Application Note 40

Configuring the FPA Support Code/FPE

Document Number: ARM DAI 0040 A
Issued: April 1997

ENGLAND
Advanced RISC Machines Limited
90 Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@armltd.co.uk

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@armltd.co.uk

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@armltd.co.uk

USA
ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

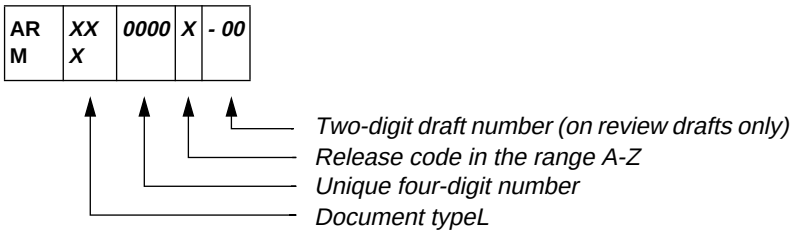
ARM
Advanced RISC Machines

Proprietary Notice

Key

Document Number

This document has a number which identifies it uniquely. It is displayed on every page.



Document Status

This describes the document's confidentiality and information status, and is shown at the bottom of each page.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above and to the staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A	April 1997	JM	First Release

Table of Contents

1 Introduction

If the `/hardfp` APCS option is used when compiling, assembling and linking floating-point code, the overall system must contain an implementation of the ARM floating-point instruction set. This implementation can be any of the following:

- The Floating Point Emulator (FPE)
- The FPA10 or FPA11 coprocessor, together with the FPA Support Code (FPASC)
- Possible future floating-point hardware, together with any support code it needs

The FPE and FPASC share a lot of code and are supplied as a single set of ARM assembler source files. Assembly-time options in these source files allow the code to be assembled as the FPE, as the FPASC, or as a combined FPASC/FPE which checks for the presence of the FPA hardware at initialisation time and behaves accordingly. In the event of further floating-point hardware being released, these options will be extended to cover the support code for that hardware.

Note *Use of the* `/hardfp` *APCS option is not recommended for code which is only ever expected to be run on systems that do not contain floating-point hardware.*

For systems that use software floating-point, the `/softfp` APCS option gives a substantial floating-point performance improvement, and generally results in reduced code size. For this reason the `FPE only` assembly-time option is only recommended when producing a system which does not have floating-point hardware, but requires binary compatibility with systems that do.

In addition to the FPE, FPASC or combined FPASC/FPE assembly-time option, the FPASC/FPE source code supplies various other options which enable it to be configured to its target environment.

This Application Note explains how these options are used. Firstly, it describes the general structure of an FPASC/FPE configuration. It then gives a detailed description of a very simple configuration of the FPASC/FPE source code and finally it expores other configuration options.

2 The structure of an FPASC/FPE configuration

An FPASC/FPE configuration consists of three parts.

2.1 The configuration veneer files

These need to be written for each configuration, and are the main means by which the FPASC/FPE is configured. There are five files which are included in the assembly by `GET` directives and are thus obligatory. These are:

`options.s`: This file specifies the assembly-time options to be used when assembling the core files (see below) for this configuration.

`ctxtws.s`: This file defines additional variables required by this configuration on a per-context basis. An example might be the process ID of a process associated with the context.

`globalws.s`: This file defines additional variables required by the veneer on a one-off basis, regardless of the number of floating-point contexts. An example might be the previous contents of the undefined instruction vector for a version of the FPASC/FPE which can be terminated, so that the vector can be restored to its previous value on finalisation.

This file can also be used to define symbolic constants, etc., which are needed by the other configuration veneer files.

Note *Unless the FPASC/FPE is configured to support multiple floating-point contexts (see* ***4 Multiple floating-point contexts*** on page 17*), there is no real difference between defining a variable in* `ctxtws.s` *and defining it in* `globalws.s`*. It is nevertheless a good idea to put variable definitions in the correct file: this will aid any later conversion to a multi-context version of the configuration.*

`start.s`: This file contains the assembler source for any veneer code which is to precede the main core code in the resulting binary.

`end.s`: This file contains the assembler source for any veneer `ctxtws.s` code which is to follow the main core code in the resulting binary.

These files will all be described in ***The start.s and end.s files*** on page 11. However, other files can easily be included from the above files via `GET` directives if desired.

2.2 The FPASC/FPE core files

These contain the bulk of the FPASC/FPE code, and should not require any modification when configuring the FPASC/FPE. The code they generate can be varied in limited ways by changing the options chosen in the veneer file `options.s`.

The most important core source file is `main.s`, which is in control of the overall assembly. It starts by including all the core and veneer definitions files in the right order, then the `start.s` code file, then all the core code files, and finally the `end.s` code file.

Other than the `GET` directives including these other files, the `main.s` file also contains a large amount of comment text describing how the core and veneer interact. These comments are also a description of how to configure the FPASC/FPE, and are an additional source of information about the facilities described in this Application Note.

WARNING: *The comments in* `main.s` *also describe various facilities which are not mentioned in this Application Note. Do not use these additional facilities, as they may be changed and/or removed entirely in future releases of the FPASC/FPE.*

The remaining core files are the following definitions files:

`armdefs.s`:	Definitions relating to the ARM instruction set
`defaults.s`:	Default values for some of the options defined in `options.s`
`fpadefs.s`:	Definitions relating to the FPA instruction set
`macros.s`:	Assembler macro definitions
`regnames.s`:	Register name definitions
`wsdefs.s`:	Workspace definitions and the following code files:
`arith.s`:	Basic arithmetic routines
`fpaundef.s`:	The main undefined instruction handler for the FPASC
`fpeundef.s`:	The main undefined instruction handler for the FPE
`interf.s`:	Routines relating to the interface between core and veneer
`ldst.s:`	Load/store routines
`nans.s`:	Routines to handle NaNs, invalid operation exceptions and divide-by-zero exceptions
`rounding.s`:	Routines to handle rounding, overflow exceptions, underflow exceptions and inexact exceptions
`toplevel.s`:	This file includes all the other core code files, and also implements a few routines which do not fit in any of the other source files
`transhw.s` and `transsw.s`:	Routines for the transcendental instructions (`POW`, `RPW`, `POL`, `LOG`, `LGN`, `EXP`, `SIN`, `COS`, `TAN`, `ASN`, `ACS` and `ATN`).

