* **ARM Procedure Call Standard (APCS)**
* **Thumb Procedure Call Standard (TPCS)**
* **ARM-Thumb Procedure Call Standard (ATPCS)**
* **Procedure Call Std. for ARM Architecture (AAPCS)**

It defines register usage and stack conventions that must be followed in order to enable separately compiled or assembled subroutines to work together.

별도로 컴파일 되거나 어셈블 된 서브 루틴이 함께 작동 할 수 있도록 하기 위해 준수해야하는 레지스터 사용 및 스택 규칙을 정의한다.

It describes a contract between a calling and a called routine:

호출과 호출 된 루틴 간의 계약을 설명한다:

* + Obligations on the caller to create a memory state in which the called routine may start to execute.

호출 된 루틴이 실행되기 시작할 수 있는 메모리 상태를 만들기 위한 호출자의 의무.

* + Obligations on the called routine to preserve the memory-state of the caller across the call.

호출을 통해 호출자의 메모리 상태를 보존하기 위해 호출된 루틴에 대한 의무.

* + The rights of the called routine to alter the memory-state of its caller

호출자의 메모리 상태를 변경하는 호출 된 루틴의 권한.

Specifies a family of Procedure Call Standard (PCS variants, generated by a cross product of user choices that reflect alternative priorities among:

Code size / Performance / Functionality e.g.) ease of debugging, run-time checking, etc.

Contains - A machine-level base standard / A set of machine-level variants / Constraints on the layout of activation records and function entry sequences to support stack unwinding / The representation of externally visible C-language and C++ extern “C” entities

* **AAPCS Base Standard**

Defines - A machine-level, integer-only calling standard common to ARM and Thumb, and a machine-level floating-point standard for ARM.

The base standard does not provide for – Interworking between ARM state and Thumb state / Position independence of either data or code / Re-entry to routines with independent data for each invocation / Stack checking: they are specified by the PCS variants

* **AAPCS Base Standard: Machine Registers**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| reg. | syn. | spc. | role in the PCS | |
| r0 | a1 |  | argument / result / scratch register 1 to 4 | |
| r1 | a2 |  |
| r2 | a3 |  |
| r3 | a4 |  |
| r4 | v1 |  | variable register 1 to 8 |  |
| r5 | v2 |  |  |
| r6 | v3 |  |  |
| r7 | v4 | WR | thumb-state work reg. |
| r8 | v5 |  |  |
| r9 | v6 | SB | Static Base in PID |
| r10 | v7 | SL | Stack Limit pointer |
| r11 | v8 | FP | ARM-state frame pointer |
| r12 |  | IP | Intra-Procedure-Call scratch register | |
| r13 |  | SP | Stack Pointer | |
| r14 |  | LR | Link Register | |
| r15 |  | PC | Program Counter | |

* **AAPCS Base Standard: Floating Point of FPA arch.**

Eight floating-point registers, f0-f7. Each may hold a value of single, double or extended precision.

* **AAPCS Base Standard: Floating Point of VFP arch.**

Sixteen double-precision registers, d0-d15. Each may also be used to hold two single precision numbers. No extended precision.

* **AAPCS Base Standard: Routine Call** can be implemented by any instruction sequence that has the effects LR (Return addr.) and PC (Destination addr.). A subroutine call preserves the values of r4-r11 and SP.

Parameter Passing Types are 32-bit integers and Single or Double precision floating point numbers.

A source language parameter value is converted to a machine parameter value as follows:

* + An int values narrower than 32bit widen to 32bit.
  + 64-bit int treated as two 32-bit int values.
  + One or more float values of the corresponding machine types if floating point hardware exists.  
    Floating HW가 존재할 경우 float값이 해당 머신 유형의 하나 이상의 float 값으로 변환.
  + A sequence of integer machine words if float operation is simulated.
  + Others converted to sequence of 32-bit integer

Variadic routines (what are passing variable number of parameters) loading the first 4 words into integers a1-a4 (lowest addressed into a1). Then pushing remaining words onto the in reverse order. (A float value can be passed in integer registers, or can be split between an integer register and memory.)

Non-variadic routines, the first N floating-point values are assigned to floating-point argument registers. The rest is the same as variadic routine.

* **AAPCS Base Standard: Result Return**

A procedure returns no result. A function returns a single value that occupies 1 or more words.

Integer function must return a 1-word value in a1, which can be 2-4 words. A longer value must be indirectly returned to memory.

Floating-point function returns float in r0, which is converted to single precision if it is half-precision. Double is returned in r0 and r1.

* **AAPCS Floating-Point Support**

The ARM process core does not contain floating-point hardware. Instead floating-point can be done in one of three ways.

* **AAPCS Floating-Point Support:** The **software floating-point library (fplib)**, supplied as part of the ARM Developer Suite C library, procides functions that can be called to implement floating-point operations using no additional hardware. This is the default tools option and most systems have historically made use of this.
* **AAPCS Floating-Point Support:** A **hardware coprocessor** attached to the ARM processor core that implements a number of instructions that provide the required floating-point operations. To date ARM has produced two such coprocessor Floating-Point Accelerator (FPA), and Vector Floating-Point (VFP). VFP implements IEEE single precision, double precision, but does not support extended double precision.
* **AAPCS Floating-Point Support: Software Floating-Point Emulation (FPE)**, where code is still generated to use coprocessor floating-point instructions, but the actual coprocessor hardware does not exist in the system to implement them. Instead and emulation of the coprocessor is provided which is attached to the ARM processor core’s undefined instruction trap.
* **ARM Procedure Call Standard (APCS)** defines how to pass function arguments and return values in ARM registers.

If there are less than 5 arguments, the first four integer arguments are passed to the first four ARM registers r0 to r3.

Else there are more than 4 arguments, subsequent integer arguments are placed on the descending stack.

The integer value is returned in r0, and the long long or double typed value is returned using r0 and r1.

