the AHB interface extensions.

Table 4-9 AHB interface extensions

Name	Direction	Description
HMASTER	Output	Enables the system to distinguish between processor initiated and debugger initiated accesses: `1` = Debugger access `0` = Processor access.
CODENSEQ	Output	This signal is valid when **HTRANS** is non-zero and **HPROT[0]** is LOW. It provides an augmentation to **HTRANS** to enable more efficient detection of sequential instruction fetches. When set to `0`, it indicates that the current instruction fetch is from an address four bytes greater than the last instruction fetch. When set to `1`, it indicates that the current instruction fetch is non-sequential from the previous instruction fetch. This signal does not take interleaved data accesses or debugger accesses into account. This signal can be used to connect to the instruction prefetchers in the system.
CODEHINTDE[2:0]	Output	Prefetch hint bus. **CODEHINTDE[2]** This indicates a branch-like instruction in decode. See the *Cortex-M0 Technical Reference Manual* for more information. **CODEHINTDE[1]** This indicates a backward unresolved conditional branch in decode. **CODEHINTDE[0]** This indicates a forwards unresolved conditional branch in decode. Use these signals to connect to a prefetch unit and influence prefetch decisions.

Table 4-9 AHB interface extensions (continued)

Name	Direction	Description
SPECHTRANS	-	A faster, speculative version of **HTRANS[1]**, active in the same cycle as **HTRANS[1]**. This signal is used to provide a version of **HTRANS[1]** to read-only slaves if timing closure on **HTRANS[1**] is problematic. This signal indicates the processor is ready to do a transaction, but does not take into account suppression of **HTRANS[1]**. This suppression can be caused by: • attempted unaligned transactions • attempted instruction fetches from `Execute-Never`, `XN`, regions • transactions that access the internal PPB. If you use **SPECHTRANS**, be sure to maintain AHB compliance. For example, it is not acceptable to introduce an **HREADY** wait-state in response to a **SPECHTRANS** for which **HTRANS[1]** does not also become asserted.

AMBA Considerations

You must abide by the rules of AMBA3 AHB-Lite irrespective of the memory interface extension interface.

4.4.3 AHB memory considerations

AHB memories implemented for the purpose of executing code must be both byte and half-word data readable for both instruction and data transactions. A slave can respond with word data.

Writable AHB memories must be capable of accepting byte writes without corrupting neighboring bytes.

4.4.4 Interrupt interface

This interface is present at both the `CORTEXM0` and `CORTEXM0INTEGRATION` levels.

Table 4-10 shows the signals of the external interrupt interface, and describes how you must connect the signals in your SoC design.

Table 4-10 Interrupt signals

Name	Direction	Description	Connection information
IRQ[31:0]	Input	External interrupt signals	When the processor is implemented with fewer than 32 external interrupts, the unused **IRQ** inputs must be tied LOW. When correctly configured by software, the Cortex-M0 processor takes an interrupt in response to an **IRQ** input either being held HIGH, or being HIGH for a single cycle of: • **FCLK** for the `CORTEXM0INTEGRATION` level configured to include WIC support for the respective **IRQ** line • **SCLK** for all other cases.
NMI	Input	Non-maskable interrupt	Irrespective of software configuration, the Cortex-M0 processor takes an NMI exception in response to **NMI** being held HIGH, or being HIGH for a single cycle of: • **FCLK** for the `CORTEXM0INTEGRATION` level configured to include WIC support for the **NMI** • **SCLK** for all other cases.

The Cortex-M0 processor does not implement synchronizers for the **IRQ[31:0]** and **NMI** inputs. If you want to use asynchronous interrupts, you must implement external synchronizers to reduce the possibility of metastability issues.

4.4.5 External debugger interface

The external debugger interface is only present at the `CORTEXM0INTEGRATION` level.

It enables a compliant, off-chip debugger to gain access to the processor control resources and external memory.

Although you can configure `CORTEXM0INTEGRATION` to implement either a JTAG debugger interface or a Serial Wire debugger interface, you cannot implement both.

Note

- If you require both Serial Wire and JTAG interfaces, you can connect a suitable DAP, like the CoreSight DAP, to the debug slave interface instead of the CORTEXM0DAP. If this is the case, you must ensure that `AHBSLV` is configured correctly for your DAP.

- If your processor is not configured for debug, you must correctly tie off the JTAG or Serial Wire debugger interface signals.

For more information on integrating:
- the JTAG interface, see *JTAG signals*
- the Serial Wire interface, see *Serial Wire signals*.

JTAG signals

Table 4-11 lists the JTAG interface signals.

For more information see *IEEE 1149.1-2001 Test Access Port and Boundary Scan Architecture*.

Table 4-11 JTAG signals

Name	Direction	Description	Connection information
nTRST	Input	JTAG nTRST	See *Resets* on page 4-7
TDI	Input	JTAG TDI	Connect to input pad for **TDI**
SWDITMS	Input	JTAG TMS	Connect to input pad for **TMS**
SWCLKTCK	Input	JTAG TCK	Connect to input pad for **TCK**
TDO	Output	JTAG TDO	Connect to optional tristate pad for **TDO**
nTDOEN	Output	JTAG TDO output pad control signal	Connect to optional tristate pad for **TDO**

Serial Wire signals

Table 4-12 lists the Serial Wire signals.

Table 4-12 Serial Wire signals

Name	Direction	Description	Connection information
SWDO	Output	Serial Wire Data Out	Connect to tristate pad for **SWDIO**
SWDOEN	Output	Serial Wire Data Out output pad control signal	Connect to tristate pad for **SWDIO**
SWCLKTCK	Input	Serial Wire Clock	Connect to input pad for **TCK**
SWDITMS	Input	Serial Wire Data In	Connect to tristate pad for **SWDIO**

4.4.6 Debug slave interface

The debug slave interface is only present at the CORTEXM0 hierarchy level. It enables memory-mapped debugger access to processor control resources and system memory.

Table 4-13 shows the signals for the debug slave interface on the Cortex-M0 processor. These signals must be connected to either the Cortex-M0 DAP or the AHB-AP port of a CoreSight DAP.

If a system does not implement debug, tie all slave port inputs LOW.

The Cortex-M0 processor debug slave port behavior is configurable through the AHBSLV parameter. This parameter is set at implementation time. See *Configuration options* on page 2-3 for more information about the options for AHBSLV.

Table 4-13 AHB-AP interface signals

Name	Direction	Connection information
SLVRDATA[31:0]	Output	Connect to the **SLVRDATA** port of the Cortex-M0 DAP, or the **HRDATA** port of a CoreSight AHB-AP.
SLVREADY	Output	Connect to the **SLVREADY** port of the Cortex-M0 DAP, or the **HREADY** port of a CoreSight AHB-AP.
SLVRESP	Output	Connect to the **SLVRESP** port of the Cortex-M0 DAP, or the **HRESP[0]** port of a CoreSight AHB-AP. When connecting to a CoreSight AHB-AP, bit[1] of the **HRESP** port on the CoreSight AHB-AP must be tied LOW.
SLVADDR[31:0]	Input	Connect to the **SLVADDR** port of the Cortex-M0 DAP, or the **HADDR** port of a CoreSight AHB-AP.
SLVTRANS[1:0]	Input	Connect to **SLVTRANS** port of the Cortex-M0 DAP or the **HTRANS** port of a CoreSight AHB-AP
SLVWRITE	Input	Connect to the **SLVWRITE** port of the Cortex-M0 DAP, or the **HWRITE** port of a CoreSight AHB-AP
SLVWDATA[31:0]	Input	Connect to the **SLVWDATA** port of the Cortex-M0 DAP, or the **HWDATA** port of a CoreSight AHB-AP
SLVSIZE[1:0]	Input	Connect to the **SLVSIZE** port of the Cortex-M0 DAP, or the **HSIZE[1:0]** port of a CoreSight AHB-AP

4.4.7 Miscellaneous signals

These signals are present at both the CORTEXM0 and CORTEXM0INTEGRATION level of hierarchy unless otherwise stated.

Table 4-14 shows the miscellaneous signals present on the processor, and describes how to connect the signals.

Table 4-14 Miscellaneous signals

Name	Direction	Description	Connection information
DGBRESTARTED[a]	Output	Handshake for **DBGRESTART**.	If you are not using this signal to connect to a debug *Cross Trigger Interface* (CTI), leave this output unconnected. If you are using this signal to connect to a CTI, contact ARM for connection information.
GATEHCLK	Output	Indicates that **HCLK** can be safely gated. You can use this signal to generate **HCLK** from **FCLK**.	This signal is only present at the `CORTEXM0INTEGRATION` level.
HALTED[a]	Output	Indicates that the processor is in debug state. **HALTED** remains asserted for as long as the processor remains in debug state.	If you are not using this signal to connect to a CTI, leave this output unconnected. If you are using this signal to connect to a CTI, contact ARM for connection information.
LOCKUP	Output	Indicates that the processor is in the architected lock-up state, as the result of an unrecoverable exception. See the *ARMv6-M Architecture Reference Manual* for more information.	You can connect this signal to a watchdog, for example, that resets the processor using **HRESETn**.
SLEEPDEEP	Output	Active only when **SLEEPING** is HIGH. Indicates that the **SLEEPDEEP** bit in the NVIC is set to 1.	-
SLEEPHOLDACKn	Output	Response to **SLEEPHOLDREQn**. If this signal is LOW, irrespective of the **SLEEPING** signal value, the processor does not advance in execution and does not perform any memory operations.	Use this to cleanly power off the processor or safely shut down PLLs for deeper levels of sleep.

Table 4-14 Miscellaneous signals (continued)

Name	Direction	Description	Connection information
SLEEPING	Output	Indicates the processor is idle, waiting for an interrupt on either the **IRQ**, **NMI**, or internal SysTick, or HIGH level on **RXEV**.	-
TXEV	Output	A single **SCLK** cycle HIGH level is generated on this output every time an SEV instruction is executed on the Cortex-M0 processor.	Use this to signal an event to other ARM processors.
DBGRESTART[a]	Input	External restart request.	If you are not using this signal to connect to a CTI, tie this input LOW. If you are using this signal to connect to a CTI, contact ARM for connection information.
ECOREVNUM[19:0][b] **ECOREVNUM[27:0]**[c]	Input	This bus provides a way to implement engineering change order modification for certain bits in the architected ID registers in the processor.	These signals must be tied LOW unless you have an *Engineering Change Order* (ECO) from ARM. ARM recommends that you ensure each of the signal drivers is distinct and readily identifiable to facilitate possible metal layer modification.
EDBGRQ[a]	Input	External debug request.	This input is asserted by a debug agent in the system. It is an external request to enter debug state. The Cortex-M0 processor only enters debug state if C_DEBUGEN in the DHCSR is set. See the *ARMv6-M Architecture Reference Manual* for more information.

Table 4-14 Miscellaneous signals (continued)

Name	Direction	Description	Connection information
IRQLATENCY[7:0]	Input	The Cortex-M0 processor supports zero jitter interrupt latency for zero wait-state memory. **IRQLATENCY** specifies the minimum number of cycles between an interrupt that becomes pended in the NVIC, and the vector fetch for that interrupt being issued on the AHB-Lite interface.	If this bus is set to `0`, interrupts are taken as quickly as possible. For zero jitter in a zero wait state memory system, set this bus to at least a decimal value of `13`. For non-zero wait state memory, zero jitter can be achieved with a higher value on **IRQLATENCY**. **IRQLATENCY** is sampled synchronously to **SCLK**.
RXEV	Input	A HIGH level on this input causes the ARM v6-M architecture defined Event Register to be set in the Cortex-M0 processor. This causes a `WFE` instruction to complete. It also awakens the processor if it is sleeping as the result of encountering a `WFE` instruction when the Event Register is clear.	The input to this signal must be constructed as the logical-OR of all non-interrupt event generating sources of interest in your system. For example, the **TXEV** output of other ARM processors, or a single cycle completion signal from peripherals not already connected to any interrupt lines.
SLEEPHOLDREQn	Input	Request to extend the processor sleeping state regardless of wake-up events. If the processor acknowledges this request driving **SLEEPHOLDACKn** LOW, this guarantees the processor remains idle even on receipt of a wake-up event.	Use this, in conjunction with **SLEEPHOLDACKn**, to cleanly power off the processor or safely shut down PLLs for deeper levels of sleep.

a. Only functional if debug is included.
b. This bus is **[19:0]** for `CORTEXM0`.
c. This bus is **[27:0]** for `CORTEXM0INTEGRATION`.

4.4.8 SysTick signals

These signals are present at both the `CORTEXM0` and `CORTEXM0INTEGRATION` levels.

The ARMv6-M system timer, SysTick, is a system-agnostic timer implementation for operating system use. When you configure the processor without the SysTick timer, both the **STCLKEN** and **STCALIB** signals are non-functional.

Software can configure the SysTick timer to select **SCLK** as its clock source, or an alternative clock source. See Appendix C *SysTick Examples* for more information.

In the Cortex-M0 processor, the alternative clock source is based on **SCLK** gated by **STCLKEN**. If you want to use an asynchronous clock to generate **STCLKEN**, you must use a synchronizer circuit. Figure 4-3 shows an example asynchronous clock generating **STCLKEN**.

If integration is performed without an alternative clock source, tie **STCLKEN** LOW and tie **STCALIB[25]** HIGH.

Table 4-15 shows the **STCALIB[25:0]** values required for the software developer using the SysTick calibration register in the Cortex-M0 NVIC memory map.

Table 4-15 SysTick signals

Name	Mapping	Description
STCALIB[25]	NOREF	Indicates that no alternative reference clock source has been integrated. Tie HIGH if **STCLKEN** has been tied off.
STCALIB[24]	SKEW	Tie this LOW if the system timer clock, the external reference clock, or **SCLK** as indicated by **STCALIB[25]**, can guarantee an exact multiple of 10ms. Otherwise, tie this signal HIGH.
STCALIB[23:0]	TENMS	Provides an integer value to compute a 10ms (100Hz) delay from either the reference clock, or **SCLK** if the reference clock is not implemented. For example, apply the value `0x07A11F` if no reference is implemented, and **SCLK** is 50MHz.

Figure 4-3 shows an example asynchronous clock generating **STCLKEN**.

