ARM Accredited Engineer

Version: 1.0

Study Guide

ARM Accredited Engineer
Study Guide

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
18 February 2013	A	Non-Confidential	

Proprietary Notice

Contents
ARM Accredited Engineer Study Guide

Preface

This preface introduces the *ARM Accredited Engineer Program Study Guide*. It contains the following sections:

- *About this book* on page v.
- *Feedback* on page vi.

About this book

The *ARM Accredited Engineer* (AAE) Program is a global certification program that allows software and hardware developers to become ARM Accredited Engineers. It provides reliable and comprehensive industry-wide qualifications that employers can use as quality benchmarks for hiring new staff and measuring the success of training and development programs. This book is the Study Guide for the first examination in the ARM Accredited Engineer Program.

Intended audience

This book is written for engineers studying for the first ARM Accredited Engineer Program examination.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this for an introduction to the AAE program.

Chapter 2 *Syllabus Information*

Read this for an introduction to the AAE syllabus.

Chapter 3 *Detailed Syllabus*

Read this for the details of the syllabus and additional information.

Typographic conventions

The following table describes the typographical conventions:

Typographical conventions

Style	Purpose
italic	Introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace`	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace` *`italic`*	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, `http://infocenter.arm.com`, for access to ARM documentation.

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

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

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

Feedback on content

If you have comments on content then send an e-mail to `errata@arm.com`. Give:

- The title.
- The number, AEG0060A.
- 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 the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1 **Introduction**

The ARM Accredited Engineer program provides a series of examinations which may be taken by engineers and developers all over the world, in any company, giving them the opportunity to become ARM Accredited Engineers.

The program defines a range of different exams, covering various different subject areas and difficulty levels. This document is related to the first of those exams – the *ARM Accredited Engineer* (AAE) exam – which is focused on software aspects of processors supporting architectures ARMv7-A and ARMv7-R.

This document is intended to explain how students may learn the factual information associated with the examination using the *Cortex-A Series Programmer's Guide* (ARM DEN 0013C), with reference to the full details of the syllabus for the entry-level exam, available from the ARM website. It is not intended to be read as a standalone text, but used in conjunction with the syllabus and Cortex-A Series Programmer's Guide. The document examines each syllabus area and provides details of where the necessary information for that syllabus area may be found. ARM expects that students would already be familiar with the fundamentals of C and Assembly Language programming, embedded systems and microprocessors.

This text should be viewed as a supplement to a course of practical experience, developing software on a real ARM-based platform. The exam assumes some familiarity with software development tools from ARM. The exam relates primarily to architectures ARMv7-A and ARMv7-R together with the processors available from ARM which conform to these architectures. Earlier versions of the architecture and earlier processors (from ARM7TDMI onwards) also form part of the syllabus.

Chapter 2
Syllabus Information

Candidates for the exam should become familiar with the AAE Entry Level Syllabus document, available from the ARM website (Document Number AEG0052C.)

The detailed syllabus provides a list of subject areas, along with a colored key indicating a difficulty level. Candidates should have a detailed knowledge of subject areas coded green and a good awareness of those colored yellow.

In this guide, we will step through the published syllabus and point out which sections of the *Cortex A-Series Programmer's Guide* cover the required knowledge and provide additional background where the *Cortex A-Series Programmer's Guide* does not fulfill all of the examination requirements. Note that this syllabus ordering does not provide a sensible suggestion for the order in which a beginner should study new topics. For example, it would be prudent to become familiar with ARM instructions, registers and modes, before learning about the *Generic Interrupt Controller* (GIC), or coherency hardware.

Chapter 3
Detailed Syllabus

In this section you can see the detailed requirements under each subject area. The syllabus text is in Italic. This is followed by non-italic text that gives a reference to where you can find the information, or additional information, where necessary.

3.1 Implementation

This section covers implementation.

3.1.1 ARM Processor Families

1. **Which are multiprocessor?**

   Candidates should be familiar with the available processors from ARM and know which of these may be used in multiprocessor configurations.

   Section 2.4 of the Cortex-A Series Programmer's Guide describes some commonly encountered ARM processors. Section 2.5 of the Cortex-A Series Programmer's Guide describes the processors in the Cortex-A series in more detail. Particular attention should be paid to Table 2-3.

2. **Coherency hardware**

   Candidates should be aware of the concept of coherency hardware, what it does, why it is required and what are the benefits of using it.

   Although this topic is mentioned at the start of the syllabus, it is a relatively advanced one. Coherency hardware is required in a multiprocessor system. Understanding how it works and its impact on software development requires a good understanding of cache and multiprocessor behavior. These areas are covered in Chapter 9 (Caches) and Chapter 21 (Introduction to Multi-processing) respectively. The coherency hardware is described in Section 22.1.

3. **Which include a GIC**

   Candidates should be aware of the concept of coherency hardware, what it does, why it is required and what are the benefits of using it.

   The *Generic Interrupt Controller* (GIC) is described in Section 13.2 of the Cortex-A Programmer's Guide. Students should also refer to Section 22.3 (Interrupt handling in a multi-core system). The GIC is found in the Cortex-A5MPCore, Cortex-A9MPCore and Cortex-R7MPCore processors. It is optionally present in the Cortex-A7 and Cortex-A15 based designs.

4. **Architecture Conformance**

   Candidates must be able to identify which ARM processors conform to which Architecture versions. Architectures from ARMv4T onwards are included, with the exception of ARMv7-M and ARMv6-M architectures.

   Table 2-2 provides this information.

5. **Performance Point**

   Candidates must be able to distinguish between applications, real-time and microcontroller profiles, be aware of the distinguishing features between them and understand the typical target market/applications of each.

   The Cortex-A Series Programmer's Guide, as the name suggests, does not provide detailed coverage of the Cortex-R or Cortex-M series. Cortex-A series processors are typically targeted at systems running platform operating systems where a memory management unit, caches and high performance are necessary.

   Cortex-R series processors target real-time applications. This typically means more deeply embedded systems with demanding real-time response requirements. This means that features such as fast, deterministic interrupt response, *Tightly-Coupled Memories* (TCM) local to the processor for fast-responding code and data and Parity detection or

Error-Correcting Code (ECC) for error detection/correction will be required. The Cortex-R Series processors will use a *Memory Protection Unit* (MPU) rather than the *Memory Management Unit* (MMU) found in the Cortex-A Series processors.

Cortex-M Series processors are optimized for cost and power sensitive MCU and mixed-signal devices. Examples of likely end applications would include smart metering, human interface devices, automotive and industrial control systems, white goods, consumer products and medical instrumentation. Key factors in this area would be smaller code (better code density), ease of use and energy efficiency. For example, Cortex-M0+ is the most power efficient ARM processor, measured by mW/MHz. Cortex-M Series processors will only support the 16-bit and 32-bit Thumb instruction set and do not usually feature caches.

3.1.2 Instruction Cycle Timing

1. **Instruction Cycle Timing**

   Candidates should be aware of the effects of pipeline and memory system on instruction execution timing.

   Pipelines are briefly discussed in Section 4.4 of the Cortex-A Series Programmer's Guide and branch prediction is described in Section 4.5. The coverage is relatively brief and assumes some prior knowledge of pipelined architectures.

   It may be useful to become familiar with the three-stage (fetch, decode, execute) pipeline of ARM7TDMI and to understand how this can effect cycle timings. The ARM7TDMI Technical Reference Manual shows this pipeline. It is easier to understand the concepts when they are applied to this pipeline.

   For example, taken branches incur a two cycle branch penalty on ARM7TDMI, because the branch is taken at the execute stage of the pipeline and the instructions already in the pipeline at the fetch and decode stages must be flushed and the pipeline refilled, so that there is a delay of three cycles until the instruction after a taken branch is executed.

   Similarly, once the operation of a simple pipeline is understood, the concept of a 'load-use penalty" becomes straightforward. This is the situation where a register is loaded with a value from a memory address. If a subsequent instruction uses the new value of that register, it must clearly wait until the value has been loaded from memory (and reached the relevant pipeline stage in the processor, possibly through dedicated forwarding paths). A compiler (or Assembly language programmer), may try to avoid this penalty, by attempting to move the instruction which uses the loaded value away from the instruction which performs the load.

3.2 Software Debug

This section covers software debug.

3.2.1 Standard Debug Facilities

1. **Difference between Hardware and Software breakpoints**

   Candidates should be aware of the differences and limitations between Hardware and Software breakpoints. They should understand the usage of watchpoint units in configuring these two types of breakpoint.

   This is described in Section 29.1 of the Cortex-A Series Programmer's Guide. On older processors (for example, ARM7TDMI), a dedicated `BKPT` (Breakpoint) instruction was not implemented. Instead, the watchpoint units were programmed to look for a specific bit pattern. The debugger enabled unlimited numbers of software breakpoints to be set on code in RAM. The debugger would read the opcode of the instruction to be replaced by a breakpoint and store this value on the host. The instruction is then over-written with a defined bit pattern (which corresponds to an invalid instruction). Typically, this might require the data cache of the processor to be cleaned and its instruction cache to be invalidated, to ensure that the store from the data side of the processor is made visible to the instruction fetch logic. Later processors, including all Cortex series processors, replace the original opcode with a `BKPT` instruction. When the breakpoint is removed, the original opcode value is written back. By contrast, for hardware breakpoints, which can be set on any type of memory, the number is limited by the available hardware units.