* **Codes: Leaf Procedure**

|  |
| --- |
| #include <stdio.h>  int checksum(int a[]){  int b =0;  int i;  **for**(i = 0 ; i < 5; i++)  b = b+a[i];  **return** b;  }  int main(int argc, char\*\* argv){  int f ;  int a[] = {0, 1, 2, 3, 4, 5,};  f = checksum(a);  printf("%d", f);  } |
| checksum PROC  MOV r2, r0  MOV r0, #0  MOV r1, r0  |L0.12|  LDR r3, [r2, r1, LSL #2]  ADD r1, r1, #1  CMP r1, #5  ADD r0, r0, r3  BLT |L0.12|  BX lr  ENDP  main PROC  PUSH {lr}  SUB sp, sp, #0x1c ; 28  LDR r1, |L0.92|  MOV r2, #0x18 ; 24  MOV r0, sp  BL \_\_aeabi\_memcpy4  MOV r0, sp  BL checksum  MOV r1, r0  ADR r0, |L0.96|  BL \_\_2printf  MOV r0,#0  ADD sp,sp,#0x1c ; 28  POP {pc}  ENDP  |L0.92|  DCD ||.constdata||  |L0.96|  DCB "%d", 0  DCB 0 |

* **Codes: Argument Passing**

|  |
| --- |
| #include <stdio.h>  int sum\_elements(  int a, int b, int c , int d, int e){  **return** (a+b+c+d+e) ;  }  int main(int argc, char\*\* argv){  int a, b, c, d, e, f ;  a = 0 ;  b = 1 ;  c = 2;  d = 3 ;  e = 4 ;  f = sum\_elements(a, b, c, d, e) ;  printf("%d", f);  } |
| sum\_elements PROC  ADD r0, r0, r1  LDR r12, [sp, #0]  ADD r0, r0, r2  ADD r0, r0, r3  ADD r0, r0, r12  BX lr  ENDP  main PROC  PUSH {r3, lr}  MOV r12, #4  MOV r0, #0  MOV r1, #1  MOV r2, #2  MOV r3, #3  STR r12, [sp, #0]  BL sum\_elements  MOV r1, r0  ADR r0, |L0.76|  BL \_\_2printf  MOV r0, #0  POP {r3, pc}  ENDP  |L0.76|  DCB "%d", 0  DCB 0 |

* **Codes: Non-leaf Procedure**

|  |
| --- |
| #include <stdio.h>  int fact(int n) {  **if** (n < 1)  **return** 1;  **else**  **return** n \* fact(n - 1);  }  int main(int argc, char\*\* argv) {  int a;  a = fact(10);  printf("%d", a);  } |
| fact PROC  PUSH {r4, lr}  CMP r0, #1  MOV r4, r0  MOVLT r0, #1  POPLT {r4, pc}  SUB r0, r4, #1  BL fact  MUL r0, r4, r0  POP {r4, pc}  ENDP  main PROC  PUSH {r4, lr}  MOV r0, #0xa  BL fact  MOV r1, r0  ADR r0, |L0.68|  BL \_\_2printf  MOV r0, #0  POP {r4, pc}  ENDP  |L0.68|  DCB "%d", 0  DCB 0 |

* **Codes: printf with Many Arguments**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  int main(int argc, char\*\* argv) {  int a, b, c, d, e;  int i;  srand(time(NULL)) ;  a = rand() ;  b = rand() ;  c = rand() ;  d = rand() ;  e = rand() ;  printf("%d %d %d %d %d", a,b,c,d,e);  } |
| main PROC  PUSH {r2 - r8, lr}  MOV r0, #0  BL time  BL srand  BL rand  MOV r5, r0  BL rand  MOV r6, r0  BL rand  MOV r7, r0  BL rand  MOV r4, r0  BL rand  STR r0, [sp, #4]  MOV r3, r7  MOV r2, r6  MOV r1, r5  ADR r0, |L0.88|  STR r4, [sp,#0]  BL \_\_2printf  MOV r0, #0  POP {r2 - r8, pc}  ENDP  |L0.88|  DCB "%d %d %d %d %d", 0  DCB 0 |

* **Codes: Arguments with Floating-Numbers**

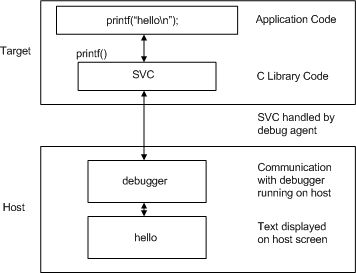
|  |
| --- |
| #include <stdio.h>  double sum\_elements(double a, float b){  **return** (a+b) ;  }  int main(int argc, char\*\* argv){  double a;  float b;  double f ;  a = 1;  b = 2;  f = sum\_elements(a, b) ;  printf("%lf", f);  } |
| sum\_elements PROC  PUSH {r4 - r6, lr}  MOV r6, r0  MOV r5, r1  MOV r0, r2  BL \_\_aeabi\_f2d  MOV r2, r6  MOV r3, r5  POP {r4 - r6, lr}  B \_\_aeabi\_dadd  ENDP  main PROC  MOV r2, #0x40000000  PUSH {r4,lr}  MOV r0, #0  SUB r1, r2, #0x100000  BL sum\_elements  MOV r2, r0  MOV r3, r1  ADR r0,|L0.80|  BL \_\_2printf  MOV r0,#0  POP {r4,pc}  ENDP  |L0.80|  DCB "%lf", 0 |

