Advanced Communications Channel

Architecture Specification

arm

IHI 0076A (ID041218)

Advanced Communications Channel

Architecture Specification

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
02 May 2018	A	Non-Confidential	First release.

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Product Status

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

Web Address

http://www.arm.com

Contents
Advanced Communications Channel Architecture Specification

Preface

This preface introduces the *Advanced Communications Channel Architecture Specification*. It contains the following sections:

About this book

This book is the *Architecture Specification* for the *Advanced Communications Channel*.

Intended audience

This specification is written for:

- System designers who require a communications channel for communication between an external debugger and a hardware or software agent on a target system.
- System designers who require communications functionality for simple on-chip communication.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the Advanced Communications Channel.

Chapter 2 *COM Encapsulation Protocol*

Read this chapter for a description of the COM Encapsulation Protocol, Flag bytes, and Session management.

Chapter 3 *COM Port programmers' model*

Read this chapter for a description of the programmers' model for the COM Port.

Chapter 4 *COM Port Peripheral programmers' model*

Read this chapter for a description of the programmers' model for the COM Port Peripheral.

Appendix A *Example Messages*

Read this appendix for a description of example Messages.

Appendix B *Pseudocode Definition*

Read this appendix for a description of the pseudocode that is used in this document.

Glossary

Read this glossary for definitions of some of the terms that are used in this manual. The *Arm Glossary* does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

Note

Arm publishes a single glossary that relates to most Arm products, see the *Arm Glossary* http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html. A definition in the glossary in this book might be more detailed than the corresponding definition in the *Arm Glossary*.

Conventions

The following sections describe conventions that this book can use:

- *Typographic conventions*.
- *Signals*.
- *Timing diagrams*.
- *Numbers* on page xi.
- *Pseudocode descriptions* on page xi.

Typographic conventions

The following table describes the typographical conventions:

italic	Introduces special terminology, and denotes citations.
bold	Denotes signal names, and is used for terms in descriptive lists, where appropriate.
`monospace`	Used for assembler syntax descriptions, pseudocode, and source code examples. Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.
SMALL CAPITALS	Used for a few terms that have specific technical meanings, and are included in the *Glossary* on page Glossary-81.
Colored text	Indicates a link. This can be: • A URL, for example `http://infocenter.arm.com`. • A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, *Pseudocode descriptions* on page xi. • A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term, for example *AMBA* on page Glossary-81.

Signals

In general this specification does not define processor signals, but it does include some signal examples and recommendations.

The signal conventions are:

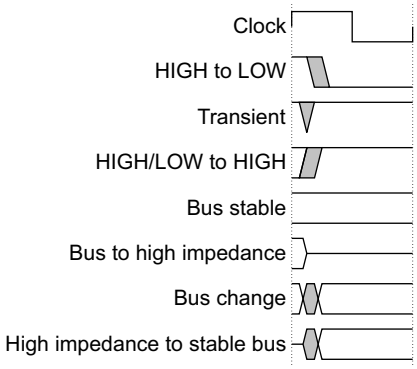
Signal level The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:
- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lower-case n At the start or end of a signal name denotes an active-LOW signal.

Timing diagrams

The figure *Key to timing diagram conventions* on page xi explains the components used in timing diagrams. 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

Timing diagrams sometimes show single-bit signals as HIGH and LOW at the same time and they look similar to the bus change shown in *Key to timing diagram conventions*. If a timing diagram shows a single-bit signal in this way then its value does not affect the accompanying description.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a `monospace` font, for example `0xFFFF0000`.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a `monospace` font, and is described in Appendix B *Pseudocode Definition*.

Additional reading

This section lists relevant publications from Arm and third parties.

See the Infocenter http://infocenter.arm.com, for access to Arm documentation.

Arm publications

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

- *ADIv5 Architecture Specification (*ARM IHI 0031).
- *ADIv6 Architecture Specification* (ARM IHI 0074).
- *Arm® CoreSight™ Architecture Specification* (ARM IHI 0029).

Other publications

This section lists relevant documents published by third parties:

- JEDEC, *Standard Manufacturers Identification Code,* JEP106 hhtp://www.jedec.org.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to errata@arm.com. Give:

- The title.
- The number, IHI 0076A.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Chapter 1
Introduction

This chapter introduces the Advanced Communications Channel. It contains the following section:

- *About the Advanced Communications Channel* on page 1-16.

1.1 About the Advanced Communications Channel

This section introduces and describes the components that make up the Advanced Communications Channel architecture. It contains the following subsections:

- *Advanced Communications Channel functionality*.
- *Duplex and Simplex communication*.

1.1.1 Advanced Communications Channel functionality

The Advanced Communications Channel is a communications channel that provides low-cost transport of byte-based protocols in a point-to-point system.

The primary objective of the Communications Channel is to enable communication from an external debugger to a hardware or software agent on a target system. The architecture in this specification also applies to simple on-chip communication.

The Advanced Communications Channel functionality can be divided into two parts:

The COM Encapsulation Protocol

The COM Encapsulation Protocol encapsulates individual arbitrary byte-based messages using byte stuffing. Extra commands are provided for session management. For more information, see Chapter 2 *COM Encapsulation Protocol*.

The COM Port

The COM Port provides a memory-mapped programmers' model for sending and receiving an arbitrary byte stream. This byte stream is encapsulated using the COM Encapsulation Protocol. For more information, see Chapter 3 *COM Port programmers' model*.

1.1.2 Duplex and Simplex communication

A system incorporating the Advanced Communications Channel can be one of two types:

Duplex

For bidirectional communication.

Simplex

For unidirectional communication.

Figure 1-1 shows a conceptual diagram of Duplex communication where typically, a Master Agent initiates communication.

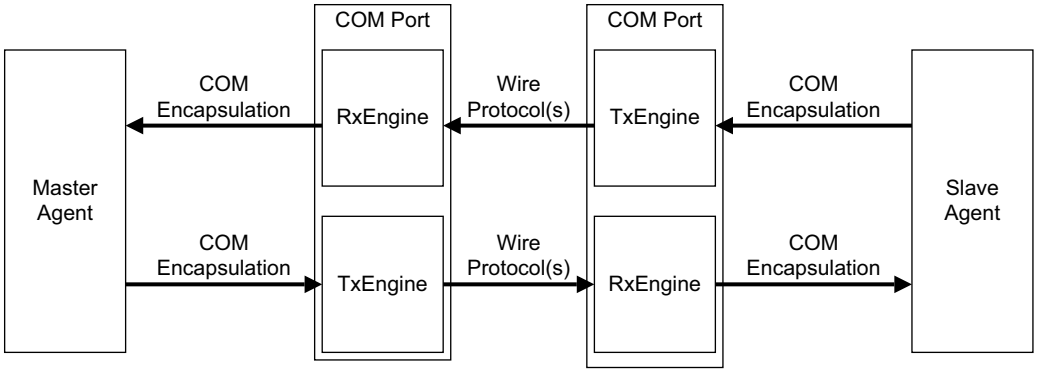


Figure 1-1 Block diagram of a typical Duplex communication

Figure 1-2 shows a conceptual diagram of Simplex communication.

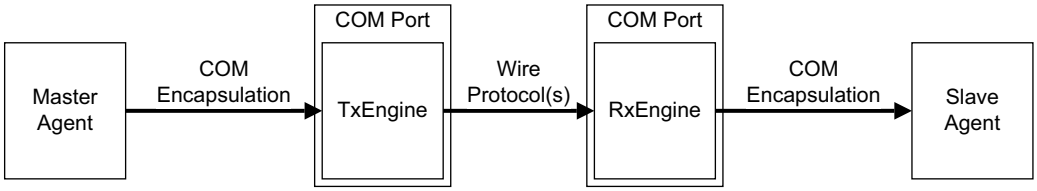


Figure 1-2 Block diagram of a typical Simplex communication

Note

For each *TxEngine,* there is a corresponding remote *RxEngine.* This is situated in the destination COM Port where the COM Encapsulation bytes are sent. In a Duplex system, each *TxEngine* also has a local *RxEngine* in the same COM Port.

Chapter 2
COM Encapsulation Protocol

This chapter describes the COM Encapsulation Protocol. It contains the following sections:

2.1 About the COM Encapsulation Protocol

This section introduces the COM Encapsulation Protocol. It contains the following subsections:

- *Byte stuffing*.
- *Flag bytes*.
- *Duplex session management*.

2.1.1 Byte stuffing

The COM Encapsulation Protocol uses byte stuffing to encapsulate an individual byte-based message of arbitrary length.

2.1.2 Flag bytes

Special Flag bytes exist for control and encapsulation purposes. These are:

- Start and End Flag bytes, for framing.
- An Escape Flag byte, for distinguishing message bytes from Flag bytes.
- A Null Flag byte, for padding at any time.
- Session management Flag bytes.
- Link persistency Flag bytes.
- ID request and ID response Flag bytes, to inform agents of the messaging protocols used.

Flag bytes are always detectable in the byte stream by analyzing any individual byte, and do not occur anywhere else.

2.1.3 Duplex session management

The COM Encapsulation Protocol provides mechanisms to manage a communication session between two agents.

In a Duplex system, the Master Agent establishes a session with the Slave Agent, before sending any normal messages. The process to establish a session consists of the following phases:

- *Phase 1:* Ensuring the remote RxEngine and TxEngine are both connected and active.
- *Phase 2:* Ensuring the Slave Agent is active, and ready to communicate.
- *Phase 3:* Determining the identification of the Slave Agent to ensure that the correct messaging protocol is used.

The following pseudocode demonstrates a possible Duplex session approach:

```
SessionEstablish()
  bool poll_complete = false;

  // Assert nSRST if needed
  if (RemoteRebootRequired && nsrstRequired()) then
    nSRST = LOW;

  // Power up
  if (RemotePowerRequired()) then
    // Release the link first, to get the link into a known state.
    SendTxEngine(LPH1RL);
    poll_complete = false;
    repeat
      if TimeoutReached() then
        Error();
      case ReadRxEngine() of
        when LPH1RL
```

```
        poll_complete = true;
      when NULL
        // do nothing
      otherwise
        Error();
  until poll_complete;

  // Send a new link power up request
  SendTxEngine(LPH1RA);
  poll_complete = false;
  repeat
    if TimeoutReached() then
      Error();
    case ReadRxEngine() of
      when LPH1RA
        poll_complete = true;
      when NULL
        // do nothing
      otherwise
        Error();
  until poll_complete;

// Link Establish
SendTxEngine(LPH2RA);
if (RemoteRebootRequired()) then
  if (nsrstRequired()) then
    nSRST = 1;
  else
    SendTxEngine(LPH2RR);
poll_complete = false;
repeat
  if TimeoutReached() then
    Error();
  case ReadRxEngine() of
    when LPH2RA
      poll_complete = true;
    when NULL
      // do nothing
    otherwise
      Error();
until poll_complete;

// ID Request
SendTxEngine(IDR);
poll_complete = false;
repeat
  case ReadRxEngine() of
    when IDA
      poll_complete = true;
    when NULL
      // do nothing
    otherwise
      Error();
until poll_complete;

next_is_non_flag = false;
poll_complete = false;
repeat
  rdbyte = ReadRxEngine();
  if ((rdbyte & 0xE0) != 0xA0) then
    if next_is_non_flag then
      rdbyte = rdbyte ^ 0x80;
      next_is_non_flag = false;
    AddByteToID(rdbyte);
  else
    case rdbyte of
      when NULL
        // do nothing
```

```
        when ESC
          next_is_non_flag = true;
        when END
          poll_complete = true;
        otherwise
          Error();
  until poll_complete;
```

When the session is established, the Master Agent sends and receives messages according to the messaging protocol used by the Master and Slave Agents. The Master Agent must terminate the session when the session is complete.

The following pseudocode demonstrates a possible approach to terminating the session:

```
SessionTerminate()
  bool poll_complete = false;

  // Link Terminate
  SendTxEngine(LPH2RL);
  poll_complete = false;
  repeat
    if TimeoutReached() then
      Error();
    case ReadRxEngine() of
      when LPH2RL
        poll_complete = true;
      when NULL
        // do nothing
      otherwise
        Error();
  until poll_complete;

  // Power release
  if (RemotePowerRequired()) then
    SendTxEngine(LPH1RL);
    poll_complete = false;
    repeat
      if TimeoutReached() then
        Error();
      case ReadRxEngine() of
        when LPH1RL
          poll_complete = true;
        when NULL
          // do nothing
        otherwise
          Error();
    until poll_complete;
```

2.2 Specification

This section outlines the specification for the COM Encapsulation Protocol. It contains the following section:

- *Rules*.

2.2.1 Rules

The following rules apply when implementing the Advanced Communication Channel Architecture Specification:

R_{TYNM} A message consists of one or more Message bytes.

