Exception Handlers

Young Won Lim
11/30/21

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

Introduction to ARM Cortex-M Microcontrollers
– Embedded Systems, Jonathan W. Valvano

Digital Design and Computer Architecture,
D. M. Harris and S. L. Harris

ARM assembler in Raspberry Pi
Roger Ferrer Ibáñez

https://thinkingeek.com/arm-assembler-raspberry-pi/

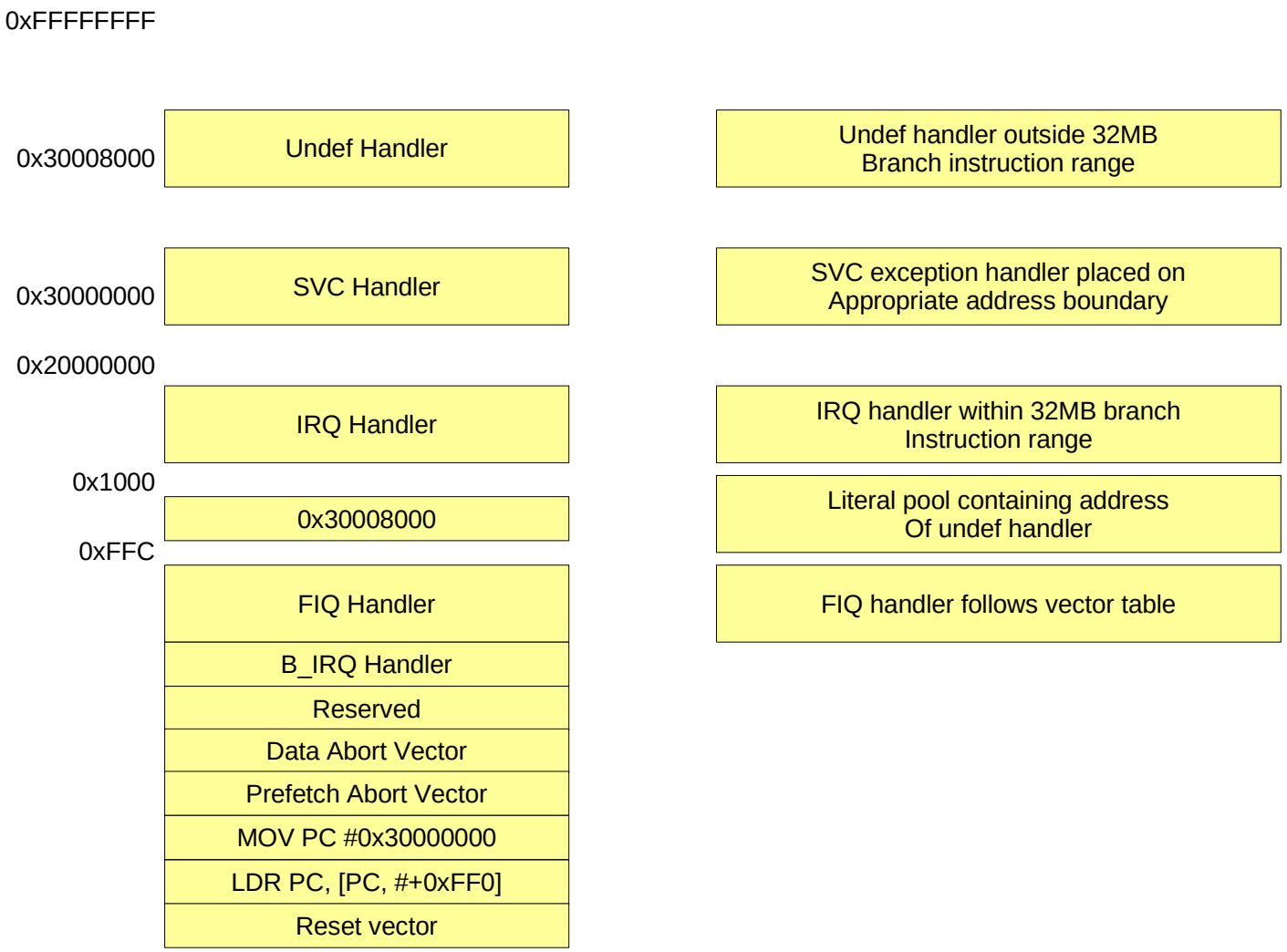
Finding the right handler

For the different kinds of exceptions, there are different handlers. When an exception occurs, the hardware determines the source of the exception as a 3-bit number, which it uses to index the vector table (which starts in memory at address 0)