   When studying for the exam, it is recommended that you gain some practical experience with a real debugger and understand its use and limitations. This will provide more useful background than reading a book description.

2. **Monitor Mode vs Halt Mode Debug**

   Candidates should be aware of the differences between debugging in each of the two modes and when one would be chosen over the other e.g. choosing monitor mode if interrupts must continue to be serviced while debugging.

   This is described in Section 29.1.1 of the Cortex-A Series Programmer's Guide.

3. **Vector Catching**

   Candidates should be aware of the function of this feature, why and when it is typically used.

   Vector catch is a means for a debugger to trap processor exceptions. This feature is typically used in the early stages of development to trap processor exceptions before the appropriate handlers are installed. Many ARM processors have dedicated vector catch hardware for this purpose. On some processors, such as ARM7TDMI, the debugger can perform vector catching by setting breakpoints at the appropriate locations in the exception vector table.

4. **Determining Exception Cause (e.g. DFSR, SPSR, Link Register)**

   Candidates should understand how to determine the actual cause of an exception by using additional resources, which might include special registers that contain fault status. For example, using the DFSR, DFAR to determine cause of a Data Abort or using the SPSR and LR to locate and retrieve SVC comment field.

   When debugging a piece of software, it is commonly necessary to understand why a particular processor exception has occurred. The processor provides a number of registers which give useful information to assist with this. The Link Register (LR) of the current exception mode will point to a location close to the instruction in the main program at which the exception was taken. It will normally be necessary to subtract 4 or 8 from the

LR (see Section 12.2 of the Cortex-A Series Programmer's Guide) to do this. Similarly, the mode bits of the current *Saved Program Status Register* (SPSR) will give information about which mode the processor was operating in prior to the exception being taken.

For certain modes, additional information is available. After a prefetch or data abort exception, it may be useful to look at certain CP15 registers, the *Fault Status Register* (FSR) and *Fault Address Register* (FAR). These can tell you the cause of the abort (for example, an external memory error, or an invalid translation table entry for this address) and the address of the memory access that generated the abort.

Note that the exception model of the ARMv6-M and ARMv7-M architecture is somewhat different to those of other ARM cores, but knowledge of this is not required for the basic examination.

5. **Trace**

Candidates should be aware of what Trace is, what on-chip and off-chip components are required to make it work, what it is used for. Also that it is non-intrusive.

This is described in Section 29.2 of the Cortex-A Series Programmer's Guide.

6. **Cross Triggering**

The candidate is not expected to demonstrate knowledge of this subject.

7. **Physical Debug Connection**

Candidates should be aware of available options for making physical connections to the target for the purpose of run-control debug (e.g. JTAG, SWD)

These are described in Chapter 29 of the Cortex-A Series Programmer's Guide. JTAG is a very widely used industry standard, which typically requires 5 device pins to physically connect a debugger running on a host computer to the ARM target to be debugged. *Serial Wire Debug* (SWD) is a 2-pin solution with similar capabilities and is likely to be found only on newer ARM target devices. Capturing trace information from the processor requires a much larger bandwidth than simply controlling execution, as multiple 32-bit address and data values may be generated on each cycle. Therefore, dedicated trace ports requiring larger numbers of pins may be required. Alternatively, the trace may be stored in an on-chip buffer and read more slowly by the debugger using a smaller number of pins.

8. **Debugger Access to Memory**

Candidates should be aware of how a debugger uses the processor to access memory and be aware of how memory translation affects this. They should also understand the potential impact of cache maintenance caused by debug operations on the state of the memory system.

The debugger will sometimes need to display the contents of memory (or peripherals) or modify them. It generally does this by having the processor perform a load or store instruction, although in some systems it is possible for a built-in debug system component to write (or read) directly to memory, without having the processor itself perform the memory operation.

A good debugger will try to minimize the change to the system caused by debug activity. For example, it will typically attempt to minimize changes to the cache while debugging. For example, if the programmer displays an area of memory in a debugger window and the debug tool must use the processor to read that memory, it will try to do so in a non-cacheable fashion so that the prior cache contents are not over-written. However, the programmer must be aware that (when the MMU is enabled) virtual addresses are displayed rather than physical addresses and that the cache contents may differ from values held in external memory.

If the memory access is performed directly by a debug component on the SoC (i.e. not done by the processor), then the access will use physical addresses rather than the processor's virtual address translation and it will directly access the memory (bypassing the processor cache).

9. **Semi Hosting**

   Candidates should be aware of semi-hosting, understand what its function is and how it can be used during development. They should also be aware of the invasive nature of using this feature and how it may affect timing etc. They should also be aware of the danger of using semi-hosting when no debugger is connected and how to retarget the standard C library.

   Semihosting is a mechanism which allows code running on an ARM target to use the facilities provided on a host computer which is running a debugger.

   Examples can include keyboard input, screen output, and disk I/O. For example, you can use this mechanism to allow C library functions, such as `printf()` and `scanf()`, to use the screen and keyboard of the host. Development hardware often does not have a full range of input and output facilities, but semihosting enables the host computer to provide these facilities.

   Semihosting is implemented by a set of defined software instructions that generate an exception. The application invokes the appropriate semihosting call and the debug agent then handles the exception. The debug agent provides the required communication with the host.

   The semihosting interface is common across all debug agents provided by ARM. Tools from ARM use `SVC 0x123456` (ARM state) or `SVC 0xAB` (Thumb) to represent semi-hosting debug functions.

   Of course, outside of the development environment, a debugger running on a host is not connected to the system. It is therefore necessary for the developer to re-target any C library functions which use semi-hosting, for example, `fputc()`. This would mean replacing the library code which used an `SVC` call with code which could output a character on this specific board.

3.2.2 Standard Debug Techniques

1. **Call Stack**

   Candidates must understand what a call stack is and how it may be used in debugging.

   Application code will use the stack to pass parameters, store local data and store return addresses. The data each function pushes on the stack is organized into a "stack frame". When a debugger stops a processor, it may be able to analyze the data on the stack to provide the user with a "Call stack" - that is, a list of function calls leading up to the current situation. This can be extremely useful when debugging, as it enables the user to deduce why the application has got into a particular state.

   In order to be able to reconstruct the call stack, the debugger must be able to determine which entries on the stack contain return address information. This information may be contained in "debugger information" (DWARF debug tables) if the code was built with these included, or by following a chain of "frame pointers" pushed on the stack by the application. Again, the code must be built to use frame pointers. If neither of these types of information are present, the call stack can not be constructed.

   In multi-threaded applications, each thread has its own stack. The call stack information will therefore only relate to the particular thread being examined.

2. **Single Step**

Candidates must understand what single stepping is and how it may be used in debugging. They should understand the difference between Step-In and Step-Over when single-stepping.

Single step simply refers to the possibility for the debugger to move through a piece of code, one instruction at a time. The difference between "Step-In" and "Step-Over" can be explained with reference to a function call. If you "Step-Over" the function call, the entire function is executed as one step, allowing you to continue after a function which you don't wish to step through. "Step-in" would mean that you instead single step through the function itself.

3. **Start/Stop**

   Candidates must understand what start/stop debugging is and how it may be used.

   This simply means that when you click "start" in your debugger, the processor is caused to exit from debug state and resume normal execution from the current PC location, until it encounters something which causes it to "stop". This could be a breakpoint, watchpoint or vector catch event or might be a debug request from an external debugger or other block in the system.

   Such start/stop debug may be contrasted with systems which permit debug without ever stopping code execution. In some embedded systems (e.g. automotive engine management systems), it is not possible for the processor to simply stop execution while debug activity is performed.

4. **Printf**

   Candidates must understand how printf may be used in code instrumentation (they may also be expected to understand what other kinds of instrumentation might typically be used).

   This refers to a simple debug technique, common to all processor architectures of inserting instructions into code to output statements which may indicate program flow, or the value of key variables at certain times, for example.

5. **Bare metal vs Application**

   Candidates should be aware of the difference in debug options between applications running on OS-based or bare metal system (e.g. Ethernet, GDB server, ICE).

   Example: Candidates should know that it is necessary to run a server program on the target system in order to debug an OS-based application.

   Section 29.4 of the Cortex-A Series Programmer's Guide provides an introduction to this topic.

6. **RAM/ROM Debug**

   Candidates should be aware of the limitations of debugging code running in different types of memory (e.g. breakpoints and image load).

   Example: Candidates should know that it is not possible to set an unlimited number of breakpoints when executing code in ROM. More advanced candidates would understand why this is the case.

   This is described earlier in this document (software breakpoints may only be set in RAM, as they require writeable memory.) One other common limitation when debugging ROM code is (of course) that it is less easy to change the code to try to understand or fix wrong behavior.