R_{PLZQ} All bytes with bits [7:5] that take the value `b101` are classified as Flag bytes.

R_{MMQW} Every byte transmitted is either a Message byte or a Flag byte.

R_{GHLK} The Flag byte values that are shown in Table 2-1 have defined meanings.

Table 2-1 Flag bytes

Value	Name	(S)implex / (D)uplex	Meaning
`0xA0`	IDR	D	Identification Request.
`0xA1`	IDA	D	Identification Acknowledge and start of ID PDU.
`0xA2 - 0xA5`	-	-	Reserved.
`0xA6`	LPH1RA	S, D	Link Phase 1 Request/Acknowledge.
`0xA7`	LPH1RL	S, D	Link Phase 1 Release/Acknowledge.
`0xA8`	LPH2RA	D	Link Phase 2 Request/Acknowledge.
`0xA9`	LPH2RL	D	Link Phase 2 Release/Acknowledge.
`0xAA`	LPH2RR	S, D	Link Phase 2 Reboot Request.
`0xAB`	LERR	S, D	Link Error.
`0xAC`	START	S, D	Start of PDU.
`0xAD`	END	S, D	End of PDU.
`0xAE`	ESC	S, D	Escape.
`0xAF`	NULL	S, D	Null.
`0xB0 - 0xBF`	-	-	Reserved.

R_{HKML} Each Message byte that matches one of the Flag bytes is immediately preceded by the ESC Flag byte, and bit [7] of the Message byte is inverted.

R_{TZVZ} Any Flag byte that immediately follows an ESC Flag byte is treated as if the ESC byte does not apply. The effect of the ESC Flag byte is delayed until the following byte. This delay continues until one of the following occurs:

- A Message byte, which has bit [7] inverted.
- A START Flag byte, which is unmodified.
- An IDR Flag byte, which is unmodified.

R_{BMNP} A Message Protocol Data Unit (Message PDU) consists of the following, in sequential order:

- A START Flag byte.
- Zero or more Message bytes, including any required ESC Flag bytes.
- An END Flag byte.

R_{PZGZ} A NULL Flag byte can occur at any time and the receiver discards it.

R_{TXOX} A LPH1RA Flag byte is a request for the link to the receiver to be enabled.

R_{TLPP} In a Duplex system, an LPH1RA Flag byte is sent back to the transmitter when an LPH1RA Flag byte is received and a link between the transmitter and receiver is enabled.

I_{XDDS} A LPH1RL Flag byte is an indication that the link to the receiver can be disabled.

R_{YYZS} In a Duplex system, when an LPH1RL Flag byte is received, an LPH1RL Flag byte is sent back to the transmitter as an acknowledgement that the first LPH1RL Flag byte has been received and processed.

R_{JBNS} A LPH2RA Flag byte is a request for the receiver to proceed to a state where the receiver can process any future bytes.

R_{SVJJ} In a Duplex system, when the RxEngine receives a LPH2RA Flag byte, the Slave Agent sends a LPH2RA Flag byte back to the transmitter when the Slave Agent is ready to receive more bytes in addition to the LPH2RA Flag byte.

R_{YPLJ} An ID Protocol Data Unit (ID PDU) consists of the following, in sequential order:

- An IDA Flag byte.
- One or more Message bytes which indicate the protocol that is used by Message PDUs. This portion might include any required ESC Flag bytes.
- An END Flag byte.

R_{YZMX} An ID PDU is not interleaved with a Message PDU.

R_{JBBC} In a Duplex system, an LERR Flag byte is queued up in the transmitter's RxEngine if the link between transmitter and receiver is disabled or terminated, and the system can detect that one or more bytes in transit have been lost.

I_{KKLD} Owing to the handshake nature of establishing a link between a Master Agent and Slave Agent, only one of the link-establishing Flag bytes is in-flight at any one time. If more than one of the following Flag bytes is queued up in an RxEngine, two or more are permitted to be replaced by a single LERR Flag byte:

- LPH1RA.
- LPH1RL.
- LPH2RA.
- LPH2RL.
- LERR.

Chapter 3
COM Port programmers' model

This chapter describes the COM Port programmers' model. It contains the following sections:

3.1 About the COM Port programmers' model

The COM Port provides a memory-mapped programmers' model to send and receive messages using the COM Encapsulation Protocol.

An optional TxEngine is provided to send one or more bytes.

An optional RxEngine is provided to receive one or more bytes.

Optional interrupt controls are provided to indicate when the RxEngine or TxEngine requires attention.

3.1.1 Interrupts

The TxEngine and RxEngine provide optional interrupt outputs to indicate to an agent that the COM Port needs attention. The TxEngine interrupt is used to instruct the agent that the TxEngine might be able to accept more bytes for transmission. The RxEngine interrupt is used to instruct the agent that there is data ready for processing.

When enabled, an TxEngine interrupt is generated when the following is true:

- A TxEngine FIFO has less than a programmed number of bytes remaining to transmit.

When enabled, an RxEngine interrupt is generated when any of the following are true:

- The RxEngine FIFO has at least a programmed number of bytes are ready to be read.
- A Flag byte is received that is not one of the following:
  - ESC.
  - NULL.
  - START.
- The RxEngine FIFO is full.

3.2 COM Port register index

Table 3-1 shows the register index for the COM Port programmers' model.

Table 3-1 Memory-mapped register map

Offset	Access	Size	Register	Description
0x00	RO	32	VIDR	Version ID Register.
0x08	RO	32	FIDTXR	Feature ID TxEngine Register.
0x0C	RO	32	FIDRXR	Feature ID RxEngine Register.
0x10	R/W	32	ICSR	Interrupt Control/Status Register.
0x20	R/W	32	DR	Data Register.
0x2C	R/W	32	SR	Status Register.
0x30	R/W	32	DBR	Data Blocking Register.
0x3C	R/W	32	SR	Status Register.

3.3 COM Port register resets

Table 3-2 shows the register reset values for the COM Port programmers' model.

Table 3-2 Register resets

Register	Field	Reset value
ICSR	RXFIS	`0`
	RXFIL	`0x1`
	TXFIS	`0`
	TXFIL	`0x0`
SR	RXLE	`0`
	RXF	`0`
	TXLE	`0`
	TXOE	`0`

All other resettable registers and fields are reset to an IMPLEMENTATION DEFINED value, which can be UNKNOWN.

3.4 COM Port register descriptions

This section contains register descriptions for the COM Port programmers' model. It contains the following subsections:

3.4.1 DBR, Data Blocking Register

The DBR characteristics are:

Purpose The DBR is used to send data via the TxEngine and receive data from the RxEngine.

The DR and DBR operate identically, except on writes where more data is written to the TxEngine than the COM port can accept, demonstrated as follows:

- When writing to the DR, the write access completes and an overflow error is logged in SR.TXOE.
- When writing to the DBR, the write access stalls until there is sufficient space.

Usage constraints The DBR accesses are as follows:

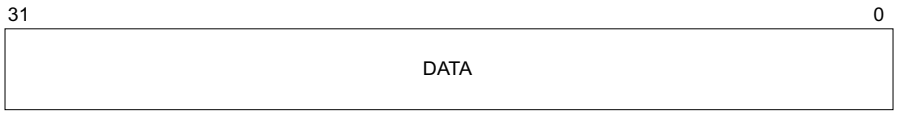
- RW when both the TxEngine and RxEngine are implemented.
- RO when only the RxEngine is implemented.
- WO when only the TxEngine is implemented.

Configurations Always implemented.

Attributes 32-bit read/write memory-mapped register at offset `0x30`.

Field descriptions

The DBR bit assignments are:



DATA, bits[31:0] This field is used for data transfer.

The FIDTXR and FIDRXR indicate the supported access sizes to the DBR.

On writes:

- Transfers bytes into the TxEngine FIFO for transmission.
- Bytes are transmitted from the least significant byte first.
- If FIDTXR.TXW indicates that the TxEngine width is less than the number of bytes written, the TxEngine ignores the upper unimplemented bytes. The upper unimplemented bytes must be written with the NULL Flag byte value.
- The TxEngine ignores any bytes which contain the NULL Flag byte value.

- If there is insufficient space in the TxEngine FIFO for all the bytes which are not NULL Flag bytes, the write to the DBR stalls until there is sufficient space. These writes do not complete immediately.
- If SR.TXOE is 1, writes are ignored.
- If SR.TXLE is 1, writes are ignored.

On reads:

- Returns bytes from the RxEngine FIFO.
- The oldest received byte is returned in the least significant byte.
- If there are fewer bytes in the RxEngine FIFO than requested by the access, the upper bytes are padded by the RxEngine with the NULL Flag byte.
- Read accesses complete immediately.

Accessing DBR

DBR can be accessed at the following address:

Offset
0x30

3.4.2 DR, Data Register

The DR characteristics are:

Purpose The Data Register is used to send data via the TxEngine and receive data from the RxEngine.

The DR and DBR operate identically except on writes where more data is written to the TxEngine than the COM port can accept, demonstrated as follows:

- When writing to the DR, the write access completes and an overflow error is logged in the SR.TXOE.
- When writing to the DBR, the write access stalls until there is sufficient space.

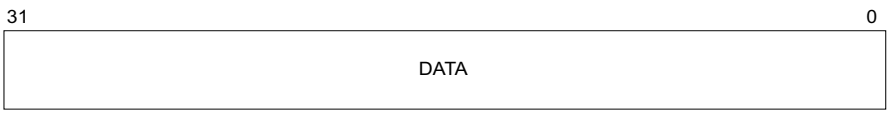
Usage constraints The DR accesses are as follows:

- RW when both the TxEngine and RxEngine are implemented.
- RO when only the RxEngine is implemented.
- WO when only the TxEngine is implemented.

Configurations Always implemented.

Attributes 32-bit read/write memory-mapped register at offset 0x20.

The DR bit assignments are:



DATA, bits[31:0] This field is used for data transfer.

The FIDTXR and FIDRXR indicate the supported access sizes to DBR.

On writes:

- Transfers bytes into the TxEngine FIFO for transmission.

- Bytes are transmitted from the least significant byte first.
- If FIDTXR.TXW indicates that the TxEngine width is less than the number of bytes written, the TxEngine ignores the upper unimplemented bytes. The upper unimplemented bytes must be written with the NULL Flag byte value.
- The TxEngine ignores any bytes which contain the NULL Flag byte value.
- If there is insufficient space in the TxEngine FIFO for all the bytes which are not NULL Flag bytes, a TxEngine Overflow error occurs and SR.TXOE is set to 1. The TxEngine discards excess bytes.
- If SR.TXOE is 1, writes are ignored.
- If SR.TXLE is 1, writes are ignored.
- Write accesses complete immediately.

On reads:

- Returns bytes from the RxEngine FIFO.
- The oldest received byte is returned in the least significant byte.
- If there are fewer bytes in the RxEngine FIFO than requested by the access, the upper bytes are padded by the RxEngine with the NULL Flag byte.
- Read accesses complete immediately.

Accessing DR

DR can be accessed at the following address:

Offset
0x20

3.4.3 FIDRXR, Feature ID RxEngine Register

The FIDRXR characteristics are:

Purpose The FIDRXR provides information about the features that are implemented in the COM Port RxEngine.

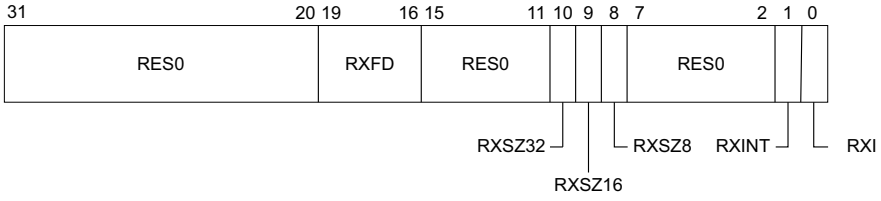
Usage constraints None.

Configurations Always implemented.

Attributes 32-bit read-only memory-mapped register at offset 0x0C.

Field descriptions

The FIDRXR bit assignments are:



Bits[31:20, 15:11, 7:2] RES0.

RXFD, bits[19:16] RxEngine FIFO depth.

The defined values of this field are:

`0x0 - 0xF` RxEngine FIFO has a capacity of at least 2^N bytes, where N is the value of this field.

This field is RES0 if the RxEngine is not implemented.

RXSZ32, bit[10] This field is for RxEngine 32-bit read support.

The defined values of this bit are:

`0b0` RxEngine does not support 32-bit wide reads.

`0b1` RxEngine supports 32-bit wide reads.

This field is RES0 if the RxEngine is not implemented.

RXSZ16, bit[9] This field is for RxEngine 16-bit read support.