0x1c	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0c	Prefetch Abort
0x08	SWI
0x04	Undefined Instruction
0x00	Reset

http://www2.unb.ca/~owen/courses/2253-2017/slides/08-interrupts.pdf

Finding the right handler



http://www2.unb.ca/~owen/courses/2253-2017/slides/08-interrupts.pdf

Prefetch and Data Aborts

A 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

- a data abort indicates a failed data access
- load/store between the core and memory system

- internal aborts are those from the core itself - the MMU/MPU
- MMU faults may indicate you need to take corrective action and
  re-execute the appropriate instruction

- external aborts are those from the memory system
  - may indicate a hardware fault
  - could be an attempted access to non-existent memory

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Abort Handlers (1)

- when an abort happens depends on the system
  - in a simple system without any memory management,
    this usually indicates a serious error (e.g. hardware fault, code bug)
  - With memory management, you may need to
    - identify the cause of the abort
    - and take corrective action.
    - for example:
      - allocate more memory for a process
      - load a new page of code or data which the process was trying to access
      - terminate the process if it did not have permission to access the aborting address
- ARM7TDMI family devices have a different abort model - see appendix

- for a prefetch abort, the offending instruction is at LR_abt - 4
- for a data abort, the offending instruction is usually at LR_abt - 8
- However, use MMU to find the faulting address
- Instruction may not be at lr - 8 if it is an imprecise abort

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

SWI Invocation

- In assembler, a SWI is called
  with an appropriate SWI number: SWI 0x24
  - Note: SWI has now been renamed to SVC (RVDS 2.2 and later)
- Note: calling a SWI when in Supervisor mode will corrupt LR_svc
  - Solution: push LR_svc onto stack before calling the SWI
- Parameters are passed in:
  - The SWI number e.g. for semihosting,
    use 0x123456 (ARM) or 0xAB (Thumb)
  - Core registers
- In C, map a call to a function onto a SWI using the keyword "__swi"
  - Pass up to 4 parameters in r0-r3
  - Note: no stack parameters because of
    the change to supervisor mode

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Software Interrupts (1)

1) The SWI is invoked in the user program (either C or ASM).
   Parameters may have been placed in registers before the SWI.
   The SWI number is embedded in the SWI instruction itself.

2) The SWI vector is taken.
   This includes switch to SVC mode.

3) The vector points to a handler which is written in ASM.
   This is where the SWI number is extracted
   from the SWI instruction (by back-tracing through LR).
   Optionally, parameters may be placed on the stack
   prior to calling a C handler.
   If a C handler is not used,
   the parameters may be used directly from the registers.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Software Interrupts (1)

4) Optionally, a C handler may be used
to carry out the bulk of the SWI processing.
In order to be able to pass the SWI number
plus four other parameters to this routine
(more than we can usually pass in registers),
the SWI number is passed in a register together
with a pointer to the other parameters
which are placed on the stack prior to the call.
The C handler is a normal C function.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

SWI Invocation Example

```
__swi(0x24) void my_swi (char *s);

void foo (void)
{
   my_swi("Hello world\n");
}
```

```
foo
   STMFD       sp!,  {r4,lr}
   LDR         r0,    =text
   SWI         0x24
   LDMFD       sp!,  {r4,pc}
...
text DCB "Hello world\n",0
```

up to only four arguments for __swi functions.

Instead of using the user SP
Design the handler interface.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

SWI Invocation Example

inline assembler SWI call

SWI numbers can also be passed
using the keyword __swi_indirect

this passes an operation code
to the SWI handler in **r12**:

```
int __swi_indirect(swi_num)
        swi_name(int real_num,  int arg1, …, int argn);
```

swi_num :
the SWI number used in the SWI instruction.

real_num :
the value passed in **r12** to the SWI handler.
use this feature to implement indirect SWIs.
the SWI handler can use **r12**
to determine the function to perform.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

SWI Invocation Example

```
int __swi_indirect(0)
      ioctl(int swino, int fn, void *argp);
```

This SWI can be called as follows:

```
ioctl(IOCTL+4, RESET, NULL);
```

It compiles to a SWI 0 with IOCTL+4 in **r12**.

To use the indirect SWI mechanism,
your system SWI handlers must
make use of the **r12** value
to select the required operation.