7. **Debugging Timing Related Problems**

   Candidates should be aware that certain debug approaches may affect the timing of the code (e.g. halt mode) and be aware of non-invasive alternatives (e.g. trace).

This is covered previously in this text.

8. **Effects of using a software debugger (observing the system may change its state)**

   Candidates should be aware of the implications/impact of debugging (e.g. reading memory locations).

   This is covered previously in this text.

3.3 Architectural

This section covers the instruction set architecture.

3.3.1 Instruction Set

1. **LDREX/STREXCLEX**

   *Candidates should be aware of how these instructions work and how they might be used to implement software synchronization (e.g. mutex). At this level candidates should be able to recognize a simple mutex implementation (in assembler) and be able to explain its operation. They will not be expected to write on*e.

   Example: Candidates should be aware of what a mutex is. More advanced candidates will be aware of the exclusive access instructions which are used to implement mutexes and similar constructs.

   Exclusive accesses and mutexes are described in Sections 22.4 and 23.4 of the Cortex-A Series Programmer's Guide. Candidates should take time to become familiar with the code in Example 22-1 and to understand its operation.

2. **DMB/DSB/ISB**

   Candidates must understand the features of a weakly ordered memory system and recognize when manual ordering is required. Candidates must also know how to achieve required access ordering using the ARM barrier instructions. They should be able to insert suitable barrier instructions in simple code sequences where particular ordering is required.

   Examples: Candidates should know that ARM systems implement a weakly-ordered memory model and that this means barriers are required to enforce ordering in some circumstances. More advanced candidates will know what these instructions are and how to use them in simple code sequences.

   The use of barriers and the ARM memory ordering model is described in detail in Chapter 11 of the Cortex-A Series Programmer's Guide. Candidates should pay particular attention to understanding the difference between `DSB` and `DMB` and to the examples presented in Section 11.2. If possible, candidates should attempt to find a practical example in which a barrier instruction might be required in their system. For example, if an interrupt source is cleared by writing to a memory mapped peripheral, it may be necessary to use a barrier instruction to force the completion of that write before a subsequent instruction re-enables interrupts. Without the barrier, a spurious interrupt could occur.

3. **VFP**

   The candidate is not expected to demonstrate knowledge of this subject.

4. **NEON**

   The candidate is not expected to demonstrate knowledge of the NEON instruction set.

5. **Exception Entry/Exit**

   Candidates must be able to describe the exception entry sequence of an ARM processor. They must understand and describe the state of the processor on entry to an exception (e.g. PC, LR, CPSR, SPSR etc). Candidates must also be familiar with the standard exception return instructions (not including ERET) and return address adjustments for each exception type.

   Example: Candidates should know what operations the core undertakes automatically on processing an exception. More advanced candidates will know the standard exception return instructions for each exception type.

This is covered in Chapters 12-14 of the Cortex-A Series Programmer's Guide.

6. **PSR modifying instructions**

   Candidates must know the instructions that allow manual control of the PSRs, which bits may and may not be modified in unprivileged mode.

   Example: Candidates should know the instructions which may be used to modify the PSR. More advanced candidates will know how to use these instructions and what their limitations are.

   The PSRs are described in Section 4.3.1 of the Cortex-A Series Programmer's Guide. The instructions which allow manual control of PSR bits are described in Appendix A (`CPS`, `MRS`/`MSR` and `SETEND`). In user (unprivileged) mode, the programmer cannot manipulate the PSR bits [4:0] which control the processor mode or the A, I and F bits which govern which exceptions are enabled or disabled.

7. **WFI, WFE, SEV**

   Candidates should be aware of the ARM hint instructions for entering low power states, what a typical implementation does on executing such an instruction and when they might be used. Candidates should also have knowledge of wakeup conditions from each low power state.

   Example: Candidates must be aware of the WFI instruction and that it may not actually do anything. More advanced candidates will be aware also of WFE and of the conditions which cause wakeup from standby mode.

   This is covered in the Cortex-A Series Programmer's Guide Chapter 25 (and see also Section 11.2.3).

8. **General Instruction Set**

   Candidates must have a good knowledge of the syntax and semantics of ARM and Thumb data processing, branch and load/store instructions. They should be able to recognise and interpret such instructions, and be able to match fragments of C code with their corresponding fragments of assembler code.

   Example: Candidates will be familiar with the core instruction set, including operand order, basic syntax and addressing modes. More advanced candidates will be able to map instructions onto C code, including knowledge of literal pools, loop structures, register spilling and simple compiler transformations.

   Chapters 5 and 6 of the Cortex-A Series Programmer's Guide provide an introduction to ARM Assembly Language, while Appendix A gives the description and syntax of all instructions. Candidates might also refer to the *ARM Compiler Toolchain Using the Assembler* document which is part of the DS-5 compilation tools product. This document is available from the ARM website. There are numerous ARM assembly language tutorials available from the internet or published books.

   However, this is an area where it is strongly recommended that candidates gain practical experience by writing simple assembly language code and by observing the compiler output for simple C code.

   Regarding the specific topics mentioned in the syllabus, the concept of a literal pool is first mentioned in the Cortex-A Series Programmer's Guide, Section 6.1.1, covering constant data. Register spilling is mentioned in Section 17.1. Section 19.3 provides some simple examples of loop structures and compiler transformations.

3.3.2 Power Modes

1. **Standby, Dormant, Shutdown**

   Candidates must know the architectural power modes, and understand the progression between these power modes. Candidates are not expected to know the power domains or power-saving features of specific processors or implementations.

   Example: Candidates will know that there are different sleep modes and be able to rank them in order of power saving. More advanced candidates will be aware of the progression between these modes.

   See Chapter 25 of the Cortex-A Series Programmer's Guide.

2. **Entry/Exit**

   Candidates should be aware of the software requirements for entering and exiting power saving modes (e.g. they should know what state must be preserved and be aware of how this is done).

   Example: Candidates will k now about the WFI instruction and when it might be used. More advanced candidates will know what state must be preserved on entry to power saving modes.

   See Chapter 25 of the Cortex-A Series Programmer's Guide.

3.3.3 Memory model

1. **Access ordering**

   Candidates must understand the characteristics of a weakly ordered memory system as used by ARM processors. They should be able to recognize examples of access patterns which could and could not be re-ordered by a processor.

   Example: Candidates will know that ARM supports a weakly-ordered memory model and what this means when writing software. More advanced candidates will be able to recognize and analyze access patterns from fragments of code.

   See Chapter 11 of the Cortex-A Series Programmer's Guide.

2. **Memory Types (i.e. Normal, Device, Strongly Ordered**

   Candidates must know the three memory types defined by the ARM architecture. They should understand the characteristics of each, what they are used for and the ordering requirements which result.

   Example: Candidates must know the three memory types defined by the ARM architecture and that peripherals should be classed as Device memory. More advanced candidates will know the characteristics and typical use of each memory type.

   See Chapter 11 of the Cortex-A Series Programmer's Guide.

3. **VMSA**

   a. **Translation Table Format**

      Candidates are not expected to demonstrate knowledge of this subject.

   b. **Translation Table Location**

      Candidates are not expected to demonstrate knowledge of this subject.

   c. **TLB**

      Candidates should be aware of the existence, location and purpose of TLBs. They should be aware of circumstances which require TLB maintenance operations.

      Example: Candidates must know the purpose of TLBs. More advanced candidates will know that TLB maintenance operations are required and when these must be carried out.

The *Translation Lookaside Buffer* (TLB) is described in the Cortex-A Series Programmer's Guide Section 10.4. The use of maintenance operations is covered in the Cortex-A Series Programmer's Guide, Section 10.5.

d. **OS Support (e.g. ASIDs)**

Candidates are not expected to demonstrate knowledge of this subject.

e. **Address Translation**

Candidates must have an awareness of virtual to physical memory translation, why it is necessary and how it is used by Operating Systems. They should be aware of where the MMU is located in the memory system hierarchy. They should be aware of which cores have virtual memory capability and which do not.

At this level, knowledge of page table and descriptor formats is not required.

Example: Candidates must know what virtual-physical address translation is and what it is used for in an Operating System environment. More advanced candidates will be aware of how Operating Systems use the MMU to achieve this and may be aware of multi-level translation tables.

Chapter 10 of the Cortex-A Series Programmer's Guide provides thorough coverage of the Memory Management Unit. As the MMU is part of the L1 memory system, you should become familiar with the contents of Chapters 9 and 11 to gain a better understanding of MMU behavior.

Candidates should attempt to gain some practical experience with using an MMU. This does not necessarily mean creating page tables. For example, candidates could take a running system with the MMU enabled and observe that with the MMU disabled, no address translation takes place and all data side memory accesses are treated as Strongly Ordered and therefore the data cache is not used. The SCTLR register in CP15 has an "M" bit which is used to enable or disable the MMU.

4. **PMSA**

Candidates should have an awareness of the features of the PMSA and what it can be used for. They should know which cores have MPUs and which do not. They should be aware of the limitations of a MPU as compared to an MMU.