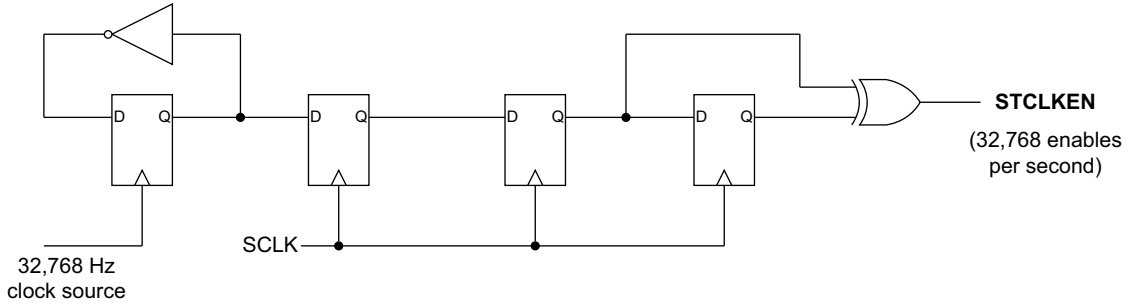


Figure 4-3 Asynchronous SysTick clock example

For an implementation where no alternative reference clock is provided, and the frequency of **SCLK** is not computable in hardware, tie:

- **STCLKEN** LOW
- **STCALIB[25]** HIGH
- **STCALIB[24:0]** LOW.

4.4.9 WIC interface

The WIC interface is only functional if a WIC is configured. The interface signals only exist at the CORTEXM0 level.

If no WIC interface is configured, tie **WICDSREQn** HIGH.

Figure 4-4 shows the WIC mode enable sequence. It includes:

- the **WICENREQ** and **WICENACK** PMU interface signals that are present at the CORTEXM0INTEGRATION level only, see *Power Management Unit interface* on page 4-24 for more information.
- the **WICDSREQn** and **WICDSACKn** WIC signals that are present at the CORTEXM0 level only, see *WIC interface* for more information.

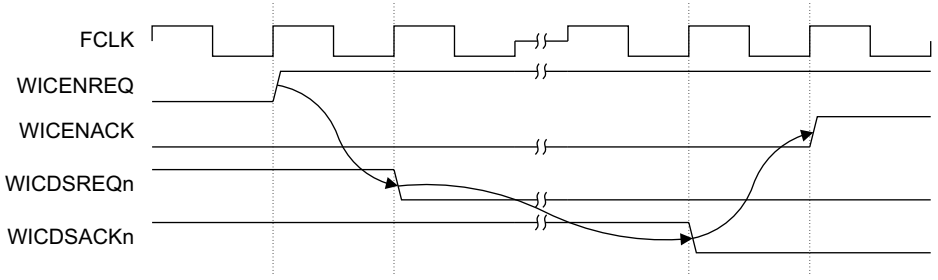


Figure 4-4 WIC mode enable sequence

Figure 4-5 on page 4-23 shows the power down timing sequence. It includes:

- the **WICLOAD**, **WICMASK[]**, **WICCLEAR**, **WICINT[]**, **WICPEND**, and **WAKEUP** WIC interface signals that are present at the CORTEXM0 level only, see *WIC interface* for more information.
- the **SLEEPING** and **SLEEPDEEP** miscellaneous interface signals that are present at the CORTEXM0 level only, see *Miscellaneous signals* on page 4-17 for more information.

- the **SYSISOLATEn**, **SYSRETAINn**, and **SYSPWRDOWN** signals that are present at the `CORTEXM0IMP` and `CORTEXM0INTEGRATIONIMP` levels for particular libraries only, see *Chapter 2 Configuration Guidelines* for more information.

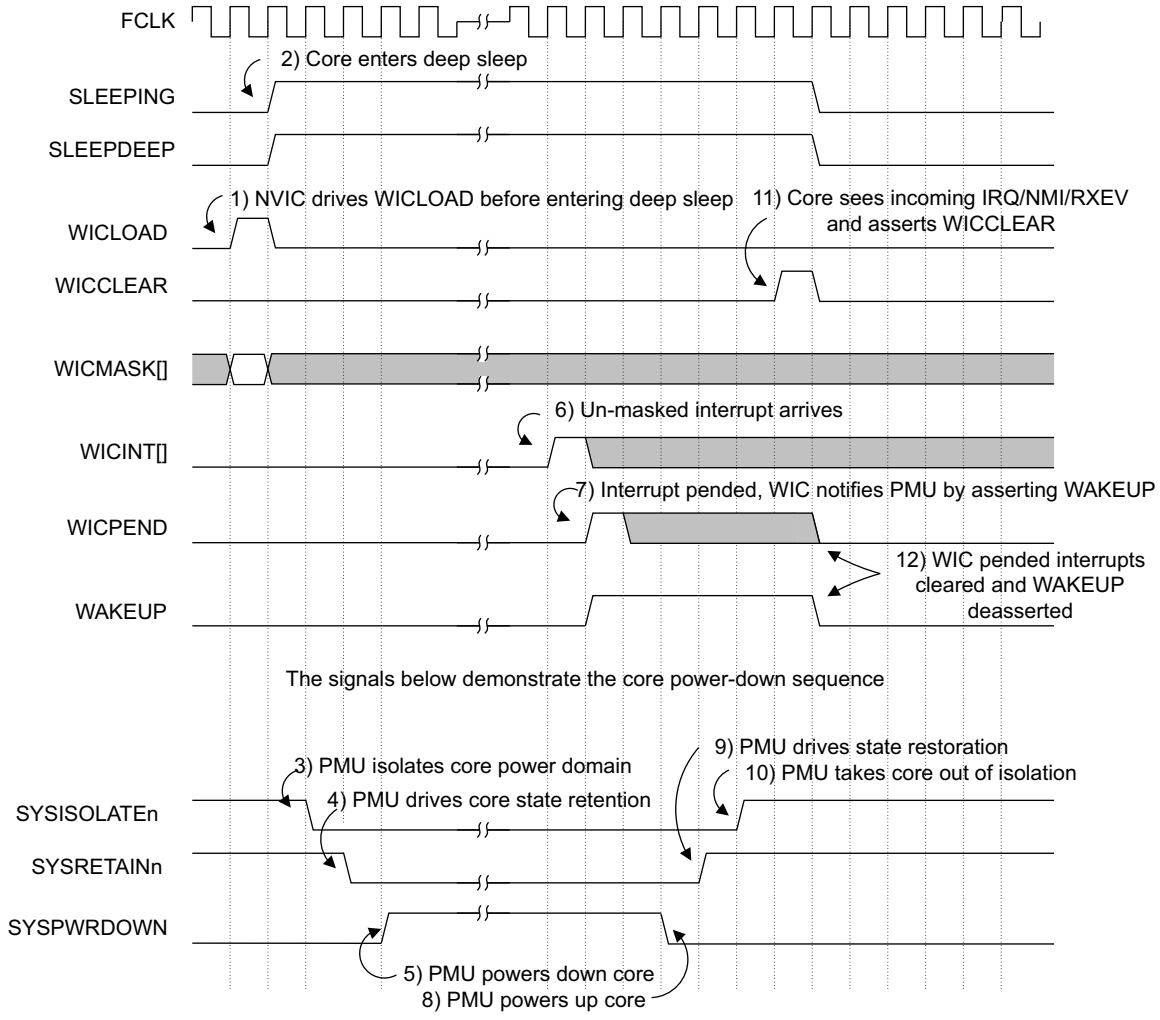


Figure 4-5 Power down timing sequence

Table 4-16 shows the WIC interface signals.

Table 4-16 WIC interface signals

Name	Direction	Description
WICDSREQn	Input	Active LOW request for deep sleep to be WIC-based deep sleep
WICDSACKn	Output	Active LOW acknowledge that deep sleep is WIC-based deep sleep
WICMASKISR	Output	Interrupt sensitivity mask used for WIC wake-up detection
WICMASKNMI	Output	**NMI** sensitivity mask used for WIC wake-up detection
WICMASKRXEV	Output	**RXEV** sensitivity for WIC wake-up detection
WICLOAD	Output	Indicates that the WIC must reload its mask from **WICMASKISR**, **WICMASKNMI** and **WICMASKRXEV**
WICCLEAR	Output	Indicates that the WIC must clear its mask

4.4.10 Power Management Unit interface

The PMU interface only exists at the `CORTEXM0` level.

Table 4-17 shows the PMU interface signals.

Table 4-17 PMU interface signals

Name	Direction	Description
WAKEUP[a]	Output	Active HIGH signal to the PMU that indicates a wake-up event has occurred and the processor system domain requires its clocks and power restored.
WICSENSE[a]	Output	Active HIGH set of signals. These indicate which input lines cause the WIC to generate the **WAKEUP** signal.
WICENREQ[a b]	Input	Active HIGH request for deep sleep to be WIC-based deep sleep. This is driven from the PMU.
WICENACK[a]	Output	Active HIGH acknowledge signal for **WICENREQ**.
CDBGPWRUPREQ	Output	Active HIGH signal that indicates an external debugger request to the PMU to power-up the debug domain in readiness for a debugging session. This signal must be synchronized before use.
CDBGPWRUPACK	Input	Active HIGH signal that indicates the debug domain is powered up in response to **CDBGPWRUPREQ** being HIGH. This signal must be driven synchronously to **FCLK**.

a. Only functional if a WIC is configured.
b. Tie LOW if WIC interface is not configured.

CDBGPWRUPREQ and **CDBGPWRUPACK** form a four-phase handshake. It is a requirement that **CDBGPWRUPACK** is deasserted out of power-on reset and is only asserted or deasserted in response to the assertion and deassertion respectively of **CDBGWPRUPREQ**.

If you are not implementing a PMU and a multi power domain system, you must synchronize **CDBGPWRUPREQ** to **FCLK** and connect the synchronized signal to **CDBGPWRUPACK**.

You must ensure that when **CDBGPWRUPACK** is HIGH, all processor power domains are powered-up and all clocks are running.

Figure 4-4 on page 4-22 and Figure 4-5 on page 4-23 show how the **WAKEUP**, **WICSENSE**, **WICENREQ**, and **WICENACK** signals are sequenced.

4.5 Test interface

Table 4-18 shows the functions of the test interface pins.

Table 4-18 Test interface pin functions

Signal	Direction	Description
RSTBYPASS	Input	CORTEXM0INTEGRATION level only. When set to HIGH, the internal reset synchronizers are bypassed. This enables test pattern generation tools to have controllability of reset flops in the design.
SE	Input	**SE** is the scan-enable input. Drive this signal HIGH during vector scan-in/scan-out, and LOW during normal operation. **SE** also bypasses any architectural clock-gate cell instantiations within the Cortex-M0 processor.

Table 4-19 shows the required test interface pins.

Table 4-19 Test interface pins

Level	Required pins
CORTEXM0	**SE**
CORTEXM0INTEGRATION	**SE** and **RSTBYPASS**.

Note

Scan insertion during synthesis introduces one or more *Scan-In* (SI) and *Scan-Out* (SO) pins.

See Chapter 8 *Design For Test* for more information about testing and the testing interface.

Chapter 5
Integration Kit

This chapter describes the Cortex-M0 *Integration Kit* (IK). It contains the following sections:

5.1 About the integration kit

The Cortex-M0 integration kit is provided as a reference to enable easy integration of the Cortex-M0 processor into your system, and contains tests to check that you have performed this integration correctly. Support for simulation of both the Cortex-M0 RTL and Cortex-M0 netlists is provided. In addition, example vector capture testbenches are provided to enable you to capture vectors for silicon testing at either the `CORTEXM0IMP` level or the `CORTEXM0INTEGRATIONIMP` level of hierarchy.

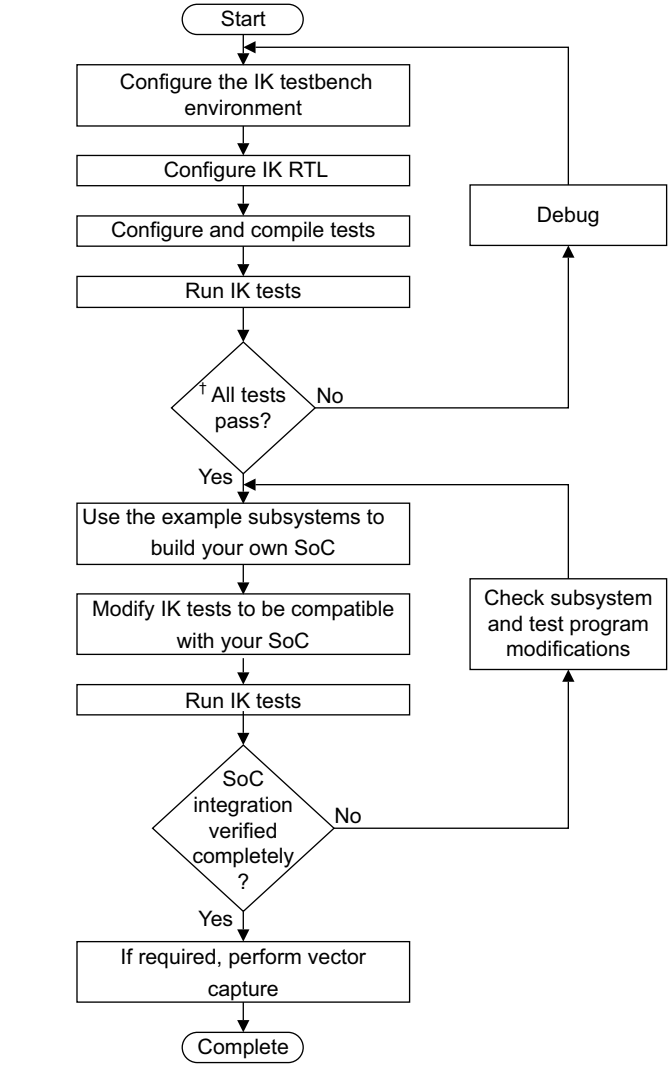
The Cortex-M0 integration kit is based on a simple example microcontroller, CM0IKMCU, that instantiates the Cortex-M0 processor, a ROM, a RAM, a Cortex-M0 DAP, a PMU, a WIC, a reset controller, and some *General Purpose Input Output* (GPIO).

Note

The integration kit tests do not exhaustively test the Cortex-M0 processor I/O because not all I/O pins have an effect that is visible to program code, and some pins are meant to be static. This means that you must not use passing of all integration kit tests as the only sign-off criteria for successful processor integration. See Chapter 10 *Sign-off* for details of your Sign-Off obligations.

5.2 Cortex-M0 IK flow

The Cortex-M0 IK is intended as a platform from which you can develop a SoC incorporating the Cortex-M0 processor. Figure 5-1 shows the Cortex-M0 IK flow.



† Depending on the configuration you choose, some tests might skip

Figure 5-1 Cortex-M0 IK flow

5.3 Test overview

Table 5-1 shows the example test programs in the `integration_kit/validation/tests` directory.