2.3 A header file ‘header.s’

This is the file which is actually assembled to produce the FPASC/FPE object file. However, all it does is set up some file-naming conventions and then use the assembler `GET` directive to start the main assembly with the `main.s` file.

This file may require minor modifications when configuring the FPASC/FPE.

The source files for the configuration described in the following section are all in the accompanying `simple_fpasc` directory. They are generally quite small—all are less than 100 lines long and only a couple are more than 50 lines.

3 A simple FPASC/FPE configuration

This section describes a simple, purely RAM-based FPASC/FPE configuration. This configuration generates an FPASC object file which can be linked into another stand-alone assembler program to allow it to use the FPA10 or FPA11 hardware.

Caution *Do not link this FPASC object file into multiple application programs running on the same hardware because the multiple copies of the FPASC code are likely to interfere with each other's operation. There should only ever be one copy of the FPASC/FPE code running on a particular ARM. The code should normally only be linked into an application program if it is genuinely a stand-alone application. If multiple application programs are running on the same hardware, there will normally be some sort of operating system (possibly a very rudimentary one): the FPASC/FPE code should then be linked into that operating system rather than into an individual application.*

The requirements on this stand-alone assembler program which will be linked with the FPASC object file are:

- It should `IMPORT` the symbol `FPASC_STARTUP` and execute the instruction:

  ```
  BL  FPASC_STARTUP
  ```

  during its initialisation code, with the ARM in supervisor mode, and before making any attempt to execute any floating-point instructions.

  Calling this routine will set up a small supervisor stack (addressed by r13_svc) and a small undefined instruction stack (addressed by r13_und), then initialise the FPASC software and enable the FPA hardware.

Note *If the stand-alone assembler program wants to set up its own supervisor and/or undefined instruction stacks, it should do so before the call to* `FPASC_STARTUP` *and the appropriate stack set-up code should be deleted from the* `start.s` *file described in* ***The start.s and end.s files*** on page 11.

- It should define and `EXPORT` four error handling routines, which will be branched to from the FPASC code if the corresponding error occurs. These should all be treated as fatal errors with this configuration, and are:

  – `FPA_ABSENT`: Initialisation time error to indicate that no FPA hardware was found.

  – `FP_TRAP`: A trapped floating-point exception has happened.

  - `COREFAULTY`: An unlikely error, probably indicating that the FPA hardware or the way it is connected to the ARM is faulty.

  – `COREDISABLED`: An error indicating an attempt to execute floating-point code while the floating-point system is disabled. This is very unlikely with this precise configuration, as it does not try to export the call to disable the floating-point system, but this error handling routine is needed for linkage purposes.

- It should define and `EXPORT` a routine called `FPASC_UNHANDLED` which will try to process non-floating-point coprocessor instructions. The entry and exit conditions of this routine are precisely those of an ARM undefined instruction handler, and its most likely implementation in a simple system is as an `Unexpected undefined instruction` fatal error.

- It should define and EXPORT three routines to manipulate the undefined instruction vector, with definitions as follows:

veneer_newhandlers

This installs a particular routine as the current undefined instruction handler.

On entry:

r1 is the new undefined instruction handler address.
r13 is the stack pointer;
r14 is the return link.

On exit:

All registers except r14 are preserved.

veneer_preservehandlers

This records the current undefined instruction handler's address.

On entry:

r13 is the stack pointer;
r14 is the return link.

On exit:

All registers except r14 are preserved; the address of the current undefined instruction handler has been recorded in a variable somewhere.

veneer_restorehandlers

This restores the undefined instruction handler to what it was at the time of the last call to `veneer_preservehandlers`.

On entry:

r13 is the stack pointer;
r14 is the return link.

On exit:

All registers except r14 are preserved.

The exact implementation of these routines will vary somewhat according to the target system. A typical implementation would be:

Instruction at undefined instruction vector (location 0x00000004) is:

```
LDR PC,UndefHandler.
```

`UndefHandler` and `OldUndefHandler` are consecutive RAM words within +/-4K of the undefined instruction vector:

```
UndefHandler DCD 0
```

```
OldUndefHandler DCD 0
```

`InitUndefHandler` is the address of the initial undefined instruction handler, which is probably a routine to report an `unexpected undefined instruction` error.

Then an implementation of the three routines above is:

```
veneer_newhandlers
      EXPORT veneer_newhandlers
      STMFD r13!,{r14}
      LDR r14,Addr_UndefHand
      STR r1,[r14]                  ;Store UndefHandler
      LDMFD r13!,{PC}
veneer_preservehandlers
      EXPORT veneer_preservehandlers
      STMFD r13!,{r1,r14}
      LDR r14,Addr_UndefHand
      LDR r1,[r14]                  ;Load UndefHandler
      STR r1,[r14,#4]               ;Store
                                    ;OldUndefHandler
      LDMFD r13!,{r1,PC}
veneer_restorehandlers
      EXPORT veneer_restorehandlers
      STMFD r13!,{r1,R14}
      LDR r14,Addr_UndefHand
      LDR r1,[r14,#4]               ;Load
                                    ;OldUndefHandler
      STR r1,[r14]                  ;Store UndefHandler
      LDMFD r13!,{r1,PC}
      Addr_UndefHand DCD UndefHandler
```

The header.s file

- The `simple_fpasc/header.s` file should be included somewhere around here.
- This file sets up assembly-time variables which define: a name for the veneer ; which directory the core source files are to be found in and which directory the veneer source files are to be found in.
- what filename extension should be used for the source files.