(ADS1.2 Compilers and Libraries Guide
- section 3.1.2 - Function qualifiers)

BKPT can also be used for semihosting
on e.g. Cortex-M3

```
int __swi_indirect(swi_num)
      swi_name(int real_num,  int arg1, …);
```

swi_num :
the SWI number used in the SWI instruction.

real_num :
the value passed in **r12** to the SWI handler.
use this feature to implement indirect SWIs.
the SWI handler can use **r12**
to determine the function to perform.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

SWI Invocation Example

As ARM state is automatically entered
on taking an exception,
and an ARM instruction is needed
to return from an exception
: exception handlers in ARM code.

when performance is critical
and the exception handling code
can be located in fast 32-bit wide memory -
for instance with interrupt handlers.

Sometimes, it may be better
- the main part of an exception handler
  in Thumb code,
- entry and exit from the handler
  with an ARM code stub

better code density and also performance
if running from narrow memory.

this does apply generally to other exceptions,
but SWIs are the most common one
that you may want to use Thumb code in
(e.g. for a large handler for OS calls)

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Accessing the SWI number & arguments

- the core provides no mechanism
  for passing the SWI number directly
  to the handler
- SWI handler must extract the comment field
  from the SWI instruction itself
- to do this, the SWI handler must
  determine which state (ARM/Thumb)
  the SWI was called from
- Check the **T** bit in the SPSR
  The SWI instruction is at
  - LR-4 for ARM state,
  - LR-2 for Thumb state

- To pass the parameters to C,
  they are typically pushed on the stack
- Pass a pointer to those parameters
  to the C subroutine implementing
  the main body of the handler

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Example SWI Handlers

```
T_bit  EQU 0x20

SWI_Handler
    STMFD     sp!, {r0-r4,r12,lr}
    MOV       r1, sp

    MRS       r0, spsr
    STR       r0, [sp, #-4]!

    TST       r0, #T_bit
    LDRNEH    r0, [lr,#-2]
    BICNE     r0, r0, #0xff00
    LDREQ     r0, [lr,#-4]
    BICEQ     r0, r0, #0xff000000
```

```
; r0 now contains SWI number
; r1 now contains pointer to parameters on stack

    BL        C_SWI_Handler

    LDR       r1, [sp], #4
    MSR       spsr_csxf, r1
    LDMFD     sp!, {r0-r4,r12,pc}^
```

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Example SWI Handlers

This routine supports the passing back
of parameters as well. If the C handler modifies
the parameters on the stack, these modified values
will be passed back to the invocation in r0-r3.

Example C handler:

```
int C_SWI_Handler(int swinum, int * params)
{
    switch(swinum)
    {
    case 0:
      return writechar(params[0], params[1]);
      break;
    case 1:
      return readchar(params[0]);
      break;
    }
    return -1;
}
```

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Reset

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

for example it may:

- Set up exception vectors
- Initialize the memory system (e.g. MMU/PU)
- 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

Unlike the other exception handlers,
there should be no need to 'return' from the Reset handler
as it should call your main application.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Undefined Instruction (1)

an Undefined Instruction exception can occur if:

- The ARM tries to execute a real undefined instruction
- The ARM encounters a coprocessor instruction for which the appropriate coprocessor hardware does not exist in the system.
- The ARM encounters a coprocessor instruction for which the appropriate coprocessor hardware exists, but has not been enabled
- The ARM encounters a coprocessor instruction for which the appropriate coprocessor does exist but rejects the instruction because the ARM is not in a privileged mode.
- For example access to cp15 - the system control coprocessor Other privileged instructions

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Undefined Instruction (2)

Coprocessors may need to be enabled via cp15 register 1, opcode2 = 4

In the case of the non-existent coprocessor,
it is possible to write an undefined instruction handler
which emulates that coprocessor in software.

An emulator can examine the instruction
to see if it is one it can emulate.

If bits 27-24 = 1110 or 110x
then the instruction is a coprocessor instruction.

Can then extract bits 8-11 which define
which coprocessor should deal with this instruction.

If this emulator is the correct one,
then process the instruction and return to the application.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Undefined Instruction (2)

This is the way that floating point used to be done
on the ARM before the introduction of the floating point library.

Floating point would either be implemented in hardware, FPA
- Floating Point Accelerator, as used by ARM7500FE,
or in software using a Floating Point Emulator.

In reality the FPA actually used a combination of
hardware coprocessor for the common cases
and software emulation for the rare cases.