Table 5-1 Example test programs

Test program	Description
`hellow`	The processor reads and checks its CPU ID and writes to the GPIO registers to print a simple message.
`dhry`	This test runs the dhrystone benchmarking program. The default number of iterations is 5. You can change the number of iterations in one of two ways: • If you are building tests using the `Makefile`, edit the value of ITERATIONS there. • If you are building tests using MDK-ARM, right-click **CM0IKMCU - Cortex-M0** within the dhry project to open the target options, then edit the value of ITERATIONS under the **C/C++** tab.
`max_power`	This test executes instructions that exercise the Cortex-M0 processor and maximize power consumption. Replay the captured test vectors on your netlist to get power values.
`speed_indicative`	This test executes instructions that exercise the critical paths in the Cortex-M0 processor. Replay the captured test vectors on your netlist to test timing accuracy.
`config_check`	This test verifies that the processor configuration matches the expected configuration values set in the `IKConfig.h` file.
`interrupt`	This test exercises **NMI**, **IRQ**, **TXEV**, and **RXEV**.
`reset`	This test checks the **SYSRESETREQ** output and that accesses to the AHB default slave cause a fault.
`sleep`	This test exercises the sleep modes of the processor, and the **SLEEPING** and **SLEEPDEEP** pins. The test uses an interrupt to wake the processor, and if the processor includes debug, also wakes the processor from sleep through the DAP.
`debug`	This test is only useful for implementations that include `DBG`. If you run this on an implementation without `DBG`, the test passes without performing useful work. The test checks the pins **LOCKUP**, **EDBGRQ**, **HALTED**, **DEBUGRESTART**, and **DEBUGRESTARTED**.
`romtable`	This test checks that it is possible for a debugger to autodetect the Cortex-M0 processor. The test uses the DAP to locate the architecturally-defined ROM table to locate the processor. If you have included system level ROM tables, the content of these is displayed, but not checked. This test is only useful for implementations that include `DBG`. If you run this on an implementation without `DBG`, the test passes without performing useful work.

5.4 Configuring the testbench

This section describes how to configure the testbench.

The testbench instantiates the CM0IKMCU level of the integration kit. See *Testbench structure* on page 5-20 for more information.

Table 5-2 shows the Verilog command files in the `integration_kit/logical/tbench/verilog/` directory.

Table 5-2 Verilog command files

File	Description
`tbench.vc`	Used for all simulations
`rtl.vc`	Used for RTL simulations
`netlist.vc`	Used for netlist simulations
`dsm.vc`	Used for DSM simulations

Note

- The Verilog command files used depend on the type of simulation required.
- The `tbench.vc` file contains defines for vector capture. For information on Verilog defines related to vector capture see *Test vector capture* on page 5-28.

The `ARM_CM0IK_INTEGRATION_LEVEL` define in the `integration_kit/logical/tbench/verilog/tbench.vc` file controls the integration level of the processor. Ensure this define is included if you want to use the `CORTEXM0INTEGRATION` level. Ensure this define is not included if you want to use the `CORTEXM0` level.

The `ARM_CM0IK_SRPG` define in the `integration_kit/logical/tbench/verilog/tbench.vc` file controls whether SRPG signals are included in the processor. Ensure this define is included if you want SRPG signals. Ensure this define is not included if you do not want SRPG signals.

The `ARM_SRPG_ON` define in the `integration_kit/logical/tbench/verilog/rtl.vc` or `integration_kit/logical/tbench/verilog/netlist.vc` file controls whether you use *Unified Power Format* (UPF) and *Common Power Format* (CPF) RTL simulations or SRPG-netlist simulations respectively. Ensure this define is included if you want UPF or CPF in RTL simulations or SRPG in netlist simulations. Ensure this define is not included if you do not UPF or CPF in RTL simulations or SRPG in netlist simulations.

The ARM_CM0IK_SDF define in the integration_kit/logical/tbench/verilog/netlist.vc file determines if delay calculation and static timing analysis information is in SDF. Ensure this define is included if you want SDF from netlist simulation. Ensure this define is not included if you want do not want SDF from netlist simulation.

The Cortex-M0 processor RTL and Cortex-M0 DAP RTL include conditionally instantiated *Open Verification Library* (OVL) assertion checkers. These dynamic checks can help you to diagnose problems with your system or test code. To run simulations with these checkers enabled, you must uncomment the appropriate lines in the logical/tbench/verilog/tbench.vc file and ensure the path to the OVL library is correct for your environment. You can download the OVL library from http://www.accellera.org.

To configure the testbench options, open the integration kit definitions file integration_kit/logical/tbench/verilog/cm0ik_defs.v. Table 5-3 shows the testbench options to configure in the definitions file.

Table 5-3 cm0ik_defs.v testbench options

option	Description
ARM_CM0IK_CLK_PERIOD	Clock for CM0IKMCU
ARM_CM0IK_POR_CYCLES	Power-on-Reset duration in cycles of CM0IKMCU clock
ARM_CM0IK_TIMEOUT_CYCLES	Runaway simulation time out duration in cycles of CM0IKMCU clock

5.5 Configuring the IK RTL

This section describes how to configure the IK RTL depending on your requirements.

5.5.1 Configuring the implementation levels

If you are integrating:

- the processor level, see *CORTEXM0 configuration options* on page 2-3 for more information
- the `CORTEXM0INTEGRATION` level, see *CORTEXM0INTEGRATION configuration options* on page 2-6 for more information.

5.5.2 Configuring the timing wrappers

The timing wrapper that you configure depends on the implementation level you use. For the `CORTEXM0INTEGRATION` level, use `cm0ik_cortexm0integration_timing` wrapper. For the `CORTEXM0` level, use `cm0ik_cortexm0_timing` wrapper.

cm0ik_cortexm0_timing

If you use the `cm0ik_cortexm0_timing` wrapper, you must modify the following in the `integration_kit/logical/cm0ikmcu/verilog/cm0ik_cortexm0_timing.v` file:

- If the WIC is used, modify the following parameters to match the processor:
  - `WIC`
  - `WICLINES`.
- If debug is used, modify the following parameters to suit your requirements:
  - `DBG`
  - `JTAGnSW`
  - `RAR`.

cm0ik_cortexm0integration_timing

If you use `cm0ik_cortexm0integration_timing` wrapper, there are no parameters to configure.

5.5.3 Setting the implementation pin options

Implementation pin options can be set at either implementation level. To set the implementation pin options, edit the integration kit definitions file `integration_kit/logical/tbench/verilog/cm0ik_defs.v`. The implementation pin options are:

ARM_CM0IK_STCALIB

SysTick calibration, see Figure 4-3 on page 4-21 for an asynchronous SysTick clock example.

ARM_CM0IK_IRQLATENCY

Interrupt Latency, see *Miscellaneous signals* on page 4-17.

5.6 Configuring and compiling tests

This section describes how to set up the test programs before running them on hardware or running a testbench simulation.

The supplied tests are written in C and must be compiled before you execute them on the Cortex-M0 processor.

The tests have been developed and tested using the full ARM RVCT toolchain running under Linux, and using RealView MDK-ARM under Windows. You can download an evaluation version of the:

- RealView Development Suite, which includes RVCT, from `http://www.arm.com`
- RealView MDK-ARM from `http://www.keil.com`.

Note

See the release note for the ARM RVCT and RealView MDK-ARM versions that have been tested with the deliverables.

You might have to modify:

- the tests if you want to use an alternative compiler
- `Makefile`, or use an alternative system, if you want to use a different OS.

Table 5-4 shows the test support files in the `integration_kit/validation/tests` directory.

Table 5-4 Test support files

Filename	Description
`boot.c`	This file provides the stack and heap initialization, vector table, default handlers and `_sys_exit` function that updates the **TESTPASS** and **TESTCOMPLETE** signals when test code completes. The CMSIS provides assembler example startup files that you can modify and use instead of `boot.c`.
`retarget_cm0ikmcu.h`	This file implements the functions necessary to retarget the C-library `printf()` function output to the `CM0IKMCU` **GPIO** pins.
`IKtests.h`	This header file describes and defines the allocation of **GPIO** pins used by the `CM0IKMCU` in the integration kit. It also declares the function prototypes the integration kit tests use to communicate with the debug driver.

Table 5-4 Test support files (continued)

Filename	Description
IKtests.c	This file provides the functions the integration kit tests uses to initialize the GPIOs and to communicate with the debug driver.
IKConfig.h	This file includes some defines that you must edit to match the implemented configuration of CORTEXM0INTEGRATION or CORTEXM0.
Makefile	This file enables you to build the integration kit tests using the ARM RVCT compiler.

5.6.1 Integration kit test configuration

The file IKConfig.h in the integration_kit/validation/tests directory defines the expected values the IK tests check against. Ensure the following IKConfig.h configuration options match the configuration of your RTL, see *Configuration options* on page 2-3 for more information:

EXPECTED_BE

Specifies the endianness of the processor in the integration kit.

Note

- If the Cortex-M0 processor is in big-endian configuration, you must also compile the IK tests as big-endian.
- If you are compiling tests using:
  - — ARM RVCT and the Makefile, edit the COMPILE_BE variable
  - — RealView MDK, consult the MDK-ARM documentation for details of the changes you must make to your project files.

EXPECTED_BKPT

Specifies the number of breakpoint comparators.

EXPECTED_DBG

Specifies whether or not debug is included.

EXPECTED_JTAGnSW

Specifies the Cortex-M0 DAP protocol.

EXPECTED_NUMIRQ

Specifies the number of external interrupts.

EXPECTED_SYST

Specifies whether the SysTick timer is included.

EXPECTED_SMUL

Specifies the implemented multiplier.

EXPECTED_WPT

Specifies the number of watchpoint comparators.

Ensure the following `IKConfig.h` configuration options match the tied-off values of the corresponding signals:

EXPECTED_STCALIB

The expected value of **STCALIB[25:0]**. See *SysTick signals* on page 4-20 for information about how to determine this value for your design.

EXPECTED_BASEADDR

The expected value of the CORTEXM0DAP CoreSight Component pointer. See *CoreSight ROM table base address* on page 2-8 for information about how to determine this value for your design.

Note

If you use RealView MDK-ARM to build your tests, you can use the Configuration Wizard to update the values in `IKConfig.h`.

5.6.2 Test program compilation, ARM RVCT

This section describes how to compile the test programs with ARM RVCT 4.0 under Linux using the `make` command. You might have to modify the `Makefile` in the `integration_kit/validation/tests` directory to suit your environment, for example:

- the name of the ARM compiler
- the name of the linker
- compiler and linker options
- test configuration options.

Test programs are compiled into the `tests` directory. The `make` command compiles:

- binary `.elf` files for use with a debugger
- binary `.bin` files for memory initialization.

To compile a test:

1. Change to the integration_kit/validation/tests directory by typing:

   ```
   cd validation/tests
   ```

2. Use the Makefile in integration_kit/validation/tests to compile the test programs. The command is in the format:

   ```
   make <test_program>
   ```
   or
   ```
   make all
   ```

   where:

<test_program>	Selects an individual test program to compile. See Table 5-1 on page 5-4 for a list of available tests.
all	Compiles all the test programs into the current directory.

   For example, to compile the test hellow.c, type:

   ```
   make hellow
   ```

   Note

   - The ARM compiler must be on your path.
   - If no individual test program is selected and the option all is not used with the make command, the default is to compile all the tests listed in Table 5-1 on page 5-4.
   - To remove the compiled tests and prepare for a fresh compilation, type:

     ```
     make clean
     ```

5.6.3 Test program compilation, RealView MDK-ARM

This section describes how to compile the test programs using the RealView MDK-ARM tools.

The integration_kit/validation/mdk directory contains a multiproject workspace that includes project files for each of the integration kit tests and the debug driver. Use the following procedure to compile the test programs:

1. Open the project workspace:

   ```
   Project -> Open Project -> IntegrationKit.mpw
   ```

2. Configure the tests:

   Locate the IKConfig.h header file within any of the projects and make any necessary changes.

3. Build the test binaries:

   ```
   Project -> Batch Build -> Select All, Build
   ```

Intermediate build files are created in the `mdk` directory. Test binaries are created in the `tests` directory.

5.7 Running IK tests

The testbench is run from the `integration_kit/validation` directory. The `RunIK` script is provided as an example script to compile the testbench and simulate a specified test. This section contains:

- *RunIK script usage and options*
- *Running RunIK with the -dsm option* on page 5-15
- *Running RunIK with -netlist option* on page 5-15
- *Simulation logs* on page 5-16
- *Modifying the RunIK script* on page 5-16
- *UPF or CPF SRPG simulation considerations* on page 5-16.

5.7.1 RunIK script usage and options

To run the `RunIK` script use the following format:

```
RunIK <options> <testname>
```

where `<options>` are:

`-build`	This switch is optional. Use this switch to compile the integration kit Verilog, including the testbench, before running the test or tests.
`-make`	This switch is optional. Use this switch to call `make` to compile the tests using the ARM RVCT tools, if these tools are present. If you do not have the ARM RVCT tools, compile the tests before you run this script.
`-all`	Use this switch to run all integration kit tests.
`-dsm`	Use a DSM for the `CORTEXM0` level.
`-netlist`	Use this switch to run the integration kit with a netlist.
`-mti`	Use the MTI simulator. The default is MTI.
`-nc`	Use the NC simulator.
`-vcs`	Use the VCS simulator.
`-sdf <delay type>`	Use SDF with the netlist. Choose `<delay type>` from `min` or `max`. If this switch is not specified, the default is zero delay.
`-upf`	Use UPF for simulation.
`-cpf`	Use CPF for simulation.

<testname> gives the name of the test image file to run. This file is in the integration_kit/tests directory. For example, if you want to compile the testbench and run the test hellow.c using the VCS simulator, type:

```
RunIK -vcs -build hellow
```

The RunIK script creates a directory within integration_kit/validation, corresponding to the specified simulator:

- MTI
- VCS
- NC.

Ensure that the simulator of your choice is on your path before you execute the RunIK script.

5.7.2 Running RunIK with the -dsm option

Ensure that the DSM has been set up correctly to run RunIK with the -dsm option. See the documentation in the DSM package for more information about DSM setup. An example file integration_kit/validation/dsmdotchrc is provided to set up the DSM. In addition, ensure that the file integration_kit/logical/tbench/verilog/dsm.vc points to the CORTEXM0 DSM.

5.7.3 Running RunIK with -netlist option

Ensure that the file integration_kit/logical/tbench/verilog/netlist.vc points to the netlist at the required level to run RunIK with -netlist option. To simulate netlist with back-annotated timing, ensure that:

- The Verilog define to include SDF is set. See *Configuring the testbench* on page 5-5.
- A symbolic link at the required level, either CORTEXM0IMP.sdf.gz or CORTEXM0INTEGRATIONIMP.sdf.gz, exists in integration_kit/validation and points to the actual .sdf file.

5.7.4 Simulation logs

When a test program passes in simulation, the following message appears in the simulation log:

```
** TEST PASSED OK **
```

When a test program fails in simulation, the following message appears in the simulation log:

```
** TEST FAILED **
```

When a test program does not complete within the time limit specified by `ARM_CM0IK_TIMEOUT_CYCLES`, the runaway simulation timer terminates the test and the following message appears in the simulation log:

```
** TEST KILLED **
```

The simulation log files are generated in `integration_kit/validation/logs`.

5.7.5 Modifying the RunIK script

You can modify the `RunIK` script to include any extra simulator options you want to use.