* **Codes: Matrix Multiplication**

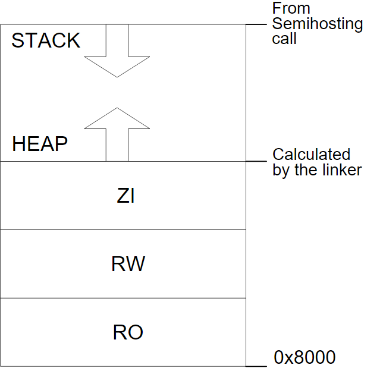
|  |
| --- |
| #include<stdio.h>  int main() {  int a[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}} ;  int b[3][3] = {{9,8,7}, {6,5,4}, {3,2,1}} ;  int c[3][3];  int i, j, k;  int sum = 0;  **for** (i = 0; i <= 2; i++) {  **for** (j = 0; j <= 2; j++) {  sum = 0;  **for** (k = 0; k <= 2; k++) {  sum = sum + a[i][k] \* b[k][j];  }  c[i][j] = sum;  }  }  printf("**\n**Multiplication Of Two Matrices : **\n**");  **for** (i = 0; i < 3; i++) {  **for** (j = 0; j < 3; j++) {  printf(" %d ", c[i][j]);  }  printf("**\n**");  }  } |
| main PROC  PUSH {r4-r9, lr}  SUB sp, sp, #0x6c  LDR r1, |L0.220|  MOV r2, #0x24  MOV r0, sp  BL \_\_aeabi\_memcpy4  LDR r1, |L0.224|  MOV r2, #0x24  ADD r0, sp, #0x24  BL \_\_aeabi\_memcpy4  MOV r3, #0  MOV r9, sp  ADD r8, sp, #0x24  ADD r6, sp, #0x48  |L0.56|  ADD r0, r3, r3, LSL #1  MOV r1, #0  ADD r5, r9, r0, LSL #2  ADD r7, r6, r0, LSL #2  |L0.72|  MOV r2, #0  MOV r0, r2  |L0.80|  ADD r4, r0, r0, LSL #1  LDR r12, [r5, r0, LSL #2]  ADD r4, r8, r4, LSL #2  LDR r4, [r4, r1, LSL #2]  ADD r0, r0, #1  MLA r2, r12, r4, r2  CMP r0, #2  BLE |L0.80|  STR r2, [r7, r1, LSL #2]  ADD r1, r1, #1  CMP r1, #2  BLE |L0.72|  ADD r3, r3, #1  CMP r3, #2  BLE |L0.56|  ADR r0, |L0.228|  BL \_\_2printf  MOV r5, #0  |L0.152|  ADD r0, r5, r5, LSL #1  MOV r4, #0  ADD r7, r6, r0, LSL #2  |L0.164|  LDR r1, [r7, r4, LSL #2]  ADR r0, |L0.264|  BL \_\_2printf  ADD r4, r4, #1  CMP r4, #3  BLT |L0.164|  ADR r0, |L0.272|  BL \_\_2printf  ADD r5, r5, #1  CMP r5, #3  MOVGE r0, #0  BLT |L0.152|  ADD sp, sp, #0x6c  POP {r4-r9, pc}  ENDP  |L0.220|  DCD ||.constdata||  |L0.224|  DCD ||.constdata||+0x24  |L0.228|  DCB "**\n**Multiplication Of Two Matrices : **\n**", 0  |L0.264|  DCB " %d ", 0  DCB 0  DCB 0  DCB 0  |L0.272|  DCB "**\n**", 0  DCB 0  DCB 0    AREA ||.constdata||, DATA, READONLY, ALIGN=2  DCD 0x00000001  DCD 0x00000002  DCD 0x00000003  DCD 0x00000004  DCD 0x00000005  DCD 0x00000006  DCD 0x00000007  DCD 0x00000008  DCD 0x00000009  DCD 0x00000009  DCD 0x00000008  DCD 0x00000007  DCD 0x00000006  DCD 0x00000005  DCD 0x00000004  DCD 0x00000003  DCD 0x00000002  DCD 0x00000001 |

* **Codes: Bubble sort**

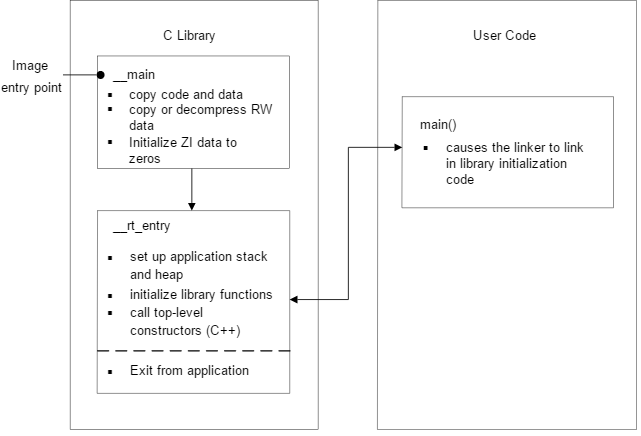
|  |
| --- |
| #include <stdio.h>  #include <math.h>  #include <stdlib.h>  #include <time.h>  void bubbleSort(int\* number, int n){  int temp, i, j ;  **for**(i = 1; i < n; i++) {  **for**(j = 0; j < n-i; j++ ) {  **if**(number[j] > number[j + 1]) {  temp = number[j];  number[j] = number[j + 1];  number[j + 1] = temp;  }  }  }  }  void printArray(int\* number, int n) {  int i;  **for**(i = 0; i < n; ++i) {  printf("number[%d] = %d**\n**", i, number[i]);  }  }  int main(){  int n = 20;  int A[20] ;  int i ;  srand(time(NULL));  **for**(i = 0; i < n; i++) {  A[i] = rand();  }  printf("Unsorted Array: **\n**");  printArray(A, n);  bubbleSort(A, n);  printf("--------------------**\n**");  printf("Sorted Array:**\n**");  printArray(A, n);  **return** 0;  } |
| bubbleSort PROC  PUSH {r4 - r6, lr}  MOV r3, #1  |L0.8|  CMP r1, r3  MOVGT r2, #0  SUBGT r6, r1, r3  POPLE {r4 - r6, pc}  |L0.24|  CMP r6, r2  ADDLE r3, r3, #1  BLE |L0.8|  ADD r4, r0, r2, LSL #2  LDR r12, [r0, r2, LSL #2]  LDR r5, [r4, #4]  CMP r12, r5  STRGT r5, [r0, r2, LSL #2]  ADD r2, r2, #1  STRGT r12, [r4, #4]  B |L0.24|  ENDP  printArray PROC  PUSH {r4-r6, lr}  MOV r6, r1  MOV r5, r0  MOV r4, #0  B |L0.108|  |L0.88|  LDR r2, [r5, r4, LSL #2]  MOV r1, r4  ADR r0, |L0.244|  BL \_\_2printf  ADD r4, r4, #1  |L0.108|  CMP r4, r6  BLT |L0.88|  POP {r4 - r6, pc}  ENDP  main PROC  PUSH {r4 - r6, lr}  SUB sp, sp, #0x50  MOV r5, #0x14  MOV r0, #0  BL time  BL srand  MOV r4, #0  MOV r6, sp  |L0.152|  BL rand  STR r0, [r6, r4, LSL #2]  ADD r4, r4, #1  CMP r4, r5  BLT |L0.152|  ADR r0, |L0.264|  BL \_\_2printf  MOV r1, r5  MOV r0, sp  BL printArray  MOV r1, r5  MOV r0, sp  BL bubbleSort  ADR r0, |L0.284|  BL \_\_2printf  ADR r0, |L0.308|  BL \_\_2printf  MOV r1, r5  MOV r0, sp  BL printArray  MOV r0, #0  ADD sp, sp, #0x50  POP {r4 - r6, pc}  ENDP  |L0.244|  DCB "number[%d] = %d**\n**", 0  DCB 0  DCB 0  DCB 0  |L0.264|  DCB "Unsorted Array: **\n**", 0  DCB 0  DCB 0  |L0.284|  DCB "--------------------**\n**", 0  DCB 0  DCB 0  |L0.308|  DCB "Sorted Array:**\n**", 0  DCB 0 |

* **Semihosting** is a mechanism that enables code running on ARM target to communicate and use the Input/Output facilities on a host computer that is running a debugger

In the process of moving from an “out-of-the-box” build to a standalone embedded application, several issues need to be considered: C Library use of HW / Target memory map / Application startup.