The reasons for the existence of this source file are mainly historical, and the recommended way of keeping the core and veneer source files in different directories is now to use armasm's -I option rather than to change the definitions of `CoreDir` and `VeneerDir` in this file. If they are changed,they should be terminated with the appropriate directory separator character. (eg. for a Unix file system, `CoreDir` should not be defined to be `core_src`, but `core_src/`.)

This particular `header.s` file defines the veneer name to be `simple`, both core and veneer source files to come from the current directory or directories specified by armasm's -I option, and the source file extension to be `.s`.

The options.s file

The `simple_fpasc/options.s` file should be included somewhere around here. This file selects the assembly-time options wanted for the FPASC/FPE core. Some have default values and do not require a GBLL, GBLS or GBLA directive in this file; others must be specified explicitly and do require such a directive in this file.

The options which must be specified are:

- `FPEWanted` and `FPASCWanted`, which specify between them whether the FPE, the FPASC or a combined FPE/FPASC is wanted. This particular configuration contains the FPASC only. (Making both options be {FALSE} is an error.)
- `UndefHandlerConvention`, which gives some control over what parts of undefined instruction handling are to be performed by the FPASC/FPE and what parts are to be performed by other system components. This configuration uses the StandAlone convention, which means that the FPASC/FPE basically does the entire job. This is the simplest option, but can be inefficient if multiple coprocessor instruction sets need to be supported: in this case, an alternative `BranchTable` convention can be used instead. See ***6 Multiple coprocessors*** on page 24 for more details of how the `StandAlone` and `BranchTable` options work.
- `MultipleContexts`, which specifies whether multiple floating-point contexts are supported. If the target system wishes to support multiple concurrent floating-point calculations, it should set `MultipleContexts` to {TRUE} to allow each to be performed in a different floating-point context and provide a quick way of switching between them. If just one floating-point calculation is wanted, setting `MultipleContexts` to {FALSE} makes the job of configuring the FPASC/FPE somewhat easier and the resulting code slightly more efficient.

  `MultipleContexts` is set to {FALSE} in this configuration because it is meant to be particularly simple and it is only supposed to support a single stand-alone program. See ***4 Multiple floating-point contexts*** on page 17 for more details on this subject.
- `EnableInterrupts`, which specifies whether interrupts are to be enabled while the FPASC/FPE code does its undefined instruction processing. Setting this to {FALSE} keeps the system simple: interrupts are disabled when the undefined instruction trap is taken and are not re-enabled until it is completely processed, and so a partially-processed floating-point instruction is generally not seen. The drawback of this simplicity is that interrupts may remain disabled for a substantial period, resulting in a greatly increased interrupt latency.

  Setting `EnableInterrupts` to {TRUE} instead means that interrupts are re-enabled as soon as possible after the undefined instruction trap is entered, and are only disabled again for brief periods (a few instructions at a time) during FPASC/FPE undefined

instruction processing. The drawback of doing this is that extra care has to be taken at some times to deal with the case of a partially-processed floating-point instruction. Forsimplicity, `EnableInterrupts` is set to {FALSE} in this configuration.

More details of the cases in which care must be taken can be found under ***4 Multiple floating-point contexts*** on page 17.

- `FPRegInitValue` and `FPSRInitValue`, which specify the values to which the floating-point registers f0-f7 and the FPSR (Floating Point Status Register) are initialised in each floating-point context. `FPRegInitValue` may be set to `SigNaN` to initialise f0-f7 to signalling NaNs, or to `Zero` to initialise them to zeros: the former option is good at making use of uninitialised registers to generate errors, while the latter is fairly good at making such uses behave "sensibly". This configuration uses the `SigNaN` option, in order to catch uses of uninitialised registers at the early development stages in which this configuration is likely to be useful.

  `FPSRInitValue` may be any legitimate value for the bottom 24 bits of the FPSR (the top 8 bits are read-only `SysID` bits). This configuration uses 0x000200, setting the NE bit: this results in the highest level of IEEE compliance from the system.

The options with a default value generally default to the highest level of functionality, with the alternative value removing some aspects of the functionality to get reduced code size and/or greater ease of configuration:

- `NoTranscendentals`, which defaults to {FALSE}. If changed to {TRUE}, this removes the implementations of the transcendental instructions (`POW`, `RPW`, `POL`, `LOG`, `LGN`, `EXP`, `SIN`, `COS`, `TAN`, `ASN`, `ACS` and `ATN`) from the FPASC/FPE; instead, they are treated as undefined floating-point instructions. The advantage of doing this is that it reduces the FPASC/FPE code size substantially (about 8K, ie. 30-40% of the total FPASC/FPE code size) when these functions are not wanted in a particular application.
- `NoPacked`, which defaults to {FALSE}. If changed to {TRUE}, this removes the implementations of the load/store packed (`LDFP` and `STFP`) instructions from the FPASC/FPE; instead, they are treated as undefined floating-point instructions. Like `NoTranscendentals`, this provides a useful reduction in code size when this functionality is not wanted. This reduction is much smaller - about 1.5K - but still represents about 10% of the total FPASC/FPE code size once `NoTranscendentals` has been specified.
- `TrapsCanReturn`, which defaults to {TRUE}. This allows a trap handler for a floating-point exception to return a result for the operation that generated the exception and resume execution, via various somewhat complicated interfaces which need careful interfacing to the rest of the system. Most applications do not need this sort of functionality, instead dealing with floating-point exceptions by generating an error and/or not trapping them: in such cases, `TrapsCanReturn` can be set to {FALSE} to remove the code that allows a trap handler to resume execution with a specified result. This reduces code size somewhat and simplifies the job of writing trap handlers.

All three of these options are set to their non-default values in this FPASC/FPE configuration, to minimise code size as far as possible.

The ctxtws.s and globalws.s files

The `simple_fpasc/ctxtws.s` and `simple_fpasc/globalws.s` files should be included somewhere around here. These files allow the configuration to allocate extra variables in the FPASC/FPE context and/or workspace, for use by the configuration's own code.