The defined values of this bit are:

`0b0` RxEngine does not support 16-bit wide reads.

`0b1` RxEngine supports 16-bit wide reads.

This field is RES0 if the RxEngine is not implemented.

RXSZ8, bit[8] This field is for RxEngine 8-bit read support.

The defined values of this bit are:

`0b0` RxEngine does not support 8-bit wide reads.

`0b1` RxEngine supports 8-bit wide reads.

This field is RES0 if the RxEngine is not implemented.

RXINT, bit[1] This field indicates whether the RxEngine generates interrupts.

The defined values of this bit are:

`0b0` RxEngine interrupts not implemented.

`0b1` RxEngine interrupts are implemented.

This field is RES0 if the RxEngine is not implemented.

If the RxEngine interrupts are implemented, the following are implemented:

- ICSR.RXFIL.
- ICSR.RXFIS.

RXI, bit[0] This field indicates whether the RxEngine is implemented.

The defined values of this bit are:

`0b0` RxEngine not implemented.

`0b1` RxEngine implemented.

If the RxEngine is implemented, the following are implemented:

- SR.RXF.
- SR.RXLE.
- Reads of DR and DBR.

Accessing FIDRXR

FIDRXR can be accessed at the following address:

Offset
0x0C

3.4.4 FIDTXR, Feature ID TxEngine Register

The FIDTXR characteristics are:

Purpose The FIDTXR provides information about the features that are implemented in the COM Port TxEngine.

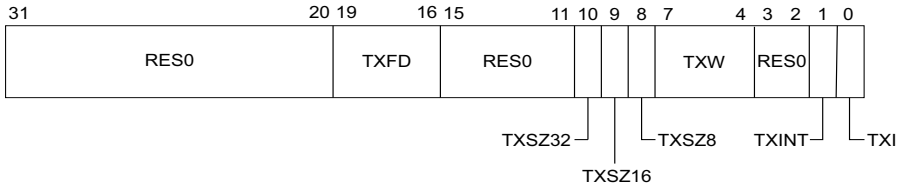
Usage constraints None.

Configurations Always implemented.

Attributes 32-bit read-only memory-mapped register at offset `0x08`.

Field descriptions

The FIDTXR bit assignments are:



Bits[31:20, 15:11, 3:2] RES0.

TXFD, bits[19:16] TxEngine FIFO depth.

The defined values of this field are:

`0x0 - 0xF` TxEngine FIFO has a capacity of at least 2^N bytes, where N is the value of this field.

This field is RES0 if the TxEngine is not implemented.

TXSZ32, bit[10] This field is for TxEngine 32-bit write support.

The defined values of this bit are:

`0b0` TxEngine does not support 32-bit wide writes.

`0b1` TxEngine supports 32-bit wide writes.

This field is RES0 if the TxEngine is not implemented.

TXSZ16, bit[9] This field is for TxEngine 16-bit write support.

The defined values of this bit are:

`0b0` TxEngine does not support 16-bit wide writes.

`0b1` TxEngine supports 16-bit wide writes.

This field is RES0 if the TxEngine is not implemented.

TXSZ8, bit[8] This field is for TxEngine 8-bit write support.

The defined values of this bit are:

`0b0` TxEngine does not support 8-bit wide writes.

`0b1` TxEngine supports 8-bit wide writes.

This field is RES0 if the TxEngine is not implemented.

TXW, bits[7:4] This field indicates the implemented width of the TxEngine.

The defined values of this field are:

`0x1` 1-byte wide TxEngine. Writes to DR or DBR must either be 1-byte wide or must have all bytes other than bits [7:0] set to the NULL Flag byte value.

`0x2` 2-byte wide TxEngine. Writes to DR or DBR must either be 1-byte or 2-bytes wide or must have all bytes other than bits [15:0] set to the NULL Flag byte value.

`0x4` 4-byte wide TxEngine.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

This field is RES0 if the TxEngine is not implemented.

TXINT, bit[1] This field indicates whether or not the TxEngine generates interrupts.

The defined values of this bit are:

`0b0` TxEngine interrupts not implemented.

`0b1` TxEngine interrupts implemented.

This field is RES0 if the TxEngine is not implemented.

If the TxEngine interrupts are implemented, the following are implemented:

- ICSR.TXFIL.
- ICSR.TXFIS.

TXI, bit[0] This field indicates whether the TxEngine is implemented.

The defined values of this bit are:

`0b0` TxEngine not implemented.

`0b1` TxEngine implemented.

If the TxEngine is implemented, the following are implemented:

- SR.TXS.
- SR.TXOE.
- SR.TXLE.
- Writes to DR and DBR.

Accessing FIDTXR

FIDTXR can be accessed at the following address:

Offset
`0x08`

3.4.5 ICSR, Interrupt Control/Status Register

The ICSR characteristics are:

Purpose The Interrupt Control/Status register controls the interrupts that the COM Port generates.

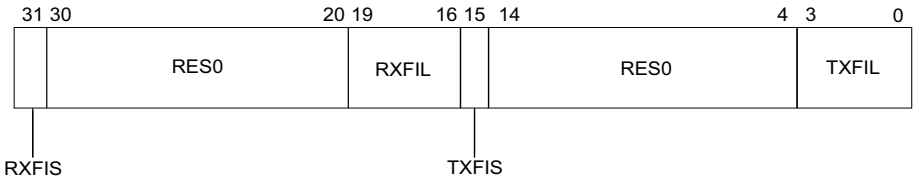
Usage constraints None.

Configurations Present only if FIDR.TXINT is 1 or FIDR.RXINT is 1.
RES0 otherwise.

Attributes 32-bit read/write memory-mapped register at offset `0x10`.

Field descriptions

The ICSR bit assignments are:



RXFIS, bit[31] This field indicates the RxEngine FIFO interrupt status.

The possible values of this bit are:

`0b0` RxEngine FIFO interrupt has not occurred.

`0b1` RxEngine FIFO interrupt has occurred.

This field is RES0 if FIDRXR.RXINT is 0.

This bit is read/write-one-to-clear.

This bit resets to zero.

Bits[30:20, 14:4] RES0.

RXFIL, bits[19:16] This field indicates the RxEngine FIFO interrupt level select.

The possible values of this field are:

`0x0` RxEngine FIFO interrupts disabled.

`0x1-0xF` Generate RxEngine FIFO interrupt when any of the following occur:

- RxEngine FIFO has at least the specified number of bytes ready.
- RxEngine FIFO is full.
- RxEngine FIFO detects a Flag byte other than one of the following:
  - ESC.
  - NULL.
  - START.

This field is RW.

This field is RES0 if FIDRXR.RXINT is 0.

This field resets to `0x1`.

TXFIS, bit[15] This field indicates the TxEngine FIFO interrupt status.

The possible values of this bit are:

`0b0` TxEngine FIFO interrupt has not occurred.

`0b1` TxEngine FIFO interrupt has occurred.

This field is RES0 if FIDTXR.TXINT is 0.

This bit is read/write-one-to-clear.

This bit resets to zero.

TXFIL, bits[3:0] This field indicates the TxEngine FIFO interrupt level select.

The possible values of this field are:

`0x0` TxEngine FIFO interrupts disabled.

`0x1-0xF` Generate TxEngine FIFO interrupt when the TxEngine FIFO has less than the specified number of bytes remaining to process.

This field is RW.

This field is RES0 if FIDTXR.TXINT is 0.

This field resets to `0x0`.

Accessing ICSR

ICSR can be accessed at the following address:

Offset
0x10

3.4.6 SR, Status Register

The SR characteristics are:

Purpose The Status Register indicates the status of the COM Port.

Usage constraints None.

Configurations Always implemented.

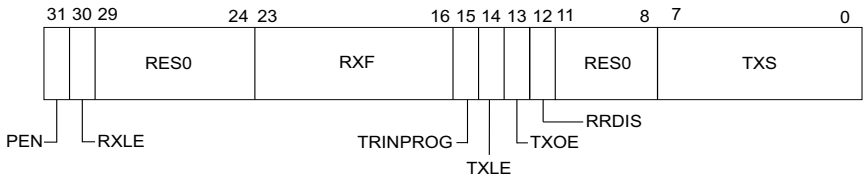
Attributes 32-bit read/write memory-mapped register located at both:

- Offset 0x2C.
- Offset 0x3C.

SR is aliased in more than one location. Accesses to any SR location access a single physical SR.

Field descriptions

The SR bit assignments are:



PEN, bit[31] This field indicates the COM Port enabled status.

The defined values of this bit are:

0b0 COM Port is disabled.
- Writes to DR and DBR are ignored.
- Reads of DR and DBR behave as if the RxEngine FIFO is empty.
- Interrupt outputs are disabled.

0b1 COM Port is enabled.

This bit is read-only.

RXLE, bit[30] This field indicates whether an RxEngine link error has been detected.

The possible values of this bit:

0b0 No link error detected.

0b1 A link error has been detected in the RxEngine.

This field is RES0 if FIDRXR.RXI is 0.

This bit is read/write-one-to-clear.

This bit resets to zero.

Bits[29:24, 11:8] RES0.

RXF, bits[23:16] This field indicates the RxEngine FIFO full level.

The possible values of this field are:

`0x00` RxEngine has no data.

`0x01-0xFF` RxEngine has at least the specified number of bytes available to read.

This field is RO.

This field is RES0 if FIDRXR.RXI is 0.

This field resets to zero.

TRINPROG, bit[15] Transaction in progress. The possible values of this bit are:

`0b0` No transaction in progress.

`0b1` An input transaction has been aborted but the internal operation of that transaction, or a previous transaction is still in progress.

This field is RES0 if the DP abort functionality is not implemented.

TXLE, bit[14] This field indicates whether a TxEngine link error has been detected.

The possible values of this bit are:

`0b0` No link error detected.

`0b1` A link error has been detected in the TxEngine.

This field is RES0 if FIDTXR.TXI is 0.

Set to 1 on any of the following:

- One or more bytes written to the TxEngine are discarded because the link to the remote RxEngine is not operating.
- A LERR Flag byte is inserted into the local RxEngine due to the link to the remote RxEngine not operating.

This bit is read/write-one-to-clear.

This bit resets to zero.

TXOE, bit[13] This field indicates the TxEngine FIFO overflow.

The possible values of this bit are:

`0b0` No overflow logged.

`0b1` At least 1 byte written to the TxEngine has not been accepted and has been lost.

This field is RES0 if FIDTXR.TXI is 0.

This bit is read/write-one-to-clear.

This bit resets to zero.

RRDIS, bits[12] This field indicates whether Remote Reboot requests are disabled. The defined values of this bit are:

`0b0` Remote Reboot requests enabled.

`0b1` Remote Reboot requests disabled.

When this field is 1, the TxEngine discards any LPH2RR Flag bytes written to the TxEngine.

This field is RES0 if FIDTXR.TXI is 0.

This bit is read-only.

TXS, bits[7:0] This field indicates the TxEngine FIFO space.

The defined values for this field are:

`0x00` TxEngine has no space for new data.

`0x01-0xFF` TxEngine has at least the specified number of bytes available.

This field resets to an IMPLEMENTATION DEFINED value.

This field is RES0 if FIDTXR.TXI is 0.

This field is read-only.

Accessing SR

SR can be accessed at the following addresses:

Offset
0x2C
0x3C

3.4.7 VIDR, Version ID Register

The VIDR characteristics are:

Purpose The Version ID Register provides information about the architecture version of the COM Port.

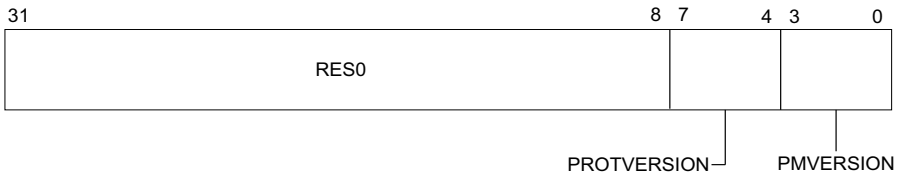
Usage constraints None.

Configurations Always implemented.

Attributes 32-bit read-only memory-mapped register at offset 0x00.

Field descriptions

The VIDR bit assignments are:



Bits[31:8] RES0.

PROTVERSION, bits[7:4]

This field indicates the COM Port Protocol version.

The defined values of this field are:

0x0 COM Port Protocol version 0 implemented.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

PMVERSION, bits[3:0]

This field indicates the COM Port programmers' model version.