* **Default Memory Map**

By default code is linked to load and execute at 0x8000. The heap is placed directly above the data region. The stack base location is read from the debugging environment by C library startup code.

* **Application Startup**

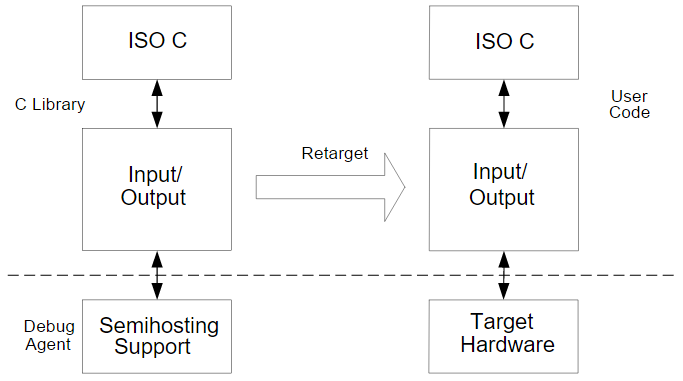
\_\_main is responsible for setting up the memory and \_\_rt\_entry is responsible for setting up the run-time environment.

\_\_main performs code and data copying, decompression, and zero initialization of the ZI data. It then branches to \_\_rt\_entry to set up the stack and heap, initialize the library functions and static data, and call any top level C++ constructors. \_\_rt\_entry then branches to main(), the entry to your application. When the main application has finished executing, \_\_rt\_entry shuts down the library, then hands control back to the debugger.

The function label main() has a special significance. The presence of a main() function forces the linker to link in the initialization code in \_\_main and \_\_rt\_entry. Without a function labeled main() the initialization sequence is not linked in, and as a result, some standard C library functionality is not supported.

* **Retargeting the C Library**

You can replace the C library’s device driver level functionality with an implementation that is tailored to your target hardware. e.g.) printf() should go to LCD screen, not debugger console.



To “Retarget” the C library, simple replace those C library functions which use Semihosting with your own implementations, to suite your system.

e.g.) The printf() family of functions (except sprint()) all ultimately call fputc(). The default implementation of fputc() uses Semihosting. Replace this with:

|  |
| --- |
| **extern** void sendchar(char \*ch);  int fputc(int ch, FILE \*f){  */\* e.g. write a character to an LCD screen \*/*  char tempch = ch;  sendchar(&tempch);  **return** ch;  } |

* **Avoiding C library Semihosting**

To ensure that no functions which use semihosting are linked in from C library, import the ‘guard’ symbol

|  |
| --- |
| #paragam import(\_\_use\_no\_semihosting) |

If there are still semihosting functions being linked in, the linker will report

|  |
| --- |
| Error: L6915E: Library reports error: \_\_use\_no\_semihosting was requested but <function> was referenced |

* **Scatterloading** defines two types of memory region.

Load Regions contain application code & data at reset/load time (typically ROM).

Execution Regions contain code and data while the application is executing. One or more execution regions will be created from each load region during application startup.

The details of the memory map used by a scatterload application are contained in a description file which is passed as a parameter to armlink. e.g.)

|  |
| --- |
| $ armlink program.o –scatter scatter.scat -o program.axf |

|  |  |  |
| --- | --- | --- |
| Load View | 0x18000 | Execute View |
|  |  |
| RAM | ZI |
| 0x10000 | RW |
|  |  |
| 0x04000 |
|  |  |
| RW | ROM |
| RO | 0x00000 | RO |
|  |  |

RO code and data stays in ROM. C library initialization code (in \_\_main) will Copy/Decompress RW data from ROM to RAM and initialize the ZI data in RAM to zero.

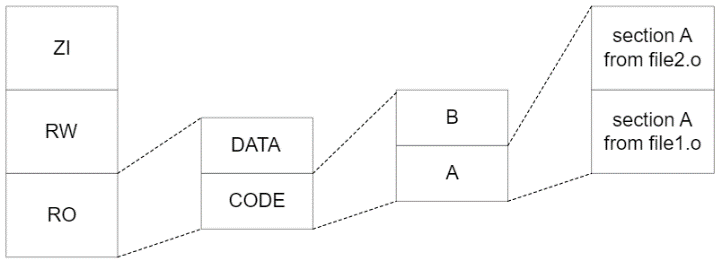
|  |
| --- |
| LOAD\_ROM 0x000 0x4000 {  EXEC\_ROM 0x0000 0x4000{  \*(+RO)  }  RAM 0x10000 0x8000{  \*(+RW, +ZI)  }  } |

The wildcard (\*) syntax allows for easy grouping of code and data.

Scatter description files can be pre-processed.

* **Linker Placement Rules**

The basic ordering is organized by attribute. RO precedes RW which provides ZI. With the same attribute, code is placed before data.

Further ordering is determined by input section name in alphabetical order, then by the order objects are specified on the armlink command line.

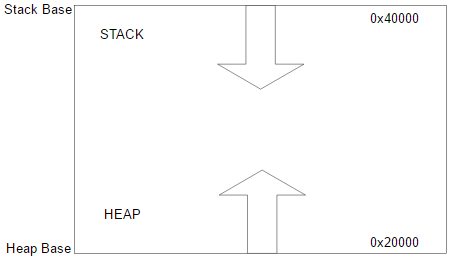
* **Ordering Objects in a Scatter File**

Use the +FIRST and +LAST directives to place individual objects first and last in an execution region. e.g.) to place the vector table at the beginning of a region.

|  |
| --- |
| ROM\_LOAD 0x000 0x4000{  ROM\_EXEC 0x0000 0x4000{  vectors.o(Vect, +RIST)  \*(+RO)  }  ...  } |

The ordering of objects in the scatter file execution region does not affect ordering in the output image.