This can be either on a one-off basis (by putting the definitions in `globalws.s`) or a per-floating-point-context basis (by putting them in `ctxtws.s`). These allocations should be done using armasm's # directive to reserve space in a storage map. The ^ directive, which is normally used to set the originof a storage map, should not be used in this file, as it is taken care of by the core files which `GET` these files.

As an example, if the configuration wants to store the process ID associated with each context in a per-context word variable, as well as an old process ID and an area in which to dump the contents of eight registers, the lines:

```
ProcessID # 4
OldProcessID # 4
RegDump # 32
```

should be placed in the `ctxtws.s` file. These variables can then be accessed as follows:

- If r10 contains the right context/workspace pointer (a common entry condition to the veneer routines), the variable names can be used directly in `LDR`, `STR` and `ADR` instructions - eg. assuming that `OldProcessID` and `RegDump` are two other such variables:

  ```
  LDR r0,ProcessID
  STR r0,OldProcessID
  ADR r8,RegDump
  STMIA r8,{r0-r7}
  ```

- If the right context/workspace pointer is in some other register `rn`, the assembler's :`INDEX`: operator can be used to get the corresponding effects - eg.:

  ```
  LDR r0,[rn,#:INDEX:ProcessID]
  STR r0,[rn,#:INDEX:OldProcessID]
  ADD r8,rn,#:INDEX:RegDump
  STMIA r8,{r0-r7}
  ```

- If the right context/workspace pointer is not in a register, it will generally need to be loaded into a register before the variables can be accessed. It is also legitimate for these files to contain other definitions - eg. `EQU` directives as in:

  ```
  BufferSize EQU 64
  Buffer # BufferSize
  Buffer2 # BufferSize
  ```

They must not however contain anything that will actually be assembled as code, since these files are assembled before any AREA directive has appeared. All configuration code should be placed in the `start.s` and `end.s` files.

The simple configuration being described here has no need for any extra workspace variables, so its `ctxtws.s` and `globalws.s` files only contain comments.

The start.s and end.s files

The `simple_fpasc/start.s` and `simple_fpasc/end.s` files should be included somewhere around here.

These two files contain all the configuration code. Most of it can be placed in either file: the main reason for making two files available is to allow system rules about the format of code to be obeyed, and in particular to allow the production of header and trailer information in the resulting binary if desired. Exceptions to this rule are:

- Macro definitions required by the core routines must be placed in the `start.s` file.
- Rarely, code using symbols defined by the core code files may have to go in the `end.s` file to ensure that the symbols are defined at the right times.

The essential elements of the configuration code are illustrated by this configuration's `start.s` and `end.s` files, and are:

- The `start.s` file must start with an AREA directive. (This is done in the configuration to allow user choice of the area name and attributes.) A GBLA variable called `ExpectsCoreVersion` must be defined and set to the `major version number` of the core files: this is used to highlight the need to update the configuration if the veneer/core interface changes in a major way. The major version number is 1 for the FPASC/FPE version being described in this Application Note.
- A macro with a prototype line:

  ```
  $label AdrWS $reg
  ```

  must be defined: its implementation should place the address of the FPASC/FPE workspace in the register `$reg`, and `$label` should be the label of the first instruction in the code to do so. It must not change the value of any register other than `$reg`.

  This macro needs to be executed at times when very little is known about the contents of any registers. All it may assume is:
- It is being executed from within the FPASC/FPE code itself.

- That r13 points to a Full Descending stack.

  This particular configuration uses an ADRL instruction to address the workspace: it is known to be within range because the configuration allocates the workspace from within the FPASC/FPE code area itself. This is the simplest solution in this purely RAM-based environment. A more usual arrangement is to allocate the workspace at a known fixed location in memory - e.g. 0x1000 - and use `MOV $reg,#0x1000` as the macro implementation. (Or put a pointer to the workspace at a similar fixed location and use e.g. `MOV $reg,#0x1000` followed by `LDR $reg,[$reg]` to implement the macro.)

- A macro with a prototype line

  ```
  $label FPInstrDone $opt
  ```

  must be defined. Its normal implementation does nothing at all if `$opt` is not specified, and clears the Z flag while leaving all registers unchanged if `$opt` is specified.The entry conditions include the fact that when `$opt` is specified, r9 is the address of the instruction which is about to be executed next after return from the undefined instruction handler.

  In this configuration, the macro has a null implementation when `$opt` is not specified, and is implemented with `TEQ r9,#0` otherwise. This implements the above in all cases except for the execution of a floating point instruction at address 0xFFFFFFFC: this isextremely unlikely, since it involves the flow of execution next wrapping around to the reset vector at 0x00000000.

  The purpose of this macro is to allow the veneer and/or an operating system in which it is embedded to take special actions at the end of the FPASC/FPE's processing of an undefined instruction: this is wanted in some environments. How this is done goes beyond the scope of this Application Note, but details can be found in the `main.s` comments. For most purposes, however, the near-null implementation in this configuration is what is wanted.

- It should make the `veneer_newhandlers` and `veneer_restorehandlers` calls available to the FPASC/FPE core. These are as documented in ***veneer_newhandlers*** on page 6, except that as far as the FPASC/FPE core is concerned, the purpose of `veneer_restorehandlers` is to restore the undefined instruction handler to what it was before the FPASC/FPE was initialised.

  This configuration implements these by IMPORTing `veneer_newhandlers`, `veneer_preservehandlers` and `veneer_restorehandlers` as documented above, then calling `veneer_preservehandlers` once during its initialisation.

- It should make a routine called `veneer_corefaulty` available. This will be branched to under various conditions that should never happen in a working system - eg. if the FPA hardware causes the undefined instruction vector to be entered, but the FPASC's subsequent read of the FPCR does not reveal any reason for this to have happened.

  The only sensible thing to do with this condition is to treat it as a fatal error. This configuration does so by branching to the COREFAULTY routine described on page 5.

- It should make a routine called `veneer_coredisabled` available. This is a call-back routine which is executed if an attempt is made to execute a floating-point instruction while the floating-point system is disabled. See ***5 Other floating-point context issues*** on page 22.

