PrimeCell® Vectored Interrupt Controller (PL190)

Revision: r1p2

Technical Reference Manual

ARM®

ARM DDI 0181E

PrimeCell Vectored Interrupt Controller (PL190)
Technical Reference Manual

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Change
30 June 2000	A	First release
August 2000	B	Small corrections to code examples
September 2000	C	VICITIP1 & 2 changed to read-only. Changes to Figs 2-5 & 2-6
02 July 2003	D	Incorporate errata, revision r1p1
30 November 2004	E	Incorporate erratum, revision r1p2

Proprietary Notice

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual

List of Tables
PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual

List of Figures
PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual

Preface

This preface introduces the *PrimeCell Vectored Interrupt Controller (PL190) r1p2 Technical Reference Manual*. It contains the following sections:

About this manual

This is the *Technical Reference Manual* for the *ARM PrimeCell Vectored Interrupt Controller* (VIC).

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this manual, 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 manual is written for hardware and software engineers implementing *System-on-Chip* (SoC) designs. It provides the necessary information to enable designers to integrate the peripheral into a target system as quickly as possible. The manual describes the VIC.

Using this manual

This manual is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the VIC and its features.

Chapter 2 *Functional Overview*

Read this chapter for a description of the major functional blocks of the VIC.

Chapter 3 *Programmer's Model*

Read this chapter for a description of the registers and programming details of the VIC.

Chapter 4 *Programmer's Model for Test*

Read this chapter for a description of the test registers and signals of the VIC.

Appendix A *Signal Descriptions*

Read this appendix for a description of the VIC signals.

Appendix B *Example Code*

Read this appendix for a description of the VIC example code.

Glossary Read the Glossary for definitions of terms used in this manual.

Conventions

Conventions that this manual can use are described in:

- *Typographical*
- *Timing diagrams* on page xii
- *Signals* on page xii
- *Numbering* on page xiii.

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.

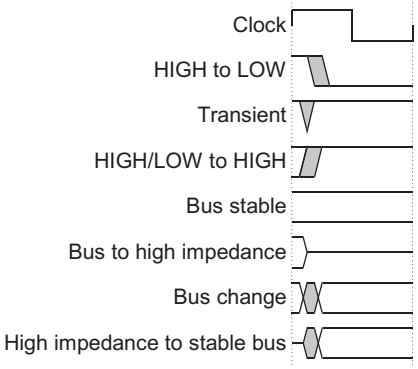
< and > Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They appear in normal font in running text. For example:

- `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`
- The Opcode_2 value selects the register that is accessed.

Timing diagrams

The figure named *Key to timing diagram conventions* explains the components that timing diagrams use. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.



Key to timing diagram conventions

Signals

The signal conventions are:

Signal level The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means HIGH for active-HIGH signals and LOW for active-LOW signals.

Prefix A Denotes *Advanced eXtensible Interface* (AXI) global and address channel signals.

Prefix B Denotes AXI write response channel signals.

Prefix C Denotes AXI low-power interface signals.

Prefix H Denotes *Advanced High-performance Bus* (AHB) signals.

Prefix n Denotes active-LOW signals except in the case of AXI, AHB, or *Advanced Peripheral Bus* (APB) reset signals.

Prefix P Denotes APB signals.

Prefix R Denotes AXI read channel signals.

Prefix W Denotes AXI write channel signals.

Suffix n Denotes AXI, AHB, and APB reset signals.

Numbering

The numbering convention is:

<size in bits>'<base><number>

This is a Verilog method of abbreviating constant numbers. For example:

- 'h7B4 is an unsized hexadecimal value.
- 'o7654 is an unsized octal value.
- 8'd9 is an eight-bit wide decimal value of 9.
- 8'h3F is an eight-bit wide hexadecimal value of `0x3F`. This is equivalent to b00111111.
- 8'b1111 is an eight-bit wide binary value of b00001111.

Further reading

This section lists publications by ARM Limited, and by third parties.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the Frequently Asked Questions list.

ARM publications

This manual contains information that is specific to the ARM PrimeCell Vectored Interrupt Controller (PL190). See the following documents for other relevant information:

- *AMBA Specification (Rev 2.0)* (ARM IHI 00011).

Feedback

ARM Limited welcomes feedback, both on the ARM PrimeCell Vectored Interrupt Controller (PL190) and its documentation.

Feedback on the product

If you have any comments or suggestions about the VIC, contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this manual

If you have any comments on this manual, send email to `errata@arm.com` giving:

- the title
- the number
- the relevant page number(s) to which your comments apply
- a concise explanation of your comments.

ARM Limited also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the *ARM PrimeCell Vectored Interrupt Controller* (VIC) (PL190). It contains the following section:

1.1 About the VIC

The VIC is an *Advanced Microcontroller Bus Architecture* (AMBA) compliant *System-on-Chip* (SoC) peripheral that is developed, tested, and licensed by ARM.

The VIC provides an interface to the interrupt system, and improves interrupt latency in two ways:

- moves the interrupt controller to the AMBA AHB bus
- provides vectored interrupt support for high-priority interrupt sources.

1.1.1 Features of the VIC

The VIC has the following features:

- compliance to the *AMBA Specification (Rev 2.0)* onwards for easy integration into SoC implementation
- support for 32 standard interrupts
- support for 16 vectored IRQ interrupts
- hardware interrupt priority
- IRQ and FIQ generation
- AHB mapped for faster interrupt response
- software interrupt generation
- test registers
- raw interrupt status
- interrupt request status
- interrupt masking
- privileged mode support
- vector interrupt controller daisy-chaining support.

1.2 Product revisions

This section describes differences in functionality between product revisions of the VIC:

r1p-r1p2 Contains no change to functionality. See the Errata document for details of erratum that have been fixed in this release.

These changes have no effect on the information provided in this manual.

Chapter 2
Functional Overview

This chapter describes the major functional blocks of the VIC (PL190). It contains the following sections:

2.1 About the VIC

The VIC provides a software interface to the interrupt system. In a system with an interrupt controller, software must determine the source that is requesting service and where its service routine is loaded. A VIC does both of these in hardware. It supplies the starting address, or vector address, of the service routine corresponding to the highest priority requesting interrupt source.

In an ARM system, two levels of interrupt are available:

Fast Interrupt reQuest (FIQ)

For fast, low latency interrupt handling.

Interrupt ReQuest (IRQ)

For more general interrupts.

Generally, you only use a single FIQ source at a time in a system to provide a true low-latency interrupt. This has the following benefits:

- You can execute the interrupt service routine directly without determining the source of the interrupt.
- It reduces interrupt latency. You can use the banked registers available for FIQ interrupts more efficiently, because you do not require a context save.

The interrupt inputs must be level sensitive, active HIGH, and held asserted until the interrupt service routine clears the interrupt. Edge-triggered interrupts are not compatible.

The interrupt inputs do not have to be synchronous to **HCLK**.

Note

The VIC does not handle interrupt sources with transient behavior. For example, an interrupt is asserted and then deasserted before software can clear the interrupt source. In this case, the CPU acknowledges the interrupt and obtains the vectored address for the interrupt from the VIC, assuming that no other interrupt has occurred to overwrite the vectored address. However, when a transient interrupt occurs, the priority logic of the VIC is not set, and lower priority interrupts can interrupt the transient interrupt service routine, assuming interrupt nesting is permitted.

There are 32 interrupt lines. The VIC uses a bit position for each different interrupt source. The software can control each request line to generate software interrupts.

There are 16 vectored interrupts. These interrupts can only generate an IRQ interrupt. The vectored and non-vectored IRQ interrupts provide an address for an *Interrupt Service Routine* (ISR). Reading from the Vector Interrupt Address Register, VICVECTADDR, provides the address of the ISR, and updates the interrupt priority hardware that masks out the current, and any lower priority interrupt requests. Writing to the VICVECTADDR Register indicates to the interrupt priority hardware that the current interrupt is serviced, enabling lower priority or the same priority interrupts to be removed, and for the interrupts to become active to go active.

The FIQ interrupt has the highest priority, followed by interrupt vector 0 to interrupt vector 15. Non-vectored IRQ interrupts have the lowest priority. A programmed interrupt request enables you to generate an interrupt under software control. This register typically downgrades an FIQ interrupt to an IRQ interrupt.

Note

- The ARM core sets the priority of the FIQ over IRQ.
- The VIC can raise both an FIQ and an IRQ at the same time.

The IRQ and FIQ request logic has an asynchronous path to the **nVICIRQ** and **nVICFIQ** outputs respectively. This enables you to assert interrupts when the VIC AHB clock, **HCLK**, is disabled in a low-power mode. It is expected that the power control logic enables the processor and VIC AHB clocks when an interrupt is received, so that the interrupt service routine can be performed.

By convention, for the IRQ interrupt, you are recommended to use bits 1 to 5 that Table 2-1 defines. Bit 0 and bit 6 upwards are available for you to use. For the FIQ interrupt, you can use all the bits.

Table 2-1 Recommended interrupt standard configuration

Bit	Interrupt source
1	Software interrupt
2	Comms Rx
3	Comms Tx
4	Timer 1
5	Timer 2

A space is reserved for the software interrupt so that you can use it without masking out a valid hardware interrupt. You can program any of the interrupt bits through software using the VICSOFTINT Register but, by reserving a specific software interrupt bit, it is easier to differentiate between hardware and software interrupts.

The Comms RX and TX lines are debug channel interrupts that the system processor uses, and they are required in any system that uses these debug features.

Spaces are reserved for two timers because a typical system has at least two timers.

Figure 2-1 shows a block diagram of the VIC.