Example: Candidates should know that an MPU can be used to partition and protect memory regions. More advanced candidates will know the number and usage of these regions.

The Cortex-A Programmers Guide does not provide any coverage of this topic, as the *Memory Protection Unit* (MPU) is associated primarily with the Cortex-R (ARMv7-R profile) Series processors and may also be found in some Cortex-M Series devices.

In systems which do not require the overhead of a full MMU, an MPU may be provided. This does not provide address translation facilities for virtual memory, but works with the L1 memory system to control accesses to and from L1 and external memory. The MPU enables you to partition memory into regions and set individual memory type attributes (normal/device/strongly ordered, cacheable attributes etc.) and protection attributes for each region. The MPU typically supports a small, fixed number (e.g. eight or twelve) of memory regions.

Each region is programmed with a base address and size, and the regions can be overlapped to enable efficient programming of the memory map. To support overlapping, the regions are assigned priorities. The MPU returns access permissions and attributes for the highest priority region where the address hits. These regions are programmed using CP15 registers c1 and c6, with access permitted only from Privileged modes.

5. **Alignment**

Candidates must have good knowledge of the data and instruction alignment requirements of the ARM architecture and be aware of the consequences of misalignment in systems with and without an MMU.

Example: Candidates will know the basic code and data alignment restrictions. More advanced candidates will be aware of specific cases (e.g. LDREX, LDRD, VFP) and of the consequences of violating alignment rules.

This is described in Section 16.2 of the Cortex-A Series Programmer's Guide. Candidates should become familiar with the specific alignment rules of the cases mentioned above (load and store exclusive instructions, load and store doubles and floating point instructions) by referring to the *ARM Architecture Reference Manual* (ARM ARM) or other assembly language documentation.

6. **Endianness**

Candidates must know what endianness means. They should know the two endian models supported by the ARMv7 architecture. Knowledge of endianness support prior to ARMv7 is not required.

Example: Candidates should know what endianness is and that ARM supports both big-endian and little-endian models. More advanced candidates would know that endianness can be changed at runtime and how to use this feature.

This is described in Section 16.1 of the Cortex-A Series Programmer's Guide.

7. **Attributes (e.g. Shareability, Executability)**

Candidates must know the memory attributes available in ARMv7, (e.g. executability) and understand when and how to apply them in real-world examples.

Example: Candidates should know and understand the read-write and execute permissions. More advanced candidates will know about shareability and how and when to use it.

These topics are covered in the Cortex-A Series Programmer's Guide, Section 10.7 and (for Shareability) Chapter 11.

The Cortex-A Series Programmer's Guide does not mention one of the key attributes that may be set in page tables, the XN (Execute Never) bit. When set, this prevents speculative instruction fetches from taking place to the memory location (useful for peripheral devices) and causes a prefetch abort to occur should execution from the memory location take place (useful for trapping problems caused by broken function pointers).

8. **Cache**

Candidates must understand the ARMv7 cache hierarchy (e.g. levels of cache, Harvard vs. unified). Candidate must know the available cache maintenance operations (e.g. Clean, Invalidate) and understand when they are required. Detailed knowledge of the instructions used to carry out these operations, nor how to address individual cache lines, is not required.

Example: Candidates should know caches exist, that they are in a hierarchy, that some are unified while others are Harvard. More advanced candidates will know that maintenance operations are required, what these operations are and in what circumstances they would be used.

Chapter 9 of the Cortex-A Series Programmer's Guide covers cache organization in some detail. Candidates may wish to obtain some simple practical experience of caches. For example, they might verify that the performance of their system is significantly lower when caches are disabled.

3.3.4 Registers

1. **VFP**

   Candidates should be aware that VFP supports a separate register bank. They should be aware of the number and size of these registers. Detailed knowledge of floating point operating and number formats is not required.

   Example: Candidates must know that VFP uses a separate register bank. More advanced candidates will know the size, format and behavior of these registers.

   Chapter 7 of the Cortex-A Series Programmer's Guide provides coverage of ARM Floating Point.

   NEON and VFPv3 use the same register bank. This is distinct from the ARM register bank. In VFPv3, you can view the register bank as thirty-two 64-bit doubleword registers, D0-D31, or as thirty-two 32-bit single word registers, S0-S31 (only half of the register bank is accessible in this view) or as a combination of registers from these views. In VFPv2 and the VFPv3-D16 variant, a smaller number of double precision registers is provided (D0-D15).

2. **NEON**

   Candidates should be aware that NEON supports a separate register bank. They should know how this overlaps with the VFP registers. Detailed knowledge of NEON operations and instructions is not required.

   Example: Candidates must know that NEON uses a separate register bank. More advanced candidates will know the size, format and behavior of these registers

   Chapter 8 of the Cortex-A Series Programmer's Guide provides coverage of NEON. Advanced candidates should refer also to Chapter 20.

3. **ARM**

   Candidates must have detailed knowledge of the ARM general purpose registers (R0 – R15), including any with special significance/use (e.g. R15 = PC). They should know which are banked between modes.

   This is covered in Section 4.3 of the Cortex-A Series Programmer's Guide.

4. **System Control Registers (e.g. CP14, CP15)**

   Candidates should be aware that system control registers are accessed via CP15 and debug registers via CP14. Although detailed knowledge is not required, they should be aware of what features are available in VMSA and PMSA processors.

   Example: Candidates must know that CP15 is used for system control and CP14 for debug. More advanced candidates will know some detail on specific registers.

   This information is given in the Cortex-A Series Programmer's Guide, Section 6.8.1. There are numerous references to specific registers throughout the Cortex-A Series Programmer's Guide.

3.3.5 Versions

1. **ARM Architecture**

   Candidates must have a high-level knowledge of the evolution of the ARM architecture between ARMv4T and ARMv7-AR. Candidates should be able to identify which architectural features are available in which versions. They should be able to match processors with the architecture to which they conform. Candidates should be aware of what is defined by the architecture (e.g. programmer's model) and what by the implementation (e.g. pipeline structure).

   This is covered in Section 2.1 of the Cortex-A Series Programmer's Guide.

2. **VFP Architecture**

   Candidates should be aware of the major differences between the VFPv3 and VFPv4. Candidates should also be aware of differences between the D16 and D32 variants (i.e. number of registers). Detailed knowledge of VFP function or number format is not required.

   This is briefly covered in Chapter 2 of the Cortex-A Series Programmer's Guide. As suggested, the D16 and D32 variants differ in the number of double precision registers available. In D16, there are 16 double-precision registers, which may also be viewed as 32 single-precision registers. In D32, there are 32 double-precision registers (D0-D31). The VFPv4 architectural extension is supported by the Cortex-A5, Cortex-A7 and Cortex-A15 processors. It extends VFPv3 by adding support for fused multiply-accumulate operations and half-precision format conversion.

3. **NEON Architecture**

   Candidates should be aware of the major differences between versions 1 and 2 of the advanced SIMD extensions. They should also be aware that Advanced SIMD extensions are only available in A class processors. Detailed knowledge of NEON instruction behavior is not required.

   This is briefly covered in Chapter 2 of the Cortex-A Series Programmer's Guide.

4. **Compatibility**

   Candidates must understand the compatibility limitations of code written for the different architecture versions (e.g. ARMv6 code cannot be guaranteed to run on a ARMv5 processor but will run on a ARMv7 processor).

   This is covered in Chapter 2 of the Cortex-A Series Programmer's Guide.

3.3.6 Exception model

1. **Modes**

   Candidates must know the names and functions of the seven basic operating modes. Knowledge of HYP and MON modes is not required. They should know about register banking, including which registers are banked in which modes. They should know which modes are privileged and which are associated with specific types of exception or interrupt.

   This is covered in Chapter 4 of the Cortex-A Series Programmer's Guide.

2. **Interrupts**

   Candidates must understand the differences between FIQ and IRQ, including banked registers, placement of FIQ in the vector table, use with an interrupt controller. Candidates should know how to enable and disable FIQ and IRQ via the CPSR.

   This is covered in Chapters 12 and 13 of the Cortex-A Series Programmer's Guide.

3. **Abort**s

   a. **Data Abort**

      Candidates should know what events can cause a Data Abort exception and what the usual corrective actions are. Knowledge of how to code solutions for these corrective actions is not required. Candidates should be aware of the high-level difference between a synchronous and asynchronous abort.

      Example: Candidates must know what events may cause a data abort. More advanced candidates would understand the difference between asynchronous and synchronous aborts.

   b. **Prefetch abort**

Candidates should know what events can cause a Prefetch Abort exception and what the usual corrective actions are. Knowledge of how to code solutions for these corrective actions is not required. Candidates should know and understand why the Prefetch Abort exception is not taken immediately on receiving the error from the memory system.

Both **Data Aborts** and **Prefetch Aborts** are covered in Chapter 12 and Section 14.1 of the Cortex-A Series Programmer's Guide.