  This configuration never attempts to disable the floating-point system, and so treats a call to `veneer_coredisabled` as a fatal error, by branching to the `COREDISABLED` routine described above.
- It should make the following routines available to handle trapped floating-point exceptions. There are quite a lot of them, as the type of exception and nature of the instruction that generated the exception change what additional information can be reported, and thus change the entry conditions to the routine.

  ```
  veneer_invalidop1_single
  veneer_invalidop2_single
  veneer_invalidop1_double
  veneer_invalidop2_double
  veneer_invalidop1_extended
  veneer_invalidop2_extended
  veneer_invalidop1_integer
  veneer_invalidop2_integer
  veneer_invalidop1_packed
  veneer_invalidop1_xpacked
  veneer_invalidop1_ldfp
  veneer_invalidop1_ldfpx
  veneer_zerodivide1_single
  veneer_zerodivide2_single
  veneer_zerodivide1_double
  veneer_zerodivide2_double
  veneer_zerodivide1_extended
  veneer_zerodivide2_extended
  veneer_overflow_single
  veneer_overflow_double
  veneer_overflow_extended
  veneer_underflow_single
  veneer_underflow_double
  veneer_underflow_extended
  veneer_inexact_single
  veneer_inexact_double
  veneer_inexact_extended
  veneer_inexact_integer
  veneer_inexact_packed
  veneer_inexact_xpacked
  ```

This configuration produces a fatal `Trapped floating-point exception` error for any of these, without paying attention to any of the information associated with the exception. So all of the above routines are implemented as a branch to the `FP_TRAP` routine described on page 5.

- It should contain an initialisation routine which is called to start the FPASC/FPE up: in this configuration, this initialisation routine is called `FPASC_STARTUP`.

  This initialisation code must carry out certain processes, and in the following order:

  - It must ensure that it is executing in supervisor mode. In this configuration, this is simply a requirement on how `FPASC_STARTUP` is called.
  - It must ensure that r13_svc and r13_und point to Full Descending stacks of a reasonable size. This is only likely to involve any real code within the FPASC/FPE configuration in very simple systems like this one: in more complex systems, setting up suitable stacks is probably the job of some other system component.
  - It must take a record of what the undefined instruction handler was before the FPASC/FPE starts changing it. In this configuration, this is done by calling the `veneer_preservehandlers` routine described on page 6.
  - It must allocate the FPASC/FPE workspace. The length of this workspace is determined by the FPASC/FPE core and made available to the veneer by setting a symbol `WorkspaceLength` to be the number of bytes required. This length includes the space required by any variables defined in the `globalws.s` and `ctxtws.s` files, so no extra allowance should be made for them.
  - The workspace must be word-aligned. Furthermore, in a single-context configuration (ie. if `MultipleContexts` is set to {FALSE} in the `options.s` file), the first items in the workspace is the one and only floating-point context. This means that the terms *workspace pointer* and *context pointer* are effectively synonymous in this case. (If `MultipleContexts` is {TRUE}, this situation changes - see ***4 Multiple floating-point contexts*** on page 17.)

  In this configuration, the workspace is allocated by means of a `Workspace % WorkspaceLength` directive at the end of `end.s`: this method works in this purely RAM-based system, but would probably need replacing in anything more sophisticated.

  - It must ask the FPASC/FPE core to initialise the workspace. In a single-context configuration like this one, this also includes initialising the single context that workspace contains; again, this changes in a multi-context configuration. The call to do this is `BL core_initws`, with the following entry and exit conditions:

    `core_initws`

    This is a call to initialise the FPASC/FPE workspace, and to determine what floating-point hardware is present, if any.

    On entry:

r1 is the address of the undefined instruction handler to pass control to if the instruction encountered is not a valid floating-point instruction;

r10 is the pointer to workspace/context;

r13 is the stack pointer;

r14 is the return link.

On exit:

r0 is `SysId` of hardware found (in bottom byte), or -1 if no hardware is found. All other registers except r14 are preserved.

This configuration sets r1 on entry to point to the `FPASC_UNHANDLED` routine described on page 5. The `SysID` returned is the value of the read-only top byte of the hardware's FPSR, if hardware is present. The core definition files contain a definition of a symbol `SysID_FPA (0x81)`.

- It should check the `SysId` returned by `core_initws` in r0 to determine whether it can actually deal with the hardware. If it is configured to be the FPASC only, failure to find the FPA hardware is clearly a fatal error. This is the case for this configuration, and the error is implemented by the `FPA_ABSENT` routine described above. If it is configured to be the FPE only and floating-point hardware is found, or to be the combined FPASC/FPE and hardware other than the FPA is found, the hardware will be left disabled and the code will act as an FPE. This does not have to cause an error, but is strong evidence of a misconfigured system since it results in the floating-point hardware lying unused. A system may therefore want to treat it as an error.
- In a single-context system like this one, it must activate the single context by making certain that the hardware is enabled and that all its register values, etc., are loaded into the hardware. The call to do this is `BL core_activatecontext` and has the following entry and exit conditions:

`core_activatecontext`

This activates a floating-point context.

On entry:

r10 is a pointer to workspace/context;

r13 is the stack pointer;

r14 is the return link.

On exit:

All registers except r14 are preserved.

As usual for anything to do with floating-point contexts, this changes in a multi-context system - see ***4 Multiple floating-point contexts*** on page 17.

- It must initialise its own `globalws.s` variables, and in a single-context configuration like this one, its own `ctxtws.s` variables as well.

  Since this configuration does not contain any such variables, no code is needed to do this.

4 Multiple floating-point contexts

The simple configuration of the FPASC/FPE described above is only designed to deal with a single floating-point calculation at a time. This is all that is wanted in some applications, but many others need to be able to deal with multiple concurrent floating-point calculations. Reasons for such a requirement include:

- Having multiple software processes, each of which is performing its own floating-point calculations. These can be switched between either co-operatively (ie. each process contains code to yield control at specified points during its execution) or pre-emptively (ie. under the control of interrupts).
- Having interrupt handlers which perform floating-point calculations, while the code being interrupted is also performing floating-point calculations.