You must modify the `RunIK` script if you want to include new tests when you use the `-all` switch for batch testing.

5.7.6 UPF or CPF SRPG simulation considerations

Before simulating the RTL or netlist with UPF or CPF ensure that:

- SRPG signals are included. See *Configuring the testbench* on page 5-5 to see how SPRG signals are included.
- The file `integration_kit/validation/power_file` exists and it has a symbolic link to the appropriate UPF or CPF file.

  Note

  Four files are available in `integration_kit/validation/srpg` for UPF or CPF SRPG simulation:

  — `CM0IKMCU.upf`
  — `CM0IKMCU_integration.upf`
  — `CM0IKMCU.cpf`
  — `CM0IKMCU_integration.cpf`

  For example, for UPF simulation with `cm0ik_cortexm0_timing`, type the following command from `integration_kit/validation`:

```
ln -s srpg/CM0IKMCU.upf power_file
```

- The `mvtools` suite is on your path for UPF simulation.

5.8 Debugging failing tests

This section describes what you can do to help diagnose problems when a test or tests fail or do not complete on hardware or testbench simulations.

If the tests are running on the simulation testbench, you can debug the tests interactively using the functions available in your chosen simulator.

If the tests are running on hardware and a debugger is available, you can use the features provided by the debugger to identify where the test fails.

All tests must pass, regardless of the configuration of the processor. If some tests are failing or being killed by the runaway simulation timer, debug the tests to determine the cause of the problem.

The recommended debug strategy is as follows:

Prioritize the failures

The simplest test is `hellow`. If this test is among your failing tests, start debugging it first.

The `config_check` test is more complex, but it is the only test that checks the configuration of the processor matches the expected values set in `IKConfig.h`. If this test is failing, you must resolve the issues, because other tests might assume that the `EXPECTED_*` values in `IKConfig.h` are correct.

Check log files for errors or warnings

The test itself might indicate the cause of the failure, for example, a mismatch in expected and actual values for a parameter.

If you have enabled OVL checks, you might see messages that indicate RTL misconfiguration or X value propagation. The latter is normally the result of accessing uninitialized memory or a malfunctioning memory subsystem. See *Configuring the testbench* on page 5-5 for information on how to enable OVL checks.

Check configuration

Check that the `EXPECTED_*` values in `IKConfig.h` match the configuration of the processor.

Enable message printing

By default, tests print progress and status messages to the simulation log using the simple character output device in the top level testbench. You can disable this by commenting out the definition of `CM0IKMCU_PRINTF` in `IKConfig.h` before compiling the tests. You might want to do this to reduce the runtime of the tests, and therefore vectors.

If a test is failing, enable message printing by defining `CM0IKMCU_PRINTF`, for example by uncommenting it in `IKConfig.h`, and recompile and rerun the test to help determine the reason for failure.

You can also enable message printing from the debug driver module by defining `DEBUGDRIVER_PRINTF` in `IKConfig.h` and recompiling the debug driver image. This can help you to debug an issue with a test that uses the debug driver, for example `config_check`, `debug`, `sleep`.

Run on an unmodified version of the integration kit, or see the golden logs in directory `glogs`, to view the output the tests should produce.

Add your own debug messages

If message printing is enabled, you can insert additional calls to `printf()` into the code to help determine where the test is failing.

Compare executed instructions against the test code

By default, RTL simulations produce a log file, `tarmac.log`, of the instructions executed on the processor. This can be used to debug test failures by comparing the executed instructions with the assembly language of the test code. You can use the ARM RVCT `fromelf` utility to show the assembly language content of the compiled test code, for example:

```
fromelf -c hellow.elf
```

Note

See your compiler documentation if you are not using ARM RVCT to compile your tests.

5.9 Modifying the IK RTL for your SoC

This section describes the testbench structure and the integration kit level, CM0IKMCU, memory map. Read this section if you want to modify the IK RTL for your own SoC requirements.

5.9.1 Testbench structure

Figure 5-2 shows the testbench structure and its instantiated integration kit components.

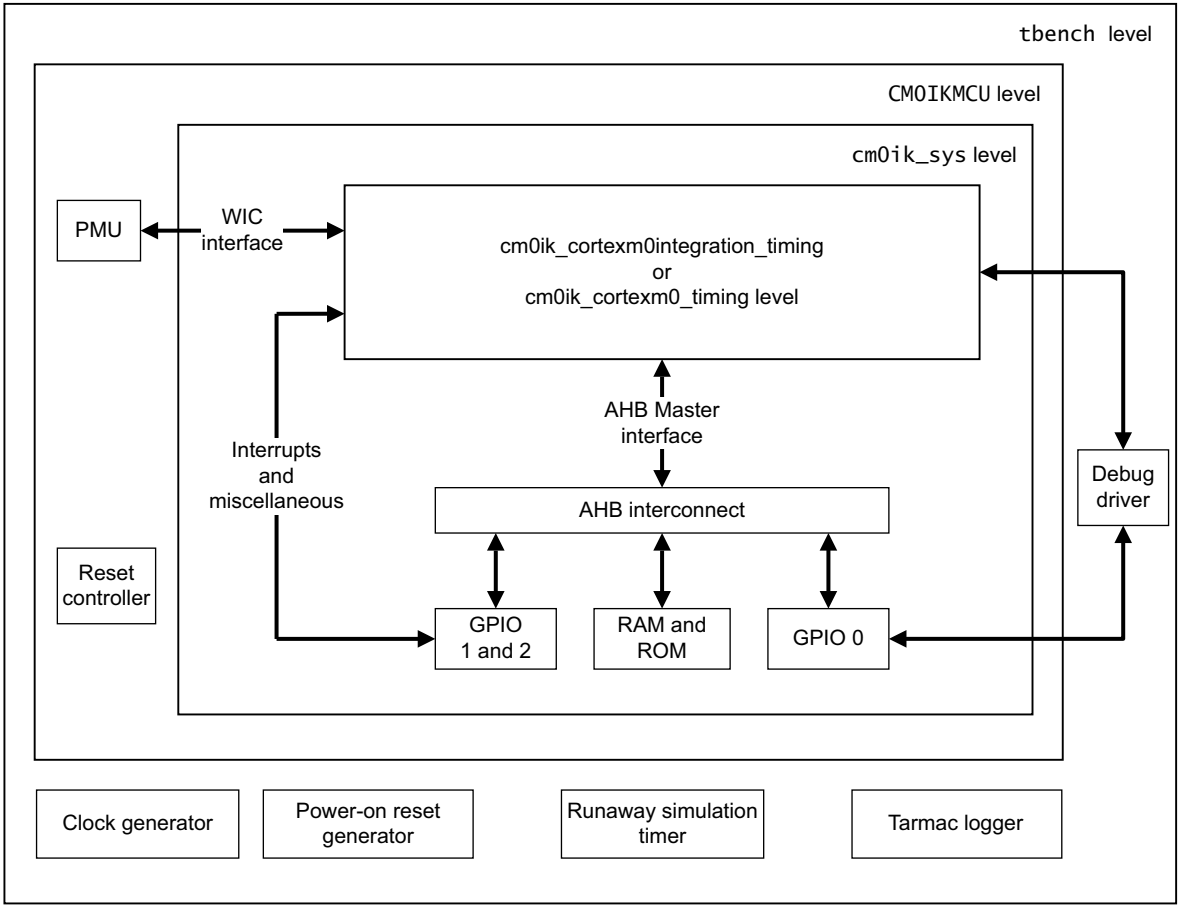


Figure 5-2 Integration kit

The integration kit contains these levels:

- *cm0ik_cortexm0_timing level* on page 5-21

- *cm0ik_cortexm0integration_timing level*
- *cm0ik_sys level*
- *CM0IKMCU level* on page 5-23
- *tbench level* on page 5-24.

The `integration_kit/` path in Figure 1-6 on page 1-11 shows the supplied integration kit directory structure.

cm0ik_cortexm0_timing level

This level instantiates the `CORTEXM0IMP` level, Cortex-M0 DAP, and the WIC. This level delays all inputs to the `CORTEXM0IMP` level to support netlist simulations with timing.

cm0ik_cortexm0integration_timing level

This level instantiates the `CORTEXM0INTEGRATIONIMP` level. This level delays all inputs to the `CORTEXM0INTEGRATIONIMP` level to support netlist simulations with timing.

cm0ik_sys level

This section describes the cm0ik_sys level and its components. The cm0ik_sys level contains:

General Purpose Input Output (GPIO)

GPIO is a general purpose I/O device.

You can configure the GPIO pins individually as inputs or outputs.

You can configure the GPIO to generate an interrupt when specific input values change.

Use the GPIO to:

- drive signals at the testbench level
- control the debug driver block to access the DAP.

General Purpose Input Output 0

This GPIO has all 32 I/O pins routed to the testbench level. These signals:

- indicate test completion and pass or fail status
- display messages using a simple character output device
- drive the Serial Wire or JTAG interface to the DAP using the debug driver block in the testbench.

For more information see Appendix B *CM0IKMCU GPIO Integration*.

General Purpose Input Output 1

This GPIO drives signals in the example CM0IKMCU, for test purposes only. It drives PMU signals and other processor signals.

For more information see Appendix B *CM0IKMCU GPIO Integration*.

General Purpose Input Output 2

This GPIO drives signals in the example CM0IKMCU, for test purposes only. It acts as a source of interrupts for the WIC and processor. All GPIO interrupts drive **IRQ[0]** of the processor.

For more information see Appendix B *CM0IKMCU GPIO Integration*.

AHB default slave

Accesses to unused address locations in the system go to the AHB default slave. The AHB default slave responds with an error each time it is accessed.

AHB ROM Memory

This is a zero wait-state, read-only AHB memory model that includes support for retention of state on power-down during power-aware simulations. The integration test code image is loaded here.

SRAM controller and SRAM

This is a synchronous SRAM model with an example AHB SRAM controller that guarantees zero wait-state responses to all AHB accesses by supporting write data buffer forwarding.

AHB bus interconnect

This block does the AHB address decoding and multiplexing.

The single slave port is connected to the AHB-Lite master interface from the `CORTEXM0INTEGRATION` level.

The seven master ports are connected to the ROM, the RAM, GPIO 0, GPIO 1, GPIO 2, the system ROM table, and the AHB default slave respectively.

Miscellaneous logic

Miscellaneous logic used for integration test purposes.

System level ROM tables

This module instantiates example system level CoreSight ROM tables. These permit debuggers to uniquely identify the Cortex-M0 based system. If you want to support system-level ROM tables, you must modify this module to include your JEP106 manufacturer ID and a part number that identifies your system.

CM0IKMCU level

The CM0IKMCU level is designed to be an example MCU and contains:

cm0ik_sys The provided example system level components. See *cm0ik_sys level* on page 5-21 for more information.

PMU

The PMU is an example system power controller that:

1. Implements three distinct power domains and manages the control for each:
   - an always-on power domain containing the WIC and part of the DAP
   - a debug power domain containing the processor debug resources and the rest of the DAP
   - a system power domain containing the rest of the processor and the NVIC.
2. Interfaces with the WIC to ensure that power-down and wake-up behavior is transparent to software.
3. Generates clocks for use by the processor, WIC and the memory system compatible with the clocking, power-domain and sleeping requirements.
4. Interfaces with the reset controller to meet the power control reset requirements.
5. Interfaces with the processor to manage extended sleep and ensure clean power-down.

The PMU provided is a verified example module that you can modify to suit your requirements.

Reset controller

An example reset controller that meets all the reset requirements of the processor. It generates synchronously asserted and synchronously de-asserted reset signals from a single asynchronous raw reset input and synchronous reset requests.

The reset controller is a verified example module that you can modify to suit your requirements.

tbench level

The `tbench` level is the top level testbench and contains:

CM0IKMCU level

The provided example CM0IKMCU level.

Clock generator This generates clocks for the CM0IKMCU level.

Power-on reset generator

This generates power-on reset.

Debug driver This programmable block controls the Cortex-M0 DAP. See Appendix D *Debug Driver* for more information.

Runaway simulation timer

This is used in simulation to end tests that have not completed within a specified cycle limit.

Tarmac logger The tarmac logger creates an output log file of the instructions executed by the Cortex-M0 processor during a test run.

5.9.2 CM0IKMCU memory map

Figure 5-3 on page 5-25 shows the CM0IKMCU memory map.

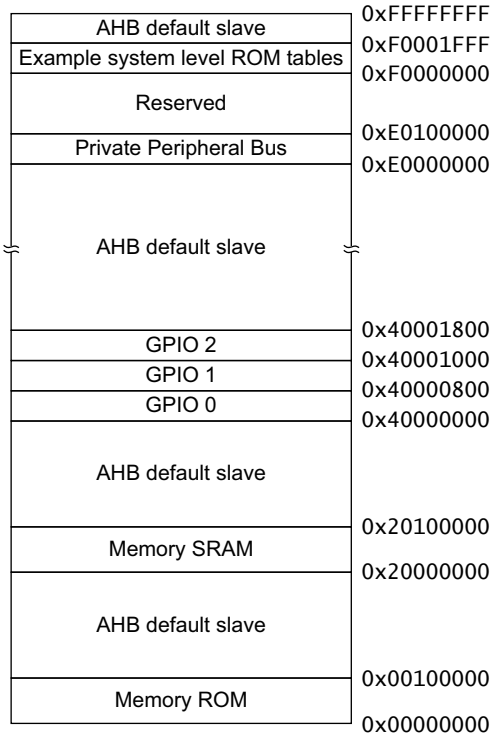


Figure 5-3 CM0IKMCU memory map

The CM0IKMCU memory map is characterized by the following:

- CM0IKMCU instantiates 2MB of physical memory. It is split into two blocks of 1MB:
  - the first 1MB, typically flash in an MCU, is read-only memory that holds the code
  - the second 1MB, typically SRAM, is where the stack and heap are located.
- There are three GPIOs, each allocated 2KB of space.
- The *Private Peripheral Bus* (PPB) space of the Cortex-M0 processor is 1MB.
- Example system level ROM tables occupy 8KB of space.
- All remaining memory regions are mapped to the AHB default slave, including the reserved space. Any access to the default slave results in a fault.

5.10 Modifying IK tests

The supplied test programs rely on the integration kit GPIO to report test status, generate interrupts, and monitor status signals. To run the test programs in a custom environment, modify the code to:

- report test passed or failed
- use available peripherals to generate external interrupts
- define appropriate exception handlers for the system.

Some tests might have to be excluded if the environment does not use a particular interface or cannot support self-checking of interface signals.

Test programs are supplied as C source code with the integration kit. The header of each source code describes the test requirements of that code. You might have to modify these test programs to work in your SoC validation environment.

As supplied, the code uses some components external to the processor, to self-check some of the interface signals and to check the functionality of processor. You must adapt the integration test programs to make use of the external components in your design for these tests. Omit one or more of the tests if your SoC validation environment:

- does not use a particular interface
- does not support self-checking of one or more of the interface signals.