The defined values of this field are:

0x0 COM Port programmers' model version 0 implemented.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Accessing VIDR

VIDR can be accessed at the following address:

Offset
0x00

Chapter 4
COM Port Peripheral programmers' model

This chapter describes the COM Port Peripheral programmers' model. It contains the following sections:

4.1 About the COM Port Peripheral

I_{FSZV} The COM Port functionality can be provided in a peripheral, which can be accessed by:

- On-chip software.
- An external debugger, via an interface that is Arm Debug Interface v6 compliant, see *ADIv6 Architecture Specification* for more information.

I_{HZLK} For usage models where an external debugger uses the COM Port to gain access to a system and communicate with on-chip software, typically two instances of the COM Port will exist. One for the external debugger to use and one for the on-chip software.

4.2 Com Port Peripheral register map

R_{ASJF} A COM Port Peripheral implements the register map that is shown in Table 4-1.

Table 4-1 COM Port Peripheral register map

Offset	Description
0x000-0xCFC	Reserved, RES0.
0xD00-0xD7C	COM Port programmers' model. See *About the COM Port programmers' model* on page 3-26.
0xD80-0xE7C	Reserved, RES0.
0xE80-0xEFC	IMPLEMENTATION DEFINED.
0xF00-0xFFC	CoreSight Management registers. See *CoreSight Management register descriptions* on page 4-45.

4.2.1 DP abort

The DP abort behavior is described as follows:

R_{OKLP} The COM Port Peripheral optionally implements the ADI DP abort mechanism.

R_{ZIJF} If there is no ongoing input transaction to the COM Port Peripheral when an abort request occurs, then the abort is ignored.

R_{TYZO} If there is an ongoing input transaction to the COM Port Peripheral when an abort request occurs:

- The input transaction must complete in finite time.
- If the input transaction did not complete normally, SR.TRINPROG is set to 0b1 until the input transaction would have completed normally.

I_{ZPLF} If the input transaction did not complete normally, Arm recommends that the COM Port Peripheral returns an error to the requestor of the input transaction.

I_{VZPF} After an abort request, the COM Port Peripheral is in an UNKNOWN state and it is IMPLEMENTATION DEFINED which COM Port Peripheral registers are accessible. Arm recommends that:

- Reads of all registers operate as normal.
- Writes to DR and DBR while SR.TRINPROG is 0b1 return an error, otherwise writes operate as normal.

I_{LKMS} Typically, only writes to the DBR have the possibility of stalling input transactions for a variable amount of time until there is space in the TxEngine FIFO. This means that DP aborts normally only affect DBR writes.

4.3 CoreSight Management register index

Table 4-2 shows the register index for the CoreSight Management registers.

Table 4-2 CoreSight component register address offsets

Offset	Type	Name	Description	
0xF00	RW	ITCTRL	Integration Mode Control Register	See the *Arm® CoreSight™ Architecture Specification* for full implementation details.
0xF04-0xF9C	RES0	-	Reserved	
0xFA0	RW	CLAIMSET	Claim Tag Set Register	Claim Tag Registers.
0xFA4	RW	CLAIMCLR	Claim Tag Clear Register	
0xFA8	RO	DEVAFF0	Device Affinity Registers	See the *Arm® CoreSight™ Architecture Specification* for full implementation details.
0xFAC	RO	DEVAFF1	Device Affinity Registers	
0xFB4	RO	LSR	Software Lock Status Register	Software Lock Register.
0xFB8	RO	AUTHSTATUS	Authentication Status Register	
0xFBC	RO	DEVARCH	Device Architecture Register	
0xFC0	RO	DEVID2	Device Configuration Register 2	
0xFC4	RO	DEVID1	Device Configuration Register 1	
0xFC8	RO	DEVID	Device Configuration Register	
0xFCC	RO	DEVTYPE	Device Type Identifier Register	
0xFD0	RO	PIDR4	Component size (deprecated) and JEP106 identification	Peripheral Identification Registers. See the *Arm® CoreSight™ Architecture Specification* for full implementation details.
0xFD4	RO	PIDR5		
0xFD8	RO	PIDR6		
0xFDC	RO	PIDR7		
0xFE0	RO	PIDR0	Part number	
0xFE4	RO	PIDR1	JEP106 identification and Part number	
0xFE8	RO	PIDR2	Revision and JEP106 identification	
0xFEC	RO	PIDR3	RevAnd and Customer modified	
0xFF0	RO	CIDR0	Preamble	Component Identification Registers.
0xFF4	RO	CIDR1	Component class and Preamble	
0xFF8	RO	CIDR2	Preamble	
0xFFC	RO	CIDR3	Preamble	

4.4 CoreSight Management register descriptions

This section contains descriptions for the CoreSight Management registers. It contains the following subsections:

Note

Full details on CoreSight Management can be found in the *Arm® CoreSight™ Architecture Specification.*

4.4.1 CLAIMSET and CLAIMCLR, Claim Tag Set Register and Claim Tag Clear Register

The characteristics of CLAIMSET and CLAIMCLR are:

Purpose

Often there are several debug agents that must cooperate to control the resources that the CoreSight components make available. For example, an external debugger and a debug monitor running on the target might both require control of the breakpoint resources of a PE. It is important that a debug agent does not reprogram debug resources that another debug agent is using.

The Claim tag registers provide various bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent. All debug agents must implement a common protocol to use these bits.

The COM Port Peripheral programmers' model specifies that at least two claim tag bits must be implemented.

This specification does not define the claim tag protocol, but consider the following examples that illustrate how these bits can be used:

Protocol 1: Set common bit to claim

In this scenario, debug functionality is only claimed on a few rare, well-defined points, for example when the target is powered up or when a debugger is connected.

Each bit in the claim tag corresponds to an area of debug functionality, which is shared between all debug agents. For example, four bits can control four areas of functionality. The following shows a pseudocode implementation of this protocol:

```
read claim tag bit
if (bit is set)
    functionality is not available
else
    set bit
    use functionality
```

Protocol 2: Set private bit to claim

In this scenario, debug functionality is also only claimed on a few rare, well-defined points, but it is necessary to be able to determine which other agent has claimed functionality.

Each bit in the claim tag corresponds to an area of debug functionality for a debug agent. For example, four bits can control two areas of functionality each for two debug agents. The following shows a pseudocode implementation of this protocol:

```
read all claim tag bits for this functionality
if (any bits are set)
    functionality is not available
else
    set bit for this agent
    use functionality
```

Protocol 3: Set private bit and check for race

In this scenario, debug functionality is claimed regularly and it is possible for two debug agents to attempt to claim it simultaneously. In common with protocol 2, each bit in the claim tag corresponds to an area of debug functionality for a debug agent. The following shows a pseudocode implementation of this protocol:

```
read all claim tag bits for this functionality
if (any bits are set)
    functionality is not available
else
    set bit for this agent
    read all claim tag bits for this functionality
    if (any bits are set by other agents)
        clear bit for this agent
        wait a random amount of time
        go back to start
    else
        use functionality
```

Usage constraints The value of CLAIMCLR must be zero at reset.

CLAIMSET and CLAIMCLR are accessible as follows:

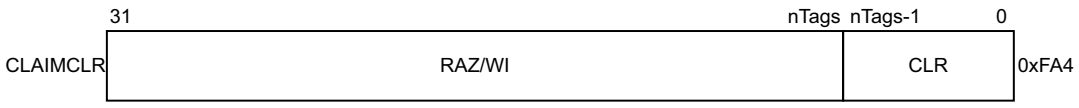
Default
RW

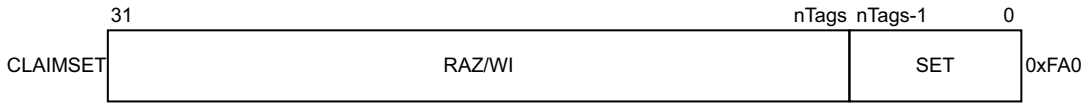
Configurations Included in all implementations. At least two claim tag bits are implemented.

Attributes CLAIMSET and CLAIMCLR are 32-bit registers.

Field Descriptions

The CLAIMSET and CLAIMCLR bit assignments are:





CLAIMCLR bits[31:nTags]

RAZ/WI

CLR, CLAIMCLR bits[nTags-1:0]

The size of this field, nTags, is IMPLEMENTATION DEFINED, and equals the number of bits set in CLAIMSET.

Permitted values of CLR[*n*] are:

Write `0` No effect.

Write `1` Clear the claim tag for bit[*n*].

Read `0` The claim tag bit is not set.

Read `1` The claim tag bit is set.

CLAIMSET bits[31:nTags]

RAZ/WI

SET, CLAIMSET bits[nTags-1:0]

The size of this field, nTags, is IMPLEMENTATION DEFINED, and equals the number of claim bits that are implemented.

Permitted values of SET[*n*] are:

Write `0` No effect.

Write `1` Set the claim tag for bit[*n*].

Read `0` The claim tag that is represented by bit[*n*] is not implemented.

Read `1` The claim tag that is represented by bit[*n*] is implemented.

Accessing CLAIMCLR and CLAIMSET

CLAIMCLR and CLAIMSET can be accessed at the following address:

Offset	
CLAIMCLR	**CLAIMSET**
`0xFA4`	`0xFA0`

4.4.2 AUTHSTATUS, Authentication Status Register

The AUTHSTATUS characteristics are:

Purpose Reports the required security level and status of the authentication interface. Where functionality changes on a given security level, the change in status must be reported in this register. For details about the authentication interface, see the *Arm® CoreSight™ Architecture Specification.*

Usage constraints Some components might not distinguish between Secure and Non-secure debug. For example, a trace component for a simple bus might connect to a Secure or a Non-secure bus, while its enable signals connect differently depending on which bus the component connects to. A failure to distinguish between Secure and Non-secure debug can result in:

- A component that indicates only Non-secure debug capabilities while performing only Secure debug functions.
- A component that indicates only Secure debug capabilities while performing only Non-secure debug functions.

Debuggers must be able to accommodate this possibility.

AUTHSTATUS is accessible as follows:

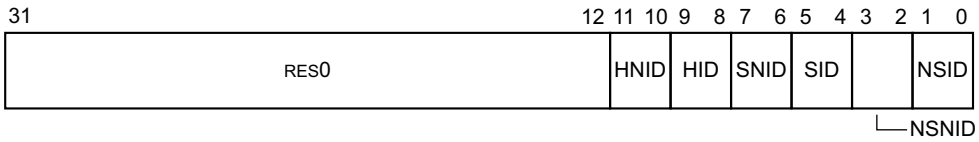
Default
RO

Configurations Included in all implementations.

Attributes AUTHSTATUS is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The AUTHSTATUS bit assignments are:



Bits[31:12] res0.

HNID, bits[11:10] Hypervisor non-invasive debug.

This field can have one of the following values:

`0b00` Separate controls for hypervisor non-invasive debug are not implemented, or no hypervisor is implemented.

All other values are reserved.

HID, bits[9:8] Hypervisor invasive debug.

This field can have one of the following values:

`0b00` Separate controls for hypervisor invasive debug are not implemented, or no hypervisor is implemented.

All other values are reserved.

SNID, bits[7:6] Secure noninvasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

All other values are reserved.

SID, bits[5:4] Secure invasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

All other values are reserved.

NSNID, bits[3:2] Non-secure noninvasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

All other values are reserved.

NSID, bits[1:0] Non-secure invasive debug.

This field can have one of the following values:

`0b00` Debug level is not supported.

All other values are reserved.

Accessing AUTHSTATUS

AUTHSTATUS can be accessed at the following address:

Offset
`0xFB8`

4.4.3 LSR, Software Lock Status Register

The characteristics of the Software lock registers are:

Purpose The Software lock mechanism prevents accidental access to the registers of CoreSight components. Software that is being debugged might accidentally write to memory used by CoreSight components. Accidental accesses might disable those components, rendering the software impossible to debug. The CoreSight programmers' model includes a Lock Status Register, LSR, and a Lock Access Register, LAR. These registers control software access to CoreSight components to minimize the likelihood of accidental access to CoreSight components.

Note

From CoreSight version 3.0 onwards, implementation of the Software lock mechanism that is controlled by LAR and LSR is deprecated.

To ensure that the software being debugged can never access an unlocked CoreSight component, a software monitor that accesses debug registers must unlock the component before accessing any registers, and lock the component again before exiting the monitor.

Arm recommends that external accesses from a debugger are not subject to the Software lock, and that external reads of the LSR return zero. For information on how CoreSight components can distinguish between external and internal accesses, see the *Arm® CoreSight™ Architecture Specification.*

A system can include several bus masters capable of accessing the same CoreSight component, for example in systems that include several PE's. In this case, it is possible for software running on one PE, PE A, to accidentally access the component while it is being programmed by a debug monitor running on another PE, PE B. Because the component that is being accessed cannot distinguish between the two PE's, PE A might disable the component and cause problems for PE B. The probability of this occurring is low, but must be considered if there are special circumstances that make this scenario more likely.