In each case, there is a need for each calculation to see its own floating-point context (ie. copies of the floating-point registers f0-f7 and of the Floating Point Status Register), and for this context not to be corrupted by any other floating-point calculation. A context switch is needed whenever the floating-point calculation being performed changes - eg. whenever a process switch occurs in the first case above, or on interrupt entry and return in the second case.

The code required to implement a floating-point context switch is conceptually simple: the f0-f7 and FPSR values belonging to the old context need to be stored away, and those belonging to the new context need to be loaded. The first part of this can be done with two `SFM` instructions, each storing 4 registers, plus an `RFS/STR` combination to store the FPSR; the second part can be done with two `LFM` instructions, each loading 4 registers, plus an `LDR/WFS` combination to load the FPSR.

Such code can easily be run on the simple FPASC/FPE configuration described above. There are, however, a number of problems with this:

- If `EnableInterrupts` is {TRUE}, it is possible for an interrupt to occur while FPASC/FPE code is being executed. Such an interrupt may result in a context switch; if so, the current values of floating-point registers when the context switch occurs may be in any one of a number of places, including FPA registers, ARM registers, memory and combinations of these, and in a number of formats. (For example, the sign and exponent may be combined in the same word or may have been unpicked into two separate words.) Similarly, the floating-point hardware may be in an unusual state, such as being temporarily enabled despite the floating-point system being disabled at the time. The net result is that there is no guarantee that the various `LFM`, `SFM`, `WFS` and `RFS` instructions can be correctly executed at the time.
- If the context switch is triggered by an interrupt which occurred when the floating-point hardware was on the point of executing a floating-point instruction, some tidying-up may be required (depending on the hardware concerned).
- With floating-point hardware, in some circumstances the instruction that causes the FPASC/FPE code to be invoked may not be the obvious one. For example, if an arithmetic instruction such as `ADF`, `MUF` or `DVF` is executed on the FPA and a floating-point exception occurs, the FPASC/FPE code to process that exception will not

normally be invoked at the time: instead, it will typically be invoked when the next floating-point instruction is executed. Normally, this next floating-point instruction belongs to the same process, and no real problems occur: the exception is imprecise, but is reported as occurring in the correct process.

- If, however, an interrupt causes a process switch before the normal next floating-point instruction is encountered, the next floating-point instruction actually executed turns out to be the `RFS` or one of the `SFM`s in the process switch code. Without special case exception-reporting code, this typically results in the floating-point exception being reported as occurring in the operating system!
- Furthermore, actually processing the exception may take a significant number of cycles, and there are cases where the FPASC/FPE code will be invoked to handle a difficult case without a floating-point exception actually occurring, or with an untrapped floating-point exception. The net result is that even in the absence of trapped floating-point exceptions, this approach may result in the process switch unexpectedly taking a lot more time than usual. This can be unacceptable in some systems.
- Finally, when the FPE is involved, this approach results in a lot of instruction emulation and data copying: six floating-point instructions need to be fetched and decoded by the FPASC and FPE, and each of two `SFM` instructions needs to copy 12 words from the part of the FPE's workspace that contains its register values to a process control block, etc.

  This greatly increases the process switch time; it is also not really necessary, since all that is needed to achieve the desired effect is to tell the FPE routines that their register values are no longer to be found in one place in memory, and are instead in some other place in memory.

All the problems above which cause incorrect functionality can be worked around by the sort of `SFM/RFS/LFM/WFS`-based context switch code described above, but only at the expense of delaying context switches yet further and adding extra software complexity.

The recommended solution is instead to configure the FPASC/FPE with `MultipleContexts = {TRUE}`. This does the following:

- All values relating to the current floating-point context are removed from the main workspace - in particular, the values of f0-f7 and the FPSR are no longer kept there. Instead, a pointer to the current floating-point context is kept in the main workspace.

  This decreases the value of `WorkspaceLength`. The `BL core_initws` call still expects to be handed a pointer to a chunk of memory of this length in r10, but now does not perform a context initialisation. Instead, it initialises the current workspace pointer to 0, to indicate that there is no current context: while this is the case, no floating-point instructions can be executed, and the FPASC/FPE will not yet have any undefined instruction handlers installed.

  As before, it is the veneer's responsibility to initialise any variables it defines in `globalws.s` at the same time. Variables that it defines in `ctxtws.s` should be left until context initialisation time.

- The FPASC/FPE configuration may now allocate as many floating-point contexts as desired. Each one must be a word-aligned chunk of memory of length `ContextLength` (like `WorkspaceLength`; this symbol is made available to the veneer by the core source code).

  Before each floating-point context may be used, it must be initialised. To do this, a `BL core_initcontext` call should be used, with entry and exit conditions described as follows.

  `core_initcontext`

  This initialises or re-initialises a floating-point context.

  On entry:

  r10 is the pointer to context;
  r13 is the stack pointer;
  r14 is the return link.

  On exit:

  All registers except r14 are preserved.

  The exact initialisation used is determined by `FPRegInitValue` and `FPSRInitValue` as described above. Note that the extra context variables defined by `ctxtws.s` are not initialised by this call: it is the veneer's responsibility to initialise them suitably at the same time.
- Once initialised, a context may be chosen via the `BL core_changecontext` call, with entry and exit conditions as follows:

  `core_changecontext`

  This changes to a new floating-point context.

  On entry:

  r10 is the pointer to context;
  r13 is the stack pointer;
  r14 is the return link.

  On exit:

  r0 is the old context pointer;
  All other registers except r14 are preserved.

  The old context pointer returned in r0 will be 0 if there was previously no floating-point context—as happens, for example, on the first call to this function after the initial `BL core_initws`. Suitable FPASC/FPE undefined instruction handlers are installed at the time of the call.

  Furthermore, r10 may be set to 0 on entry. If this is done, the call will cause the system to revert to the state of having no current floating point context: this includes using `veneer_restorehandlers` to remove the FPASC/FPE undefined instruction handlers.