The Cortex-A Programmer's Guide does not cover a topic which is mainly relevant to older versions of the ARM Architecture. The ARM7TDMI differs from later ARM processors in that it uses a "base updated" Data Abort model. Newer ARM designs use a "base restored" model.

The base restored Data Abort model means that when a data abort exception occurs during the execution of a memory access instruction, the processor hardware always restores the base register to the value it contained before the instruction was executed. This removes the requirement for the Data Abort handler to unwind any base register update that the aborted instruction might have specified. This makes it much simpler to write a Data Abort handler that can fix the cause of the abort and re-execute the failed instruction.

4. **UNDEFINED instruction**

   Candidates should know what events can cause an Undefined Instruction exception, how to determine which event has occurred and what the usual corrective actions are for each cause. At this level, knowledge of how to code these solutions is not required.

   This is covered in the Cortex-A Series Programmer's Guide, Section 14.2.

5. **Supervisor calls**

   Candidates must know what Supervisor Calls are typically used for, what causes an Supervisor Call exception, how to use the SVC instruction, what happens to the comment field, how to pass parameters to a Supervisor Call exception handler.

   Example: Candidates should know what an SVC instruction is used for. More advanced candidates will know how to pass parameters to an SVC handler.

   This is covered in the Cortex-A Series Programmer's Guide, Section 14.3.

6. **SMC**

   Candidates should know that the SMC instruction exists and that it is associated with entry to Mon mode in TrustZone systems. Detailed knowledge of its behavior is not required.

   This is covered in the Cortex-A Series Programmer's Guide, Chapter 26.

7. **RESET**

   Candidates must know what can cause a system reset (debug, reset pin etc). They should also be aware of configuration options which can be sensed during reset (endianness, vector table location etc).

   Example: Candidates must know the behavior and state of the processor on reset. More advanced candidates should know which configuration options can be sensed during reset.

   Chapters 12 and 15 of the Cortex-A Series Programmer's Guide cover the behavior of the reset exception.

   It is usual to consider reset as a kind of exception in the ARM architecture. This is in spite of the fact that it does not make sense to consider an exception return – reset is the starting point of the system, not something that we can return from. The Cortex-A Series Programmer's Guide does not specifically list all of the configuration options, although

HIVECS vector table location is mentioned in Section 12.5. On some processors, it is possible to configure the value of the endianness bit (and/or support for unaligned accesses) in a similar fashion.

8. **HVC**

   Candidates are not expected to demonstrate knowledge of this subject.

9. **Entry sequence**

   Candidates must know the actions taken by the core during exception entry, including any modifications made to registers.

   This is covered in the Cortex-A Series Programmer's Guide, Section 12.3.

10. **Exit sequence**

    Candidates must know what actions which must be taken in order to return from an exception handler. They should know the standard instructions for returning from each exception type and why these are used in each case.

    This is covered in the Cortex-A Series Programmer's Guide, Section 12.4.

11. **Nested/Re-entrant**

    Candidates should be aware that exceptions of different types may be nested easily but that there are issues with return addresses when interrupts are re-entrant. Also that standard good practice is not to make FIQ re-entrant. Detailed knowledge of coding a solution for re-entrant interrupts is not required.

    Example: Candidates must know what is meant by a re-entrant interrupt handler and that there are issues with nesting interrupts on ARM systems. More advanced candidates will know what these issues are and be aware of common solutions to them.

    This is covered in the Cortex-A Series Programmer's Guide, Section 13.1.3.

12. **Low Latency Interrupt Mode**

    Candidates are not expected to demonstrate knowledge of this subject.

13. **Vector table**

    Candidates should know the two standard locations of the vector table, how these are selected (HIVECS), that the vector table contains instructions not addresses, that RESET is the first entry and FIQ is the last. Knowledge of extra vector tables in systems implementing the Security Extensions is not required.

    Example: Candidates must know that the default location for the vector table is at 0x0 and that it contains instructions. More advanced candidates will know that the vector table can be relocated and to where, and will know the layout of the table.

    This is covered in the Cortex-A Series Programmer's Guide, Section 12.5.

3.3.7 Extensions

1. **Security Extensions**

   a. **Secure Monitor**

      Candidates should know that a Secure Monitor is used for handling the transition between Secure and Normal worlds. They should have a general understanding of how monitor mode is used to achieve this. Detailed knowledge of its internal function is not required.

   b. **Secure Exceptions**

      Candidates are not expected to demonstrate knowledge of this subject

   c. Secure Memory

Candidates should be aware that, in a TrustZone system, areas of memory can be made Secure as the S/NS information is propagated on the external bus. They should understand why this is useful.

This is covered in the Cortex-A Series Programmer's Guide, Chapter 26.

d. **Virtualization Extensions**

Candidates are not expected to demonstrate knowledge of this subject.

3.4 Software development

This section covers software development.

3.4.1 Synchronization (e.g. Mutexes, Semaphores, Spinlocks)

1. **Synchronization**

   Candidates should be aware of the need for this kind of construct and be able to insert them in example code sequences. They should be able to read and understand synchronization functions which use LDREX/STREX. They should understand that memory barriers are sometimes required to ensure correct and efficient operation.

   See the Cortex-A Series Programmer's Guide, Section 22.4.

3.4.2 Memory Management

1. **Cache Maintenance**

   Candidates should be aware that operations are required to maintain coherency between contents of cache and main memory. They should understand the terms "flush", "clean" and "invalidate". They should know that these operations are achieved via CP15 but detailed knowledge of instructions used to achieve these operations is not required.

   See the Cortex-A Series Programmer's Guide, Section 9.8.

2. **Barriers**

   Candidates must know what ISB, DSB and DMB instructions do. They should also be able to determine where these instructions are required in typical use cases.

   See the Cortex-A Series Programmer's Guide, Section 11.2

3.4.3 OS Architecture

1. **SMP vs AMP**

   Candidates should be aware of the principal differences between SMP and AMP systems. Given examples, they should be able to differentiate between them.

   Example: Candidates must know what is meant by SMP. More advanced candidates will also understand AMP and be able to differentiate between the two.

   See the Cortex-A Series Programmer's Guide, Chapter 21.

2. **Threads, Tasks and Processes**

   Candidates should be aware of the meaning of these terms in SMP and AMP systems.

   *Example: Candidates must know what these terms mean. More advanced candidates will know specifics of how some Operating Systems treat the*m.

   See the Cortex-A Series Programmer's Guide, Chapter 21 (in particular, Section 21.2).

3. **Kernel vs User Space**

   Candidates should be aware that Operating Systems partition memory into Kernel (privileged) and User (non-privileged) regions. They should know that this is configured via the MMU. They should understand that a transition to privileged mode is required to access Kernel space and that this is usually achieved via an exception. They should be aware of typical operations (e.g. memory map maintenance, exception handling) which must be carried out in a privileged mode.

   Example: Candidates must know that different privilege levels exist in an Operating System environment. More advanced candidates will know how Operating Systems partition memory and how user programs gain access to privileged operations.

The concept of privilege levels is introduced in the Cortex-A Series Programmer's Guide, Chapter 4. The association of exception handling with specific privileged modes is also covered there. The idea that the kernel is responsible for memory management is covered in the Cortex-A Series Programmer's Guide, Chapter 10 and Section 10.8 gives more detail on this. Advanced candidates may wish to read about the use of `syscall` in Linux as an example of how user programs gain access to privileged operations (e.g. by reading the `syscall man` page.)

4. **BSP**

   Candidates should be aware that some level of board/device-specific code is usually required by an operating system, and this is usually referred to as a Board Support Package (BSP).

   The Board Support Package is described in the Cortex-A Series Programmer's Guide, Section 3.1.4.

3.4.4 Booting

1. **Multiprocessor Boot**

   Candidates are not expected to demonstrate knowledge of this subject.

2. **Enable VFP and NEON**

   Candidates are not expected to demonstrate knowledge of this subject.

3. **From RAM**

   Candidates are not expected to demonstrate knowledge of this subject.

4. **From ROM**

   Candidates are not expected to demonstrate knowledge of this subject.

5. **From TCM**

   Candidates are not expected to demonstrate knowledge of this subject.

6. **Initializing Stack Pointer**

   Candidates are not expected to demonstrate knowledge of this subject.

7. **Placing Vector Table**

   Candidates are not expected to demonstrate knowledge of this subject.

8. **Populating Vector Table**

   Candidates are not expected to demonstrate knowledge of this subject.

9. **Peripheral Configuration**

   Candidates are not expected to demonstrate knowledge of this subject.

10. **Cache Initialization (e.g. TLB, Branch Prediction)**

    Candidates are not expected to demonstrate knowledge of this subject.

11. **MMU/MPU Initialization**

    Candidates are not expected to demonstrate knowledge of this subject.

12. **Secure Boot**

    Candidates are not expected to demonstrate knowledge of this subject.

13. **Feature Interrogation (e.g. CP15 feature registers)**

    Candidates are not expected to demonstrate knowledge of this subject.