VFP10 (for use with ARM1020T) also uses
this combined hardware/software approach
for implementing floating point as coprocessor instructions.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

The “Return Address”

The way the return address is set comes from the ARM7 pipeline
Newer cores have the same effective behaviour for backwards-compatibility

In ARM state:

- Upon exception occurring the core sets LR_mode = PC - 4
- Handler may need to adjust LR_mode
  (depending on the exception which occurred)
  to return to the correct address

In Thumb state:

- The address stored in LR_mode is
  amended automatically by the processor
  depending upon the exception that has occurred
- This ensures the ARM return instruction from the handler
  will return to the correct address (and to the correct state),
  regardless of the state that was current
  when the exception occurred

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Return instruction (1) **SWI** and **Undef**

Exception handler called, in effect, by instruction itself.
Thus PC has not been updated at the point that LR is calculated

		ARM	Thumb	
	SWI	PC-8	PC-4	Exception taken here
⇨	xxx	PC-4	PC-2	LR = next instruction
	yyy	PC	PC	

- Return instruction

  MOVS pc,lr

- Note : the arrow denotes instruction to execute after return from exception

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Return instruction (2) **FIQs** and **IRQs**

Exception handler called after instruction has finished executing.
Thus PC has been updated at the point LR is calculated

		ARM	Thumb	
	www	PC - 12	PC - 6	Interrupt occurred during execution
⇨	xxx	PC - 8	PC - 4	
	yyy	PC - 4	PC - 2	ARM lr = next instruction
	zzz	PC	PC	Thumb lr = two instructions ahead

- Return instruction

  SUBS pc,lr,#4

- Note : the arrow denotes instruction to execute after return from exception

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Return instruction (3) **Prefetch Aborts**

- Exception taken when instruction reaches execute stage of pipeline.
  Thus PC has not been updated at the point LR is calculated
- Need to attempt re-execution of the instruction which caused the abort

		ARM	Thumb	
⇨	www	PC – 8	PC - 4	Prefetch Abort occurred here
	xxx	PC – 4	PC - 2	ARM lr = next instruction
	yyy	PC	PC	Thumb lr = two instructions ahead

- Return instruction

```
SUBS    pc,lr,#4
```

- Note : the arrow denotes instruction to execute after return from exception

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Return instruction (4) **Data Aborts**

- Exception taken (and LR calculated) after PC has been updated
- Need to attempt re-execution of instruction which caused the abort

	ARM	Thumb	
⇨ www	PC - 12	PC - 6	Data abort occurred here
xxx	PC - 8	PC - 4	
yyy	PC - 4	PC - 2	ARM lr = two instructions ahead
zzz	PC	PC	
aaa	PC + 4	PC + 2	Thumb lr = four instructions ahead

- Return instruction

```
SUBS    pc,lr,#8
```

- Note : the arrow denotes instruction to execute after return from exception

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Returning from (1) **SWI** and **Undef**

The SWI and Undefined Instruction exceptions
are generated by the instruction itself,
so the program counter is not updated
when the exception is taken.

The processor stores (pc - 4) in lr_ mode.
This makes lr_mode point to the next instruction to be executed.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Returning from (2) **FIQs** and **IRQs**

After executing each instruction, the processor checks
to see whether the interrupt pins are LOW and
whether the interrupt disable bits in the CPSR are clear.
As a result, IRQ or FIQ exceptions are generated
only after the program counter has been updated.
The processor stores (pc - 4) in lr_mode.
This makes lr_mode point one instruction
beyond the end of the instruction in which the exception occurred.

When the handler has finished,
execution must continue from the instruction
prior to the one pointed to by lr_mode.
The address to continue from is
one word (four bytes) less than that in lr_mode,

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Returning from (3) **Prefetch Aborts**

If the processor attempts to fetch
an instruction from an illegal address,
the instruction is flagged as invalid.
Instructions already in the pipeline continue to execute
until the invalid instruction is reached,
at which point a Prefetch Abort is generated.

The exception handler loads the unmapped instruction
into physical memory and uses the MMU,
if there is one, to map the virtual memory location
into the physical one.

The handler must then return to retry the instruction
that caused the exception.

The instruction should now load and execute.
Because the program counter is not updated
at the time the prefetch abort is issued,
lr_ABT points to the instruction
following the one that caused the exception.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Returning from (4) **Data Aborts**

When a load or store instruction tries to access memory,
the program counter has been updated.
The stored value of (pc - 4) in lr_ABT
points to the second instruction
beyond the address where the exception occurred.
When the MMU, if present, has mapped
the appropriate address into physical memory,
the handler should return to the original, aborted instruction
so that a second attempt can be made to execute it.
The return address is therefore
two words (eight bytes) less than that in lr_ABT,

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Handler entry and exit code (1) **SWI** and **Undef**

Restoring the program counter from the lr with:

```
MOVS    pc, lr
```

returns control from the handler.