* **Run-Time Memory Models: One-Region Model**



|  |
| --- |
| LOAD\_FLASH ... {  ...  ARM\_LIB\_STACKHEAP 0x20000 EMPTY 0x20000 {}  ...  } |

To implement a two-region model, import \_\_use\_two\_region\_memory

* **Root Region** is an execution region whose load address is equal to its execution address. Each scatter description must have at least one root region.

Some C library code (e.g., \_\_main.o, etc) and linker-generated tables (e.g., Region&&Table) must be placed in a root region. Otherwise the linker will report:

|  |
| --- |
| Error: L6202E: Section Region&&Table cannot be assigned to a non-root region |

Forward-compatible way to specify these is:

|  |
| --- |
| LOAD\_FLASH 0x4000{  EXE\_ROM 0x4000{  \*(InRoot&&Section)  ... |

Alternatively,

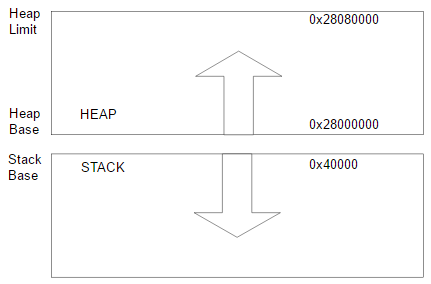
|  |
| --- |
| LOAD\_FLASH 0x4000{  EXE\_ROM 0x4000{  \_\_main.o(+RO) ; entry point.  \_\_scatter\*.o(+RO) ; copy/zero code.  \_\_dc\*.o(+RO) ; decompression code.  \*(Region$$Table) ; addresses of region  ... ; to copy/zero. |

If \*(+RO) is located in a root region, the above will be located there automatically.

The main application entry point must also lie in a root region. If not, linker will report:

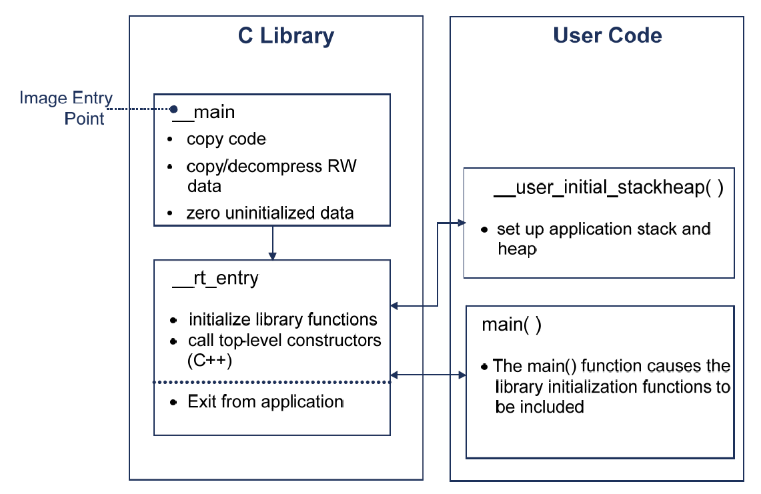
|  |
| --- |
| Error: L6203E: Entry point (0x4000) lies within non-root region |

* **Run-Time Memory Models: Two-Region Model**



|  |
| --- |
| LOAD\_FLASH ... {  ...  ARM\_LIB\_STACK 0x20000 EMPTY -0x20000 {}  ARM\_LIB\_HEAP 0x28000000 EMPTY 0x80000 {}  ...  } |

* **Run-Time Memory Management**



To place the stack and heap, you can re-implement \_\_user\_initial\_stackhep(). It may be written in C or assembly. It should return HB in r0, SB in r1, HL in r2

|  |
| --- |
| #include <rt\_misc.h>  \_\_value\_in\_regs **struct** \_\_initial\_stackheap \_\_user\_initial\_stackheap(  unsigned R0, unsigned SP, unsigned R2, unsigned SL){  **struct** \_\_initial\_stackheap config;  config.heap\_base = (unsigned int)Image$$HEAP$$ZI$$Base;  config.stack\_base = (unsigned int)Image$$STACKS$$ZI$$Base;  config.heap\_limit = (unsigned int)Image$$HEAP$$ZI$$Limit;  config.stack\_limit = (unsigned int)Image$$STACKS$$ZI$$Limit;  **return** config;  } |
| EXPORT \_\_user\_initial\_stackheap  \_\_user\_initial\_stackheap  LDR r0, = Image%%ZI$$Limit ; HB  LDR r1, = $top\_of\_memory ; SB  MOV pc, lr |

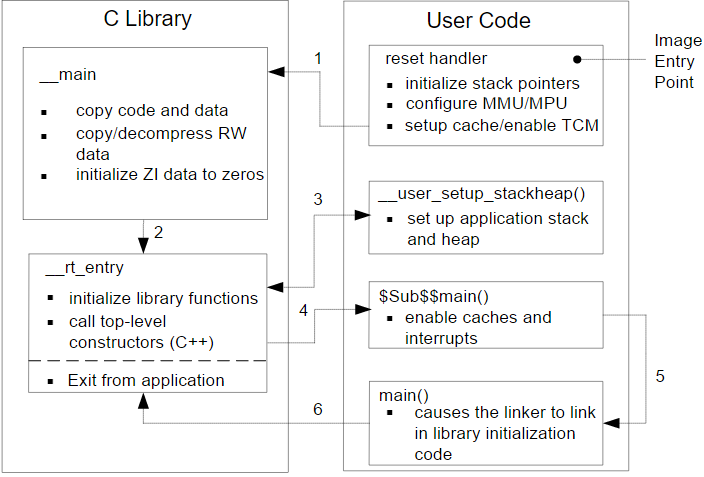
Heap Limit is not used in one region model.

* **Vector Table**

|  |
| --- |
| AREA Vectors, CODE, READONLY  IMPORT Reset\_Handler  ENTRY  B Reset\_Handler  B Undefined\_Handler  B SWI\_Handler  B Prefetch\_Handler  B Abort\_Handler  NOP ; Reserved vector  B IRQ\_Handler  B FIQ\_Handler  END |

Locate this table at 0x000 (or 0xFFFF0000) using the +FIRST directive. ENTRY directive tells linker that this is entry point.