3.4.5 ABI

1. **AAPCS – Procedure Call Standard for the ARM Architecture**

   Candidates should be familiar with the AAPCS rules in respect of register usage, parameter passing and result return, stack alignment and register preservation.

   Example: Candidates must be aware of the AAPCS register usage rules and the requirement for stack alignment. More advanced candidates will know how parameters are mapped to registers and stack.

   The *ARM Architecture Procedure Call Standard* (AAPCS) is covered in Chapter 17 of the Cortex-A Series Programmer's Guide. It is important to understand that variable sizes in the ARM architecture may differ from other processors (int is 32-bit, a short is 16-bit, char is 8-bit; a double precision float is 64-bit) and that local variables may be stored in registers (if the number of currently live variables is not too high) or on the stack.

2. **Hard/Soft Linkage**

   Candidates should be familiar with the concepts of hard and soft linkage in procedure calls with respect to floating point. Given examples, candidates should be able to differentiate between the two. They should be able to explain why this matters.

   This is covered in the Cortex-A Series Programmer's Guide 17.1.2, although familiarity with Chapter 7 and Cortex-A Series Programmer's Guide 17.1.1 is assumed.

   Floating-point arithmetic may use either hardware coprocessor instructions or library functions. Floating-point linkage, however, is concerned with passing arguments between functions that use floating-point variables.

   Software floating-point linkage means that the parameters and return value for a function are passed using the ARM integer registers R0 to R3 and the stack. Hardware floating-point linkage uses the VFP coprocessor registers to pass the arguments and return value. The benefit of using software floating-point linkage is that the code can be linked with code compiled to run on a core without a VFP coprocessor. This is often important if using libraries built to be portable across processors with and without hardware VFP support. The benefit of using hardware floating-point linkage is that it is more efficient than software floating-point linkage, but runs only on systems which have a VFP coprocessor and all objects / libraries used must be compiled with hardware floating-point linkage.

3. **Stack Alignment (Exception entry)**

   Candidates are not expected to demonstrate knowledge of this subject.

3.4.6 Compiler

1. **Automatic Vectorization**

   Candidates must know what vectorization is and that the compiler can do this automatically. They should be aware of what properties of a loop make this easier and more efficient for the compiler. They should know that certain combinations of compiler options may be required to enable this feature.

   Example: Candidates must know what vectorization is. More advanced candidates will know how to write code and configure the compiler in order to facilitate this.

   Section 20.1.1 of the Cortex-A Series Programmer's Guide describes vectorization. The compiler can perform automatic vectorization of source code. As the C language does not specify parallelizing behavior, you may have to tell the compiler where this behavior is safe. This may require the use of the `__restrict` keyword to guarantee that the pointers do not address overlapping regions of memory.

You can do this without compromising the portability of the source code between different platforms or toolchains.

Example 1.4 `http://infocenter.arm.com/help/topic/com.arm.doc.dht0002a/ch01s04s03.html#CACEHEAG` shows a small function that the compiler can safely and optimally vectorize. This is possible because the programmer has used the `__restrict` keyword to guarantee that the pointers `pa` and `pb` will not address overlapping regions of memory.

It may also be necessary to indicate to the compiler that a for loop will produce a number of iterations which is a multiple of 4, by masking the bottom two bits of `n` for the limit test, with code similar to:

```
for(i = 0; i < (n & ~3); i++)
```

The necessary command line syntax differs between compilers; in GCC you must add `-mfpu=neon` and `-ftree-vectorize`. In some toolchain versions you have to add `-mfloat-abi=softfp` to indicate that NEON variables must be passed in general purpose registers. For the ARM compiler, you must specify a target processor that includes NEON technology, compile for optimization level `-O2` or higher, and add `-Otime` and `--vectorize` to the command line.

2. **Intrinsics**

Candidates should know what an intrinsic function is and why they are to be preferred to writing native assembly code. They should be aware that a large number of intrinsics are supported by the C compiler and be able to explain what certain example functions do.

Example: Candidates must know what is in intrinsic function and that the C compiler supports a large number of them. More advanced candidates will know how to use some of these functions.

NEON instrinsics are mentioned in the Cortex-A Series Programmer's Guide, Section 20.1.3. Other intrinsics (such as barriers and semaphore instructions) are mentioned in passing in other sections (e.g. 11.2). Intrinsics are typically used to access ARM Assembly Language functions which do not map directly to C operators. This is done either to access system level features, or (as in the NEON case) to implement an algorithm more efficiently.

3. **Compiler options**

Candidates should be aware of:

- *Space/time optimization*
- *Optimization level*
- *C90/c99/cpp*
- *Architecture/CPU specification*

The student should become familiar with these options through using them with their chosen toolchain. The GCC optimization options (`-O1`, `-O2` etc.) are covered in the Cortex-A Series Programmer's Guide, Section 19.1.4, while their armcc equivalents are covered in Section 19.1.5. The `-arch` and `-cpu` switches are similar in both tools. In gcc, one specifies `-std=c90` etc. in armcc, the corresponding options are `--c90` or `--c99`.

4. **ARM/Thumb**

Candidates should know that it is possible to configure a compiler for ARM or Thumb compilation. They should be able to make sensible suggestions for when to use each instruction set.

Example: Candidates must know that it is possible to compile for ARM or Thumb. More advanced candidates will be able to make suggestions as to when each option is appropriate.

This topic is not well covered in the Cortex-A Series Programmer's Guide as, for Cortex-A Series processors, the Thumb-2 extension of the Thumb instruction set is usually preferred and in the Cortex-M processors only the Thumb instruction set is supported.

In older ARM processors, systems often contained code which was compiled for ARM state and code which was compiled for Thumb state. ARM code, with 32-bit instructions, was more powerful and required fewer instructions to perform a particular task and so might be preferred for more performance critical parts of the system. It was also used for exception handler code, as exceptions could not be handled in Thumb state on e.g ARM7 or ARM9 Series processors.

Thumb code (using 16-bit instructions) needed more instructions to carry out the same task, when compared with ARM. Thumb code could typically encode smaller constant values easily within instructions and has shorter relative branches. The available range for relative branches is approximately ±32MB for ARM instructions and ±16MB for the Thumb-2 extension. Thumb is further limited where only 16-bit instructions are used, with conditional branches having a range of ±256 Bytes and unconditional relative branches being limited to ±2048 bytes.

However, as the instructions were only half of the size, programs would be typically a third smaller than their ARM equivalent. Thumb was therefore used for reasons of code density and to reduce system memory requirements. Thumb code would also outperform ARM when the processor was directly connected to a narrow (16-bit) memory, without the benefit of cache. One Thumb instruction could be fetched on each cycle, whereas each 32-bit ARM instruction would need 2 clock cycles per fetch.

When executing a Thumb instruction, the PC reads as the address of the current instruction plus 4. The only 16-bit Thumb instructions which can directly modify the PC are certain encodings of `MOV` and `ADD`. The value written to the PC is forced to be halfword-aligned by ignoring its least significant bit, treating that bit as being 0.

In ARMCC, the option `--thumb` or `-arm` (the default) allows selection of the instruction set used for compilation.

5. **Cross vs. Native**

   Candidates must know the difference between Cross compilation and Native compilation, and recognize advantages and disadvantages of each.

   Section 3.1.3 of the Cortex-A Series Programmer's Guide describes cross-compilation and the advantages and disadvantages associated with it.

3.4.7 Linker

1. **Dynamic vs. Static**

   Candidates must know the nature of the difference between Static and Dynamic linkage. They should be aware that dynamic linkage is usually used in OS environments while static linkage is used for bare-metal applications.

   When code is compiled, the tools will convert your source code into object modules. Linking then combines object modules to produce an executable program. This linkage step may also include code from different languages (e.g. an Assembler) or libraries, for which source code may or may not be available. If a file is statically linked into an executable, this simply means that the contents of the file are included in the executable. In dynamic linkage, a pointer to the object is included in the executable, but the file contents are not. Instead, it is at run-time that the content of the dynamically linked file is placed into the executable program in memory.

Statically-linked files are fixed at link time and do not change. A dynamically linked file referenced by an executable may be changed simply by changing the file content (e.g. on an external drive or other device). Changes to functionality may therefore be achieved without having to re-link the code as the loader re-links on each execution.

Students should be familiar with the idea that the Linux kernel (or other complex OS) does not consist of a single executable object, but that executable code is dynamically loaded, whereas an embedded system may well be statically linked.

2. **Memory Layout**

Candidates must be aware that they can provide a description of memory layout to a linker, and understand when and why this is necessary. They are not expected to know the format used by a particular tool.

In many systems, it is necessary for the programmer to specify the memory system layout to the linker. For example, the linker may need to know the location of RAM in the system which is used to store variables and the location of ROM which holds the code to be executed. This information may be passed by linker command line options or by using a “scatter file.”