Note

The claim tag cannot be used to manage accesses to the Software lock registers, because access to the claim tag is subject to the Software lock mechanism.

Usage constraints LSR is accessible as follows:

Default	LSR
RO	

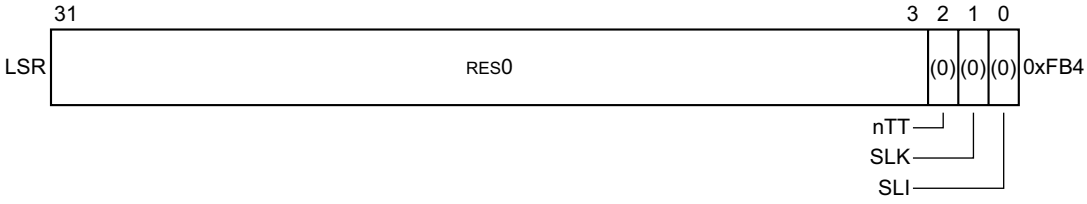
Configurations LSR is included in all implementations, and LSR.SLI indicates whether LAR is implemented.

LAR is not implemented in a COM Port Peripheral.

Attributes The Software lock registers are 32-bit registers.

Field Descriptions

The bit assignments of the Software lock registers are:



LSR, bits[31:3] RES0.

nTT, LSR bits[2] This bit is always zero, which indicates that the component implements a 32-bit LAR.

SLK, LSR bits[1] This field is used to return the current software lock status.

Permitted values of SLK are:

`0` Writing to the other registers in the component is permitted.

SLI, LSR bits[0] This field indicates whether a Software lock mechanism is implemented.

Permitted values of SLI are:

`0` Software lock mechanism is not implemented.

Accessing LSR

LSR can be accessed at the following address:

Offset	LSR
0xFB4	

4.4.4 DEVARCH, Device Architecture Register

The DEVARCH characteristics are:

Purpose Identifies the architect and architecture of a CoreSight component. The architect might differ from the designer of a component, for example when Arm defines the architecture but another company designs and implements the component.

Usage constraints DEVARCH is accessible as follows:

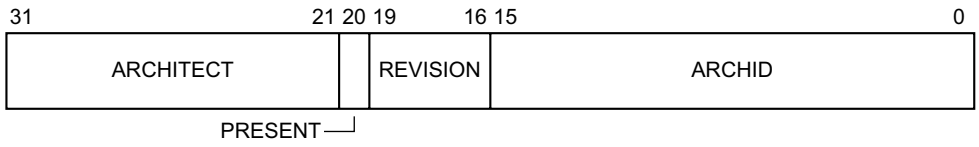
Default
RO

Configurations Included in all implementations.

Attributes DEVARCH is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVARCH bit assignments are:



ARCHITECT, bits[31:21]

Defines the architect of the component:

Bits[31:28] Indicates the JEP106 continuation code.

Bits[27:21] Indicates the JEP106 identification code.

See the *Standard Manufacturers Identification Code* for information about JEP106.

This field returns `0x23B` indicating Arm Ltd.

PRESENT, bit[20] Indicates the presence of this register:

1 = DEVARCH is present.

REVISION, bits[19:16]

Architecture revision.

This field returns `0x0`.

ARCHID, bits[15:0] Architecture ID.

This field returns `0x0A57`.

Accessing DEVARCH

DEVARCH can be accessed at the following address:

Offset
0xFBC

4.4.5 DEVTYPE, Device Type Identifier Register

The DEVTYPE characteristics are:

Purpose If the part number field is not recognized, a debugger reports the information that is provided by DEVTYPE about the component instead.

Usage constraints DEVTYPE is accessible as follows:

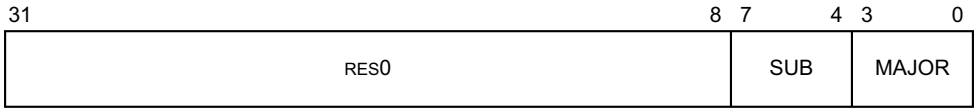
Default
RO

Configurations Included in all implementations.

Attributes DEVTYPE is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVTYPE bit assignments are:



Bits[31:8] RES0.

SUB, bits[7:4] Sub type for the component device type, as described in Table 4-3.

MAJOR, bits[3:0] Major type for the component device type, as described in Table 4-3.

Table 4-3 Device type encoding

MAJOR type [3:0]		SUB type [7:4]	
Value	**Description**	**Value**	**Description**
0x0	Miscellaneous.	0x0	Other, undefined.

Accessing DEVTYPE

DEVTYPE can be accessed at the following address:

Offset
0xFCC

4.4.6 DEVID, Device Configuration Register

The DEVID characteristics are:

Purpose Indicates the capabilities of the component.

Usage constraints This register is IMPLEMENTATION DEFINED for each part number and designer.

The entire 32-bit field can be used because the data width is determined by the component itself.

Unused bits must be RES0.

If the component is configurable, Arm recommends that this register reflects any changes to a standard configuration.

DEVID is accessible as follows:

Default
RO

Configurations Included in all implementations.

Attributes DEVID is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVID bit assignments are:

Bits[31:0] RES0.

Accessing DEVID

DEVID can be accessed at the following address:

Offset
0xFC8

4.4.7 DEVID1, Device Configuration Register 1

The DEVID1 characteristics are:

Purpose Indicates the capabilities of the component.

Usage constraints This register is IMPLEMENTATION DEFINED for each part number and designer.

The entire 32-bit field can be used because the data width is determined by the component itself.

Unused bits must be res0.

If the component is configurable, Arm recommends that this register reflects any changes to a standard configuration.

DEVID1 is accessible as follows:

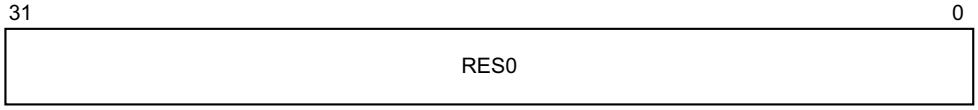
Default
RO

Configurations Included in all implementations.

Attributes DEVID1 is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVID1 bit assignments are:



Bits[31:0] RES0.

Accessing DEVID1

DEVID1 can be accessed at the following address:

Offset
0xFC4

4.4.8 DEVID2, Device Configuration Register 2

The DEVID2 characteristics are:

Purpose Indicates the capabilities of the component.

Usage constraints This register is IMPLEMENTATION DEFINED for each part number and designer.

The entire 32-bit field can be used because the data width is determined by the component itself.

Unused bits must be RES0.

If the component is configurable, Arm recommends that this register reflects any changes to a standard configuration.

DEVID2 is accessible as follows:

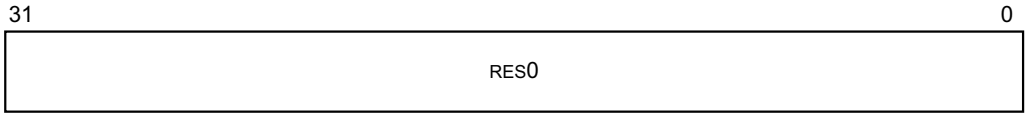
Default
RO

Configurations Included in all implementations.

Attributes DEVID2 is a 32-bit register that returns an IMPLEMENTATION DEFINED value.

Field Descriptions

The DEVID2 bit assignments are:



Bits[31:0] RES0.

Accessing DEVID2

DEVID2 can be accessed at the following address:

Offset
0xFC0

4.4.9 CIDR0-CIDR3, Component Identification Registers

The CIDR characteristics are:

Purpose Provide information that can be used to identify a CoreSight component.

Usage constraints CIDR0-CIDR3 are accessible as follows:

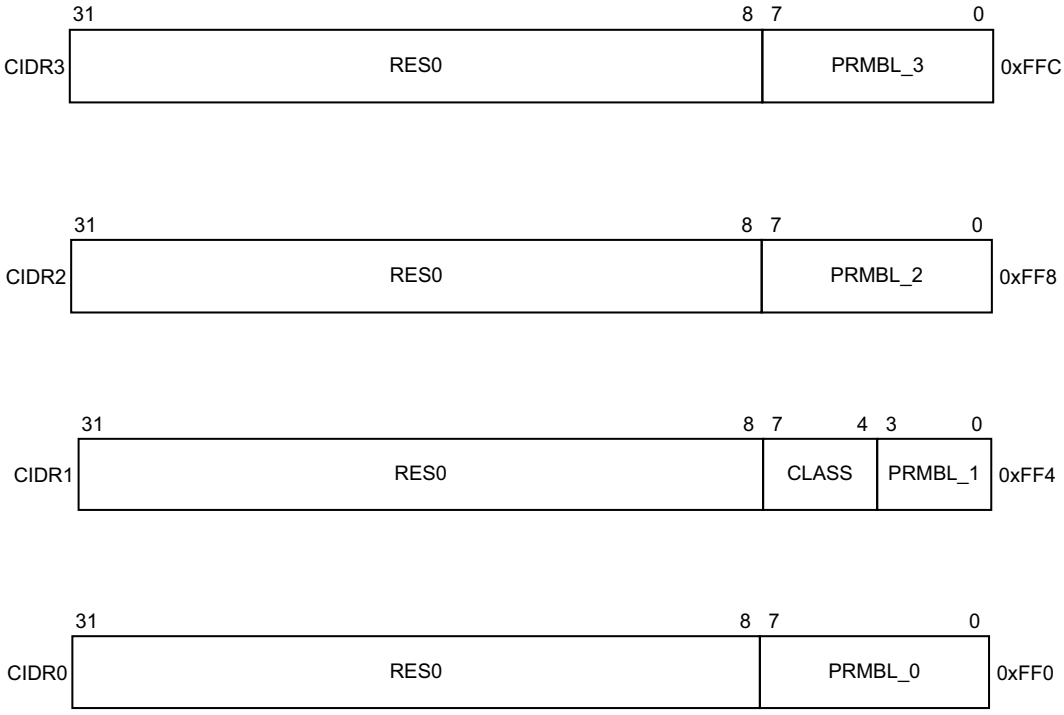
Default
RO

Configurations Included in all implementations.

Attributes CIDR0-CIDR3 are four 32-bit management registers.

Field Descriptions

The CIDR bit assignments are:



Bits[31:8] of CIDR3 RES0.

PRMBL_3, CIDR3 bits[7:0]

Preamble, segment 3. Must be 0xB1.

Bits[31:8] of CIDR2 RES0.

PRMBL_2, CIDR2 bits[7:0]

Preamble, segment 2. Must be 0x05.

Bits[31:8] of CIDR1 RES0.

CLASS, CIDR1 bits[7:4]

The component class, which can be one of the values that are listed in Table 4-4.

Table 4-4 CLASS field encodings

Value	Description
0x9	CoreSight component. See the *Arm® CoreSight™ Architecture Specification.*

PRMBL_1, CIDR1 bits[3:0]

Preamble, segment 1. Must be 0x0.

Bits[31:8] of CIDR0 RES0.

PRMBL_0, CIDR0 bits[7:0]

Preamble, segment 0. Must be 0x0D.

Accessing CIDR

CIDR0-CIDR3 can be accessed at the following addresses:

Offset			
CIDR0	CIDR1	CIDR2	CIDR3
0xFF0	0xFF4	0xFF8	0xFFC

Appendix A
Example Messages

This appendix lists some example messages. It contains the following section:

- *Examples* on page A-58.

A.1 Examples

This section contains example messages in the following subsections:

- *Simple Message*.
- *Simple Message with escaped byte 1*.
- *Simple Message with escaped byte 2*.
- *ID request* on page A-59.

A.1.1 Simple Message

A message consisting of the bytes `0x15`, `0xED` is transmitted as the sequence shown in Table A-1.

Table A-1 Simple Message

Byte	Notes
`0xAC`	START.
`0x15`	Message byte.
`0xED`	Message byte.
`0xAD`	END.

A.1.2 Simple Message with escaped byte 1

A message consisting of the bytes `0x15`, `0xAD` is transmitted in the sequence as shown in Table A-2.

Table A-2 Simple Message with escaped byte 1

Byte	Notes
`0xAC`	START.
`0x15`	Message byte.
`0xAE`	ESC.
`0x2D`	Message byte `0xAD` with MSB inverted.
`0xAD`	END.

A.1.3 Simple Message with escaped byte 2

A message consisting of the bytes `0x15`, `0xAE` is transmitted in the sequence as shown in Table A-3.

Table A-3 Simple Message with escaped byte 2

Byte	Notes
`0xAC`	START.
`0x15`	Message byte.
`0xAE`	ESC.
`0x2E`	Message byte `0xAE` with MSB inverted.
`0xAD`	END.

A.1.4 ID request

The Master Agent sends a single IDR byte.