The tests supplied with the integration kit are written in C and are compliant with the *Cortex Microcontroller Software Interface Standard* (CMSIS). The CMSIS enables end users to write code that is portable across all ARM Cortex-M based microcontrollers.

The integration kit includes a minimal subset of the CMSIS for the Cortex-M0 processor that is sufficient to support the supplied tests. See `http://onarm.com` for the full version of the CMSIS and other embedded software development resources.

See the CMSIS documentation for information about how to provide your customer with the latest CMSIS library, and how to provide headers tailored to your Cortex-M0-based device.

Table 5-5 on page 5-27 shows the files in the `integration_kit/validation/tests` and `integration_kit/validation/tests/CMSIS` directories that constitute the CMSIS.

Table 5-5 CMSIS file descriptions

Filename	Location	Supplied by	Description
CMSIS_Core.htm	tests/CMSIS/Core/Documentation/	ARM	HTML format documentation for the CMSIS Core files.
core_cm0.h	tests/CMSIS/Core/CM0/	ARM	Defines the core peripherals for the Cortex-M0 processor and core peripherals.
core_cm0.c	tests/CMSIS/Core/CM0/	ARM	Provides helper functions that access processor registers
cm0ikmcu.h	tests/	Device vendor	Device specific file that defines the peripherals for the CM0IKMCU example mircrocontroller device.
system_cm0ikmcu.h	tests/	Device vendor	Header file that provides device specific configuration for the CM0IKMCU example microcontroller device.
system_cm0ikmcu.c	tests/	Device vendor	C file that provides device specific configuration for the CM0IKMCU example microcontroller device.

5.11 Test vector capture

The vector capture testbenches provided permit you to capture vectors at the CORTEXM0IMP or CORTEXM0INTEGRATIONIMP levels of hierarchy.

You can capture vectors for any code that you run in the integration kit. If vectors are captured from simulations using the RunIK script, they can be found at integration_kit/validation/vectors. The vectors are in the CRF format. The .crf filename includes either CORTEXM0IMP or CORTEXM0INTEGRATIONIMP, depending on the level where it is captured. For example, vectors for sample_test.c at CORTEXM0IMP level are named sample_test_CORTEXM0IMP.crf and those at CORTEXM0INTEGRATIONIMP level are named sample_test_CORTEXM0INTEGRATIONIMP.crf.

The verilog vector capture modules are integration_kit/logical/tbench/verilog/CORTEXM0IMP_Capture.v and integration_kit/logical/tbench/verilog/CORTEXM0INTEGRATIONIMP_Capture.v. If you change the pinout of the CORTEXM0IMP or CORTEXM0INTEGRATIONIMP levels, you must update the corresponding vector capture module accordingly.

The integration_kit/logical/tbench/verilog/tbench.vc file contains Verilog defines that affect the way vectors are captured. They are:

ARM_CM0IK_VECTOR_CAPTURE

If this is defined, vectors are captured.

ARM_CM0IK_XVAL

An X detected at the input of CORTEXM0INTEGRATIONIMP or CORTEXM0IMP is captured as logic LOW or L in the .crf file.

If this is not defined, an X at the input is captured as logic HIGH or H in the .crf file.

ARM_CM0IK_SKIPVECNUM

This takes an integer value and specifies the number of initial vectors whose outputs are not checked during replay. For example, the first few clock ticks before reset. In the .crf file, the output for a skipped vector is marked with a dot.

ARM_CM0IK_PWRDOWNACK

If this is defined, the values of **SYSPWRDOWNACK** and **DBGPWRDOWNACK** are captured in the vectors and checked during vector replay. If this is not defined, **SYSPWRDOWNACK** and **DBGPWRDOWNACK** are not captured in the vectors and are not checked during vector replay.

If you want to capture vectors at a higher level of hierarchy in your SoC, you must create your own vector capture testbench or modify either of the provided capture testbenches to reflect the I/O and clocking structure of your SoC, and ensure the vectors are saved to the appropriate directory.

Replay the test vectors using the replay testbench to perform quick netlist checks or obtain characterization information like speed and power from your netlist. You can use the replay testbench to verify that your netlist matches the cycle-by-cycle behavior of the RTL reference. See *Test vector simulation environment and vector replay* on page 9-6 for more information.

See the Reference methodology documents from your EDA tool vendor for information about how to replay test vectors.

Chapter 6
Key Implementation Points

This chapter describes the key implementation points you must consider when you implement the Cortex-M0 processor. It contains the following sections:

Note

This chapter references steps not described in this guide. See the Reference Methodology documents from your EDA tool vendor for descriptions of these steps.

6.1 About key implementation points

This chapter contains a list of the main points to consider when you implement the Cortex-M0 processor. Read this chapter in conjunction with the rest of the information in this guide, and your Reference Methodology documentation.

You can use this chapter to check that you have covered the implementation steps described in the other chapters.

6.2 Key implementation tasks

Table 6-1 lists the key tasks for implementation.

Table 6-1 Key implementation tasks

Key task		Description
1.	Select level of hierarchy to implement. This can be either: a. The integration level, `CORTEXM0INTEGRATION`. b. The processor component level, `CORTEXM0`. c. A higher level in your SoC that includes the Cortex-M0 processor.	See *About configuration guidelines* on page 2-2 and *Configuration options* on page 2-3
2.	Configure the processor parameters.	See *Configuration options* on page 2-3
3.	Select appropriate library cells for clock gating and *Clock-Domain Crossing* (CDC) purposes.	*Other considerations for implementation* on page 6-4
4.	Determine optimum floorplan.	See *Considerations for floorplans* on page 7-6
5.	Perform synthesis.	See the Reference methodology documents from your EDA tool vendor for information on equivalence checking tools.
6.	Create layout.	
7.	Perform LVS and DRC checks.	
8.	Perform timing verification.	
9.	Perform characterization.	
10.	Run DFT.	See Chapter 8 *Design For Test* and the Reference methodology documents from your EDA tool vendor.
11.	Perform netlist dynamic verification.	See Chapter 9 *Netlist Dynamic Verification*.
12.	Perform functional verification using logical equivalence checking tools. Optionally, you can also replay test vectors.	See the Reference methodology documents from your EDA tool vendor.
13.	Perform sign-off in accordance with the agreed criteria and your sign-off obligations.	See Chapter 10 *Sign-off*
14.	Sign off your implementation.	

Note

The outputs from the tasks in Table 6-1 must produce complete and verified deliverables as described in *Requirements for sign-off* on page 10-4.

6.3 Other considerations for implementation

This section describes points that you must consider when implementing your design that are not covered by the configuration options described in Chapter 2 *Configuration Guidelines*.

6.3.1 Architectural clock gating

The Cortex-M0 design instantiates some clock gating cells to reduce the dynamic power dissipation by gating the clock to groups of registers within the design. These clock gates are called architectural clock gates to distinguish them from the clock gates that the synthesis tools use during the implementation process. The architectural clock gating cell must be provided as a module named `cm0_acg.v`.

Architectural clock gating is optional because correct operation of the processor is not dependent on it. If architectural clock gating is not required:

1. Configure your processor to have the `ACG` parameter set to 0.
2. Ensure that your `cm0_acg` module drives the **CLKOUT** output directly from the **CLKIN** input.

If architectural clock gating is required:

1. Configure your processor to have the `ACG` parameter set to 1.
2. Ensure that your `cm0_acg` module directly instantiates a positive-edge clock gating cell from your library.
3. Connect the clock inputs and outputs, clock enable and scan enable signals correctly.

The `logical/models/cells/cm0_acg.v` is a configurable module that you can use for simulation and logical equivalence checking. Do not use this version for synthesis.

6.3.2 Special purpose cells

The Cortex-M0 processor and its associated example system components have a number of interfaces that require special consideration other than the logical behavior described in the RTL. These include, for example, CDC boundaries and reset synchronizers.

To ensure that all these cases are implemented correctly, any single logic function with special requirements has its own individual cell description. Each cell can be instantiated a number of times within the design. For each different cell, you must

provide a module definition that uses chosen elements from your library that meet the specific cell requirements. The Reference Methodology provides modules for any given library. See the Reference Methodology release note for the location of these files.

For each individual cell, there is a reference version provided in `logical/models/cells`. This is provided to enable RTL simulation and logical equivalence checking for Sign-off. You must not use these versions for synthesis.

The cells that you must provide depend on the components you include in your design. Table 6-2 shows all the cells in the Cortex-M0 processor and its associated example system components. If you do not include a particular component in your SoC, then follow the directions given in the reference versions of each cell to ensure no extra logic is included in your design as a result of the cells that particular component includes.

Table 6-2 Cortex-M0 special purpose cells

Component	Cells
CORTEXM0DAP [a]	cm0_dap_cdc_capt_sync.v
	cm0_dap_cdc_comb_and.v
	cm0_dap_cdc_comb_and_addr.v
	cm0_dap_cdc_comb_and_data.v
	cm0_dap_cdc_send.v
	cm0_dap_cdc_send_addr.v
	cm0_dap_cdc_send_data.v
	cm0_dap_cdc_send_reset.v
	cm0_dap_jt_cdc_comb_and.v
	cm0_dap_sw_cdc_capt_reset.v
CORTEXM0INTEGRATION[b]	cm0_dbg_reset_sync.v

Table 6-2 Cortex-M0 special purpose cells (continued)

Component	Cells
System components	cm0_rst_send_set.v
	cm0_rst_sync.v
	cm0_pmu_acg.v
	cm0_pmu_cdc_send_reset.v
	cm0_pmu_cdc_send_set.v
	cm0_pmu_sync_reset.v
	cm0_pmu_sync_set.v

a. This is included if you integrate CORTEXM0INTEGRATION configured with debug.
b. This is only included if configured with debug.

Table 6-3 shows the module logical behavior and required DAP properties for correct CDC and glitch safe operation.

Table 6-3 Required properties for the processor and the DAP

Module	Parameter	Description
cm0_acg	ACG	Logically, these modules are architectural clock gates consisting of a latch and an AND gate. You must replace the clock gate simulation model in these modules with an architectural clock gate from your library to avoid glitches on the gated clock output. You must also connect the input module input **SE** to the bypass input of the architectural clock gate to ensure that implementation inserted scan chains can be clocked when in shift mode.
cm0_dap_cdc_capt_sync	PRESENT	Logically, these modules are two-flop synchronizers used to sample signals asynchronous to the reference clock. Modules with a _reset suffix have an asynchronous reset and modules with a _set suffix have an asynchronous set. You must choose flops designed to minimize the time required to resolve metastable outputs when the input setup or hold constraints are violated.
cm0_dap_sw_cdc_capt_reset	PRESENT	Logically, this module is a capture register that has a synchronous enable input with an asynchronous reset. The reset state of this register must be 1 for correct operation. This module samples input signals from a different clock domain. The logic of this flop must sample the data input only when the enable is asserted synchronously with the module clock. When the synchronous enable is deasserted, they must not be affected by a changing or metastable input that violates the data input setup or hold constraints. The output must remain stable around clock edges when the synchronous enable is deasserted.
cm0_dap_cdc_send	PRESENT	Logically, these modules are sets of launch flops with a synchronous enable input. Modules with a _reset suffix have an asynchronous reset and modules with a _set suffix have an asynchronous set. The other modules have a reset input that only requires connection if the reset all registers option is used at the CORTEXM0INTEGRATION level. The output of these registers must remain stable around clock edges when the synchronous enable is deasserted, or when the synchronous enable is asserted and the input to be loaded does not logically change the output.
cm0_dap_cdc_send_reset	PRESENT	
cm0_dap_cdc_send_addr	PRESENT	
cm0_dap_cdc_send_data	PRESENT	

Table 6-3 Required properties for the processor and the DAP (continued)

Module	Parameter	Description
cm0_dap_cdc_comb_and	PRESENT	Logically, this module is a set of AND gates that mask a vector of input signals, generated in a source clock domain, because they are passed to logic in a destination clock domain. The default instance is a single AND gate. The _addr and _data provide 4-bit and 32-bit AND masks, respectively. The mask input is driven synchronously to the destination clock domain, and the vector of output signals is sampled synchronously to the destination clock domain. When the mask input is LOW, the module holds the outputs LOW, regardless of the state of the other input signals, even if they are changing or metastable. The design ensures that the mask input is asserted only when the input signals are known to be stable.
cm0_dap_cdc_comb_and_addr	PRESENT	
cm0_dap_cdc_comb_and_data	PRESENT	
cm0_dap_jt_cdc_comb_and	PRESENT	
cm0_dbg_reset_sync	PRESENT	Logically, these modules are three-flop shift registers with an asynchronous reset, where the input of the shift register is connected to logic 1. The purpose of this module is to produce a reset that is asynchronously asserted and synchronously deasserted from a reset that is both asserted and deasserted asynchronously. In the case of cm0_rst_sync, the reset must also be asserted synchronously relative to a synchronous, non-glitching reset request input. The final D-type in the synchronizer must be guaranteed to change cleanly, that is, never glitch while reset is held inactive.

Table 6-4 shows the module logical behavior and required PMU and reset controller properties for correct CDC and glitch safe operation.

Table 6-4 Required properties for the example PMU and the reset controller

Module	Parameter	Description
cm0_pmu_acg	ACG	Logically, these modules are architectural clock gates consisting of a latch and an AND gate. You must replace the clock gate simulation model in these modules with an architectural clock gate from your library to avoid glitches on the gated clock output. You must also connect the input module input **SE** to the bypass input of the architectural clock gate to ensure that implementation inserted scan chains can be clocked when in shift mode.
cm0_pmu_sync_reset	-	Logically, these modules are two-flop synchronizers used to sample signals asynchronous to the reference clock. Modules with a _reset suffix have an asynchronous reset and modules with a _set suffix have an asynchronous set. You must choose flops designed to minimize the time required to resolve metastable outputs when the input setup or hold constraints are violated.
cm0_pmu_sync_set	-	
cm0_rst_send_set	-	Logically these modules are sets of launch flops with a synchronous enable input. Modules with a _reset suffix have an asynchronous reset and modules with a _set suffix have an asynchronous set. The other modules have a reset input that only requires connection if the reset all registers option is used at the CORTEXM0INTEGRATION level. The output of these registers must remain stable around clock edges when the synchronous enable is deasserted, or when the synchronous enable is asserted and the input to be loaded does not logically change the output.
cm0_pmu_cdc_send_reset	-	
cm0_pmu_cdc_send_set	-	
cm0_rst_sync	-	Logically, these modules are three-flop shift registers with an asynchronous reset where the input of the shift register is connected to logic 1. The purpose of this module is to produce a reset that is asynchronously asserted and synchronously deasserted from a reset that is both asserted and deasserted asynchronously. In the case of cm0_rst_sync, the reset must also be asserted synchronously relative to a synchronous, non-glitching reset request input. The final D-type in the synchronizer must be guaranteed to change cleanly, that is, never glitch while reset is held inactive.