The ELF image file may contain program segments, region, input sections and output sections. An input section is an individual section from an input object file. It contains code, initialized data, or describes a fragment of memory that is not initialized or that must be set to zero before the image can execute. These properties are represented by attributes such as RO, RW and ZI. These attributes are used by the linker to group input sections into bigger building blocks called output sections and regions. An output section is a group of input sections with the same RO, RW, or ZI attribute which are placed contiguously in memory by the linker. A region is a contiguous sequence of output sections which typically maps onto a physical memory device, such as ROM or RAM. A Program Segment corresponds to a load region and contains output sections. Program Segments hold information such as text and data.

Output sections may be RO (Read-Only), RW (Read-Write) or ZI (Zero-Initialized). An example of a read-only region would be constant data contained within a program. A read-write region might contain variables with a defined non-zero initial value. The initial value must be stored within the image, but placed in a location where the executing code can change the variable. The Zero-Initialized region may contain program variables whose initial value is zero or unspecified.

An image may contain a number of separate regions and output sections. Each region may have a position in memory at which it is stored and a different location from which it will execute (for dynamic loading, or where code is copied from ROM to RAM at boot time to speed its execution). Code which is position-independent or relocatable may have its execution address changed at run time.

3. **Entry Points**

Candidates should know what constitutes an “entry point”. They should know why it is necessary to define entry points e.g. to indicate to the debugger where execution starts, or to prevent unwanted deletion of unreferenced sections. They should know the various methods of defining these to the linker.

Example: Candidates must know what an entry point is and that the linker cannot remove these. More advanced candidates would understand how to define entry points when linking code.

An Entry point is a location within an image where program execution can start. The initial entry point for an image is a single location where image execution starts. You must specify exactly one initial entry point for a program otherwise the linker produces a warning. Not every source file has to have an entry point. Multiple entry points in a single source file are not permitted. The initial entry point must always lie within an execution region, must not overlay another execution region, and must be a root execution region (the load address is the same as the execution address).

You can set multiple entry points using the ENTRY directive in an assembly file. In embedded systems, this directive is used to mark code that is entered through the processor exception vectors, such as RESET, IRQ, and FIQ. It marks the output code section with the ENTRY keyword. This instructs the linker not to remove the section when it performs unused section elimination. For C and C++ programs, the `__main()` function in the C library is an entry point. The `--entry` command line switch may also be used to specify an initial entry point, for example, `--entry 0x0` might be used in an embedded system with ROM at 0.

The linker will remove code and data sections from the image which are unreachable. If no entry point is specified, this identification of unused sections cannot be performed.

4. **Software Floating Point Libraries**

   Candidates should know that options exist for support of floating point in software.

   See the Cortex-A Series Programmer's Guide, Chapter 7.

3.4.8 C Library

1. **Retargeting(e.g. printf)**

   Candidates should be aware of the concept of semihosting, of why it is important to remove it before producing final code and of how to retarget basic functions in the C library. Exhaustive knowledge of these functions is not required.

   This is described elsewhere in this document.

2. **Configuring (e.g. Stack, Heap)**

   Candidates should be aware of the possible heap and stack configurations (one-region, two region) and how these are configured. Candidates are not required to know precise details of how to achieve this using different tool chains.

   Example: Candidates must know that stack and heap occupy separate regions of memory. More advanced candidates would know how to configure this in the final image.

   The stack is used to hold scratch data for the executing thread. On ARM processors, it always grows down in memory from a particular address. The heap is used for dynamic allocation of memory. ARM tools allow the initial location of the stack and heap to be configured by using specific region names in the scatter file for the linker, or by retargeting the `__user_initial_stackheap()` or `__user_setup_stackheap()` functions. ARM tools support two main types of stack and heap implementations. In the one region model, the stack and heap share a single area of memory. The heap grows up from the bottom of the memory region while the stack grows down from the top. In the two region model the heap and the stack each have their own memory region. These regions can be placed at any suitable address in memory. The heap still grows upwards through memory in its region and the stack still descends from the top of its region.

   The Application Binary Interface (ABI) for the ARM architecture requires that the stack must be eight-byte aligned on all external interfaces, such as calls between functions in different source files. However, code does not need to maintain eight-byte stack alignment internally, for example in leaf functions. A consequence of this is that when an interrupt or exception occurs the stack might not be correctly eight-byte aligned.

   A common initial value for the stack pointer is the top of RAM. For example, if a system has RAM from 0 to `0x7FFF`, the initial value of R13 might be `0x8000`. This means that the 4 bytes at `0x7FFC` would be the first empty location on the stack.

3. **Libraries (e.g. Bare Metal vs Operating System, C Library Variants)**

Candidates should be aware of the need for standard libraries, and understand that different implementations of the libraries are needed for bare-metal or Operating System environments.

Example: Candidates must know that here are several variants of the standard library. More advanced candidates would know what the differences are between these variants.

The standard C library is defined in the ANSI specification for the C language and provides functions (and associated macros and definitions) for common tasks such as input/output, string handling, mathematical operations and memory allocation.

Different implementations of libraries are required for a number of reasons. Embedded systems, where a small memory footprint is important, might use a smaller version of a library (e.g. “newlib”) in which some functions may be omitted, or have fewer options. Library variants may be associated with a particular Operating System/ Compiler.

For example, GNU/Linux will typically have the glibc standard library. Embedded ARM libraries might use soft or hard floating point, be big or little endian and might be in ARM or Thumb. As discussed elsewhere in this document, libraries may also be statically or dynamically linked with an application.

3.4.9 Target

1. **Models (e.g. PV and CA)**

   Candidates must know what types of ARM CPU model are available to them as software developers and be able to recognize advantages and disadvantages of each, as well as the advantages and disadvantages of using them over Development Boards and Final Hardware. They are not required to have knowledge of products from any specific company.

   Models of ARM processors can be used for software development in the absence of “real” hardware. Such models allow systems to be simulated on a PC and provide a functionally accurate representation of the processor and the memory and peripherals it is connected to. Such models often provide useful debug features, allowing the values of registers and variables to be observed without the need for a JTAG debugger. Models are, of course, significantly slower than real hardware and it may require a long time to (for example) boot Linux.

   ARM’s DS-5 product includes a *Real-Time System Model* (RTSM) which allows simulation speeds of over 250MHz on a typical desktop PC. In addition to the processor, it simulates peripherals such as an LCD controller, keyboard, mouse, touchscreen, UARTs and Ethernet controllers, using the host PC resources. Stop-mode debug using CADI connection for bare-metal or Linux kernel development is provided, as is run-mode debug using gdbserver connection for Linux application development.

2. **Development Boards**

   Candidates must know what types of Development Board are available to them to target as software developers, and be able to recognize advantages and disadvantages of each, as well as the advantages and disadvantages of using them over Models and Final Hardware. They are not required to have knowledge of products from any specific company.

3. **Final Hardware**

   Candidates should be able to recognize advantages and disadvantages of using Final Hardware over Models and Development Boards. They are not required to have knowledge of products from any specific company.

There are a wide range of development boards utilizing ARM Cortex-A Series processors available at low cost and it is recommended that students obtain such hardware and gain practical experience on running code. Some suitable boards are listed in Section 3.5 of the Cortex-A Series Programmer's Guide, but many newer boards are also available.

3.5 System

This section covers the system.

3.5.1 GIC Architecture

1. **Distribution Models**

   Candidates should be aware that multiprocessor systems require an interrupt controller/distributor such as a GIC. They should be aware of and know the differences between 1-N and 1-1 interrupt distribution models.

   The Generic Interrupt Controller (GIC) is described in Section 13.2 and Section 22.3 of the Cortex-A Series Programmer's Guide . The GIC is mainly found with the Cortex-A MPCore series processor and is part of the ARM processor private bus interface. It manages interrupts in systems containing one or more processors and essentially consists of a distributor shared between all processors plus one or more separate, private processor interfaces.

   The difference between 1-N and 1-1 distribution models is that in the 1-N case, an interrupt is routed to all available processors, whereas with a 1-1 model, each interrupt is routed to just one processor.

2. **Priority Management**

   Candidates should be aware that the priority of interrupts is configured via the GIC, how many levels of priority are available and what effect this has on interrupt handling.

   Priority management of the GIC is briefly mentioned in the Cortex-A Series Programmer's Guide, Section 13.2.1. Students may also wish to refer to Chapter 3 of the GIC Architecture specification.

3. **Interrupt State Model**

   Candidates should know the possible states of an interrupt as it is handled via the GIC, together with the necessary actions which are taken in each state. They should be aware of what causes transitions between states.

   The Interrupt State Model of the GIC is mentioned in the Cortex-A Series Programmer's Guide, Section 13.2. Students may also wish to refer to Chapter 3 of the GIC Architecture specification.

4. **Software Generated Interrupts**

   Candidates should be aware that the SGI mechanism exists within the GIC and what it might be used for.

   An SGI is generated by writing to the *Software Generated Interrupt Register* (ICDSGIR). It is most commonly used for inter-processor communication.

3.5.2 Software Storage and Upload

1. **Flash**

   Candidates must know what types of code and/or data are typically stored in Flash memory, and what the advantages and disadvantages of Flash memory are over other types of storage. They should also have a basic (high-level) understanding of how code and/or data are uploaded to Flash memory (e.g. that a specific algorithm is required to program it and that this is usually achieved by running a dedicated programming routine on the target hardware).