The Slave Agent responds with an IDA byte plus six Message bytes 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, then an END byte as shown in Table A-4.

Table A-4 ID request

Byte	Notes
0xA1	IDA.
0x01	Message byte.
0x02	Message byte.
0x03	Message byte.
0x04	Message byte.
0x05	Message byte.
0x06	Message byte.
0xAD	END.

If a Message byte matches a Flag value, an ESC Flag byte is sent and then the Message byte with the MSB inverted. For example, for a Message byte sequence of 0x01, 0xA1, 0x03, 0xAE, 0x05, 0x06 as shown in Table A-5.

Table A-5 ID request 2

Byte	Notes
0xA1	IDA.
0x01	Message byte.
0xAE	ESC.
0x21	Message byte 0xA1 with MSB inverted.
0x03	Message byte.
0xAE	ESC.
0x2E	Message byte 0xAE with MSB inverted.
0x05	Message byte.
0x06	Message byte.
0xAD	END.

Appendix B
Pseudocode Definition

This appendix provides a definition of the pseudocode used in this document, and lists the *helper* procedures and functions used by pseudocode to perform useful architecture-specific jobs. It contains the following sections:

B.1 About Arm pseudocode

Arm pseudocode provides precise descriptions of some areas of the architecture. The following sections describe the Armv7 pseudocode in detail:

- *Data types* on page B-63.
- *Expressions* on page B-67.
- *Operators and built-in functions* on page B-69.
- *Statements and program structure* on page B-74.

B.1.1 General limitations of Arm pseudocode

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `SUBARCHITECTURE_DEFINED`, `UNDEFINED`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

- Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
- No subsequent behavior indicated by the pseudocode occurs. This means that these statements terminate pseudocode execution.

For more information, see *Simple statements* on page B-74.

B.2 Data types

This section describes:

- *General data type rules*.
- *Bitstrings*.
- *Integers* on page B-64.
- *Reals* on page B-64.
- *Booleans* on page B-64.
- *Enumerations* on page B-64.
- *Lists* on page B-65.
- *Arrays* on page B-66.

B.2.1 General data type rules

Arm architecture pseudocode is a strongly-typed language. Every constant and variable is of one of the following types:

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- List.
- Array.

The type of a constant is determined by its syntax. The type of a variable is normally determined by assignment to the variable, with the variable being implicitly declared to be of the same type as whatever is assigned to it. For example, the assignments `x = 1`, `y = '1'`, and `z = TRUE` implicitly declare the variables `x`, `y`, and `z` to have types integer, bitstring of length 1, and Boolean, respectively.

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. This is most often done in function definitions for the arguments and the result of the function.

The remaining subsections describe each data type in more detail.

B.2.2 Bitstrings

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 1.

The type name for bitstrings of length `N` is `bits(N)`. A synonym of `bits(1)` is `bit`.

Bitstring constants are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants of type `bit` are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

A special form of bitstring constant with `'x'` bits is permitted in bitstring comparisons, see *Equality and non-equality testing* on page B-69.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length N is bit (N–1) and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings derived from encoding diagrams, this order matches the way they are printed.

Bitstrings are the only concrete data type in pseudocode, in the sense that they correspond directly to the contents of, for example, registers, memory locations, and instructions. All of the remaining data types are abstract.

B.2.3 Integers

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as integers.

The type name for integers is `integer`.

Integer constants are normally written in decimal, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal, such as `0x55` or `0x80000000`. Hexadecimal integer constants are treated as positive unless they have a preceding minus sign. For example, `0x80000000` is the integer $+2^{31}$. If -2^{31} must be written in hexadecimal, it must be written as `-0x80000000`.

B.2.4 Reals

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as reals.

The type name for reals is `real`.

Real constants are written in decimal with a decimal point. This means `0` is an integer constant but `0.0` is a real constant.

B.2.5 Booleans

A Boolean is a logical true or false value.

The type name for Booleans is `boolean`. This is not the same type as `bit`, which is a length–1 bitstring. Boolean constants are `TRUE` and `FALSE`.

B.2.6 Enumerations

An enumeration is a defined set of symbolic constants, such as:

```
enumeration InstrSet {InstrSet_A32, InstrSet_T32, InstrSet_A64};
```

An enumeration always contains at least one symbolic constant, and a symbolic constant must not be shared between enumerations.

Enumerations must be declared explicitly, although a variable of an enumeration type can be declared implicitly by assigning one of the symbolic constants to it. By convention, each of the symbolic constants starts with the name of the enumeration followed by an underscore. The name of the enumeration is its *type name*, or *type*, and the symbolic constants are its possible *constants*.

Note

A Boolean is a pre-declared enumeration that does not follow the normal naming convention and it has a special role in some pseudocode constructs, such as `if` statements, for example:

```
enumeration boolean {FALSE, TRUE};
```

B.2.7 Lists

A list is an ordered set of other data items, separated by commas and enclosed in parentheses, for example:

```
(bits(32) shifter_result, bit shifter_carry_out)
```

A list always contains at least one data item.

Lists are often used as the return type for a function that returns multiple results. For example, this list at the start of this section is the return type of the function `Shift_C()` that performs a standard Arm shift or rotation, when its first operand is of type `bits(32)`.

Some specific pseudocode operators use lists surrounded by other forms of bracketing than the `(…)` parentheses. These are:

- Bitstring extraction operators, that use lists of bit numbers or ranges of bit numbers surrounded by angle brackets `<…>`.
- Array indexing, that uses lists of array indexes surrounded by square brackets `[…]`.
- Array-like function argument passing, that uses lists of function arguments surrounded by square brackets `[…]`.

Each combination of data types in a list is a separate type, with type name given by listing the data types. This means that the example list at the start of this section is of type `(bits(32), bit)`. The general principle that types can be declared by assignment extends to the types of the individual list items in a list. For example:

```
(shift_t, shift_n) = ('00', 0);
```

implicitly declares `shift_t`, `shift_n`, and `(shift_t, shift_n)` to be of types `bits(2)`, `integer`, and `(bits(2), integer)`, respectively.

A list type can also be explicitly named, with explicitly named elements in the list. For example:

```
type ShiftSpec is (bits(2) shift, integer amount);
```

After this definition and the declaration:

```
ShiftSpec abc;
```

the elements of the resulting list can then be referred to as `abc.shift`, and `abc.amount`. This qualified naming of list elements is only permitted for variables that have been explicitly declared, not for those that have been declared by assignment only.

Explicitly naming a type does not alter what type it is. For example, after the definition of `ShiftSpec`, `ShiftSpec`, and `(bits(2), integer)` are two different names for the same type, not the names of two different types. To avoid ambiguity in references to list elements, it is an error to declare a list variable multiple times using different names of its type or to qualify it with list element names not associated with the name by which it was declared.

An item in a list that is being assigned to can be written as `"-"` to indicate that the corresponding item of the assigned list value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

List constants are written as a list of constants of the appropriate types, for example the `('00', 0)` in the earlier example.

B.2.8 Arrays

Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then `..`, then the upper inclusive end of the range.

For example:

```
// The names of the Banked core registers.

enumeration RName {RName_0usr, RName_1usr, RName_2usr, RName_3usr, RName_4usr, RName_5usr,
                   RName_6usr, RName_7usr, RName_8usr, RName_8fiq, RName_9usr, RName_9fiq,
                   RName_10usr, RName_10fiq, RName_11usr, RName_11fiq, RName_12usr, RName_12fiq,
                   RName_SPusr, RName_SPfiq, RName_SPirq, RName_SPsvc,
                   RName_SPabt, RName_SPund, RName_SPmon, RName_SPhyp,
                   RName_LRusr, RName_LRfiq, RName_LRirq, RName_LRsvc,
                   RName_LRabt, RName_LRund, RName_LRmon,
                   RName_PC};


array bits(8) _Memory[0..0xFFFFFFFF];
```

Arrays are always explicitly declared, and there is no notation for a constant array. Arrays always contain at least one element, because:

- Enumerations always contain at least one symbolic constant.
- Integer ranges always contain at least one integer.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as `R[i]`, `MemU[address, size]` or `Elem[vector, i, size]`. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing.

B.3 Expressions

This section describes:

- *General expression syntax*.
- *Operators and functions - polymorphism and prototypes* on page B-68.
- *Precedence rules* on page B-68.

B.3.1 General expression syntax

An expression is one of the following:

- A constant.
- A variable, optionally preceded by a data type name to declare its type.
- The word `UNKNOWN` preceded by a data type name to declare its type.
- The result of applying a language-defined operator to other expressions.
- The result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register described in the text is to be regarded as declaring a correspondingly named bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as RAZ/WI, then the corresponding bit of its variable reads as 0 and ignore writes.

An expression like `bits(32) UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not constitute a security hole and must not be promoted as providing any useful information to software.

Note

Some earlier documentation describes this as an UNPREDICTABLE value. UNKNOWN values are similar to the definition of UNPREDICTABLE, but do not indicate that the entire architectural state becomes unspecified.

Only the following expressions are assignable. This means that these are the only expressions that can be placed on the left-hand side of an assignment.

- Variables.
- The results of applying some operators to other expressions.

  The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. For example, those circumstances might require that one or more of the expressions the operator operates is an assignable expression.
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

Every expression has a data type:

- For a constant, this data type is determined by the syntax of the constant.
- For a variable, there are the following possible sources for the data type:
  - An optional preceding data type name.
  - A data type the variable was given earlier in the pseudocode by recursive application of this rule.
  - A data type the variable is being given by assignment, either by direct assignment to the variable, or by assignment to a list of which the variable is a member.

  It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.
- For a language-defined operator, the definition of the operator determines the data type.

- For a function, the definition of the function determines the data type.

B.3.2 Operators and functions - polymorphism and prototypes

Operators and functions in pseudocode can be polymorphic, producing different functionality when applied to different data types. Each resulting form of an operator or function has a different prototype definition. For example, the operator + has forms that act on various combinations of integers, reals, and bitstrings.

One particularly common form of polymorphism is between bitstrings of different lengths. This is represented by using `bits(N)`, `bits(M)`, or similar, in the prototype definition.

B.3.3 Precedence rules

The precedence rules for expressions are:

1. Constants, variables, and function invocations are evaluated with higher priority than any operators using their results.

2. Expressions on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.

3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but this is not necessary if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j`, and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

B.4 Operators and built-in functions

This section describes:

- *Operations on generic types*.
- *Operations on Booleans*.
- *Bitstring manipulation*.
- *Arithmetic* on page B-72.

B.4.1 Operations on generic types

The following operations are defined for all types.

Equality and non-equality testing

Any two values `x` and `y` of the same type can be tested for equality by the expression `x == y` and for non-equality by the expression `x != y`. In both cases, the result is of type `boolean`.

A special form of comparison is defined with a bitstring constant that includes `'x'` bits in addition to `'0'` and `'1'` bits. The bits corresponding to the `'x'` bits are ignored in determining the result of the comparison. For example, if opcode is a 4-bit bitstring, `opcode == '1x0x'` is equivalent to `opcode<3> == '1' && opcode<1> == '0'`.

Note

This special form is permitted in the implied equality comparisons in `when` parts of `case … of …` structures.

Conditional selection

If `x` and `y` are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

B.4.2 Operations on Booleans

If `x` is a Boolean, then `!x` is its logical inverse.

If `x` and `y` are Booleans, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

If `x` and `y` are Booleans, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

If `x` and `y` are Booleans, then `x ^ y` is the result of exclusive-ORing them together.

B.4.3 Bitstring manipulation

The following bitstring manipulation functions are defined:

Bitstring length and most significant bit

If `x` is a bitstring:

- The bitstring length function `Len(x)` returns the length of `x` as an integer.
- `TopBit(x)` is the leftmost bit of x. Using bitstring extraction, this means:

  `TopBit(x)= x<Len(x)-1>`.

Bitstring concatenation and replication

If `x` and `y` are bitstrings of lengths `N` and `M` respectively, then `x:y` is the bitstring of length `N+M` constructed by concatenating `x` and `y` in left-to-right order.

If x is a bitstring and n is an integer with `n > 0`:

- `Replicate(x, n)` is the bitstring of length `n*Len(x)` consisting of n copies of x concatenated together
- `Zeros(n)` = `Replicate('0', n)`, `Ones(n)` = `Replicate('1', n)`.

Bitstring extraction

The bitstring extraction operator extracts a bitstring from either another bitstring or an integer. Its syntax is `x<integer_list>`, where x is the integer or bitstring being extracted from, and `<integer_list>` is a list of integers enclosed in angle brackets rather than the usual parentheses. The length of the resulting bitstring is equal to the number of integers in `<integer_list>`. In `x<integer_list>`, each of the integers in `<integer_list>` must be:

- `>= 0`
- `< Len(x)` if x is a bitstring.