For each module, the parameter indicates whether the particular instance is required, according to the RTL configuration. When the appropriate parameter is non-zero, the corresponding cells must be present.

Chapter 7
Floorplan Guidelines

This chapter describes the floorplan used as a starting point for your design. It contains the following sections:

7.1 About floorplanning

ARM recommends that you perform the synthesis and layout at the CORTEXM0INTEGRATION level of hierarchy, or a higher level of hierarchy in your SoC.

Figure 7-1 shows the top level inputs, resources, outputs, and controls and constraints for floorplanning.

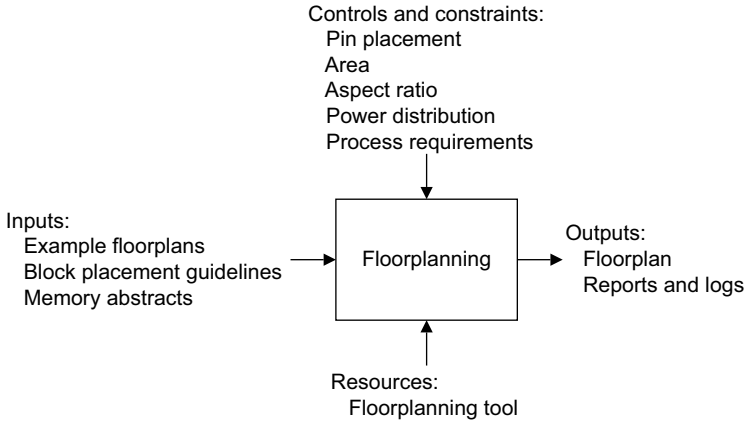


Figure 7-1 Floorplan process

7.2 Resource requirements for floorplanning

This guide assumes that you have suitable EDA tools and compute resources for floorplanning.

7.3 Controls and constraints for floorplanning

The aspect ratio of the floorplan depends on the size and number of memories in your system. You must minimize the paths from the processor to your memory blocks and the paths from your memory blocks to the processor. You must also ensure that the standard cells for the processor are grouped together to ensure good timing performance. A poor floorplan or incorrectly grouped standard cells reduce the timing performance.

ARM does not expect you to perform hierarchical floorplanning of the processor.

7.4 Inputs to floorplanning

Inputs are specific to your floorplanning tool, and can include the following:

- example floorplans
- block placement guidelines
- memory abstracts.

7.5 Considerations for floorplans

ARM recommends that you incorporate the power grid in the floorplan passed to your place and route tool so that a more accurate representation of the available routing resource is presented.

You must design the grid to meet the requirements of your library. However, ARM recommends a grid that satisfies an IR drop of 2% for V_{DD} and V_{SS} combined.

7.6 Outputs from floorplans

Output files are specific to your floorplanning tool and might include:

- logs
- floorplan with power grid and pin locations
- placement and route guides.

Chapter 8
Design For Test

This chapter describes the *Design for Test* (DFT) features of the processor. It contains the following sections:

8.1 About design for test

The processor uses *Automatic Test Pattern Generation* (ATPG) test vectors for production test. To support ATPG, the synthesis tools insert scan chains into the macrocell. See the Reference Methodology documents supplied by your EDA tool vendor for more information.

During synthesis, your synthesis tools might support optional insertion of a test wrapper. This enables the creation of a black-box view of the macrocell, without reducing the testability of the macrocell or your SoC.

The processor is a fully synchronous design with all registers clocked off the rising clock edge, which means you can achieve a very high test coverage.

Figure 8-1 shows the top level inputs, resources, outputs, and controls and constraints for DFT.

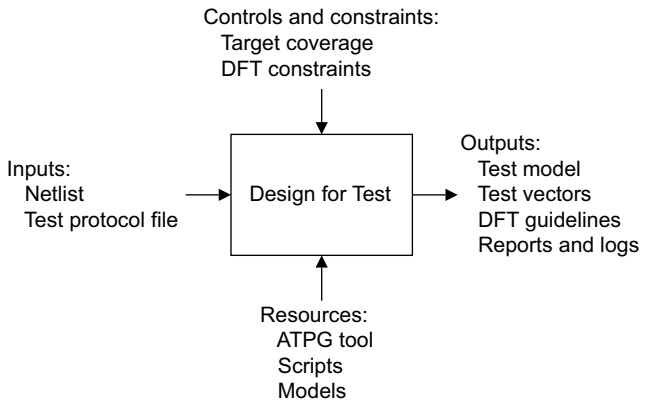


Figure 8-1 Design for Test process

8.2 Requirements for DFT

This guide assumes that you have suitable EDA tools and compute resources for DFT.

8.3 Controls and constraints for DFT

The following sections describe the controls and constraints for DFT:

- *Clock gating*
- *Reset*.

8.3.1 Clock gating

To ensure that the scan chains operate correctly, without affecting the fault coverage, you must ensure that the clock gating cells are forced to enable when scan enable is asserted. This applies to:

- clock gates that are inferred in the RTL and inserted automatically by the tools during the implementation flow
- architectural clock gates instantiated in the RTL.

Use **SE** to enable the clock gating cells during scanning in or out of test patterns.

Drive **SE** HIGH during scan testing, and LOW during normal operation.

8.3.2 Reset

You can use **RSTBYPASS** to bypass the reset synchronizers during scan testing.

You must drive **RSTBYPASS** HIGH during the scan part of scan testing, and LOW during normal operation.

Note

This reset bypass pin is available only in the `CORTEXM0INTEGRATION` level.

The `CORTEXM0` level does not implement reset synchronizers.

8.4 DFT features

See the Reference Methodology documents from your EDA tool vendor for information on DFT features.

8.5 Reference data for DFT

The following sections give reference data for DFT:

- *Scan test insertion*
- *Test wrapper insertion*.

8.5.1 Scan test insertion

Table 8-1 lists the scan test ports that access the internal scan chains in the macrocell.

Table 8-1 Scan test ports

Name	Direction	Description
RSTBYPASS	Input	Bypass reset synchronizers during scan testing so test tools have direct control over the resets.
SE	Input	Enable Scan chains and force enable clock gate cells. High fan out of this signal means this can be a multicycle path, and cannot be switched at-speed. This signal must be tied LOW during functional mode.

Note

See the Reference Methodology documentation from your EDA tools vendor for details of the scan ports.

8.5.2 Test wrapper insertion

If you implement a test wrapper, it provides production test access to the inputs and outputs of the processor. This gives increased test coverage when access to the primary inputs and outputs is impossible because the macrocell is deeply embedded in your SoC design.

Note

- See the Reference Methodology documentation from your EDA tools vendor for details of the test wrapper ports.
- If you use a top-down flow you do not require a test wrapper because the test tools have complete visibility of all logic paths. Because the processor does not have registered interfaces, ARM recommends that you do not have a test wrapper.

Chapter 9
Netlist Dynamic Verification

This chapter describes how to use test vectors to test the functionality of your implementation of the processor. It contains the following sections:

9.1 Netlist dynamic verification

You must use equivalence checking tools to verify your post-synthesis and post-layout netlists. This is described in the Reference Methodology documentation from your EDA tools vendor.

Note

ARM requires you to use equivalence tools to verify your netlist. Equivalence checking provides a complete method for verifying your netlist and does not require the extensive simulation compute resources that other methods require.

In addition, you can use test vectors to perform quick netlist checks before you perform formal or functional verification. This is described in this chapter, and Figure 9-1 shows the top level inputs, resources, outputs, and controls and constraints for this netlist dynamic verification.

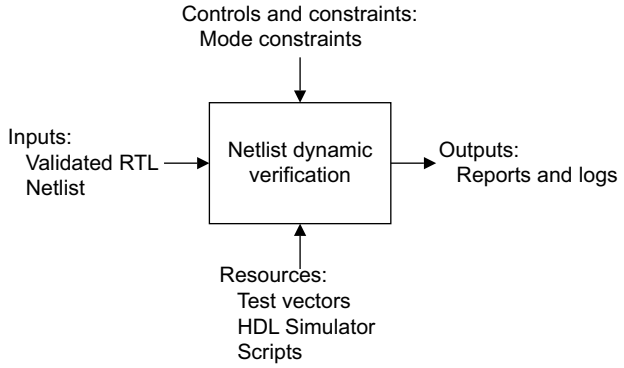


Figure 9-1 Netlist dynamic verification process

This verification is described in *Verification using test vectors* on page 9-5.

9.2 Resource requirements for netlist dynamic verification

To run netlist dynamic verification, you require various test vectors. The test vectors are generated by integration kit tests. See *Test vector capture* on page 5-28 for details of how to capture test vectors from your configured RTL.

Table 9-1 shows the netlist dynamic verification test vector locations and descriptions.

Table 9-1 Netlist dynamic verification test vector descriptions

Test vector directory	Content description
`implementation/vectors/CORTEXM0IMP/crf`	Test vector files captured at the `CORTEXM0IMP` implementation level
`implementation/vectors/CORTEXM0IMP/srpg`	UPF and CPF files for the `CORTEXM0IMP` level
`implementation/vectors/CORTEXM0IMP/tbench`	Verilog testbench for replaying the test vectors at the `CORTEXM0IMP` implementation level
`implementation/vectors/CORTEXM0INTEGRATIONIMP/crf`	Test vector files captured at the `CORTEXM0INTEGRATIONIMP` implementation level
`implementation/vectors/CORTEXM0INTEGRATIONIMP/srpg`	UPF and CPF files for `CORTEXM0INTEGRATIONIMP` level
`implementation/vectors/CORTEXM0INTEGRATIONIMP/tbench`	Verilog testbench for replaying the test vectors at the `CORTEXM0INTEGRATIONIMP` implementation level
`implementation/vectors/tools`	Scripts to process vectors and run vector replay testbench
`logical`	Verilog source for the macrocell

9.3 Inputs for netlist dynamic verification

Inputs are specific to your verification tool, and can include:

- validated RTL
- netlist
- test vectors.

9.4 Flow for netlist dynamic verification

This section describes verification using test vectors, and it contains the following subsections:

- *Verification using test vectors*
- *Test vector directory structure*
- *Test vector simulation environment and vector replay* on page 9-6.

Note

You must also perform verification using equivalence checking tests. See the Reference Methodology documentation from your EDA tool vendor for more information.

9.4.1 Verification using test vectors

You can use test vectors to perform quick netlist checks before performing formal verification. Because the vectors are captured using the configured source RTL as a reference, replaying the vectors checks that your netlist matches the cycle-by-cycle behavior of the RTL reference. The test vectors are not suitable for design sign-off because they focus on signal connectivity rather than processor functionality.

9.4.2 Test vector directory structure

The Cortex-M0 test vectors are located in the following directories:

`implementation/vectors/CORTEXM0IMP/crf`

This directory contains the test vector `crf` files at the `CORTEXM0IMP` implementation level. You must generate these files using the integration kit tests and copy them to the `crf` directory.

See *Test vector capture* on page 5-28 for more information about how to capture test vectors at the `CORTEXM0IMP` level.

`implementation/vectors/CORTEXM0INTEGRATIONIMP/crf`

This directory contains the test vector `crf` files at the `CORTEXM0INTEGRATIONIMP` implementation level. You must generate these files using the integration kit tests and copy them to the `crf` directory.

See the *Test vector capture* on page 5-28 for more information on how to capture test vectors at `CORTEXM0INTEGRATIONIMP` level.

`implementation/vectors/CORTEXM0IMP/tbench`

This directory contains the testbench for replaying vectors at the `CORTEXM0IMP` implementation level.

`implementation/vectors/CORTEXM0INTEGRATIONIMP/tbench`

This directory contains the testbench for replaying vectors at the `CORTEXM0INTEGRATIONIMP` implementation level.

9.4.3 Test vector simulation environment and vector replay

This section describes how to build the replay testbench and replay the vectors on the testbench.

From the `implementations/vectors` directory execute the following command:

```
source dotcshrc
```

Run the testbench from the `implementation/vectors/CORTEXM0IMP/tbench` or `implementation/vectors/CORTEXM0NTEGRATIONIMP/tbench` directory, depending on the level at which you want to replay vectors. The `Replay` script is provided as an example script to build the testbench and replay the test vectors. Ensure that the RTL or netlist configuration in the replay testbench is the same as the one used to capture vectors in the integration kit.

Replay script usage and options

The `Replay` script has the following command format:

```
Replay <options> <testname>
```

Where `<options>` are:

`-build`	Optional switch. Use this to compile the testbench before running the specified test.
`-all`	Use this switch to run a batch of tests.
`-integration`	Use the `CORTEXM0INTEGRATIONIMP` level. Default is `CORTEXM0IMP`.
`-srpg`	Use the SRPG pins for vector replay
`-netlist`	Use this switch to use netlist for simulation.
`-sdf <delay type>`	Use SDF with the netlist. `<delay type>` can be `min` or `max`. If this switch is not used, zero delay is assumed.
`-mti`	Use the MTI simulator. `-mti` is the default.

-nc	Use the NC simulator.
-vcs	Use the VCS simulator.
-upf	Use UPF for simulation. This can only be used with the -srpg option.

<testname> gives the name of the .crf file to run. Depending on the level at which you are replaying the vectors, this file is in either the implementation/vectors/CORTEXM0IMP/crf directory or the implementation/vectors/CORTEXM0INTEGRATIONIMP/crf directory. <testname> must give the name of the .crf file without the .crf extension.

Replaying testbench vectors

To build the CORTEXM0IMP Replay testbench with SRPG pins and run the vector sample_test_CORTEXM0IMP.crf using VCS, type the following command from the directory implementation/vectors/CORTEXM0IMP/tbench:

```
Replay -vcs -srpg -build sample_test
```

Depending on the simulator you specify, the Replay script creates the following directories for compilation and simulation:

- MTI
- VCS
- NC.

Ensure that your chosen simulator is on your path before running the Replay script.

To simulate with UPF, ensure that mvtools suite is on your environment path variable.

You can modify the Replay script to include extra simulator options you want to use. For example, modify the array @testlist in Replay to add tests for batch test replay.

The testbench prints a summary at the end of simulation. Verify that all vectors replay with zero errors.

9.5 Outputs from netlist dynamic verification

The log files output are specific to your verification.

Chapter 10
Sign-off

In addition to your normal ASIC flow sign-off checks, you must satisfy certain verification criteria before you sign off your design. This chapter describes the sign-off criteria. It contains the following sections:

10.1 About sign-off

Figure 10-1 shows the top level inputs, resources, outputs, and controls and constraints for sign-off.