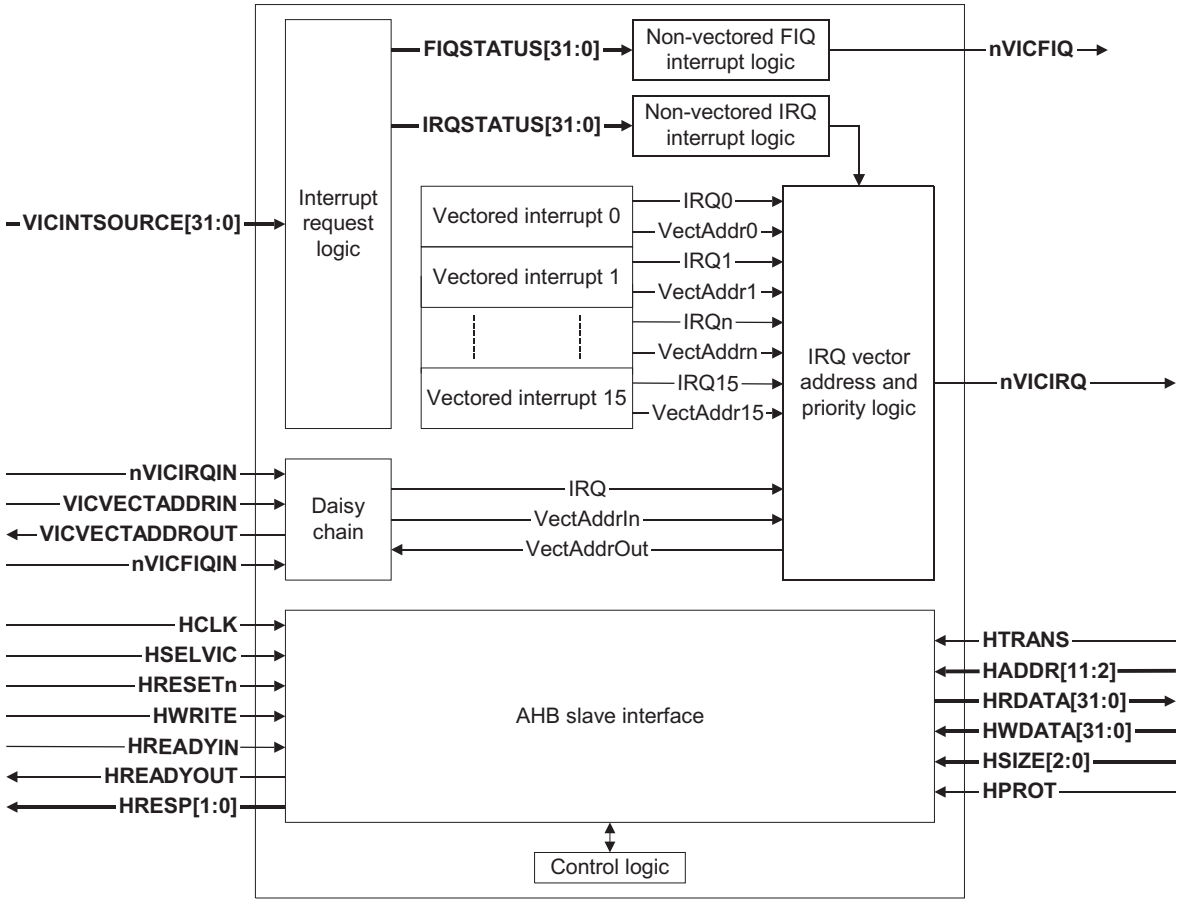


Figure 2-1 VIC block diagram

The following sections describe the main components of the VIC:

- *Interrupt request logic*
- *Non-vectored FIQ interrupt logic*
- *Non-vectored IRQ interrupt logic* on page 2-6
- *Vectored interrupt block* on page 2-6
- *Interrupt priority logic* on page 2-7.

2.1.1 Interrupt request logic

The interrupt request logic receives the interrupt requests from the peripheral and combines them with the software interrupt requests. It then masks out the interrupt requests that are not enabled, and routes the enabled interrupt requests to either **IRQSTATUS** or **FIQSTATUS**. Figure 2-2 shows a block diagram of the interrupt request logic.

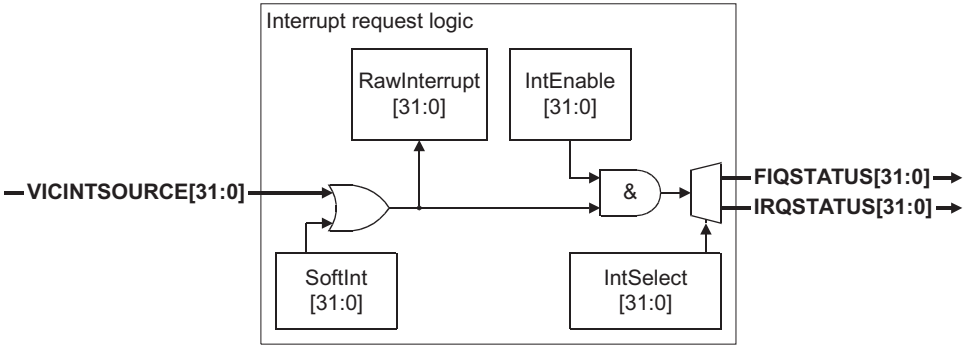


Figure 2-2 Interrupt request logic

2.1.2 Non-vectored FIQ interrupt logic

The non-vectored FIQ interrupt logic generates the FIQ interrupt signal by combining the FIQ interrupt requests in the interrupt controller and any requests from daisy-chained interrupt controllers. Figure 2-3 on page 2-6 shows a block diagram of the non-vectored FIQ interrupt logic.

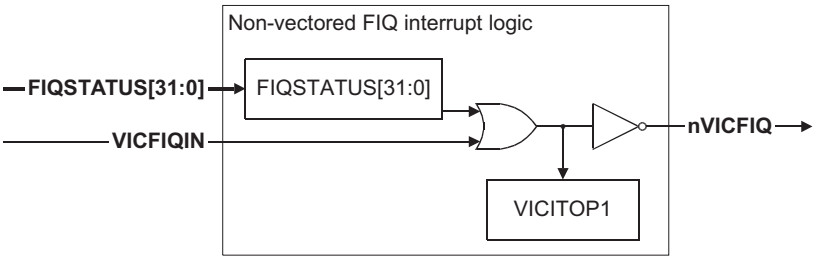


Figure 2-3 Non-vectored FIQ interrupt logic

2.1.3 Non-vectored IRQ interrupt logic

The non-vectored IRQ interrupt logic combines the non-vectored interrupt requests to generate the non-vectored IRQ interrupt signal. This signal is then sent to the IRQ vector address and priority logic. Figure 2-4 shows a block diagram of the non-vectored IRQ interrupt logic.

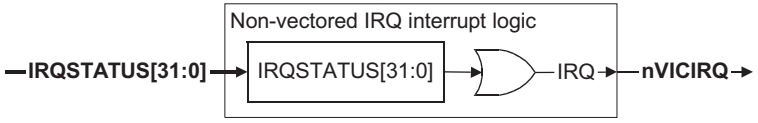


Figure 2-4 Non-vectored IRQ interrupt logic

2.1.4 Vectored interrupt block

There are 16 vectored interrupt blocks. The vectored interrupt blocks receive the IRQ interrupt requests and set **VECTIRQX** if the following are true:

- the selected interrupt is active
- the selected interrupt is currently the highest requesting interrupt.

Each vectored interrupt block also provides a **VECTADDRX[31:0]** output that you can use in the interrupt priority block. Figure 2-5 on page 2-7 shows a block diagram of two of the vectored interrupt blocks.

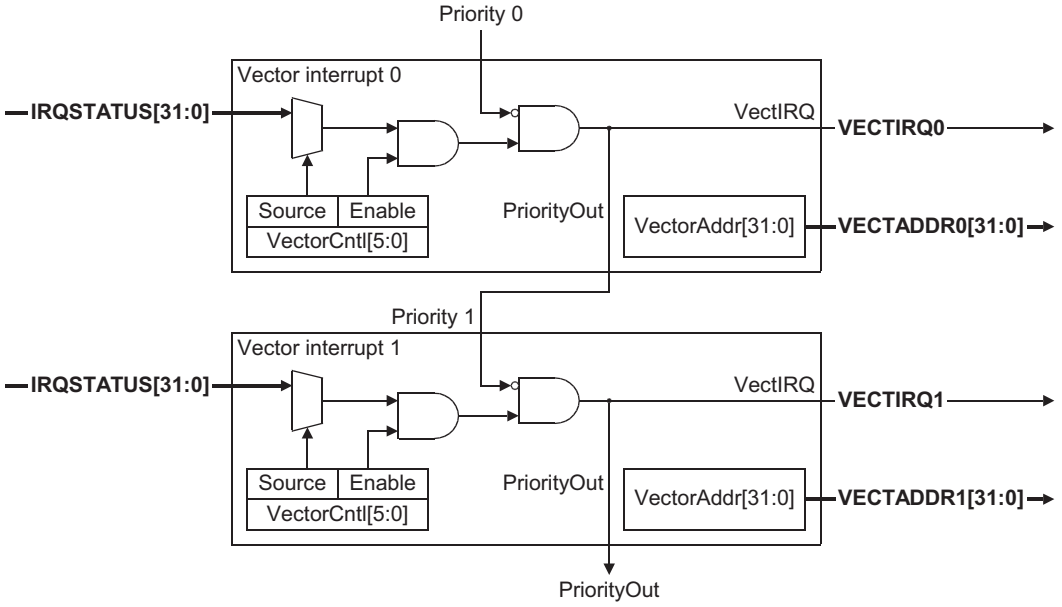


Figure 2-5 Vectored interrupt block

2.1.5 Interrupt priority logic

The interrupt priority block prioritizes the following requests:

- non-vectored interrupt requests
- vectored interrupt requests
- external interrupt requests.

The highest priority request generates an IRQ interrupt if the interrupt is not currently being serviced. Figure 2-6 on page 2-8 shows a block diagram of the interrupt priority logic.

Note

nVICIRQIN is the daisy-chained IRQ request input.

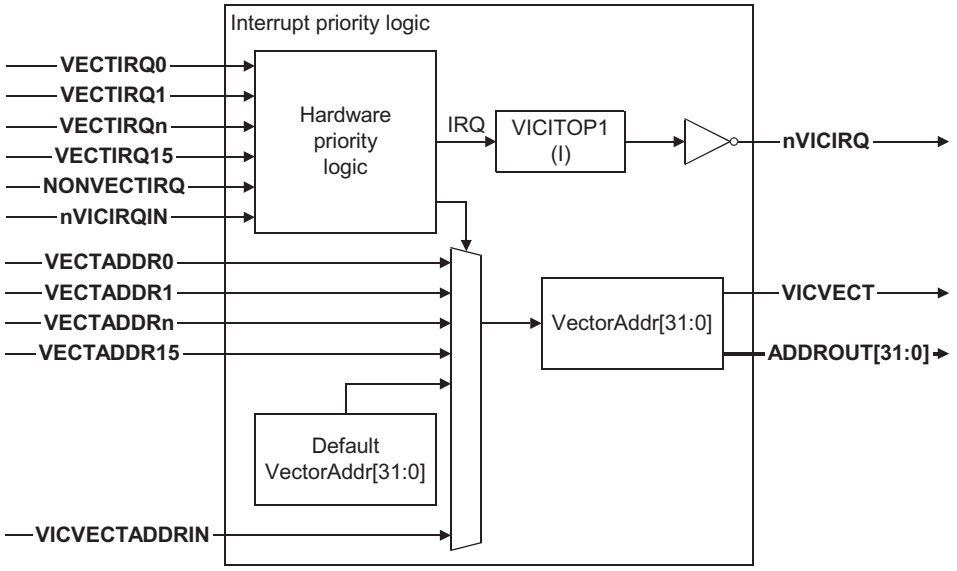


Figure 2-6 Interrupt priority logic

2.1.6 Vectored interrupts

A vectored interrupt is only generated if the following are true:

- you enable it in the interrupt enable register, VICIntEnable
- you set it to generate an IRQ interrupt in the interrupt select register, VICIntSelect
- you enable it in the relevant vector control register, VICVectCntl[0-15].