* **Initialization Steps**



* **Long Branch Veneers**

|  |  |
| --- | --- |
| int main(){  farfunc();  }  ...  void farfunc(){  ...  } | ROM\_LOAD {  ROM\_EXEC 0x000{  \*(+RO)  }  RAM 0x80000000{  farfunc.o(+RO)  \*(+RW, +ZI)  }  } |
| 0x00000000 bl Ven$AA$L$$Sfarfunc    ..    Ven$AA$L$$Sfarfunc  ldr pc, [pc, #-4]  dcd 0x80000000  ..  0x80000000 ..  bx lr | |

The linker can automatically add Long Branch Veneers, so that ‘far’ functions can be called successfully.

* **Endianness**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data Bus Pins | Byte Accesses | | | Halfword Accesses | | | Word Accesses | | |
| LE | BE-8 | BE-32 | LE | BE-8 | BE-32 | LE | BE-8 | BE-32 |
| 63:56 | A7 | A7 | A4 | A6:MS | A6:LS | A4:MS | A4:MS | A4:LS | A4:MS |
| 55:48 | A6 | A6 | A5 | A6:LS | A6:MS | A4:LS | A4:MS -1 | A4:LS +1 | A4:MS -1 |
| 47:40 | A5 | A5 | A6 | A4:MS | A4:LS | A6:MS | A4:LS +1 | A4:MS -1 | A4:LS +1 |
| 39:32 | A4 | A4 | A7 | A4:LS | A4:MS | A6:LS | A4:LS | A4:MS | A4:LS |
| 31:24 | A3 | A3 | A0 | A2:MS | A2:LS | A0:MS | A0:MS | A0:LS | A0:MS |
| 23:16 | A2 | A2 | A1 | A2:LS | A2:MS | A0:LS | A0:MS -1 | A0:LS +1 | A0:MS -1 |
| 15:8 | A1 | A1 | A2 | A0:MS | A0:LS | A2:MS | A0:LS +1 | A0:MS -1 | A0:LS +1 |
| 7:0 | A0 | A0 | A3 | A0:LS | A0:MS | A2:LS | A0:LS | A0:MS | A0:LS |

* **Exception** is any condition that needs to halt the normal sequential execution of instructions.
* **Exceptions of ARM processor**

Reset (RESET) / Undefined Instruction (UNDEF) / Software Interrupt (SWI) / Prefetch Abort (PABT) / Data Abort (DABT) / Interrupt Request (IRQ) / Fast Interrupt Request (FIQ)

* **Exception Handling Process**

1. Save Processor Status

Copies CPSR into SPSR\_<mode>  
Stores the return address in LR\_<mode>

1. Change Processor Status for Exception

Mode field bits  
ARM or Thumb (T2) state  
Interrupt disable bits (if appropriate)  
Sets PC to vector address

1. Execute Exception Handler

<users code>

1. Return to Main Application

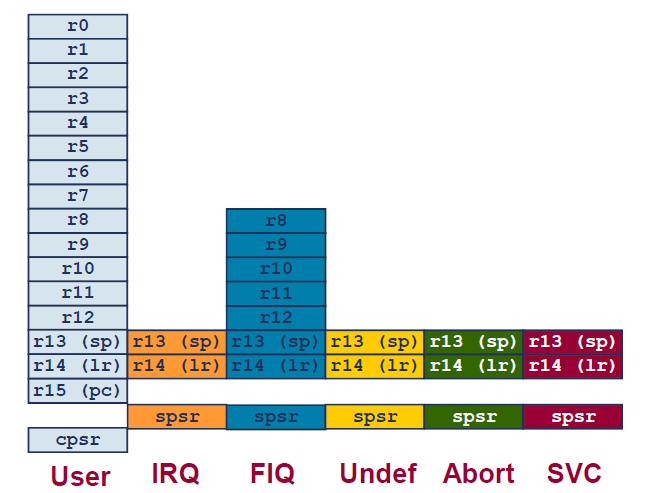
Restore CPSR from SPSR\_<mode>  
Restore PC from LR\_<mode>

1 and 2 performed automatically by the core.  
3 and 4 responsibility of software.

* **Exception Priorities and Modes**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Except | Mode | I | F | Main purpose |
| 1 | Reset | SVC | 1 | 1 | Handling under operating system |
| 2 | DABT | abort | 1 | 0 | Handling memory protection |
| 3 | FIQ | FIQ | 1 | 1 | Fast interrupt request handling. |
| 4 | IRQ | IRQ | 1 | 0 | General interrupt request handling |
| 5 | PABT | abort | 1 | 0 | Handling memory protection |
| 6 | SWI | SVC | 1 | 0 | Handling under operating system |
| 6 | UNDEF | undef | 1 | 0 | Handling undefined instruction |

* **ARM Register Set and Modes**



ARM has 37 registers: PC (1) / CPSR (1) / SPSR (5) / General purpose register (30) / Banked register (20)

Execution modes have some private registers which are banked in when the mode is changed.

Non-banked registers are shared between modes

* **Vector Table** is a table of addresses that the ARM core branches to when an exception occurs.

|  |
| --- |
| B label |
| LDR pc, [pc, $offset] |
| MOV pc, #immediate |

Branch instructions contained in the vector table. Do not use a BL (as it will corrupt the link register)

* **Link Register Offsets**

|  |  |  |
| --- | --- | --- |
| Excep | Addr | Description |
| Reset | - | lr 값이 정의되지 않음 |
| DABT | lr - 8 | DABT 예외를 발생시킨 명령어를 가리킴 |
| FIQ | lr - 4 | FIQ 핸들러로 부터의 복귀 주소 |
| IRQ | lr - 4 | IRQ 핸들러로 부터의 복귀 주소 |
| PABT | lr - 4 | PABT 예외를 발생시킨 명령어를 가리킴 |
| SWI | lr | SWI 명령어 다음 명령어를 가리킴 |
| UNDEF | lr | Undef Instruction 다음 명령어를 가리킴 |

* **Interrupt latency**

외부에서 interrupt가 발생했을 때 interrupt service routine의 첫 명령어를 수행할 때까지의 시간.

* **Low-latency Interrupt Mode**

Accesses to normal memory can be abandoned. Interrupt taken immediately and then abandoned access is repeated after interrupt returns.

* **ARM Hardware Interrupts**

Exception raised by an external peripheral.

IRQ: General-purpose interrupts.