The handler entry and exit code
to stack the return address and pop it on return is:

```
STMFD   sp!,{reglist,lr}
;...
LDMFD   sp!,{reglist,pc}^
```

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Handler entry and exit code (2) **FIQs** and **IRQs**

so the return instruction is:

```
SUBS        pc, lr, #4
```

The handler entry and exit code
to stack the return address and pop it on return is:

```
SUB         lr,lr,#4
STMFD       sp!,{reglist,lr}
;...
LDMFD       sp!,{reglist,pc}^
```

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Handler entry and exit code (3) **Prefetch Aborts**

The handler must return to lr_ABT - 4 with:

```
SUBS        pc,lr, #4
```

The handler entry and exit code
to stack the return address and pop it on return is:

```
SUB         lr,lr,#4
STMFD       sp!,{reglist,lr}
;...
LDMFD       sp!,{reglist,pc}^
```

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Handler entry and exit code (4) **Data Aborts**

making the return instruction:

```
SUBS        pc, lr, #8
```

The handler entry and exit code
to stack the return address and pop it on return is:

```
SUB         lr,lr,#8
STMFD       sp!,{reglist,lr}
;...
LDMFD       sp!,{reglist,pc}^
```

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Handler entry and exit code summary

SWI / Undef	FIQ / IRQ	Prefetch Abort	Data Abort
MOVS pc, lr	SUBS pc, lr, #4	SUBS pc, lr, #4	SUBS pc, lr, #8
	SUB lr, lr,#4	SUB lr, lr,#4	SUB lr, lr,#8
STMFD sp!, {reglist,lr}	STMFD sp!, {reglist,lr}	STMFD sp!, {reglist,lr}	STMFD sp!, {reglist,lr}
;...	;...	;...	;...
LDMFD sp!, {reglist,pc}^	LDMFD sp!, {reglist,pc}^	LDMFD sp!, {reglist,pc}^	LDMFD sp!, {reglist,pc}^

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

The Abort model

Many ARM memory access instructions cause
the base register to be updated e.g.

```
LDR     r0,[r1,#8]!
```

causes r1 to be updated

- If the memory access results in a data abort, the effect on the base register is
- dependent on the particular ARM core in use
- “Base Restored Abort Model”
- Supported by StrongARM, ARM9 and ARM10, ARM11 and later families
- Base register is restored automatically by the ARM core
- “Base Updated Abort Model”
- Supported by ARM7TDMI family
- Base register may have to be restored by the handler before instruction can be re-executed
- Example code for both models is included in RVDS examples directory

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Install an exception handler – Method I

```
Vector_Init_Block
        b       Reset_Addr
        b       Undefined_Addr
        b       SWI_Addr
        b       Prefetch_Addr
        b       Abort_Addr
        NOP                     ;Reserved vector
        b       IRQ_Addr
        b       FIQ_Addr

Reset_Addr          …
Undefined_Addr      …
SWI_Addr            …
Prefetch_Addr       …
Abort_Addr          …
IRQ_Addr            …
FIQ_Addr            …
```

http://osnet.cs.nchu.edu.tw/powpoint/Embedded94_1/Chapter%207%20ARM%20Exceptions.pdf

Install an exception handler – Method II

```
Vector_Init_Block
                    LDR PC, Reset_Addr
                    LDR PC, Undefined_Addr
                    LDR PC, SWI_Addr
                    LDR PC, Prefetch_Addr
                    LDR PC, Abort_Addr
                    NOP ;Reserved vector
                    LDR PC, IRQ_Addr
                    LDR PC, FIQ_Addr
Reset_Addr          DCD Start_Boot
Undefined_Addr      DCD Undefined_Handler
SWI_Addr            DCD SWI_Handler
Prefetch_Addr       DCD Prefetch_Handler
Abort_Addr          DCD Abort_Handler
                    DCD 0 ;Reserved vector
IRQ_Addr            DCD IRQ_Handler
FIQ_Addr            DCD FIQ_Handler
```

http://osnet.cs.nchu.edu.tw/powpoint/Embedded94_1/Chapter%207%20ARM%20Exceptions.pdf

References

[1] http://wiki.osdev.org/ARM_RaspberryPi_Tutorial_C
[2] http://blog.bobuhiro11.net/2014/01-13-baremetal.html
[3] http://www.valvers.com/open-software/raspberry-pi/
[4] https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html