This prevents multiple interrupts being generated from a single interrupt request if the controller is incorrectly programmed.

2.1.7 Software interrupts

The software can control the source interrupt lines to generate software interrupts. These interrupts are generated before interrupt masking, in the same way as external source interrupts. You clear software interrupts by writing to the Software Interrupt Clear Register, VICSoftIntClear. See *Software Interrupt Clear Register* on page 3-8. This is normally done at the end of the interrupt service routine.

2.2 Operation

The following sections describe the operation of the VIC:

- *Vectored interrupt flow sequence*
- *Simple interrupt flow*.

2.2.1 Vectored interrupt flow sequence

The following procedure shows the sequence for the vectored interrupt flow:

1. An interrupt occurs.
2. The ARM processor branches to either the IRQ or FIQ interrupt vector.
3. If the interrupt is an IRQ, read the VICVectAddr Register and branch to the interrupt service routine. You can do this using an `LDR PC` instruction. Reading the VICVectorAddr Register updates the interrupt controllers hardware priority register.
4. Stack the workspace so that you can re-enable IRQ interrupts.
5. Enable the IRQ interrupts so that a higher priority can be serviced.
6. Execute the *Interrupt Service Routine* (ISR).
7. Clear the requesting interrupt in the peripheral, or write to the VICSoftIntClear Register if the request was generated by a software interrupt.
8. Disable the interrupts and restore the workspace.
9. Write to the VICVectAddr Register. This clears the respective interrupt in the internal interrupt priority hardware.
10. Return from the interrupt. This re-enables the interrupts.

2.2.2 Simple interrupt flow

The following procedure describes how to use the interrupt controller without using vectored interrupts or the interrupt priority hardware. For example, you can use it for debugging:

1. An interrupt occurs.
2. Branch to the IRQ or FIQ interrupt vector.
3. Branch to the interrupt handler.

4. Interrogate the VICIRQStatus Register to determine the source that generated the interrupt, and prioritize the interrupts if there are multiple active interrupt sources. This takes a number of instructions to compute.
5. Branch to the correct ISR.
6. Execute the ISR.
7. Clear the interrupt. If a software interrupt generated the request, you must write to the VICSoftIntClear Register.
8. Check the VICIRQStatus Register to ensure that no other interrupt is active. If there is an active request, go to Step 4.
9. Return from the interrupt.

Note

If you use this flow, do not read or write to the VICVectorAddr Register.

2.3 Connectivity

You normally use the VIC as a standalone interrupt controller. Where required, you can daisy-chain a second VIC.

Note

The interrupt latency increases if you use daisy-chaining. See *Daisy-chained interrupts* on page 3-20.

The VIC connects to the processor as a standard AHB slave, with the FIQ and IRQ signals connected to the FIQ and IRQ inputs on the processor. The interrupt request lines from the peripheral connect to the **VICINTSOURCE** inputs of the VIC. To ensure that you can read the vector address register in a single instruction, you must put the VIC in the upper 4K of memory, at `0xFFFFF000`. See *Vector Address Register* on page 3-9.

Note

If the VIC is located at a different address, interrupt latency increases.

The following sections describe the connectivity for the two options:

- *Standalone interrupt controller*
- *Daisy-chained interrupt controller* on page 2-12.

2.3.1 Standalone interrupt controller

If you use the VIC as a standalone interrupt controller, connect the signals as follows:

- tie **nVICIRQIN** and **nVICFIQIN** HIGH
- tie **VICVECTADDRIN[31:0]** LOW.

Figure 2-7 shows the connections between the VIC and the processor when you use it as a standalone interrupt controller.

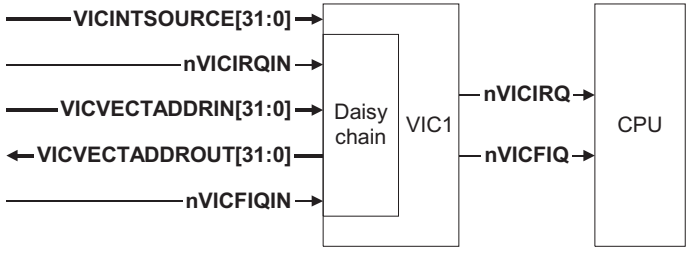


Figure 2-7 Standalone interrupt controller connectivity

2.3.2 Daisy-chained interrupt controller

If you use the VIC in a daisy-chain, connect the signals between the VICs as follows:

- **nVICIRQIN** on the first VIC to the **nVICIRQ** output of the second VIC
- **nVICFIQIN** on the first VIC to the **nVICFIQ** output of the second VIC
- **VICVECTADDRIN[31:0]** on the first VIC to the **VICVECTADDROUT[31:0]** of the second VIC.

Standalone interrupt controller on page 2-11 describes how to connect the final VIC in the chain, the VIC furthest from the processor.

Figure 2-8 shows the connections between the VICs and the processor when you use them in a daisy-chain.

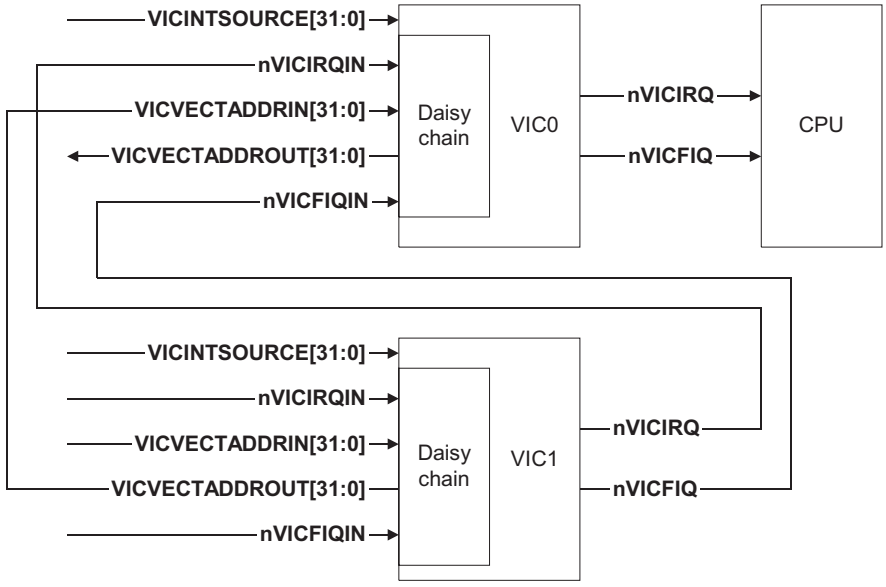


Figure 2-8 Daisy-chained interrupt controller connectivity

Daisy-chain interrupt priority

The interrupt priority in Figure 2-8 on page 2-12 is as follows:

1. FIQ
2. VIC0: VIRQ0, VIRQ1-VIRQ15, NonVectIRQ
3. VIC1: VIRQ0, VIRQ1-VIRQ15, NonVectIRQ
4. VICn: ...

Chapter 3
Programmer's Model

This chapter describes the VIC (PL190) registers. It provides information that you require to program the microcontroller. It contains the following sections:

3.1 About the programmer's model

To ensure that you can read the vector address register in a single instruction, the VIC base address must be `0xFFFFF000`, that is the upper 4K of memory. See *Vector Address Register* on page 3-9. Placing the VIC anywhere else in memory increases interrupt latency because the ARM processor cannot access the VICVectorAddr Register using a single instruction.

The read, `LDR`, instruction has a maximum address offset of 12 bits, equivalent to 4K, meaning that it can read from an address up to 4K away from the current address with a single read instruction. If the address to be read from is more than 4K away, you require a second instruction to read in the full address value. This takes longer to perform.When an interrupt occurs, the current address is either the IRQ or FIQ exception vector location, `0x00000018` or `0x0000001C` for normal low exception vectors. A 4K offset from the exception address is the upper 4K of memory, so placing the VIC in this area of memory enables the read of the VICADDRESS Register, at `0xFFFFFF00`, to be performed using an address offset with a single instruction. For example, at location `0x18 LDR pc, [pc, #-0x120]` to access VICADDRESS at location `0xFFFFFF00`.

If you use a processor that supports high exception vectors, and you tie the **HIVECS** configuration pin HIGH, you must put the VIC at `0xFFFEF000` to enable the exception vectors that are located at `0xFFFFF000`. The VIC is not located at `0x00000000`, because this is the standard location for the system memory. The offset of any particular register from the base address is fixed.

3.2 Summary of VIC registers

Table 3-1 lists the VIC registers.