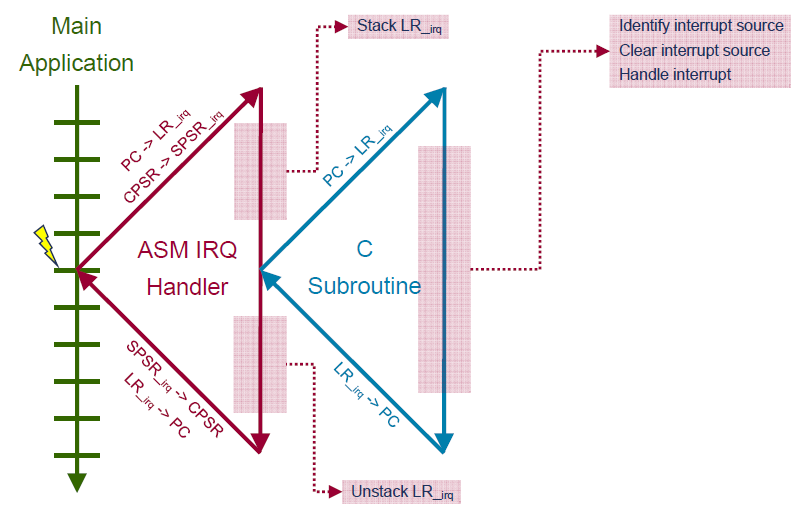
FIQ: Interrupts sources that require a fast response time.

* **ARM Software Interrupts**

A specific instruction that causes an exception.

SQI: Used to call privileged operating system calls.

* **Simple Interrupt Example**



|  |
| --- |
| IRQ\_Handler  PUSH {r0-r3, r12, lr}  BL identify\_and\_clear\_source  BL C\_irq\_handler  POP {r0-r3, r12, lr}  SUBS pc, lr #4 |

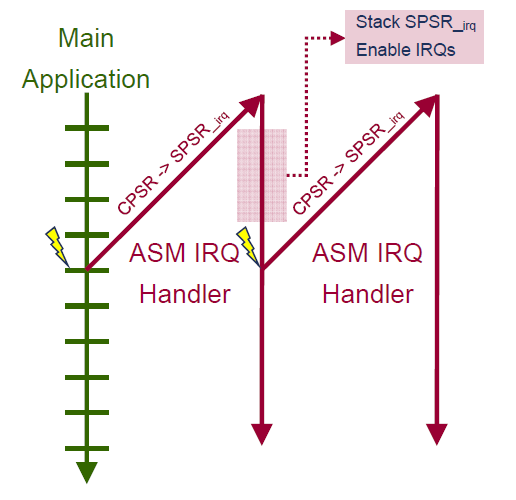
* **Interrupt Pre-emption**

When IRQ request receive a while serving another, the new one disabled on entry to handlers.

A new IRQ will no pre-empt the current handler unless explicitly re-enabled. IRQs are automatically re-enabled when handler returns.

* **Issues with Re-enabling Interrupts: PSR**

If interrupts are nested, the programmer must take special steps to prevent the core state from being lost.



SPSR needs to be preserved before interrupts are re-enabled. If not, new IRQ will overwrite SPSR\_irq

So, before re-enabling IRQs, stack the SPSR by using one of the following

|  |
| --- |
| SRSFD sp!, #0x12 |
| MRS r0, SPSR  PUSH r0 |

SRS store return state onto a stack

SRS{addr\_mode}{cond} sp{!}, #modenum

addr\_mode for IA|IB|DA|DB

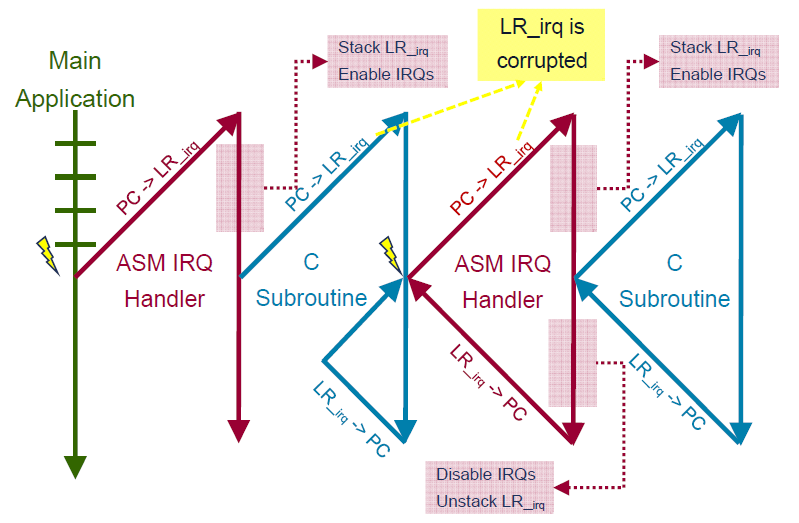
! is present, the final address is written back into the SP of the mode specified by modenum,

modenum specifies the number of the mode whose banked SP is used as the base register.

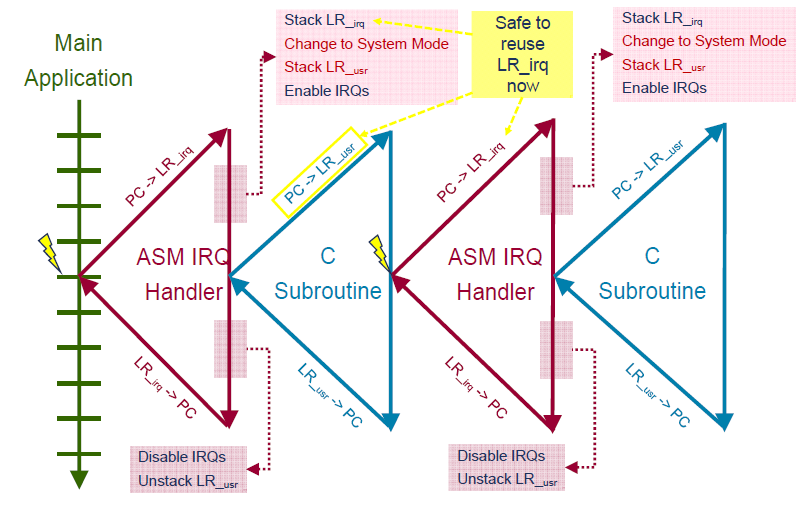
* **Processor Mode**

|  |  |  |  |
| --- | --- | --- | --- |
| Mode | CPSR | Mode | CPSR |
| USER | b10000 | ABORT | b10111 |
| FIQ | b10001 | UNDEF | b11011 |
| RIQ | b10010 | SYS | b11111 |
| SVC | b10011 |  |  |

* **Issues with Re-enabling Interrupts: LR**



BL with interrupt handlers will cause LR\_irq corruption. So, change to SVC mode before re-enabling IRQs (BL uses LR\_svc).