The definition of `x<integer_list>` depends on whether `integer_list` contains more than one integer:

- If `integer_list` contains more than one integer, `x<i, j, k,…, n>` is defined to be the concatenation:

  `x<i> : x<j> : x<k> : … : x<n>`.

- If `integer_list` consists of one integer `i`, `x<i>` is defined to be:
  - If x is a bitstring, `'0'` if bit `i` of x is a zero and `'1'` if bit `i` of x is a one.
  - If x is an integer, let y be the unique integer in the range `0` to `2^(i+1)-1` that is congruent to x modulo `2^(i+1)`. Then `x<i>` is `'0'` if `y < 2^i` and `'1'` if `y >= 2^i`.

    Loosely, this definition treats an integer as equivalent to a sufficiently long two's complement representation of it as a bitstring.

In `<integer_list>`, the notation `i:j` with `i >= j` is shorthand for the integers in order from `i` down to `j`, with both end values included. For example, `instr<31:28>` is shorthand for `instr<31, 30, 29, 28>`.

The expression `x<integer_list>` is assignable provided x is an assignable bitstring and no integer appears more than once in `<integer_list>`. In particular, `x<i>` is assignable if x is an assignable bitstring and `0 <= i < Len(x)`.

Encoding diagrams for registers frequently show named bits or multi-bit fields.

Logical operations on bitstrings

If x is a bitstring, `NOT(x)` is the bitstring of the same length obtained by logically inverting every bit of x.

If x and y are bitstrings of the same length, `x AND y`, `x OR y`, and `x EOR y` are the bitstrings of that same length obtained by logically ANDing, ORing, and exclusive-ORing corresponding bits of x and y together.

Bitstring count

If x is a bitstring, `BitCount(x)` produces an integer result equal to the number of bits of x that are ones.

Testing a bitstring for being all zero or all ones

If x is a bitstring:

- `IsZero(x)` produces `TRUE` if all of the bits of x are zeros and `FALSE` if any of them are ones.
- `IsZeroBit(x)` produces `'1'` if all of the bits of x are zeros and `'0'` if any of them are ones.

`IsOnes(x)` and `IsOnesBit(x)` work in the corresponding ways. This means:

```
IsZero(x)    = (BitCount(x) == 0)

IsOnes(x)    = (BitCount(x) == Len(x))

IsZeroBit(x) = if IsZero(x) then '1' else '0'

IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

Lowest and highest set bits of a bitstring

If x is a bitstring, and `N = Len(x)`:

- `LowestSetBit(x)` is the minimum bit number of any of its bits that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of its bits that are ones. If all of its bits are zeros, `HighestSetBit(x) = -1`.
- `CountLeadingZeroBits(x)` is the number of zero bits at the left end of x, in the range 0 to N. This means:

  `CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)`.
- `CountLeadingSignBits(x)` is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to N–1. This means:

  `CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>)`.

Zero-extension and sign-extension of bitstrings

If x is a bitstring and i is an integer, then `ZeroExtend(x, i)` is x extended to a length of i bits, by adding sufficient zero bits to its left. That is, if `i == Len(x)`, then `ZeroExtend(x, i) = x`, and if `i > Len(x)`, then:

```
ZeroExtend(x, i) = Replicate('0', i-Len(x)) : x
```

If x is a bitstring and i is an integer, then `SignExtend(x, i)` is x extended to a length of i bits, by adding sufficient copies of its leftmost bit to its left. That is, if `i == Len(x)`, then `SignExtend(x, i) = x`, and if `i > Len(x)`, then:

```
SignExtend(x, i) = Replicate(TopBit(x), i-Len(x)) : x
```

It is a pseudocode error to use either `ZeroExtend(x, i)` or `SignExtend(x, i)` in a context where it is possible that `i < Len(x)`.

Converting bitstrings to integers

If x is a bitstring, `SInt(x)` is the integer whose two's complement representation is x:

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

`UInt(x)` is the integer whose unsigned representation is x:

```
// UInt()
```

```
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

`Int(x, unsigned)` returns either `SInt(x)` or `UInt(x)` depending on the value of its second argument:

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

B.4.4 Arithmetic

Most pseudocode arithmetic is performed on integer or real values, with operands being obtained by conversions from bitstrings and results converted back to bitstrings afterwards. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

Unary plus, minus, and absolute value

If `x` is an integer or real, then `+x` is `x` unchanged, `-x` is `x` with its sign reversed, and `Abs(x)` is the absolute value of `x`. All three are of the same type as `x`.

Addition and subtraction

If `x` and `y` are integers or reals, `x+y` and `x-y` are their sum and difference. Both are of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

Addition and subtraction are particularly common arithmetic operations in pseudocode, and so it is also convenient to have definitions of addition and subtraction acting directly on bitstring operands.

If `x` and `y` are bitstrings of the same length `N`, so that `N = Len(x) = Len(y)`, then `x+y` and `x-y` are the least significant `N` bits of the results of converting them to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>
x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

If `x` is a bitstring of length `N` and `y` is an integer, `x+y` and `x-y` are the bitstrings of length `N` defined by `x+y = x + y<N-1:0>` and `x-y = x - y<N-1:0>`. Similarly, if `x` is an integer and `y` is a bitstring of length `M`, `x+y` and `x-y` are the bitstrings of length `M` defined by `x+y = x<M-1:0> + y` and `x-y = x<M-1:0> - y`.

Comparisons

If `x` and `y` are integers or reals, then `x == y`, `x != y`, `x < y`, `x <= y`, `x > y`, and `x >= y` are equal, not equal, less than, less than or equal, greater than, and greater than or equal comparisons between them, producing Boolean results. In the case of `==` and `!=`, this extends the generic definition applying to any two values of the same type to also act between integers and reals.

Multiplication

If `x` and `y` are integers or reals, then `x * y` is the product of `x` and `y`. It is of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

Division and modulo

If x and y are integers or reals, then x/y is the result of dividing x by y, and is always of type real.

If x and y are integers, then x DIV y and x MOD y are defined by:

```
x DIV y = RoundDown(x/y)
x MOD y = x - y * (x DIV y)
```

It is a pseudocode error to use any of x/y, x MOD y, or x DIV y in any context where y can be zero.

Square root

If x is an integer or a real, Sqrt(x) is its square root, and is always of type real.

Rounding and aligning

If x is a real:

- RoundDown(x) produces the largest integer n such that n <= x.
- RoundUp(x) produces the smallest integer n such that n >= x.
- RoundTowardsZero(x) produces RoundDown(x) if x > 0.0, 0 if x == 0.0, and RoundUp(x) if x < 0.0.

If x and y are both of type integer, Align(x, y) = y * (x DIV y) is of type integer.

If x is of type bitstring and y is of type integer, Align(x, y) = (Align(UInt(x), y))<Len(x)-1:0> is a bitstring of the same length as x.

It is a pseudocode error to use either form of Align(x, y) in any context where y can be 0. In practice, Align(x, y) is only used with y a constant power of two, and the bitstring form used with y = 2^n has the effect of producing its argument with its n low-order bits forced to zero.

Scaling

If n is an integer, 2^n is the result of raising 2 to the power n and is of type real.

If x and n are of type integer, then:

- x << n = RoundDown(x * 2^n).
- x >> n = RoundDown(x * 2^(-n)).

Maximum and minimum

If x and y are integers or reals, then Max(x, y) and Min(x, y) are their maximum and minimum respectively. Both are of type integer if x and y are both of type integer, and real otherwise.

B.5 Statements and program structure

The following sections describe the control statements used in the pseudocode:

- *Simple statements*.
- *Compound statements* on page B-75.
- *Comments* on page B-79.

B.5.1 Simple statements

Each of the following simple statements must be terminated with a semicolon, as shown.

Assignments

An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

Procedure calls

A procedure call takes the form:

```
<procedure_name>(<arguments>);
```

Return statements

A procedure return takes the form:

```
return;
```

and a function return takes the form:

```
return <expression>;
```

where `<expression>` is of the type declared in the function prototype line.

UNDEFINED

This subsection describes the statement:

```
UNDEFINED;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that the Undefined Instruction exception is taken.

UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is UNPREDICTABLE.

SEE…

This subsection describes the statement:

```
SEE <reference>;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The `<reference>` indicates where that other pseudocode can be found.

It usually refers to another instruction but can also refer to another encoding or note of the same instruction.

IMPLEMENTATION_DEFINED

This subsection describes the statement:

```
IMPLEMENTATION_DEFINED {<text>};
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is IMPLEMENTATION DEFINED. An optional `<text>` field can give more information.

SUBARCHITECTURE_DEFINED

This subsection describes the statement:

```
SUBARCHITECTURE_DEFINED <text>;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is SUBARCHITECTURE DEFINED. An optional `<text>` field can give more information.

B.5.2 Compound statements

Indentation normally indicates the structure in compound statements. The statements contained in structures such as `if … then … else …` or procedure and function definitions are indented more deeply than the statement itself, and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level.

if … then … else …

A multi-line `if … then … else …` structure takes the form:

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    …
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    …
    <statement z>
else
    <statement A>
    <statement B>
    …
    <statement Z>
```

The block of lines consisting of `elsif` and its indented statements is optional, and multiple such blocks can be used.

The block of lines consisting of `else` and its indented statements is optional.

Abbreviated one-line forms can be used when there are only simple statements in the `then` part and in the `else` part, if it is present, such as:

```
if <boolean_expression> then <statement 1>

if <boolean_expression> then <statement 1> else <statement A>

if <boolean_expression> then <statement 1> <statement 2> else <statement A>
```

Note

In these forms, `<statement 1>`, `<statement 2>` and `<statement A>` must be terminated by semicolons. This and the fact that the `else` part is optional are differences from the `if … then … else …` expression.

repeat … until …

A `repeat … until …` structure takes the form:

```
repeat
    <statement 1>
    <statement 2>
    …
    <statement n>
until <boolean_expression>;
```

while … do

A `while … do` structure takes the form:

```
while <boolean_expression> do
    <statement 1>
    <statement 2>
    …
    <statement n>
```

for …

A `for …` structure takes the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>
    <statement 2>
    …
    <statement n>
```

case … of …

A `case … of …` structure takes the form:

```
case <expression> of
    when <constant values>
        <statement 1>
        <statement 2>
        …
        <statement n>
    … more "when" groups …
    otherwise
        <statement A>
        <statement B>
        …
        <statement Z>
```

In this structure, `<constant values>` consists of one or more constant values of the same type as `<expression>`, separated by commas. Abbreviated one line forms of `when` and `otherwise` parts can be used when they contain only simple statements.

If `<expression>` has a bitstring type, `<constant values>` can also include bitstring constants containing `'x'` bits. For details see *Equality and non-equality testing* on page B-69.

Procedure and function definitions

A procedure definition takes the form:

```
<procedure name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    …
    <statement n>
```

where `<argument prototypes>` consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

Note

This first prototype line is not terminated by a semicolon. This helps to distinguish it from a procedure call.