Table 3-1 VIC register summary

Register	Address offset	Type	Reset value	Description
VICIRQSTATUS	0x000	RO	0x00000000	See *IRQ Status Register* on page 3-6
VICFIQSTATUS	0x004	RO	0x00000000	See *FIQ Status Register* on page 3-6
VICRAWINTR	0x008	RO	-	See *Raw Interrupt Status Register* on page 3-6
VICINTSELECT	0x00C	R/W	0x00000000	See *Interrupt Select Register* on page 3-7
VICINTENABLE	0x010	R/W	0x00000000	See *Interrupt Enable Register* on page 3-7
VICINTENCLEAR	0x014	Write	-	See *Interrupt Enable Clear Register* on page 3-7
VICSOFTINT	0x018	R/W	0x00000000	See *Software Interrupt Register* on page 3-8
VICSOFTINTCLEAR	0x01C	WO	-	See *Software Interrupt Clear Register* on page 3-8
VICPROTECTION	0x020	R/W	0x0	See *Protection Enable Register* on page 3-8
VICVECTADDR	0x030	R/W	0x00000000	See *Vector Address Register* on page 3-9
VICDEFVECTADDR	0x034	R/W	0x00000000	See *Default Vector Address Register* on page 3-10
VICVECTADDR0	0x100	R/W	0x00000000	See *Vector Address Registers* on page 3-10
VICVECTADDR1	0x104	R/W	0x00000000	
VICVECTADDR2	0x108	R/W	0x00000000	
VICVECTADDR3	0x10C	R/W	0x00000000	
VICVECTADDR4	0x110	R/W	0x00000000	
VICVECTADDR5	0x114	R/W	0x00000000	
VICVECTADDR6	0x118	R/W	0x00000000	
VICVECTADDR7	0x11C	R/W	0x00000000	
VICVECTADDR8	0x120	R/W	0x00000000	
VICVECTADDR9	0x124	R/W	0x00000000	

Table 3-1 VIC register summary (continued)

Register	Address offset	Type	Reset value	Description
VICVECTADDR10	0x128	R/W	0x00000000	See *Vector Address Registers* on page 3-10
VICVECTADDR11	0x12C	R/W	0x00000000	
VICVECTADDR12	0x130	R/W	0x00000000	
VICVECTADDR13	0x134	R/W	0x00000000	
VICVECTADDR14	0x138	R/W	0x00000000	
VICVECTADDR15	0x13C	R/W	0x00000000	
VICVECTCNTL0	0x200	R/W	0x00	See *Vector Control Registers* on page 3-10
VICVECTCNTL1	0x204	R/W	0x00	
VICVECTCNTL2	0x208	R/W	0x00	
VICVECTCNTL3	0x20C	R/W	0x00	
VICVECTCNTL4	0x210	R/W	0x00	
VICVECTCNTL5	0x214	R/W	0x00	
VICVECTCNTL6	0x218	R/W	0x00	
VICVECTCNTL7	0x21C	R/W	0x00	
VICVECTCNTL8	0x220	R/W	0x00	
VICVECTCNTL9	0x224	R/W	0x00	
VICVECTCNTL10	0x228	R/W	0x00	
VICVECTCNTL11	0x22C	R/W	0x00	
VICVECTCNTL12	0x230	R/W	0x00	
VICVECTCNTL13	0x234	R/W	0x00	
VICVECTCNTL14	0x238	R/W	0x00	
VICVECTCNTL15	0x23C	R/W	0x00	
VICPERIPHID0	0xFE0	RO	0x90	See *Peripheral Identification Registers* on page 3-11

Table 3-1 VIC register summary (continued)

Register	Address offset	Type	Reset value	Description
VICPERIPHID1	0xFE4	RO	0x11	See *Peripheral Identification Registers* on page 3-11
VICPERIPHID2	0xFE8	RO	0x04	
VICPERIPHID3	0xFEC	RO	0x00	
VICPCELLID0	0xFF0	RO	0x0D	See *PrimeCell Identification Registers* on page 3-14
VICPCELLID1	0xFF4	RO	0xF0	
VICPCELLID2	0xFF8	RO	0x05	
VICPCELLID3	0xFFC	RO	0xB1	

3.3 Register descriptions

This section describes the VIC registers.

3.3.1 IRQ Status Register

The read-only VICIRQSTATUS Register, with address offset of `0x000`, provides the status of interrupts [31:0] after IRQ masking. Table 3-2 lists the bit assignments for this register.

Table 3-2 VICIRQSTATUS Register bit assignments

Bits	Name	Function
[31:0]	IRQStatus	Shows the status of the interrupts after masking by the VICINTENABLE and VICINTSELECT Registers. A HIGH bit indicates that the interrupt is active, and generates an interrupt to the processor.

3.3.2 FIQ Status Register

The read-only VICFIQSTATUS Register, with address offset of `0x004`, provides the status of the interrupts after FIQ masking. Table 3-3 lists the bit assignments for this register.

Table 3-3 VICFIQSTATUS Register bit assignments

Bits	Name	Function
[31:0]	FIQStatus	Shows the status of the interrupts after masking by the VICINTENABLE and VICINTSELECT Registers. A HIGH bit indicates that the interrupt is active, and generates an interrupt to the processor.

3.3.3 Raw Interrupt Status Register

The read-only VICRAWINTR Register, with address offset of `0x008`, provides the status of the source interrupts, and software interrupts, to the interrupt controller. Table 3-4 lists the bit assignments for this register.

Table 3-4 VICRAWINTR Register bit assignments

Bits	Name	Function
[31:0]	RawInterrupt	Shows the status of the interrupts before masking by the enable registers. A HIGH bit indicates that the appropriate interrupt request is active before masking.

3.3.4 Interrupt Select Register

The read/write VICINTSELECT Register, with address offset of 0x00C., selects whether the corresponding interrupt source generates an FIQ or an IRQ interrupt. Table 3-5 lists the bit assignments for this register.

Table 3-5 VICINTSELECT Register bit assignments

Bits	Name	Function
[31:0]	IntSelect	Selects the type of interrupt for interrupt requests: 1 = FIQ interrupt 0 = IRQ interrupt.

3.3.5 Interrupt Enable Register

The read/write VICINTENABLE Register, with address offset of 0x010, enables the interrupt request lines, by masking the interrupt sources for the IRQ interrupt. Table 3-6 lists the bit assignments for this register.

Table 3-6 VICINTENABLE Register bit assignments

Bits	Name	Function
[31:0]	IntEnable	Enables the interrupt request lines: 1 = Interrupt enabled. Enables interrupt request to processor. 0 = Interrupt disabled. On reset, all interrupts are disabled. A HIGH bit sets the corresponding bit in the VICINTENABLE Register. A LOW bit has no effect.

3.3.6 Interrupt Enable Clear Register

The write-only VICINTENCLEAR Register, with address offset of 0x014, clears bits in the VICIntEnable Register. Table 3-7 lists the bit assignments for this register.

Table 3-7 VICINTENCLEAR Register bit assignments

Bits	Name	Function
[31:0]	IntEnable Clear	Clears bits in the VICINTENABLE Register. A HIGH bit clears the corresponding bit in the VICINTENABLE Register. A LOW bit has no effect.

3.3.7 Software Interrupt Register

The read/write VICSOFTINT Register, with address offset of `0x018`, generates software interrupts. Table 3-8 lists the bit assignments for this register.

Table 3-8 VICSOFTINT Register bit assignments

Bits	Name	Function
[31:0]	SoftInt	Setting a bit generates a software interrupt for the specific source interrupt before interrupt masking. A HIGH bit sets the corresponding bit in the VICSOFTINT Register. A LOW bit has no effect.

3.3.8 Software Interrupt Clear Register

The write-only VICSOFTINTCLEAR Register, with address offset of `0x01C`, clears bits in the VICSoftInt Register. Table 3-9 lists the bit assignments for this register.

Table 3-9 VICSOFTINTCLEAR Register bit assignments

Bits	Name	Function
[31:0]	SoftIntClear	Clears bits in the VICSOFTINT Register. A HIGH bit clears the corresponding bit in the VICSOFTINT Register. A LOW bit has no effect.

3.3.9 Protection Enable Register

The read/write VICPROTECTION Register, with address offset of `0x020`, enables or disables protected register access. Figure 3-1 shows the bit assignments for this register.

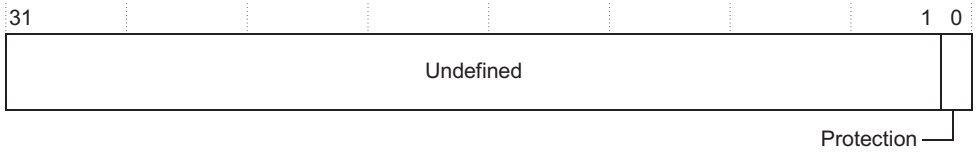


Figure 3-1 VICPROTECTION Register bit assignments

Table 3-10 lists the bit assignments for this register.

Table 3-10 VICPROTECTION Register bit assignments

Bits	Name	Function
[31:1]	-	Read undefined. Write as zero.
[0]	Protection	Enables or disables protected register access. When enabled, only privileged mode accesses, reads and writes, can access the interrupt controller registers. When disabled, both User mode and Privileged mode can access the registers. This register is cleared on reset, and can only be accessed in privileged mode.

Note

If the bus master cannot generate accurate protection information, leave this register in its reset state to enable User mode access.

3.3.10 Vector Address Register

The read/write VICVECTADDR Register, with address offset of `0x030`, contains the *Interrupt Service Routine* (ISR) address of the currently active interrupt. Table 3-11 lists the bit assignments for this register.

Table 3-11 VICVECTADDR Register bit assignments

Bits	Name	Function
[31:0]	VectorAddr	Contains the address of the currently active ISR. Any writes to this register clear the interrupt.

Note

Reading from this register provides the address of the ISR, and indicates to the priority hardware that the interrupt is being serviced. Writing to this register indicates to the priority hardware that the interrupt has been serviced. You must use this register as follows:

- the ISR reads the VICVectAddr Register when an IRQ interrupt is generated
- at the end of the ISR, the VICVectAddr Register is written to, to update the priority hardware.

Reading from or writing to the register at other times can cause incorrect operation.

3.3.11 Default Vector Address Register

The read/write VICDEFVECTADDR Register, with address offset of 0x034, contains the default ISR address. Table 3-12 lists the bit assignments for this register

Table 3-12 VICDEFVECTADDR Register bit assignments

Bits	Name	Function
[31:0]	Default VectorAddr	Contains the address of the default ISR handler

3.3.12 Vector Address Registers

The read/write VICVECTADDR[0-15] Registers span address locations 0x100-0x13C and contain the ISR vector addresses. Table 3-13 lists the bit assignments for these registers.

Table 3-13 VICVECTADDR Registers bit assignments

Bits	Name	Function
[31:0]	VectorAddr 0-15	Contains ISR vector addresses

3.3.13 Vector Control Registers

The read/write VICVECTCNTL[0-15] Registers span address locations 0x200-0x23C and select the interrupt source for the vectored interrupt. Figure 3-2 shows the bit assignments for these registers.

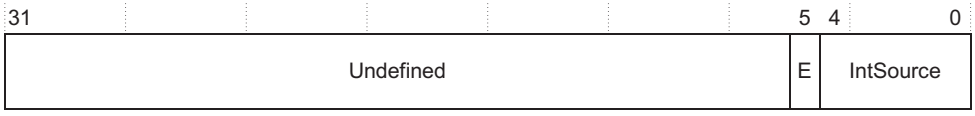


Figure 3-2 VICVECTCNTL Register bit assignments

Table 3-14 lists the bit assignments for these registers.

Table 3-14 VICVECTCNTL Registers bit assignments

Bits	Name	Function
[31:6]	-	Read undefined. Write as zero.
[5]	E	Enables vector interrupt. This bit is cleared on reset.
[4:0]	IntSource	Selects interrupt source. You can select any of the 32 interrupt sources.