   Flash memory is a non-volatile memory which can be electronically erased and re-written. The number of times that the device can be erased is limited and typically dedicated software is required to handle re-writing flash.

There are two main types of flash memory, with names deriving from the NAND and NOR logic gates. NAND flash bits have a '1' value when erased. NOR flash bits read as '0' after being erased.

NAND flash may be written and read in pages which are smaller than the entire memory, in a similar fashion to a disk. The pages are typically between 512 and 4096 bytes in size. The memory may be erased on a per-block basis, where the blocks might typically contain between 32 to 128 pages. The device will typically have some ECC facility to cope with faulty bits and software will also have to cope with the fact that some blocks on the device will also be bad. NOR flash supports random-access for reads and so it is possible for code to be executed directly from this type of memory, whereas code would typically be copied from NAND flash to RAM before being executed. NOR flash is more expensive than NAND flash and might be used to hold boot code (for example), where NAND flash might be used for USB "flash drives" or memory cards.

2. **Removeable**

   Candidates should know the common types of removable storage, and what types of code and/or data are typically stored in them. They should know what the advantages and disadvantages of removable storage are over other types of storage. They should also have a basic (high-level) understanding of how code and/or data are uploaded to removable storage.

   Computer systems may have a variety of removable storage media. Common examples might include CDs or DVDs, floppy disks and memory cards. Such media provide a much larger storage area compared with system memory, but with a significantly longer access time. Any introductory computer textbook will provide further details on this syllabus area.

3. **Network Storage**

   Candidates should know the common types of network storage, and what types of code and/or data are typically stored in them. They should know what the advantages and disadvantages of network storage are over other types of storage. They should also have a basic (high-level) understanding of how code and/or data is uploaded to network storage.

   Network storage simply refers to the ability of computer systems to store code and/or data (usually at a file level) on an external device to which some network connection is available. This may be, for example, one or more hard drives, often arranged with some element of redundancy in the case of failure. Files will typically be made available using protocols such as NFS or SMB. Most introductory computer textbooks will provide further details on this syllabus area.

3.5.3 Power management

1. **Power Management**

   Candidates must be aware of the four basic power modes support by ARM cores (Run, Standby, Dormant, and Shutdown), of the comparative power saving available in each mode and of when these states might be used in a typical system. They should be aware of what system state is preserved in each state and of the comparative complexity and cost of entering and exiting each state. Detailed knowledge of entry/exit sequences is not required, nor is knowledge of any external PMC support circuitry.

   This is covered in Chapter 25 of the Cortex-A Series Programmer's Guide.

3.5.4 Memory System Hierarchy

1. **Integrated vs Non-Integrated Caches (POC and POU)**

Candidates are not expected to demonstrate knowledge of this subject.

2. **Cache Hierarchy**

   Candidates should be aware that systems frequently have more than one level of cache and the differences between unified and Harvard caches. The terminologies and topologies of simple example architectures should be known. They should know the cache topologies of some common systems, for example, the Cortex-A8 processor.

   This is covered in Chapter 9 of the Cortex-A Series Programmer's Guide.

3. **Memory Map**

   Candidates should be aware of the constraints of memory map structures and be able to distinguish between common/desirable physical memory maps for ARM-based systems and those that are uncommon/undesirable.

   In general, the physical memory map of Cortex-A series processors is not fixed by the architecture and so the main consideration is ensuring that the reset exception vector at 0 (or `0xFFFF 0000`) is present at boot time and perhaps also in ensuring that the vector table location is in RAM to enable faster execution when compared with having the code in flash or ROM device. The position of the "private" memory region giving access to GIC, timers, etc. may also need to be considered.

   Cortex-R series and Cortex-M series devices will typically have a default memory map.

   Cortex-R series processors (and older ARM11 and ARM9 series devices) may contain *Tightly Coupled Memory* (TCM). Some systems require fast, deterministic access to memory which contains critical code or data. The TCM is designed to provide low-latency memory, local to the processor (part of the L1 memory system) that can be used without the unpredictability of access time associated with caches. There is typically a separate TCM for both the instruction and data side of the processor and in certain processors, each TCM may itself be split into two separately accessible blocks.

   TCM may be used to hold critical routines, such as interrupt handling routines or real-time tasks where the indeterminacy of a cache is highly undesirable. It might also be used to hold scratch pad data, data types whose locality properties are not well suited to caching, and critical data structures such as interrupt stacks.

   Unlike cached memory, TCM forms part of the physical memory map of the processor. TCM status and control is achieved through access to dedicated registers in CP15. These allow the base address of the TCMs to be programmed, the size to be read by code etc.

3.6 Software Optimization

This section covers Software optimization.

3.6.1 General

1. **PMU**

   Candidates should be aware that ARMv7-AR cores implement a set of hardware performance counters. They should be aware of the types of event which can be counted, of combinations of counts which can provide useful information and of the use of interrupts to extend the range of the counters. They should know that use of the performance counters is non-intrusive.

   Example: Candidates must be aware of the PMU and its basic purpose. More advanced candidates will be aware of some of the events which can be counted and the kind of information which they can be used to derive.

   The Cortex-R and Cortex-A processors contain a *Performance Monitor Unit* (PMU) which can be used for benchmarking and profiling code. This allows code profiled to provide useful statistics, such as cycle and instruction counts. The PMU has a cycle counter, which can be configured to increment for every processor cycle or for every 64 processor cycles. It also has some configurable event counters, which can be set to count a chosen event (for example instructions executed, data cache misses or mispredicted branches). The number of counters and the range of available events to be counted vary between processor implementations.

   The performance counters can be configured and accessed through software accessing CP15, or with debug tools. The counters are 32-bit registers. Therefore, you should check that the performance counters have not overflowed after performing benchmarking. In some cases, it may be necessary to use software techniques to enable counting of larger numbers of events. For example, it is possible to have the processor automatically generate an interrupt upon a counter overflow and this could be used to increment a software variable and reset the PMU counter.

2. **Coding Techniques for Low Power Consumption**

   Candidates should be aware of simple coding techniques to minimize external memory accesses, minimize instruction count and make use of available power modes. They should be aware that these are all techniques which may be used to reduce power consumption.

   System energy consumption may be minimized by producing code which minimizes the number of instructions required to be executed in order to achieve the desired outcome and also by minimizing the number of external memory accesses. In particular, memory accesses which must go off-chip have a significantly higher power cost. Chapter 19 of the Cortex-A Series Programmer's Guide covers this syllabus area.

3. **Coding Techniques for Performance**

   Candidates should be aware of simple coding techniques which may be used to improve performance. For example, using integers for data processing operations, passing no more than four parameters, writing decrementing loops to zero, naturally aligning data.

   Chapter 19 of the Cortex-A Series Programmer's Guide covers this syllabus area.

4. **Coding Techniques for Small Memory Footprint**

   Candidates should be aware of simple coding techniques which may be used to reduce memory footprint of code and data. For example, use of __packed to reduce padding in structures, configuring the compiler to optimize for code size, using Thumb code.

Chapter 19 of the Cortex-A Series Programmer’s Guide covers this syllabus area. Students should be aware of the fact that the `--Ospace` command line option may be used to tell the compiler to optimize for reduced memory use. Candidates should understand that the 16-bit instructions of the Thumb instruction set allow Thumb code to be significantly smaller than a corresponding section of ARM code. Section 16.3.2 of the Cortex-A Series Programmer’s Guide provides some descriptions of structures, padding and the `__packed` attribute.

5. **Cache-Sensitive Coding (Coding for efficient memory access)**

   Candidates are not expected to demonstrate knowledge of this subject.

3.6.2 Techniques for Identifying Performance Bottlenecks

1. **Profiling**

   Candidates must know what profiling is and, given a description of an application performance problem, the candidate should know what can be learned and what cannot be learned from profiling the application.

   Chapter 18 of the Cortex-A Series Programmer’s Guide provides an introduction to profiling.

2. **CPU Frequency Changing**

   Given a description of an application performance problem, the candidate should know what can be learned and what cannot be learned from changing the CPU frequency.

   In many systems, the processor clock speed may be a performance bottleneck. Therefore, increasing the processor clock speed will increase system performance. Conversely, if increasing the processor clock speed does not produce a corresponding increase system performance, it may be concluded that the processor clock speed is not the limiting factor in the system. Instead, it may be necessary to look at other parts of the system. For example, it may be the access latency of a particular peripheral, or insufficient memory system bandwidth which limits system performance.

3. **Wall Clock Timing**

   Given a description of an application performance problem, the candidate should know what can be learned and what cannot be learned from measuring the wall clock time elapsed for different parts of the application. They should know when this method is not suitable (e.g. when using a model).

   Measuring “real” time elapsed is generally useful only for software running on “real” hardware. A simulation model of a processor might run at the rate of less than 1 million clock cycles per second and be affected by other, unrelated activity on the same PC.