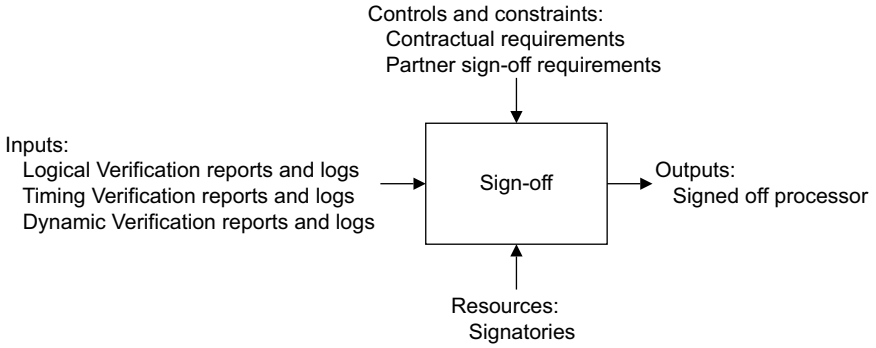


Figure 10-1 Sign-off process

10.2 Obligations for sign-off

Signatories must approve the sign-off of the design in accordance with:

- the terms of the contract with ARM
- any other partner sign-off requirements.

See *Implementation controls and constraints* on page 1-7 for more information.

10.3 Requirements for sign-off

The following sections describe the two types of requirement for sign-off:

- *Mandatory for sign-off*
- *Recommended for sign-off*.

10.3.1 Mandatory for sign-off

All ARM partners must fulfill the terms of their contract with ARM to complete sign-off.

In most cases, you must complete the following implementation stages successfully for sign-off:

- Logical Equivalence Check.
  See the Reference Methodology documents supplied by your EDA tool vendor.

Reports and logs from each of these stages are required for sign-off.

A minimum set of deliverable outputs is required at the end of the implementation. See *Completion of sign-off* on page 10-6.

Note

You can change the timing constraints to suit your design provided it still meets all the mandatory requirements for sign-off.

10.3.2 Recommended for sign-off

Table 10-1 shows the recommended stages for sign-off.

Table 10-1 Recommended stages for sign-off

Sign-off stage	Notes
Design Rule Check (DRC)	See the Reference Methodology documents supplied by your EDA tool vendor
Layout Versus Schematic (LVS)	
Timing Verification	
Characterization	
Vector replay with back annotated netlist	If you are using a pre-hardened processor at the `CORTEXM0` or `CORTEXM0INTEGRATION` levels

10.4 Steps for sign-off

To sign off the processor you must meet the criteria in each of the following stages in the design flow:

1. *RTL integration*
2. *Post-synthesis and place-and-route*
3. *Post place-and-route timing*.

Note

You must also ensure you meet any additional verification or usage criteria that might be identified in the legal agreement between your company and ARM.

10.4.1 RTL integration

You must verify the RTL deliverables before you begin the synthesis stage by running the supplied integration kit on the configured RTL. This confirms that the RTL has been successfully installed and configured.

Note

You must use the files provided in `models/cells/` for RTL integration.

Do not use the implementation versions of files for RTL integration. See *Special purpose cells* on page 6-4 for additional information.

10.4.2 Post-synthesis and place-and-route

You must verify the functionality of the final placed-and-routed netlist before you sign off the macrocell. This verification requires you to prove logical equivalence between the validated processor RTL and the final place-and-routed netlist, using formal verification tools. See the Reference Methodology documents supplied by your EDA tool vendor.

10.4.3 Post place-and-route timing

You must verify the timing of the post place-and-route netlist before you sign off the netlist using *Static Timing Analysis* (STA). ARM also recommends that you run:

- all functional vectors appropriate to your configured build
- some or all of the supplied validation tests on a netlist with back-annotated timing as a final check.

10.5 Completion of sign-off

For successful completion of sign-off, you must have the required completed and verified ARM related deliverables resulting from the implementation process.

These include:

- GDS II output
- test vectors
- extracted timing model
- simulation model
- all required reports and logs.

Appendix A
GPIO Programmers Model

This appendix describes the GPIO programmers model. It contains the following section:

- *GPIO programmers model* on page A-2.

A.1 GPIO programmers model

This section describes the GPIO programmers model. It contains the following sections:

- *Data Register - GPIODATA*
- *Direction Register - GPIODIR* on page A-4
- *Interrupt Enable Register - GPIOIE* on page A-4.

The GPIO is a general purpose I/O device. It has the following properties:

- three registers
- 32 input or output lines with programmable direction
- word and halfword read and write access
- address-masked byte write to facilitate quick bit set and clear operations
- address-masked byte read to facilitate quick bit test operations
- maskable interrupt generation based on input value change.

Table A-1 lists the GPIO registers.

Table A-1 GPIO registers

Address offset	Name	Type	Description
0x00000000-0x000003FF	**GPIODATA**	R/W	Reads current value of **GPIOIN** pins, or sets value driven onto **GPIOUT** pins.
0x00000400	**GPIODIR**	R/W	Configures the direction of the I/O. The value is driven on **GPIOEN.** Setting a bit defines that bit as an output.
0x00000410	**GPIOIE**	R/W	Configures the interrupt on input change feature.

A.1.1 Data Register - GPIODATA

Use this register to read the value of the **GPIOIN** pins when the corresponding **GPIODIR** is 0.

Use this register to drive the value onto the **GPIOOUT** pins when the corresponding **GPIODIR** is 1.

Note

Reading **GPIODATA** when **GPIOEN** is 1 returns the value seen on the **GPIOIN** pin.

The register address, access type, and reset state are:

Address GPIO_BASE to GPIO_BASE + 0x000003FF

Access Read/write

Reset state `0x00000000`

Byte accesses to the Data Register use the bits **HADDR[9:2]** as a mask. This enables you to perform various bit set and bit clear operations efficiently because it avoids the requirement for a read-modify-write to the **GPIODATA** register.

Bit set example

To set bit [9] of GPIO 0 Data Register, perform a byte write access to address `0x40000009` with **HWDATA** set to `0x200` and **HSIZE** set to `3'b000`.

The mask extracted from the address is `0x02` and applied to the second byte of the GPIO 0 Data Register.

This sets bit [9] but preserves all other bits.

Bit clear example

To clear bit [9] of GPIO 0 Data Register, perform a byte write access to address `0x40000009` with **HWDATA** set to `0x0` and **HSIZE** set to `3'b000`.

The mask extracted from the address is `0x02` and applied to the second byte of the GPIO 0 Data Register.

This clears bit [9] but preserves all other bits.

Bit read example

To read bit [9] of GPIO 0 Data Register, perform a byte read access to address `0x40000009` with **HSIZE** set to `3'b00`.

The mask extracted from the address is `0x02` and applied to the second byte of the GPIO 0 Data register.

HRDATA contains the value of bit [9], all other bits are zeroes.

Figure A-1 on page A-4 shows the structure of the GPIO Data Register.

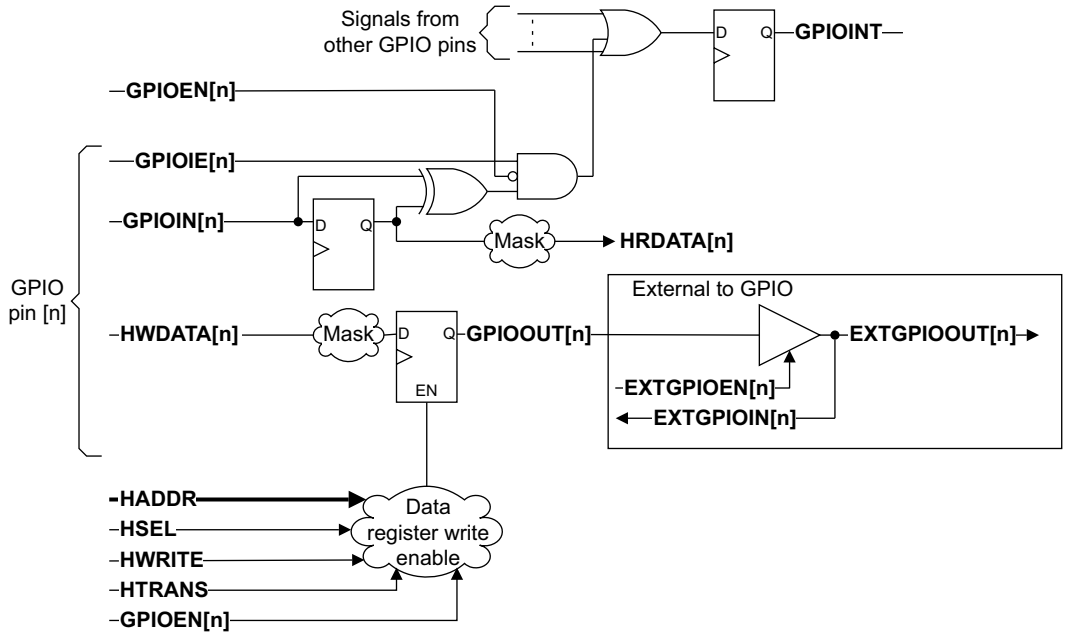


Figure A-1 GPIO Data Register

The external signals **EXTGPIOOUT**, **EXTGPIOIN**, and **EXTGPIOEN** map to **GPIOOUT**, **GPIOIN**, and **GPIOEN** respectively.

You can use the **GPIOINT** output pin of the GPIO as an interrupt source.

A.1.2 Direction Register - GPIODIR

Use this register to configure the direction of the **Data Register** as either `input` or `output`. This connects to the **GPIOEN** pin and is used as a tristate enable. Set bits in this register to `1'b1` when you want to use the bit as an output.

The register address, access type, and reset state are:

Address `GPIO_BASE + 0x00000400`

Access Read/write

Reset state `0x00000000`

A.1.3 Interrupt Enable Register - GPIOIE

Use this register to enable input signal changes on **GPIOIN** to trigger an interrupt through the **GPIOINT** output.

You can set **GPIOIE[n]** to enable changes on **GPIOIN[n]** to pulse the **GPIOINT** pin.

The register address, access type, and reset state are:

Address `GPIO_BASE + 0x00000410`
Access Read/write
Reset state `0x00000000`

Appendix B
CM0IKMCU GPIO Integration

This appendix describes the integration of the GPIO within the CM0IKMCU example system. It contains the following section:

- *GPIO Integration* on page B-2.

B.1 GPIO Integration

This section describes the GPIO bit assignments used in the integration kit. It contains:

- *GPIO 0 bit assignments* on page B-4
- *GPIO 1 bit assignments* on page B-5
- *GPIO 2 bit assignments* on page B-6.

The Cortex-M0 integration kit instantiates three GPIOs internally, that is, GPIO 0, 1, and 2. For more information about GPIO, see Appendix A *GPIO Programmers Model*.

Figure B-1 shows the connection of the GPIO interrupt lines.

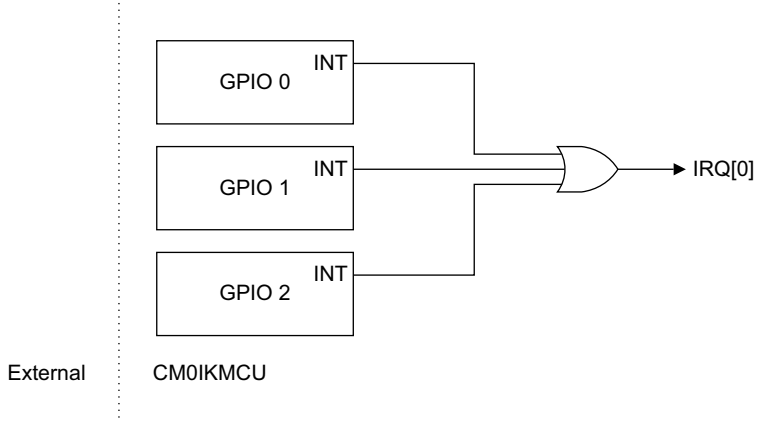


Figure B-1 GPIO interrupt line connections

Figure B-2 on page B-3 shows the GPIO 0 connections.

Note

IRQ[0] is the logical OR of the **INT** outputs from the three GPIOs. This enables the integration kit interrupt tests to work even when the processor is configured to have only one external interrupt.

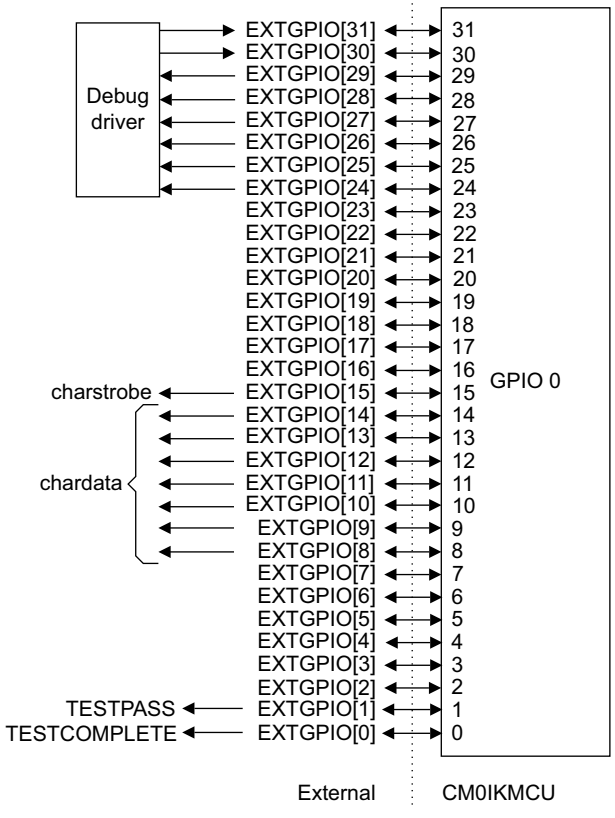


Figure B-2 GPIO 0 connections

Figure B-3 on page B-4 shows the GPIO 1 connections.

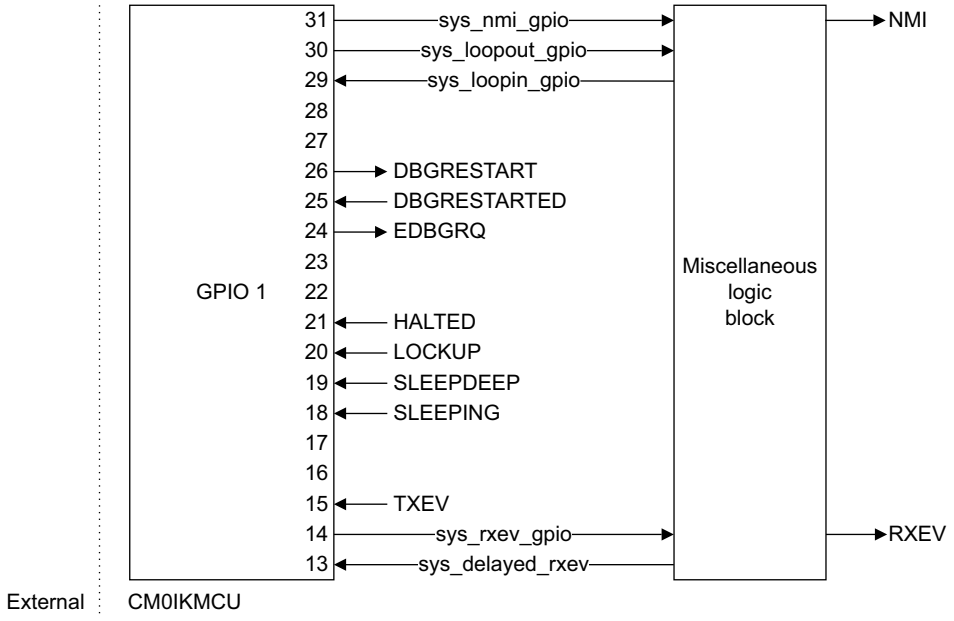


Figure B-3 GPIO 1 connections

Figure B-4 shows the GPIO 2 connections.