Note

Vectored interrupts are only generated if the interrupt is enabled. The specific interrupt is enabled in the VICIntEnable Register, and the interrupt is set to generate an IRQ interrupt in the VICIntSelect Register. This prevents multiple interrupts being generated from a single request if the controller is incorrectly programmed.

3.3.14 Peripheral Identification Registers

The read-only VICPeriphID0-3 Registers are four 8-bit registers, that span address locations `0xFE0-0xFEC`. You can treat the registers conceptually as a single 32-bit register. The read-only registers provide the following options for the peripheral:

Part number [11:0]

This identifies the peripheral. The VIC uses the three-digit product code `0x90`.

Designer [19:12]

This is the identification of the designer. ARM Limited is `0x41`, ASCII A.

Revision number [23:20]

This is the revision number of the peripheral. The revision number starts from 0.

Configuration [31:24]

This is the configuration option of the peripheral. The configuration value is 0.

Figure 3-3 on page 3-12 shows the bit assignments for these registers.

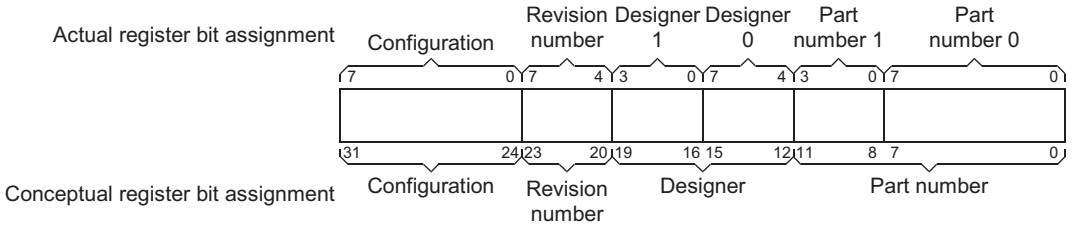


Figure 3-3 Peripheral Identification Register bit allocation

The following sections describe the four 8-bit Peripheral Identification Registers:

- *VICPERIPHID0 Register*
- *VICPERIPHID1 Register* on page 3-13
- *VICPERIPHID2 Register* on page 3-13
- *VICPERIPHID3 Register* on page 3-14.

VICPERIPHID0 Register

The read-only VICPERIPHID0 Register, with address offset of `0xFE0`, is hard-coded, and the fields within the register determine the reset value. Figure 3-4 shows the bit assignments for this register.

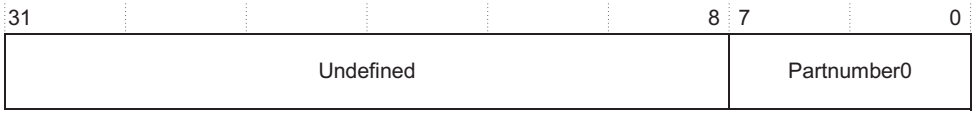


Figure 3-4 VICPERIPHID0 Register bit assignments

Table 3-15 lists the bit assignments for this register.

Table 3-15 VICPERIPHID0 Register bit assignments

Bits	Name	Function
[31:8]	-	Read undefined
[7:0]	Partnumber0	These bits read back as `0x90`

VICPERIPHID1 Register

The read-only VICPERIPHID1 Register, with address offset of `0xFE4`, is hard-coded, and the fields within the register determine the reset value. Figure 3-5 shows the bit assignments for this register.

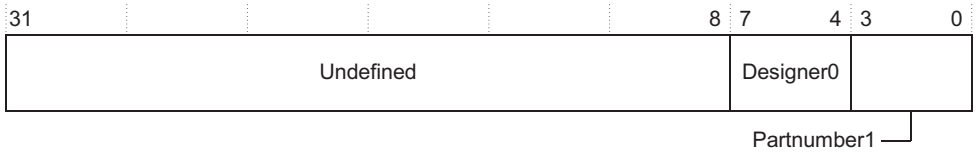


Figure 3-5 VICPERIPHID1 Register bit assignments

Table 3-16 lists the bit assignments for this register.

Table 3-16 VICPERIPHID1 Register bit assignments

Bits	Name	Function
[31:8]	-	Read undefined
[7:4]	Designer0	These bits read back as `0x1`
[3:0]	Partnumber1	These bits read back as `0x1`

VICPERIPHID2 Register

The read-only VICPERIPHID2 Register, with address offset of `0xFE8`, is hard-coded and the fields within the register determine the reset value. Figure 3-6 shows the bit assignments for this register.

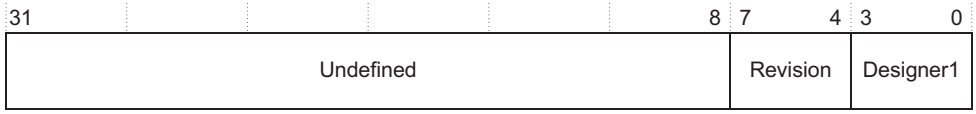


Figure 3-6 VICPERIPHID2 Register bit assignments

Table 3-17 lists the bit assignments for this register.

Table 3-17 VICPERIPHID2 Register bit assignments

Bits	Name	Function
[31:8]	-	Read undefined
[7:4]	Revision	These bits read back as 0x1
[3:0]	Designer1	These bits read back as 0x0

VICPERIPHID3 Register

The read-only VICPERIPHID3 Register, with address offset of 0xFEC., is hard-coded and the fields within the register determine the reset value. Figure 3-7 shows the bit assignments for this register.

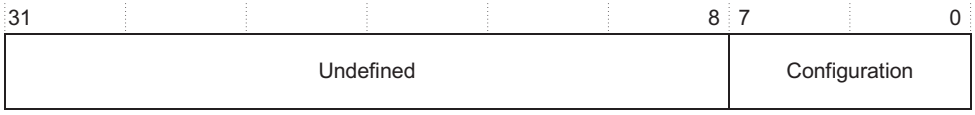


Figure 3-7 VICPERIPHID3 Register bit assignments

Table 3-18 lists the bit assignments for this register.

Table 3-18 VICPERIPHID3 Register bit assignments

Bits	Name	Function
[31:8]	-	Read undefined
[7:0]	Configuration	These bits read back as 0x0

3.3.15 PrimeCell Identification Registers

The read-only VICPCELLID0-3 Registers are four 8-bit registers that span address locations 0xFF0-0xFFC. You can treat them conceptually as a single 32-bit register. Use the register as a standard cross-peripheral identification system. on page 3-15 shows the bit assignment for these registers.

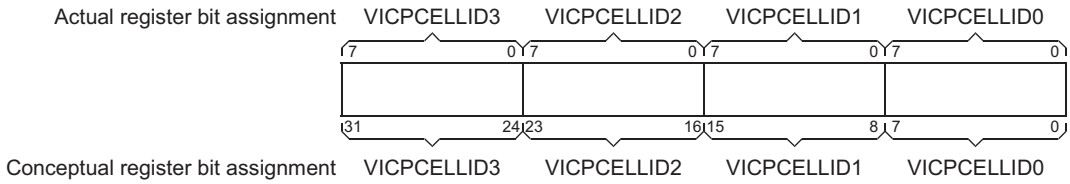


Figure 3-8 PrimeCell Identification Register bit assignment

The four 8-bit registers are described in the following subsections:

- *VICPCELLID0 Register*
- *VICPCELLID1 Register*
- *VICPCELLID2 Register* on page 3-16
- *VICPCELLID3 Register* on page 3-16.

VICPCELLID0 Register

The read-only VICPCELLID0 Register, with address offset of `0xFF0`, is hard-coded and the fields within the register determine the reset value. Figure 3-9 shows the bit assignments for this register.

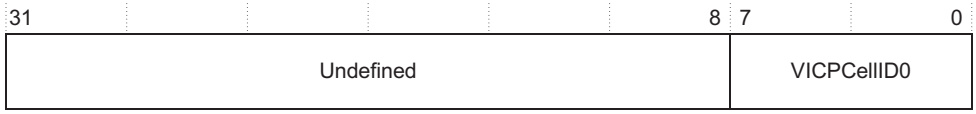


Figure 3-9 VICPCELLID0 Register bit assignments

Table 3-19 lists the bit assignments for this register.

Table 3-19 VICPCELLID0 Register bit assignments

Bits	Name	Function
[31:8]	-	Read undefined
[7:0]	VICPCellID0	These bits read back as `0x0D`

VICPCELLID1 Register

The read-only VICPCELLID1 Register, with address offset of `0xFF4`, is hard-coded and the fields within the register determine the reset value. Figure 3-10 on page 3-16 shows the bit assignments for this register.

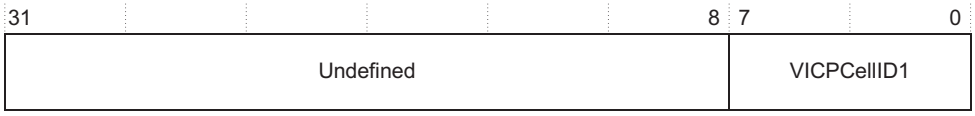


Figure 3-10 VICPCELLID1 Register bit assignments

Table 3-20 lists the bit assignments for this register.

Table 3-20 VICPCELLID1 Register bit assignments

Bits	Name	Function
[31:8]	-	Read undefined
[7:0]	VICPCellID1	These bits read back as `0xF0`

VICPCELLID2 Register

The read-only VICPCELLID2 Register, with address offset of `0xFF8`, is hard-coded and the fields within the register determine the reset value. Figure 3-11 shows the bit assignments for this register.

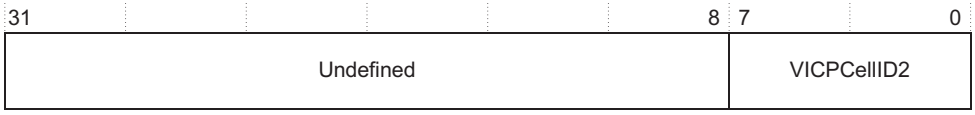


Figure 3-11 VICPCELLID2 Register bit assignments

Table 3-21 lists the bit assignments for this register.

Table 3-21 VICPCELLID2 Register bit assignments

Bits	Name	Function
[31:8]	-	Read undefined
[7:0]	VICPCellID2	These bits read back as `0x05`

VICPCELLID3 Register

The read-only VICPCELLID3 Register, with address offset of `0xFFC`, is hard-coded and the fields within the register determine the reset value. Figure 3-12 on page 3-17 shows the bit assignments for this register.