A function definition is similar but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    …
    <statement n>
```

An array-like function is similar but with square brackets:

```
<return type> <function name>[<argument prototypes>]
    <statement 1>
    <statement 2>
    …
    <statement n>
```

An array-like function also usually has an assignment prototype:

```
<function name>[<argument prototypes>] = <value prototypes>
    <statement 1>
    <statement 2>
    …
    <statement n>
```

B.5.3 Comments

Two styles of pseudocode comment exist:

- // starts a comment that is terminated by the end of the line.
- /* starts a comment that is terminated by */.

Glossary

This glossary describes some of the terms that are used in Arm documentation.

Abort
An abort occurs when an illegal memory access causes an exception. An abort can be generated by the hardware that manages memory accesses, or by the external memory system.

ADI
See Arm Debug Interface (ADI).

AHB
An AMBA bus protocol supporting pipelined operation, with the address and data phases occurring during different clock periods, meaning that the address phase of a transfer can occur during the data phase of the previous transfer. AHB provides a subset of the functionality of the AMBA AXI protocol.

See also AMBA.

Aligned
A data item stored at an address that is exactly divisible by the number of bytes that defines its data size. Aligned doublewords, words, and halfwords have addresses that are divisible by eight, four, and two respectively. An aligned access is one where the address of the access is aligned to the size of each element of the access.

AMBA
The AMBA family of protocol specifications is the Arm open standard for on-chip buses. AMBA provides solutions 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.

APB
An AMBA bus protocol for ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. It connects to the main system bus through a system-to-peripheral bus bridge that helps reduce system power consumption.

Arm Debug Interface (ADI)

The ADI connects a debugger to a device. The ADI is used to access memory-mapped components in a system, such as processors and CoreSight components. The ADI protocol defines the physical wire protocols permitted, and the logical programmers model.

AXI
An AMBA bus protocol that supports:

- Separate phases for address or control and data.

- Unaligned data transfers using byte strobes.
- Burst-based transactions with only start address issued.
- Separate read and write data channels.
- Issuing multiple outstanding addresses.
- Out-of-order transaction completion.
- Optional addition of register stages to meet timing or repropagation requirements.

The AXI protocol includes optional signaling extensions for low-power operation.

Big-endian In the context of the Arm architecture, big-endian is defined as the memory organization in which the least significant byte of a word is at a higher address than the most significant byte, for example:

- 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 and Endianness.

Boundary scan chain

A boundary 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. A core can contain several shift registers, enabling a scan to access selected parts of the device..

Burst A group of transfers that form a single transaction. With AMBA protocols, only the first transfer of the burst includes address information, and the transfer type determines the addresses used for subsequent transfers.

Cold reset A cold reset has the same effect as starting the processor by turning the power on. This clears main memory and many internal settings. Some program failures can lock up the core and require a cold reset to restart the system.

This is also known as power-on or powerup reset.

See also Processing Element (PE), Warm reset.

Core reset *See* Warm reset.

DAP *See* Debug Access Port (DAP).

Data Link layer The layer of an ADIv5 implementation that provides the functional and procedural means to transfer data between the external debugger and the *Debug Port* (DP). ADIv5 and upwards define two Data Link layers, one based on the IEEE 1149.1 Standard Test Access Port and Boundary Scan Architecture, referred to as JTAG, and one based on the Arm Serial Wire Debug protocol interface, referred to as SW-DP.

DATA LINK DEFINED

Means that the behavior is not defined by the base architecture, but must be defined and documented by individual Data Link layers of the architecture.

When DATA LINK DEFINED appears in body text, it is always in SMALL CAPITALS.

DBGTAP *See* Debug Test Access Port (DBGTAP).

Debug Access Port (DAP)

A block that acts as an AMBA, AHB, or AHB-Lite master on a system bus, to provide access to the debug target.

Debug Test Access Port (DBGTAP)

A debug control and data interface based on IEEE 1149.1 JTAG Test Access Port (TAP).

Debugger A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.

Doubleword A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

Doubleword-aligned

A data item having a memory address that is divisible by eight.

Embedded Trace Macrocell (ETM)

A hardware macrocell that, when connected to a core, outputs trace information on a trace port. The ETM provides core-driven trace through a trace port compliant to the ATB protocol. An ETM always supports instruction trace, and might support data trace.

Endianness The scheme that determines the order of the successive bytes of data in a larger data structure when that structure is stored in memory.

See also Little-endian and Big-endian.

ETM See *Embedded Trace Macrocell (ETM)*.

Halfword A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

Halfword-aligned

A data item having a memory address that is divisible by 2.

Host A computer that provides data and other services to another computer. In the context of an Arm debugger, a computer providing debugging services to a target being debugged.

IMP DEF *See* IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED

Behavior that is not defined by the architecture, but must be defined and documented by individual implementations.

When IMPLEMENTATION DEFINED appears in body text, it is always in SMALL CAPITALS.

Joint Test Action Group (JTAG)

An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a Joint Test Action Group (JTAG) interface port to communicate with processors.

See IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture specification available from the IEEE Standards Association.

JTAG *See* Joint Test Action Group (JTAG).

JTAG Access Port (JTAG-AP)

An optional component of the DAP that provides debugger access to on-chip scan chains.

JTAG Debug Port (JTAG-DP)

An optional external interface for the DAP that provides a standard JTAG interface for debug access.

JTAG-AP *See* JTAG Access Port (JTAG-AP).

JTAG-DP *See* JTAG Debug Port (JTAG-DP).

Little-endian In the context of the Arm architecture, little-endian is defined as the memory organization in which the most significant byte of a word is at a higher address than the least significant byte.

See also Big-endian and Endianness.

PE *See* Processing Element (PE).

Powerup reset *See* Cold reset.

Processing Element (PE)

The abstract machine defined in the Arm architecture, as documented in the Arm *Architecture Reference Manual*. A PE implementation that is compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm *Architecture Reference Manual*.

RAO *See* Read-As-One (RAO).

RAO/WI Read-as-One, Writes Ignored.

Hardware must implement the field as Read-as-One, and must ignore writes to the field. Software can rely on the field reading as all 1s, and on writes being ignored. This description can apply to a single bit that reads as `0b1`, or to a field that reads as all 1s.

See also Read-As-One (RAO).

RAZ *See* Read-As-Zero (RAZ).

RAZ/WI Read-as-Zero, Writes ignored.

Hardware must implement the field as Read-as-Zero, and must ignore writes to the field. Software can rely on the field reading as all 0s, and all writes being ignored. This description can apply to a single bit that reads as `0b0`, or to a field that reads as all 0s.

See also Read-As-Zero (RAZ).

Read-As-One (RAO)

Hardware must implement the field as reading as all 1s. Software can rely on the field reading as all 1s. This description can apply to a single bit that reads as `0b1`, or to a field that reads as all 1s.

Read-As-Zero (RAZ)

Hardware must implement the field as reading as all 0s. Software can rely on the field reading as all 0s. This description can apply to a single bit that reads as `0b0`, or to a field that reads as all 0s.

RES0 A reserved bit or field with Should-Be-Zero-or-Preserved (SBZP) behavior. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Note

RES0 is not used in descriptions of instruction encodings.

Within the architecture, there are some cases where a register bit or bitfield:

- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

This means the definition of RES0 for register fields is:

If a bit is RES0 in all contexts

It is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to `0b0`. In this case:
   - Reads of the bit always return `0b0`.
   - Writes to the bit are ignored.

   The bit might be described as RES0, WI, to distinguish it from a bit that behaves as described in 2.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to `0b0`.
   - A read of the bit returns the last value successfully written to the bit.

     Note

     As indicated in this list, this value might be written by an indirect write to the register.

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location associated with the bit.
- The value of the bit must have no effect on the operation of the core, other than determining the value read back from the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is implementation defined on a field-by-field basis.

If a bit is RES0 only in some contexts

When the bit is described as RES0:

- An indirect write to the register sets the bit to `0b0`.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an unknown value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the core, other than determining the value read back from that bit.

For any RES0 bit, software:

- Must not rely on the bit reading as `0b0`.
- Must use an *SBZP* policy to write to the bit.

The RES0 description can apply to bits or bitfields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as `0b0`, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as *SBZ*.

This RES0 description can apply to a single bit that should be written as its preserved value or as `0b0`, or to a field that should be written as its preserved value or as all 0s.

In body text, the term RES0 is shown in SMALL CAPITALS.

See also Read-As-Zero (RAZ), Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

RES1

A reserved bit or field with Should-Be-One-or-Preserved (SBOP) behavior. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Note

RES1 is not used in descriptions of instruction encodings.

Within the architecture, there are some cases where a register bit or bitfield:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

This means the definition of RES1 for register fields is:

If a bit is RES1 in all contexts

It is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to `0b1`. In this case:
   - Reads of the bit always return `0b1`.
   - Writes to the bit are ignored.

   The bit might be described as RES1, WI, to distinguish it from a bit that behaves as described in 2.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to `0b1`.
   - A read of the bit returns the last value successfully written to the bit.

     Note

     As indicated in this list, this value might be written by an indirect write to the register.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location associated with the bit.
   - The value of the bit must have no effect on the operation of the core, other than determining the value read back from the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is implementation defined on a field-by-field basis.

If a bit is RES1 only in some contexts

When the bit is described as RES1:

- An indirect write to the register sets the bit to `0b1`.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an unknown value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the core, other than determining the value read back from that bit.

For any RES1 bit, software:

- Must not rely on the bit reading as `0b1`.
- Must use an *SBOP* policy to write to the bit.

The RES1 description can apply to bits or bitfields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as `0b1`, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as *SBO*.

This RES1 description can apply to a single bit that should be written as its preserved value or as `0b0`, or to a field that should be written as its preserved value or as all 1s.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

Reserved Unless otherwise stated in the architecture or product documentation:

- Reserved instruction and 32-bit system control register encodings are unpredictable.
- Reserved 64-bit system control register encodings are undefined.
- Reserved register bit fields are UNK/SBZP.

SBO *See* Should-Be-One (SBO).

SBOP *See* Should-Be-One-or-Preserved (SBOP).

SBZ *See* Should-Be-Zero (SBZ).

SBZP *See* Should-Be-Zero-or-Preserved (SBZP).

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.

Serial Wire debug (SWD)
A debug implementation that uses a serial connection between the SoC and a debugger. This connection normally requires a bidirectional data signal and a separate clock signal, rather than the four to six signals required for a JTAG connection.

Serial-Wire Debug Port (SW-DP)

The interface for Serial Wire Debug.

Serial Wire JTAG Debug Port (SWJ-DP)

The SWJ-DP is a combined JTAG-DP and SW-DP that you can use to connect either a Serial Wire Debug (SWD) or JTAG probe to a target.

Should-Be-One (SBO)

Hardware must ignore writes to the field.

Software should write the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as `0b1`, or to a field that should be written as all 1s.

Should-Be-One-or-Preserved (SBOP)

The Armv7 Large Physical Address Extension modified the definition of SBOP to apply to register fields that are SBOP in some but not all contexts. From the introduction of Armv8 such register fields are described as RES1, see *RES1*. The definition of SBOP given here applies only to fields that are SBOP in all contexts.

Hardware must ignore writes to the field.

If software has read the field since the core implementing the field was last reset and initialized, it should preserve the value of the field by writing the value that it previously read from the field. Otherwise, it should write the field as all 1s.

If software writes a value to the field that is not a value previously read for the field and is not all 1s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as `0b1`, or to a field that should be written as its preserved value or as all 1s.

See also Should-Be-Zero-or-Preserved (SBZP), Should-Be-One (SBO).

Should-Be-Zero (SBZ)

Hardware must ignore writes to the field.

Software should write the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as `0b0`, or to a field that should be written as all 0s.

Should-Be-Zero-or-Preserved (SBZP)

The Armv7 Large Physical Address Extension modified the definition of SBZP to apply to register fields that are SBZP in some but not all contexts. From the introduction of Armv8 such register fields are described as RES0, see *RES0*. The definition of SBZP given here applies only to field that are SBZP in all contexts.

Hardware must ignore writes to the field.

If software has read the field since the core implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 0s.

If software writes a value to the field that is not a value previously read for the field and is not all 0s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as `0b0`, or to a field that should be written as its preserved value or as all 0s.

See also Should-Be-One-or-Preserved (SBOP), Should-Be-Zero (SBZ).

SWD *See* Serial Wire debug (SWD).

SW-DP *See* Serial-Wire Debug Port (SW-DP).

SWJ-DP *See* Serial Wire JTAG Debug Port (SWJ-DP)

TAP *See* Test Access Port (TAP).

Test Access Port (TAP)

The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. In the JTAG standard, the **nTRST** signal is optional, but this signal is mandatory in Arm processors because it is used to reset the debug logic.

See also Joint Test Action Group (JTAG), Debug Test Access Port (DBGTAP).

Trace port A port on a device, such as a processor or ASIC, to output trace information.

Unaligned An unaligned access is an access where the address of the access is not aligned to the size of the elements of the access.

See also Aligned.

UNKNOWN An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE and do not return UNKNOWN values.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

When UNKNOWN appears in body text, it is always in SMALL CAPITALS.

UNP *See* UNPREDICTABLE.

UNPREDICTABLE

For an Arm processor, UNPREDICTABLE means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect. An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

In an implementation that supports Virtualization, the Non-secure execution of unpredictable instructions at a lower level of privilege can be trapped to the hypervisor, provided that at least one instruction that is not unpredictable can be trapped to the hypervisor if executed at that lower level of privilege.

For an Arm trace macrocell, UNPREDICTABLE means that the behavior of the macrocell cannot be relied on. 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 trace macrocell can cause the core to enter Debug state, and external outputs can be used for other purposes.

Note

In issue A of this document, UNPREDICTABLE also meant an UNKNOWN value.

When UNPREDICTABLE appears in body text, it is always in SMALL CAPITALS.

W1C Hardware must implement the bit as follows:

- Writing a `0b1` to the bit clears the bit to `0b0`.
- Writing a `0b0` to the bit has no effect.

Warm reset Also known as a core reset. Initializes most of the processor functionality, excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

See also Cold reset.

WI Hardware must must ignore writes to the field. Software can rely on writes being ignored. This description can apply to a single bit, or to a field.

Word A 32-bit data item. Words are normally word-aligned in Arm systems.

Word-aligned A data item having a memory address that is divisible by four.