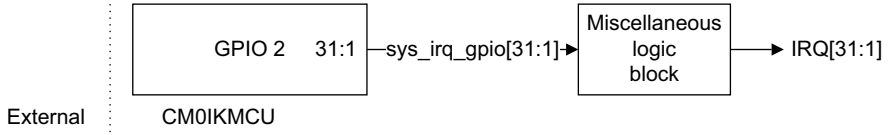


Figure B-4 GPIO 2 connections

B.1.1 GPIO 0 bit assignments

GPIO 0 is connected to the external GPIO pins **EXTGPIO[31:0]**. GPIO 0 provides the I/O capabilities of the example CM0IKMCU, with its signals routed to the top level of the CM0IKMCU.

The **EXTGPIO** signals are used in the integration kit testbench to indicate test completion and pass or fail status. They also provide a simple character output device and control the debug driver block.

Table B-1 shows the GPIO 0 bit assignments in the Cortex-M0 integration kit.

Table B-1 GPIO 0 bit assignments

Bit	Receive input from	Send output to	Connection information
[31]	**EXTGPIO[31]**	-	Running signal from Debug Driver
[30]	**EXTGPIO[30]**	-	Error signal from Debug Driver
[29]	-	**EXTGPIO[29]**	Function Strobe to Debug Driver
[28:24]	-	**EXTGPIO[28:24]**	Function Select to Debug Driver
[23:16]	-	-	Not used
[15]	-	**EXTGPIO[15]**	Connected to **charstrobe** in the top level testbench
[14:8]	-	**EXTGPIO[14:8]**	Connected to **chardata** in the top level testbench
[7:2]	-	-	Not used
[1]	-	**EXTGPIO[1]**	**TESTPASS** in the top level testbench
[0]	-	**EXTGPIO[0]**	**TESTCOMPLETE** in the top level testbench

B.1.2 GPIO 1 bit assignments

GPIO 1 is used internally to the example CM0IKMCU to drive and monitor core signals for testing purposes only.

These signals can drive, or can be driven by, other blocks in your SoC.

Table B-2 shows the bit assignments.

Table B-2 GPIO 1 bit assignments

Bit	Receive input from	Send output to	Connection information
[31]	-	**NMI**	Drives the **NMI** pin of the Cortex-M0 processor, after a delay
[30]	-	Miscellaneous logic block	Drives **GPIOIN[29]**, after a delay
[29]	Miscellaneous logic block	-	Delayed version of **GPIOOUT[30]**
[28:27]	-	-	Not used
[26]	-	**DBGRESTART**	Drives the **DBGRESTART** pin of the Cortex-M0 processor

Table B-2 GPIO 1 bit assignments (continued)

Bit	Receive input from	Send output to	Connection information
[25]	**DBGRESTARTED**	-	Connects to the **DBGRESTARTED** pin of the Cortex-M0 processor
[24]	-	**EDBGRQ**	Drives the **EDBGRQ** pin of the Cortex-M0 processor
[23:22]	-	-	Not used
[21]	**HALTED**	-	Connects to the **HALTED** pin of the Cortex-M0 processor
[20]	**LOCKUP**	-	Connects to the **LOCKUP** pin of the Cortex-M0 processor
[19]	**SLEEPDEEP**	-	Connects to the **SLEEPDEEP** pin of the Cortex-M0 processor
[18]	**SLEEPING**	-	Connects to the **SLEEPING** pin of the Cortex-M0 processor
[17:16]	-	-	Not used
[15]	**TXEV**	-	Connects to the **TXEV** pin of the Cortex-M0 processor
[14]	-	**RXEV**	Drives the **RXEV** pin of the Cortex-M0 processor
[13]	Miscellaneous logic block	-	Delayed version of the **RXEV** pin of the Cortex-M0 processor
[12:0]	-	-	Not used

B.1.3 GPIO 2 bit assignments

GPIO 2 is used internally to the example CM0IKMCU to drive **IRQ[31:1]** for testing purposes only.

These signals can be driven by other blocks in your SoC.

Table B-3 shows the GPIO 2 bit assignments.

Table B-3 GPIO 2 bit assignments

Bit	Receive input from	Send output to	Connection information
[31:1]	-	Miscellaneous logic block	Drives the corresponding **IRQ** pin of the Cortex-M0 processor, after a delay. For example, bit[31] drives **IRQ[31]**.
[0]	-	-	Not used.

Appendix C
SysTick Examples

This appendix provides some examples of setting up SysTick timing. It contains the following section:

C.1 SysTick examples

Example C-1 shows 25MHz **SCLK** with no reference provided.

Example C-1 25MHz SCLK, no reference provided

```
STCALIB[25]   = 1'b1; // no reference provided
STCALIB[24]   = 1'b0; // no skew
STCALIB[23:0] = (25MHz x 10mS) - 1
              = 249,999 decimal
              = 24'h03D08F
```

Example C-2 shows 14.31818MHz **SCLK** with no reference provided.

Example C-2 14.31818MHz SCLK, no reference provided

```
STCALIB[25]   = 1'b1; // no reference provided
STCALIB[24]   = 1'b1; // skew
STCALIB[23:0] = (14.31818MHz x 10mS) - 1
              = 143,180.8 decimal
              = 143,181 with skew
              = 24'h022F4D
```

Example C-3 shows 1MHz **SCLK** with **STCLKEN** enabled once every four cycles.

Example C-3 1MHz SCLK, STCLKEN enabled once every four cycles

```
STCALIB[25]   = 1'b0; // reference provided
STCALIB[24]   = 1'b0; // no skew
STCALIB[23:0] = (1MHz x 10mS / 4) - 1
              = 2,499 decimal
              = 24'h0009C3
```

Example C-4 shows an 32.768kHz asynchronous example.

Example C-4 32.768kHz asynchronous example

```
STCALIB[25]   = 1'b0; // reference provided
STCALIB[24]   = 1'b1; // skew
STCALIB[23:0] = (32768 x 10mS) -1
              = 326.68 decimal
              = 327 with Skew
              = 24'h000147
```

Appendix D
Debug Driver

This appendix describes the debug driver supplied with the integration kit. It contains the following section:

- *Debug driver* on page D-2.

D.1 Debug driver

The debug driver block is a testbench component that controls the Cortex-M0 DAP using the JTAG or SW pins of CM0IKMCU. The debug driver contains an instantiation of the `cm0ik_cortexm0_timing` level or `cm0ik_cortexm0integration_timing` level, memories, and a GPIO. You can control the debug driver block using the pins **EXTGPIO[31:24]** of CM0IKMCU to initiate one of several predetermined operations. This arrangement enables testing of the integration kit even when the processor in the CM0IKMCU is sleeping.

The debug driver always executes the binary file `debugdriver.bin`, regardless of the test run on CM0IKMCU. The `debugdriver.bin` binary must be built from the supplied source code. See *Test program compilation, ARM RVCT* on page 5-11 and *Test program compilation, RealView MDK-ARM* on page 5-12 for information on how to compile and build the integration kit tests.

Table 4-1 lists the software source files used by the debug driver.

Table 4-1 Debug driver source files

File	Location	Description
`core_cm0.h`	`tests/CMSIS/Core/CM0/`	CMSIS file that defines the core peripherals for the Cortex-M0 processor and core peripherals.
`core_cm0.c`	`tests/CMSIS/Core/CM0/`	CMSIS file that provides helper functions that access processor registers.
`cm0ikdebugdriver.h`	`tests/`	CMSIS device specific file that defines the peripherals for the `cm0ikdebugdriver` block.
`system_cm0ikdebugdriver.h`	`tests/`	CMSIS device vendor Header file that provides device specific configuration for the `cm0ikdebugdriver` block.
`system_cm0ikdebugdriver.c`	`tests/`	CMSIS device vendor C file that provides device specific configuration for the `cm0ikdebugdriver` block.
`boot_cm0debugdriver.c`	`tests/`	This file provides the stack and heap initialization, vector table, default handlers and `_sys_exit` function used by `cm0ikdebugdriver`.
`retarget_cm0debugdriver.c`	`tests/`	This file implements the functions necessary to retarget the C-library `printf()` function output to the `cm0ikdebugdriver` GPIO pins.

Table 4-1 Debug driver source files (continued)

File	Location	Description
debugdriver.h	tests/	This header file contains various defines used by the debug driver block, including the GPIO pin allocations.
debugdriver.c	tests/	This is the main source code file for the debug driver block. It includes the routines for communicating with CM0IKMCU through the GPIO and the routines to drive the CM0IKMCU Serial Wire or JTAG interface.
debugdriver_functions.h	tests/	This header file contains a C `enum` representing the functions that the debug driver makes available to CM0IKMCU.

Figure D-1 shows the debug driver.

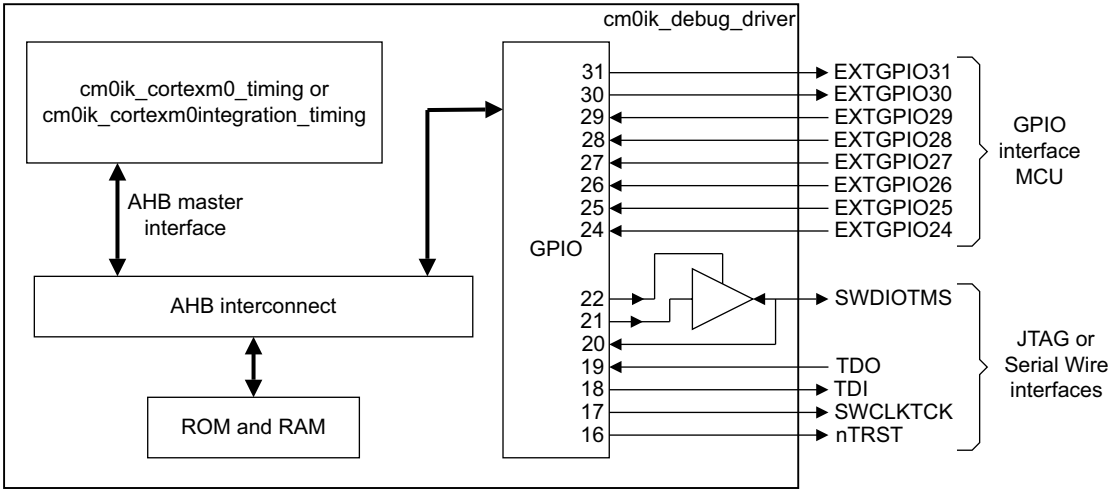


Figure D-1 Debug driver

Note

- The timing wrapper in the debug driver is the same as that used in the CM0IKMCU level.
- During netlist simulations, the implementation levels in the timing wrappers in CM0IKMCU and the debug driver are replaced by netlists.

- If any of the pins at the implementation level are modified, they must be suitably connected in the debug driver to preserve the intended behavior. For example, if you change the SRPG pins to match your requirements, ensure that these pins are tied inactive in the debug driver module.

Appendix E
Revisions

This appendix describes the technical changes between released issues of this book.

Table E-1 Issue A

Change	Location	Affects
This is the first release of this guide.	-	-

Glossary

This glossary describes some of the terms used in technical documents from ARM.

Advanced High-performance Bus (AHB)
A bus protocol with a fixed pipeline between the address or control and data phases. It supports only a subset of the functionality provided by the AMBA AXI protocol. The full AMBA AHB protocol specification includes a number of features that are not commonly required for master and slave implementations and ARM recommends using the AMBA AHB-Lite subset of the protocol.

See also Advanced Microcontroller Bus Architecture and AHB-Lite.

Advanced Microcontroller Bus Architecture (AMBA)
The AMBA family of protocol specifications is the ARM open standard for on-chip buses that provides a strategy for the interconnection and management of the functional blocks that make up a System-on-Chip (SoC). Applications include the development of embedded systems with one or more processors or signal processors and multiple peripherals. AMBA defines a common backbone for SoC modules, and therefore complements a reusable design methodology.

AHB *See* Advanced High-performance Bus.

AHB-Lite AHB-Lite is a subset of the full AHB specification. It is intended for use in designs where only a single AHB master is used. This can be a simple single AHB master system or a multi-layer AHB system where there is only one AHB master on a layer.

Aligned Refers to data items stored so that their address is divisible by the highest power of two that divides their size. Aligned words and halfwords therefore have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore refer to addresses that are divisible by four and two respectively. The terms byte-aligned and doubleword-aligned are defined similarly.

AMBA *See* Advanced Microcontroller Bus Architecture.

ATPG *See* Automatic Test Pattern Generation.

Automatic Test Pattern Generation (ATPG)
The process of automatically generating manufacturing test vectors for an ASIC design, using a specialized software tool.

Big-endian memory Memory in which:

- a byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address
- a byte at a halfword-aligned address is the most significant byte in the halfword at that address.

See also Little-endian memory.

Clock gating Gating a clock signal for a macrocell with a control signal, and using the modified clock that results to control the operating state of the macrocell.

Debug Test Access Port (DBGTAP)
A debug control and data interface based on the IEEE 1149.1 JTAG *Test Access Port* (TAP). The interface has four or five signals.

DBGTAP *See* Debug Test Access Port.

Endianness The scheme that determines the order of successive bytes of a data word when it is stored in memory. An aspect of system memory mapping.

See also Little-endian memory and Big-endian memory.

Little-endian memory
Memory in which:

- a byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address
- a byte at a halfword-aligned address is the least significant byte in the halfword at that address.

See also Big-endian memory.

Read Reads are defined as memory operations that have the semantics of a load.

See the *ARM Architecture Reference Manual* for more information.

RealView ICE A system for debugging embedded processor cores using a DBGTAP interface.

See also Debug Test Access Port.

Register A temporary storage location used to hold binary data until it is ready to be used.

Scan chain A scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

SDF *See* Standard Delay Format.

TAP *See* Test Access Port.

Test Access Port (TAP)
The collection of four mandatory terminals and one optional terminal that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **nTRST**.

Unaligned Memory accesses that are not appropriately word-aligned or halfword-aligned.

See also Aligned.