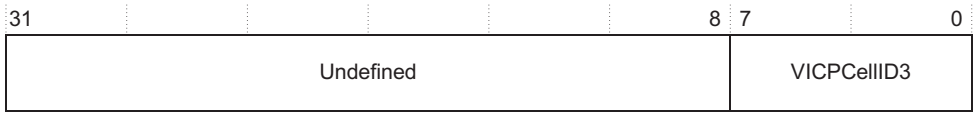


Figure 3-12 VICPCELLID3 Register bit assignments

Table 3-22 lists the bit assignments for this register.

Table 3-22 VICPCELLID3 Register bit assignments

Bits	Name	Type	Function
[31:8]	-	-	Read undefined
[7:0]	VICPCellID3	RO	These bits read back as `0xB1`

3.4 Interrupt latency

The calculations in this section show the number of cycles required to service interrupts, using the following types of interrupt:

- *FIQ interrupts*
- *IRQ interrupts* on page 3-19
- *Fast IRQ interrupts* on page 3-19
- *Daisy-chained interrupts* on page 3-20.

Note

The calculations are based on the assumption that the ISRs are in zero wait state memory.

3.4.1 FIQ interrupts

FIQ interrupts have the highest priority in the VIC, and are not nested. In FIQ mode, seven 32-bit registers are banked into the system. This enables the VIC to process the interrupt as quickly as possible. Table 3-23 lists the worst case cycles for FIQ interrupts.

Table 3-23 FIQ interrupt latency

Event	Worst case (cycles)
Interrupt synchronization.	3
Worst case instruction execution. This assumes that a standard switch reduces `STM` and `LDM`. You can reduce this to 7 cycles to avoid data aborts.	7
Entry to first instruction.	2
Total.	12

Note

For the best results, start the FIQ handler at the FIQ vector address, `0x1c`.

3.4.2 IRQ interrupts

In IRQ mode, you can nest interrupt levels lower than the highest priority FIQ interrupt level. To provide this nesting, the return address, stored in the *Link Register* (LR), and the status register, stored in the *Saved Processor Status Register* (SPSR) must be available before more IRQ interrupts can be accepted. This increases the interrupt latency, but provides a scalable nested interrupt system. Table 3-24 lists the worst case cycles for IRQ interrupts.

Table 3-24 IRQ interrupt latency

Event	Worst case (cycles)
Interrupt synchronization	3
Worst case interrupt disable period	10
Entry to first instruction	2
Nesting, assuming single-state AHB	10
Total	25

3.4.3 Fast IRQ interrupts

Fast IRQ mode is similar to IRQ mode, except that the highest-level IRQ interrupt handler assumes that no other IRQ interrupt occurs during its operation, and therefore, you do not require LR and SPSR. Table 3-25 lists the worst case cycles for fast IRQ interrupts.

Table 3-25 Fast IRQ interrupt latency

Event	Worst case (cycles)
Interrupt synchronization	3
Worst case interrupt disable period	10
Entry to first instruction	2
Load IRQ vector into PC	5
Total	20

3.4.4 Daisy-chained interrupts

Because of the additional read required to read both the primary VICVectAddr Register and the daisy-chained VICVectAddr Register, the worst-case latency of the primary VIC increases by one cycle, to 26 cycles. The worst-case latency for the secondary, daisy-chained VIC increases by two cycles, to 27 cycles. This latency applies to any number of secondary VICs. See *Daisy-chained vectored interrupt service routine* on page B-6 for more information.

3.5 Interrupt priority

The hardware regulates the interrupt priority. FIQ interrupts have the highest priority, followed by vectored interrupt 0 to vectored interrupt 15. Non-vectored interrupts have the lowest priority.

To reduce interrupt latency, you can re-enable the IRQ interrupts in the processor after the *Interrupt Service Routine* (ISR) is entered. See *Interrupt latency* on page 3-18. In this case, the current ISR is interrupted, and the higher-priority ISR is executed. The VIC then only enables a higher priority interrupt than the interrupt currently being serviced. If a higher priority interrupt goes active, the current ISR is interrupted and the higher-priority ISR is executed.

Before the interrupt enable bits in the processor can be re-enabled, the LR and SPSR must be saved, preferably on a software stack. When the ISR is exited, you must disable the interrupts, reload the LR and SPSR, and write to the Vector Address Register, VICVectAddr. See *Vectored interrupt service routine* on page B-6.

When you daisy-chain VICs, the interrupt priority is as follows:

- FIQ interrupts
- primary VIC vectored interrupts
- primary VIC non-vectored interrupts
- daisy-chained VIC vectored interrupts
- daisy-chained VIC non-vectored interrupts.

Chapter 4
Programmer's Model for Test

This chapter describes the additional logic for functional verification and provisions made for production testing. It contains the following sections:

4.1 VIC test harness overview

The additional logic for functional verification and production testing enables:

- capture of input signals to the block
- stimulation of the output signals.

The integration vectors provide a way of verifying that the VIC is correctly wired into a system. Do this by separately testing two groups of signals:

AMBA signals

Test these by checking the connections of all the address and data bits.

Intra-chip signals

The tests for these signals are system-specific, and enable you to write the necessary tests. Additional logic is implemented enabling you to read/write to each intra-chip input/output signal.

Test registers control these test features, and enable you to test the VIC in isolation from the rest of the system using only transfers from the AMBA AHB.

Off-chip test vectors are supplied using a 32-bit parallel *External Bus Interface* (EBI) and converted to internal AMBA bus transfers. The *Test Interface Controller* (TIC) AMBA bus master module controls the application of test vectors.

4.2 Scan testing

The VIC simplifies:

- insertion of scan test cells
- use of *Automatic Test Pattern Generation* (ATPG).

This provides an alternative method of manufacturing test.

4.3 Summary of test registers

Table 4-1 lists how the VIC test registers are memory-mapped.

Table 4-1 Test registers memory map

Register	Address offset	Type	Reset value	Description
VICITCR	0x300	R/W	-	See *Test Control Register*
VICITIP1	0x304	RO	0x0	See *Integration Test Input Registers* on page 4-5
VICITIP2	0x308	RO	-	
VICITOP1	0x30C	RO	0x0	See *Integration Test Output Registers* on page 4-6
VICITOP2	0x310	RO	0x00000000	

4.3.1 Test Control Register

The read/write VICITCR Register, with address offset of 0x300, is a single-bit test control register. The ITEN bit in this register controls the input test multiplexors. Figure 4-1shows the bit assignments for this register.

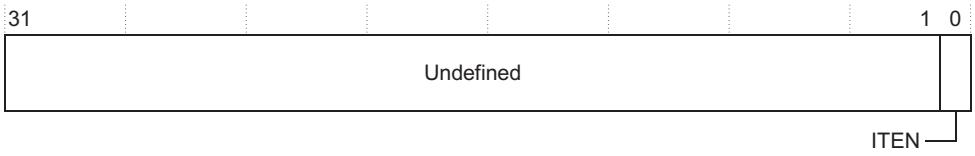


Figure 4-1 VICITCR Register bit assignments

Table 4-2 lists the bit assignments for this register.

Table 4-2 VICITCR Register bit assignments

Bits	Name	Description
[31:1]	-	Read undefined. Write as zero.
[0]	ITEN	Integration test enable: 0 = normal mode 1 = test mode.

4.3.2 Integration Test Input Registers

The read-only VICITIP1 Register, with address offset of `0x304`, is a 2-bit register that returns the values of the **nVICIRQIN** and **nVICFIQIN** inputs. Figure 4-2 shows the bit assignments for this register.

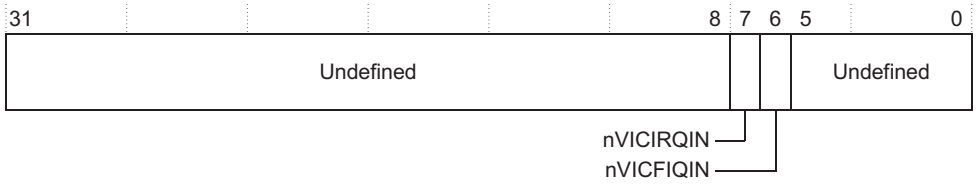


Figure 4-2 VICITIP1 Register bit assignments

Table 4-3 lists the bit assignments for this register.

Table 4-3 VICITIP1 Register bit assignments

Bits	Name	Description
[31:8]	-	Read undefined
[7]	nVICIRQIN	Reads return the value on **nVICIRQIN** when the VICITCR Register is LOW
[6]	nVICFIQIN	Reads return the value on **nVICFIQIN** when the VICITCR Register is LOW
[5:0]	-	Read undefined

The VICITIP2 Register, with address offset of `0x308`, is a read-only register. It is a 32-bit register that returns the value of the **VICVECTADDRIN** input. Table 4-4 lists the bit assignments for this register.

Table 4-4 VICITIP2 Register bit assignments

Bits	Name	Description
[31:0]	VICVECTADDRIN	Reads return the value on **VICVECTADDRIN** when the VICITCR Register is LOW.

4.3.3 Integration Test Output Registers

The read-only VICITOP1 Register, with address offset of `0x30C`, is a 32-bit register that controls the **nVICIRQ** and **nVICFIQ** outputs. Figure 4-3 shows the bit assignments for this register.

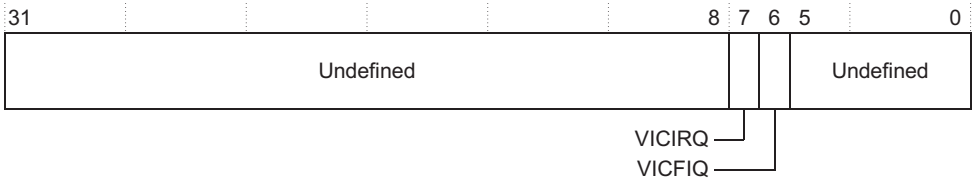


Figure 4-3 VICITOP1 Register bit assignments

Table 4-5 lists the bit assignments for this register.

Table 4-5 VICITOP1 Register bit assignments

Bits	Name	Description
[31:8]	-	Read undefined.
[7]	VICIRQ	Reads return the value on the internal **VICIRQ** line. This is the pre-inverted version of the final output, and is inverted to create the final **nVICIRQ** output.
[6]	VICFIQ	Reads return the value on the internal **VICFIQ** line. This is the pre-inverted version of the final output, and is inverted to create the final **nVICFIQ** output.
[5:0]	-	Read undefined.

The read-only VICITOP2 Register, with address offset of `0x310`, is a 32-bit register that controls the **VICVECTADDROUT** output. Table 4-6 lists the bit assignments for this register.

Table 4-6 VICITOP2 Register bit assignments

Bits	Name	Description
[31:0]	VICVECTADDROUT	Reads return the value on the **VICVECTADDROUT** lines.