* **Interrupt Handling Schemes**

중첩을 허용하지 않는다는 것은 각 인터럽트들을 순서대로 하나씩 처리하는 것이다. 따라서 인터럽트가 처리되는 동안 다른 인터럽트를 처리할 수 없기 때문에 인터럽트 지연이 크다. 구현이 간단하지만 복잡한 임베디드 시스템을 처리하는 데 사용 불가하다.

중첩이 허용된다면 인터럽트 지연이 적어진다. Last Come, First Served 방식의 순차적으로 처리한다.

현재 처리중인 인터럽트보다 높은 우선순위의 인터럽트에 대해서만 중첩 인터럽트 처리를 허용하는 중첩과 우선순위를 함께 적용한 인터럽트 처리 방식은 인터럽트 지연을 가장 최소화할 수 있다. 높은 우선순위의 인터럽트는 낮은 순위의 인터럽트보다 지연이 적기 때문이다.

* **Stack Issues**

AAPCS requires 8-byte stack alignments at all external boundaries. Need to ensure we maintain this when calling a C subroutine from ASM IRQ handler. But the SVC mode SP may not be 8-byte aligned when interrupt occurs.

Therefore, if switching to SVC mode, handler should check the stack alignment from SVC mode. Correct the stack alignment if necessary, and flag that you need to “undo” this change before returning. the stack must be aligned before re-enabling interrupts.

* **Nested Interrupt Example**

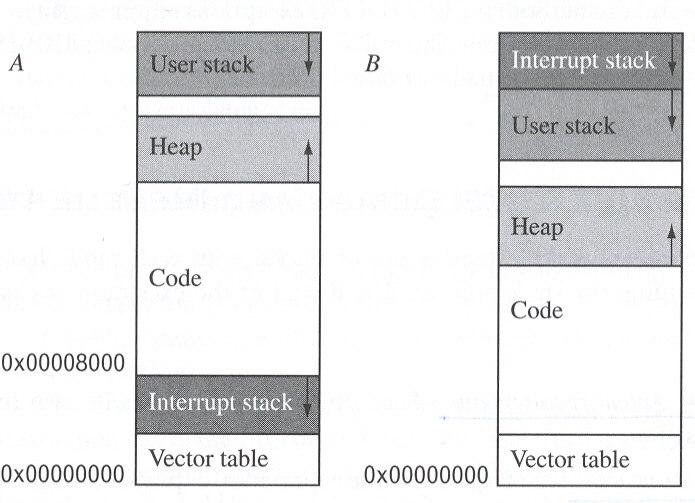
|  |
| --- |
| IRQ\_Handler  SUB lr, lr #4  SPSFD sp!, #0x13  CPS #0x13  PUSH {r0-r3, r12}  AND r1, sp, #4  SUB sp, sp, r1  PUSH {r1, lr}  BL identify\_and\_clear\_source  CPSIE i  BL C\_irq\_handler  CPSID i  POP {r1, lr}  ADD sp, sp, r1  POP {r0-r3, r12}  RFEFD sp! |

* FIQ vs IRQ

FIQs are designed to service interrupts as quickly as possible. Serviced first when multiple interrupts arise. Servicing and FIQ disables IRQs (and further FIQs). IRQ will not be serviced until FIQ handler exits. FIQ vector is last in vector table that allows handler to be run sequentially from that address.

However, most ARM based systems have more than 2 interrupt sources. Therefore, need an interrupt controller (typically memory mapped) to control how interrupts are passed to the ARM. In many systems some interrupts will be higher priority than others – these will need to pre-empt any other priority interrupts handlers.

* **Basic Interrupt Stack Design**



A에 비해 B 방식은 Stack Overflow가 발생하여도 Vector Table을 유지한다.

* **Prefetch & Data Aborts**

Prefetch abort indicates a failed instruction fetch. Tagged as aborting when the fetch occurs. Abort only taken if instruction reaches the execute stage of the pipeline.

Data Abort indicates a failed data access. Load/Store between the core and memory system. Protection faults.

When an abort happens, it depends on the system. In a simple system without any memory management, this usually indicates a serious error.

With memory management, it may need to identify the cause of the abort and take corrective action.

E.g.) Allocate more memory for a process / Load a new page of code or data which a process was trying to access

* **Data Abort Types**

Internal Aborts are those from the core itself. MMU faults may indicate that we need to map in more memory and re-execute the appropriate instruction.

External Aborts are those from the memory system.

Precise data aborts are where we know the address of the instruction that caused the data abort.

* **Identifying the Abort Source**

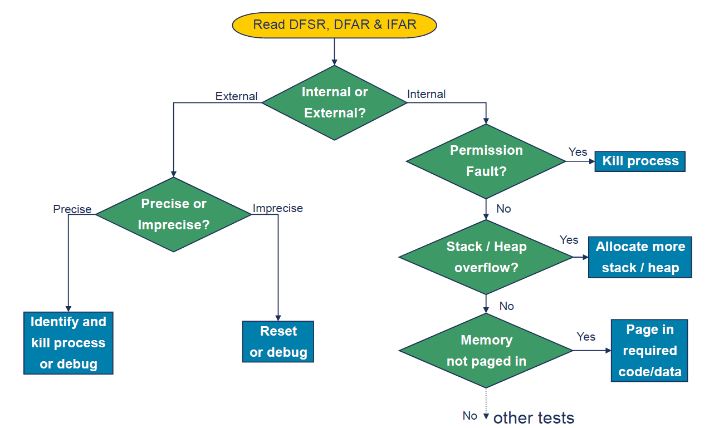
To handle a Data Abort. It needs the reason that abort occurred and the address of the data access that caused the abort.

Use MMU/MPU Fault Status Register (FSR) and Fault Address Register (FAR).

Reason for abort is given in the FSR.  
E.g.) External abort probably fatal.

Aborting address is given in FAR/DFAR.

* **Example Data Abort Handler**

****

* **Reset**

The operations carried out by a Reset handler depend upon the system in question.

For example, the reset handler may:  
- Set up exception vectors.  
- Initialize the memory system (e.g. MMU/MPU)  
- Initialize all required processor mode stacks and registers.  
- Initialize variables required by C  
- Initialize any critical I/O devices  
- Enable interrupts  
- Change processor mode and/or state  
- Call the main application.