Note *It is legitimate to use* `BL core_activatecontext` *when* `MultipleContexts = {TRUE}`*, as long as it is known that there is no current floating-point context at the time. (If there is, the old context and/or any floating-point hardware may be left in an inconsistent state.) There is also another call:*

`core_deactivatecontext`

This deactivates the current floating-point context.

On entry:

r13 is the stack pointer;

r14 is the return link.

On exit:

r0 is the old context pointer;

All other registers except r14 are preserved.

This is effectively equivalent to a call to `core_changecontext` with r10 = 0, putting the system into the state of having no floating-point context. If floating-point hardware is present, this includes making certain that all important data about the context has been transferred to memory, and that the hardware state has been tidied up.

A `BL core_changecontext` is precisely equivalent to a sequence of `BL core_deactivatecontext` followed by `BL core_activatecontext`.

Note *It is also legitimate to use any of* `BL core_changecontext`*,* `BL core_activatecontext` *and* `BL core_deactivatecontext` *when* `MultipleContexts = {FALSE}`*, though the only legitimate context pointers in this case are the main workspace pointer and 0. There are some uses for this—for instance, if the above-mentioned SFM/RFS/LFM/WFS context switch is used,* `BL core_changecontext` *should be executed first, with* r10 *equal to the workspace pointer as a software work-around to ensure that the new context is entered with the hardware in a sensible state. If, however, such uses are encountered, it is a strong indication that using* `MultipleContexts = {TRUE}` *would probably be a better solution.*

Use of `MultipleContexts = {TRUE}`and the `BL core_changecontext` call generally solves the problems mentioned above. The main reason for this is that the exact implementation used for the `BL core_changecontext` call will depend on whether floating-point hardware is present (and potentially in the future, on what kind of floating-point hardware is present). This allows it to perform hardware-specific tidy-ups and to avoid processing any pending floating-point exceptions at the time. Instead, it can record any hardware information about such an exception in the old process without processing it then and there, and reinstate any pending exception information for the new context, to be dealt with next time a floating-point instruction is executed in that context. Also, in the case where no floating-point hardware is present, the context switch can be optimised down to simply changing the current context pointer: no instruction decoding nor data movement is required.

Between them, these solve almost all the problems mentioned above. The rest can be dealt with, as long as the supervisor and undefined instruction stack pointers are changed appropriately on context switches (so that, for example,

an interrupt within the FPASC/FPE in one process does not allow its undefined instruction stack contents to be picked up by the FPASC/FPE in another process).

5 Other floating-point context issues

For some purposes, it is desirable to be able to inspect the contents of a floating-point context other than the current one. (For example, a debugger program may wish to inspect the f0-f7 and FPSR values of the program being debugged.) This could be performed by switching the floating-point context, then using SFM and RFS instructions, then switching back to the debugger's floating-point context. Doing this, however, means that any pending floating-point exception relating to the context will be processed at that point, which is likely to be undesirable.

The FPASC/FPE therefore provides a call `BL core_savecontext` with the following functionality and entry and exit conditions:

core_savecontext

This call saves the register and FPSR values (held in a floating-point context) to a specified 25-word area of memory in the format shown in ***Table 5-1: Core_savecontext register details***.

Byte offset	Value
0	FPSR value
4	f0 value, as an extended precision number
16	f1 value, as an extended precision number
28	f2 value, as an extended precision number
40	f3 value, as an extended precision number
52	f4 value, as an extended precision number
64	f5 value, as an extended precision number
76	f6 value, as an extended precision number
88	f7 value, as an extended precision number

Table 5-1: Core_savecontext register details

This is done without processing any pending exceptions associated with the context, or even completing the processing of any exception which is currently being processed.

On entry:

r0 points to the 25-word area described above;

r1 is the PC value associated with the context;

r10 points to the context;

r13 is the stack pointer;

r14 is the return link.

On exit:

r0 contains a non-zero value if there is a pending or incompletely processed exception associated with the context, and zero if there is not;

r14 may be corrupt;

All other registers are preserved.

The value returned in r0 is of use because f0-f7 and the FPSR may contain strange or inconsistent values while there is a pending or incompletely-processed exception. (For example, a debugger might print messages warning that these values could not be relied upon while this is happening.)

The PC value supplied in r1 is used to test for incompletely-processed exceptions: a PC value lying within the FPASC/FPE's undefined instruction handlers indicates that this is the case.

Similar facilities exist to allow a debugger to change the f0-f7 and FPSR values in a floating-point context. Even more care needs to be taken about pending or incompletely-processed exceptions in this case; otherwise, the values loaded may cause a totally-inappropriate exception to occur, or get overwritten by the result of the original exception, or cause other strange behaviour. The calls involved are `BL core_loadcontext` to actually change the f0-f7 values in a context, and `BL core_abort` to clean up any pending or incompletely-processed exceptions that may exist at the time. Full details of these calls may be found in the `main.s` comments.

Finally, facilities exist to temporarily disable the floating-point system, so that when an attempt is next made to execute a floating-point instruction, it is rejected and the veneer is instead called. The main purpose of this is that it allows floating-point context switches to be avoided on systems where most processes rarely (or never) execute floating-point instruction. The idea is that when a floating-point context switch would normally occur, you instead disable the floating-point system, and only actually change floating-point context if and when the callback occurs. This is probably only of real value when the floating-point context switch time is long—for example, if the non-recommended `MultipleContexts = {FALSE}` option is being used on the FPE. The calls concerned are `core_disable`, `core_enable` and `veneer_coredisabled`; again, details of these calls may be found in the `main.s` comments.