Appendix A
Signal Descriptions

This appendix describes the signals that interface with the ARM PrimeCell Vectored Interrupt Controller (PL190). It contains the following sections:

A.1 AMBA AHB signals

The VIC module is connected to the AMBA AHB as a bus slave. Table A-1 lists the AHB signals that are used and produced.

Table A-1 AMBA AHB signal descriptions

Name	Type	Source/ destination	Description
HCLK	Input	Clock source	AMBA AHB bus clock. Times all bus transfers. All signal timings are related to the rising edge of **HCLK**.
HRESETn	Input	Reset controller	AHB bus reset, active LOW.
HADDR[11:2]	Input	Master	System address bus.
HTRANS	Input	Master	Transfer type. This can be NONSEQUENTIAL, SEQUENTIAL, IDLE, or BUSY. You must connect this signal to **HTRANS[1]** on the AHB interface. **HTRANS[0]** is not used.
HWRITE	Input	Master	Transfer direction. Indicates a write transfer when HIGH, and a read transfer when LOW.
HSIZE[2:0]	Input	Master	Size of the transfer. This must be a word, 32-bit, for the VIC, **HSIZE[2:0]** = `0b010`.
HPROT	Input	Master	Memory access protection type. This can be User mode (0) or privileged mode (1). You must connect this signal to **HPROT[1]** on the AHB interface. **HPROT[3]**, **HPROT[2]** and **HPROT[0]** are not used.
HWDATA[31:0]	Input	Master	Write data bus. Transfers data from bus master to bus slaves during write operations.
HSELVIC	Input	Decoder	Slave select signal. This is a combinatorial decode of the address bus. It indicates that the current transfer is intended for the selected slave.
HRDATA[31:0]	Output	Slave	Read data bus. Transfers data from bus slaves to bus master during read operations.
HREADYIN	Input	External slave	Transfer done signal, generated by an alternate slave. When HIGH, indicates that a transfer is complete. You can drive it LOW to extend a transfer.
HREADYOUT	Output	Slave	Transfer done signal, generated by the VIC. When HIGH, indicates that a transfer is complete. You can drive it LOW to extend a transfer.
HRESP[1:0]	Output	Slave	Transfer response. This provides additional transfer status information. The response can be OKAY, ERROR, RETRY, or SPLIT. The VIC responds with either OKAY or ERROR.

A.2 Interrupt controller signals

Table A-2 lists the signals for the VIC that interface to the processor interrupt sources.

Table A-2 Interrupt controller signals

Name	Type	Source/ destination	Description
VICINTSOURCE [31:0]	Input	Peripheral interrupt request	Interrupt source input
nVICIRQ	Output	Interrupt controller	Interrupt request to processor
nVICFIQ	Output	Interrupt controller	Fast interrupt request to processor

A.3 Daisy-chain signals

You use daisy-chain signals when two or more VICs are daisy-chained. See *Daisy-chained interrupt controller* on page 2-12. Table A-3 lists the daisy-chain signals.

Table A-3 Daisy-chain signals

Name	Type	Source/ destination	Description
VICVECTADDRIN [31:0]	Input	External interrupt controller	Connects to the **VICVECTADDROUT[31:0]** signal of the previous VIC if you use daisy-chaining. Connects to logic 0 if the VIC is not daisy-chained.
VICVECTADDROUT [31:0]	Output	Interrupt controller	Connects to the **VICVECTADDRIN[31:0]** signal of the next VIC if you use daisy-chaining. Left unconnected if the VIC is not daisy-chained.
nVICIRQIN	Input	External interrupt controller	Connects to the **nVICIRQ** signal of the previous VIC if you use daisy-chaining. Connects to logic 1 if the VIC is the last in the daisy-chain, or if VIC is not daisy-chained.
nVICFIQIN	Input	External interrupt controller	Connects to the **nVICFIQ** signal of the previous VIC if you use daisy-chaining. Connects to logic 1 if the VIC is the last in the daisy-chain, or if VIC is not daisy-chained.

A.4 Scan test control signals

Table A-4 lists the internal scan test control signals.

Table A-4 Scan test control signals

Name	Type	Source/ destination	Description
SCANENABLE	Input	Scan controller	Scan enable
SCANINHCLK	Input	Scan controller	Scan data input for **HCLK** domain
SCANOUTHCLK	Output	Scan controller	Scan data output for **HCLK** domain

Appendix B
Example Code

This appendix provides examples of the code required when setting up the ARM PrimeCell Vectored Interrupt Controller (PL190). It contains the following section:

- *About the example code* on page B-2.

B.1 About the example code

This section provides the following examples of code:

- *Enable interrupts*
- *Disable interrupts*
- *Interrupt polling* on page B-3
- *Generate software interrupt* on page B-3
- *Clear software interrupt* on page B-3
- *FIQ interrupt initialization* on page B-4
- *FIQ interrupt handler* on page B-4
- *Simple interrupt initialization* on page B-4
- *Simple interrupt service routine* on page B-5
- *Vectored interrupt initialization* on page B-5
- *Vectored interrupt service routine* on page B-6
- *Daisy-chained vectored interrupt service routine* on page B-6
- *Highest level vectored IRQ interrupt service routine* on page B-7.

B.1.1 Enable interrupts

Example B-1 gives an example of the enable interrupt code.

Example B-1 Enable interrupts

```
LDR    r0, =IntCntlBase                    ;  where IntCntlBase is a predefined constant
                                           ;  for example, IntCntlBase EQU 0xFFFFF000
MOV    r1, #<interrupt to enable>
STR    r1, [r0, #IntEnableOffset]
```

B.1.2 Disable interrupts

Example B-2 gives an example of the disable interrupt code.

Example B-2 Disable interrupts

```
LDR    r0, =IntCntlBase
MOV    r1, #<interrupt to disable>
STR    r1, [r0, #IntEnableClearOffset]
```

B.1.3 Interrupt polling

Example B-3 gives an example of the interrupt polling code.

Example B-3 Interrupt polling

```
      LDR    r0, =IntCntlBase
Loop  LDR    r1, [r0, #RawInterruptOffset]
      CMP    r1, #0
      BEQ    loop

      ;  Scan r1 for source of interrupt & branch to relevant routine
```

B.1.4 Generate software interrupt

Example B-4 gives an example of the generate software interrupt code.

Example B-4 Generate software interrupt

```
      ;  Generate software interrupt on interrupt request line 1

      LDR    r0, =IntCntlBase
      MOV    r1, #2                          ;  Interrupt source/request 1
      STR    r1, [r0, #SoftIntOffset]
```

B.1.5 Clear software interrupt

Example B-5 gives an example of the clear software interrupt code.

Example B-5 Clear software interrupt

```
      ;  Clear software interrupt on interrupt request line 1.

      LDR    r0, =IntCntlBase                ;  where IntCntlBase is a predefined constant,
                                             ;  for example, IntCntlBase EQU 0xFFFFF000
      MOV    r1, #2
      STR    r1, [r0, #SoftIntClearOffset]
```

B.1.6 FIQ interrupt initialization

Example B-6 gives an example of the FIQ interrupt initialization code.

Example B-6 FIQ interrupt initialization

```
      LDR    r0, =IntCntlBase
      MOV    r1, #<interrupt_to_enable>
      STR    r1, [r0, #IntSelectOffset]              ;  Select FIQ interrupt and clear other FIQs

      STR    r1, [r0, #IntEnableOffset]              ;  Enable interrupt

      MRS    CPSR_c, #(DISABLE_IRQ + MODE_SYS_32)    ;  Enable FIQ interrupts
```

B.1.7 FIQ interrupt handler

Example B-7 gives an example of the FIQ interrupt handler code.

Example B-7 FIQ interrupt handler

```
      ;  IRQ and FIQ interrupts are automatically masked until return from interrupt performed

0x1c  ;  Interrupt service routine
      ;  Clear interrupt request
      SUBS    pc, r14, #4
```

B.1.8 Simple interrupt initialization

Example B-8 shows how you can use the interrupt controller without using vectored interrupts, or the interrupt priority hardware. For example, you can use it for debugging.

Example B-8 Simple interrupt initialization

```
      LDR    r0, =IntCntlBase
      MOV    r1, #<interrupt_to_enable>
      LDR    r2, [r0, #IntSelectOffset]              ;  Select IRQ interrupt
      BIC    r2, r2, r1
      STR    r2, [r0, #IntSelectOffset]
      STR    r1, [r0, #IntEnableOffset]              ;  Enable interrupt

      MRS    CPSR_c, #(DISABLE_IRQ + MODE_SYS_32)    ;  Enable FIQ interrupts
```

B.1.9 Simple interrupt service routine

Example B-9 shows how you can use the interrupt controller without using vectored interrupts, or the interrupt priority hardware. For example, you can use it for debugging.

Example B-9 Simple interrupt service routine

```
      ;  This interrupt service routine assumes that there are no vectored interrupts. It also
      ;  assumes that interrupts are disabled until the interrupt service routine has been exited.

      ;  IRQ interrupts are masked until a return from interrupt is performed
      ;  The FIQ interrupt is enabled

0x18  B     IRQ_ISR                                    ;  Branch to interrupt service routine

IRQ_ISR
      STMFD  sp!, {r0, r1}                             ;  Store r0 and r1
      LDR    r0, [IntCntlBase]
      LDR    r1, [r0, #IRQStatusOffset]                ;  Discover source of interrupt

      Scan r1 for source of interrupt & branch to relevant routine ISR
      Interrupt service routine
      Clear interrupt request

      LDMFD  sp!, {r0, r1}                             ;  Restore r0 and r1
      SUBS   pc, r14, #4                               ;  Exit from IRQ
```

B.1.10 Vectored interrupt initialization

Example B-10 gives an example of the vectored interrupt initialization code.

Example B-10 Vectored interrupt initialization

```
      LDR    r0, =IntCntlBase
      MOV    r1, #<interrupt_to_enable>
      STR    r1, [r0, #IntEnableClearOffset]           ;  Disable interrupt

      LDR    r2, =default_vector_address               ;  Set default vector address
      STR    r2, [r0, #DefaultVectorAddressOffset]     ;  Setup and enable vectored interrupt 15
      MOV    r2, #vector_address                       ;  Set vector address
      STR    r2, [r0, #VectorAddr15Offset]
      MOV    r2, #interrupt_source                     ;  Set interrupt source
      ORR    r2, r2, #0x20                             ;  and enabled vector interrupt
      STR    r2, [r0, #VectorCntl15Offset]
      LDR    r2, [r0, #IntSelectOffset]                ;  Select IRQ interrupt
```

```
       BIC    r2, r2, r1
       STR    r2, [r0, #IntSelectOffset]
       STR    r1, [r0, #IntEnableOffset]                    ;  Enable interrupt
       MRS    CPSR_c, #(DISABLE_IRQ + MODE_SYS_32)          ;  Enable FIQ interrupts
```

B.1.11 Vectored interrupt service routine

Example B-11 gives an example of the vectored interrupt service routine code.

Example B-11 Vectored interrupt service routine

```
0x18    LDR pc,    [pc, #-0xff0]    ;  Load Vector into PC
;  ..................................................

vector_handler
       ;  Code to enable interrupt nesting
       STMFD r13!, {r12, r14}      ;  stack lr_irq and r12 [plus other regs used below, if appropriate]
       MRS r12, spsr               ;  Copy spsr into r12...
       STMFD r13!, {r12}           ;  and save to stack

;  Read from VICIRQStatus to determine the source of the interrupt
       MSR cpsr_c, #0x1f           ;  Switch to SYS mode, re-enable IRQ
       STMFD r13!, {r0-r3, r14}    ;  stack lr_sys and r0-r3

;  Interrupt service routine...
;  NOTE: ADS 1.2 requires preservation of 8-byte stack alignment with respect to all external
;  interfaces. See ADS 1.2 Developer Guide - Section 2.3.3
;  ...
       BL 2nd_level_handler        ;  this corrupts lr_sys and r0-r3
;  ...

;  Add code to clear the interrupt source;  Code to exit handler
       LDMFD r13!, {r0-r3, r14}     ;  unstack lr_sys and r0-r3
       MSR cpsr_c, #0x92            ;  Disable IRQ, and return to IRQ mode
       LDMFD r13!, {r12}            ;  unstack r12...
       MSR spsr_cxsf, r12           ;  and restore spsr...
       LDMFD r13!, {r12, r14}       ;  unstack registers
       LDR r1, =VectorAddr
       STR r0, [r1]                 ;  Acknowledge VIRQ serviced
       SUBS pc, lr, #4              ;  Return from ISR
```

B.1.12 Daisy-chained vectored interrupt service routine

Example B-12 on page B-7 gives an example of the daisy-chained vectored interrupt service routine code.

Example B-12 Daisy-chained vectored interrupt service routine

```
0x18   LDR    pc, [pc, #-0xff0]              ;  Load vector into PC

vector_handler      ;  Code to enable interrupt nesting. First, stack off registers you know will be
corrupted      STMFD r13!, {r0-r3, r12, r14}      ;  Use r12 to stack off the spsr      MRS   r12, spsr
 ; Copy spsr to r12      STMFD r13!, {r12} ; Stack spsr in r12       ;  Change from IRQ mode to System
mode, and re-enable interrupts      MSR   cpsr_c, #0x1F       ;  Branch to the function that:      ;
 1. Clears the peripheral interrupt. Do this first      ;  2. Performs the interrupt function
      ;  Stack the link register of System mode

      STMFD SP!, {lr}      BL some_interrupt_code      LDMFD SP!, {lr}       ;  When the interrupt has
finished, disable interrupts so that you can update the VIC and your
      ;  mode without worrying about being interrupted      MSR   cpsr_c, #0x92     ;  Disable
interrupts and return to IRQ mode      ;  Acknowledge that the IRQ has finished being serviced. You can
do this because the interrupts
      ;  are now disabled, so the ARM core runs this section of code up until the end,
uninterrupted      LDR   r12, =VectorAddr  ;  VectorAddr should be = 0xFFFFF030      STR   r0,
[r12]         ;  Not important what r0 contains      ;  Stacking operations - first, restore the spsr
using r12 as a temporary register      LDMFD r13!, {r12}       ;  Pop the spsr off the stack      MSR
 spsr_cxsf, r12     ;  and restore it      ;  Pop remaining registers off the stack. This corresponds
to the first STMFD of this function      LDMFD r13!, {r0-r3, r12, r14}      ;  Return from the
interrupt handler      SUBS  pc, lr, #4
```

B.1.13 Highest level vectored IRQ interrupt service routine

Example B-13 gives an example of the highest level vectored IRQ interrupt service routine code.

Example B-13 Highest level vectored IRQ interrupt service routine

```
0x18   LDR    pc, [pc, #-0xff0]              ;  Load vector into PC

highest_priority_vector_handler

       Interrupt_service_routine

       ;  Code to exit handler
       STR    r0, VectorAddr                 ;  Acknowledge Vectored IRQ has
                                             ;  finished
       SUBS   pc, r14, #4                    ;  Return from IRQ
```


Glossary

This glossary describes some of the terms used in ARM manuals. Where terms can have several meanings, the meaning presented here is intended.

Abort

A mechanism that indicates to a core that the value associated with a memory access is invalid. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory. An abort is classified as either a Prefetch or Data Abort, and an internal or External Abort.

See also Data Abort, External Abort and Prefetch Abort.

Advanced eXtensible Interface (AXI)

This is a bus protocol that supports separate address/control and data phases, unaligned data transfers using byte strobes, burst-based transactions with only start address issued, separate read and write data channels to enable low-cost DMA, ability to issue multiple outstanding addresses, out-of-order transaction completion, and easy addition of register stages to provide timing closure.The AXI protocol also includes optional extensions to cover signaling for low-power operation.

AXI is targeted at high performance, high clock frequency system designs and includes a number of features that make it very suitable for high speed sub-micron interconnect.

Advanced High-performance Bus (AHB)
The AMBA Advanced High-performance Bus system connects embedded processors such as an ARM core to high-performance peripherals, DMA controllers, on-chip memory, and interfaces. It is a high-speed, high-bandwidth bus that supports multi-master bus management to maximize system performance.

See also Advanced Microcontroller Bus Architecture and AHB-Lite.

Advanced Microcontroller Bus Architecture (AMBA)
AMBA is the ARM open standard for multi-master on-chip buses, capable of running with multiple masters and slaves. It is an on-chip bus specification that details a strategy for the interconnection and management of functional blocks that make up a *System-on-Chip* (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules. AHB, APB, and AXI conform to this standard.

Advanced Peripheral Bus (APB)
The AMBA Advanced Peripheral Bus is a simpler bus protocol than AHB. It is designed for use with ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption.

See also Advanced High-performance Bus.

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.

AMBA *See* Advanced Microcontroller Bus Architecture.

APB *See* Advanced Peripheral Bus.

Architecture The organization of hardware and/or software that characterizes a processor and its attached components, and enables devices with similar characteristics to be grouped together when describing their behavior, for example, Harvard architecture, instruction set architecture, ARMv6 architecture.

ARM instruction A word that specifies an operation for an ARM processor to perform. ARM instructions must be word-aligned.

ASIC *See* Application Specific Integrated Circuit.

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.

AXI *See* Advanced eXstensible Interface.

Beat Alternative word for an individual transfer within a burst. For example, an INCR4 burst comprises four beats.

See also Burst.

Burst A group of transfers to consecutive addresses. Because the addresses are consecutive, there is no requirement to supply an address for any of the transfers after the first one. This increases the speed at which the group of transfers can occur. Bursts over AHB buses are controlled using the **HBURST** signals to specify if transfers are single, four-beat, eight-beat, or 16-beat bursts, and to specify how the addresses are incremented.

See also Beat.

Byte An 8-bit data item.

Core A core is that part of a processor that contains the ALU, the datapath, the general-purpose registers, the Program Counter, and the instruction decode and control circuitry.

Data Abort An indication from a memory system to the core of an attempt to access an illegal data memory location. An exception must be taken if the processor attempts to use the data that caused the abort.

See also Abort, External Abort, and Prefetch Abort.

Direct Memory Access (DMA)
An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

DMA *See* Direct Memory Access.

Event 1 (Simple) An observable condition that can be used by an ETM to control aspects of a trace.

2 (Complex) A boolean combination of simple events that is used by an ETM to control aspects of a trace.

Exception A fault or error event that is considered serious enough to require that program execution is interrupted. Examples include attempting to perform an invalid memory access, external interrupts, and undefined instructions. When an exception occurs,

normal program flow is interrupted and execution is resumed at the corresponding exception vector. This contains the first instruction of the interrupt handler to deal with the exception.

Exception vector *See* Interrupt vector.

External Abort An indication from an external memory system to a core that the value associated with a memory access is invalid. An external abort is caused by the external memory system as a result of attempting to access invalid memory.

See also Abort, Data Abort and Prefetch Abort.

High vectors Alternative locations for exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

Interrupt handler A program that control of the processor is passed to when an interrupt occurs.

Interrupt vector One of a number of fixed addresses in low memory, or in high memory if high vectors are configured, that contains the first instruction of the corresponding interrupt handler.

Microprocessor *See* Processor.

Multi-master An AMBA bus sharing scheme (not in AMBA Lite) where different masters can gain a bus lock (Grant) to access the bus in an interleaved fashion.

Prefetch Abort An indication from a memory system to the core that an instruction has been fetched from an illegal memory location. An exception must be taken if the processor attempts to execute the instruction. A Prefetch Abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory.

See also Data Abort, External Abort and Abort.

Processor A processor is the circuitry in a computer system required to process data using the computer instructions. It is an abbreviation of microprocessor. A clock source, power supplies, and main memory are also required to create a minimum complete working computer system.

Read Reads are defined as memory operations that have the semantics of a load. That is, the ARM instructions LDM, LDRD, LDC, LDR, LDRT, LDRSH, LDRH, LDRSB, LDRB, LDRBT, LDREX, RFE, STREX, SWP, and SWPB, and the Thumb instructions LDM, LDR, LDRSH, LDRH, LDRSB, LDRB, and POP.

Java instructions that are accelerated by hardware can cause a number of reads to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

Reserved A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as 0 and read as 0.

Unaligned A data item stored at an address that is not divisible by the number of bytes that defines the data size is said to be unaligned. For example, a word stored at an address that is not divisible by four.

Unpredictable Means that you cannot rely on the behavior of the ETM. Such conditions have not been validated. When applied to the programming of an event resource, only the output of that event resource is Unpredictable.Unpredictable behavior can affect the behavior of the entire system, because the ETM is capable of causing the core to enter debug state, and external outputs can be used for other purposes.

Unpredictable For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. Unpredictable instructions must not halt or hang the processor, or any part of the system.

Word A 32-bit data item.

Write Writes are defined as operations that have the semantics of a store. That is, the ARM instructions SRS, STM, STRD, STC, STRT, STRH, STRB, STRBT, STREX, SWP, and SWPB, and the Thumb instructions STM, STR, STRH, STRB, and PUSH.

Java instructions that are accelerated by hardware can cause a number of writes to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

Index

H

I

N

P

R

S

T

V