6 Multiple coprocessors

The undefined instruction handler generated by the simple configuration described above is designed to be branched to directly from the ARM's undefined instruction vector at address 0x00000004. When entered, it does the following:

- Stores the entry values of all the registers, the CPSR and the SPSR on the stack;
- Fetches the instruction that caused the undefined instruction trap from memory;
- Determines whether this is indeed a coprocessor instruction by examining bit 27 of the instruction (which is 1 for all coprocessor instructions, 0 for all other undefined instructions);
- If it is a coprocessor instruction, it determines whether it is a floating-point instruction by determining whether the coprocessor number field in bits 11:8 of the instruction is 0001 or 0010.
- If it is not a floating-point instruction, then everything stored on the stack (the entry values of all the registers, the CPSR and the SPSR) is reloaded and control is transferred to the routine whose address was passed in r1 to the `core_initws` routine at the time that the FPASC/FPE was initialised (see above).
- If it is a floating-point instruction, more detailed examination of the instruction and/or hardware status takes place, ending up with one of the following occurring.

An FPA exception is processed by the FPASC:

The instruction is emulated by the FPE or FPASC. The instruction is decoded to be an unallocated floating-point instruction: in this case, it is treated just as a non-floating-point instruction.

This behaviour of the undefined instruction handlers is selected by including the line:

```
UndefHandlerConvention SETS StandAlone
```

in the veneer's `options.s` file.

The idea is that a system is initialised with the undefined instruction vector pointing to an `Unknown instruction` error routine. When the FPASC/FPE is initialised, the current destination of the undefined instruction vector is passed to `core_initws` in r1. Thereafter, the FPASC/FPE will filter out and process floating-point instructions, while others will be passed on to the previous destination of the undefined instruction vector.

If a system contains support code for other coprocessors, or emulation code for other coprocessor instruction sets, or emulation code for undefined instructions, each such piece of code can adopt the same approach. The net

result is that the instruction follows the chain of undefined instruction handlers (in reverse order of initialisation), until either it is recognised and processed by one of them, or it reaches the original `Unknown instruction` error routine.

This scheme is simple, but has one drawback: each undefined instruction handler will take a substantial number of cycles (50+) just to decide that it does not recognise the instruction. As a result, with a lot of undefined instruction handlers on the chain, it may be several hundred cycles before the later ones are processed. This can severely impact floating-point performance, particularly in the case of the FPE.

If a system contains just one performance-sensitive undefined instruction handler, this can be solved by making that undefined instruction handler be the last one to be initialised. Some systems, however, will contain more than one performance-sensitive undefined instruction handler, or will not have this level of control over the order in which undefined instruction handlers are initialised.

For such systems, an alternative scheme may be used. Under this scheme, the operating system supplies the initial parts of the undefined instruction handler, which does the following:

- Stores the entry values of all the registers, the CPSR and the SPSR on the stack.
- Fetches the instruction that caused the undefined instruction trap from memory.
- Determines whether this is indeed a coprocessor instruction by examining bit 27 of the instruction (which is 1 for all coprocessor instructions, 0 for all other undefined instructions). If it is not a coprocessor instruction, control is passed to a handler for non-coprocessor undefined instructions.
- If it is a coprocessor instruction, it extracts the coprocessor number field in instruction bits 11:8 and uses them as an index into a table of undefined instruction handler addresses. Control is then passed to the selected undefined instruction handler.

The result is that, provided the various undefined instruction handlers do not conflict in their use of coprocessor numbers, registers are only stored on the stack once, and the offending instruction fetched and decoded once before the correct undefined instruction handler is entered. This saves a lot of time compared with the `StandAlone` scheme, in which these operations occur multiple times for all but the last undefined instruction handler to be initialised.

To use this scheme, carry out the following:

Include the definition:

`UndefHandlerConvention` SETS `BranchTable` in the veneer's `options.s` file.

The operating system undefined instruction handler should be coded as follows:

```
SUB r13,r13,#16*4          ;Claim 16 words of stack
                           ;space
STMIA r13,{r0-r14}^        ;Store user mode r0-r14
                           ;in 15 words
STR r14,[r13,#15*4]        ;And store return link in
                           ;other word
MOV     r12,r13            ;Set r12 to point to
                           ;stackedregisters
MRS     r10,CPSR_all       ;Preserve CPSR and SPSR
                           ;on stack
MRS     r9,SPSR_all
STMFD   r13!,{r9,r10}
TST     r9,#0xF            ;Is caller in user mode?
SUB     r9,r14,#4          ;Address the undefined
                           ;instruction
LDREQT  r11,[r9]           ;Fetch the undefined
                           ;instruction,
LDRNE   r11,[r9]           ;using unprivileged
                           ;access if caller was in
                           ;user mode(for security)
TST     r11,#0x08000000    ;Test if a coprocessor
                           ;instruction
ANDNE   r10,r11,#&F00      ;If so, isolate the
                           ;coprocessor
ADRNE   r9,BranchTable     ;number and use it to set
                           ;the PC to
LDRNE   r15,[r9,r10,LSR#6];the right branch table
                           ;entry
LDR     r15,RealUndef      ;If not, set the PC to
                           ;the address of the
                           ;correct handler
```

`BranchTable` should be the address of a table containing the entry points for the handlers for undefined instructions belonging to the 16 coprocessors, and `RealUndef` the address of a word containing the entry point for the handler for non-coprocessor undefined instructions. All 17 of these entry points would typically be initialised to the address of an `Unknown instruction` error routine.

The code above can be varied in minor ways, as long as:

- It leaves r11-r14 set precisely as the above code does.

  In other words:

  r11 = the instruction that caused the undefined instruction trap.

  r12 = pointer to 16-word dump of user mode register r0-r14 values, followed by the return link, with the entry CPSR stored at [r12,#-4] and the entry SPSR at [r12,#-8].

  r13 = stack pointer.

  r14 = return link.

- It leaves the processor in “Undef” mode.

- It does not enable interrupts, even temporarily.

The entry and exit conditions for some of the routines described above change as follows:

`veneer_newhandlers`

Install a pair of routines as the current undefined instruction handlers for coprocessors 1 and 2.

On Entry:

r1 is the address of new undefined instruction handler for coprocessor 1;

r2 is theaddress of new undefined instruction handler for coprocessor 2;

r13 is the stack pointer;

r14 is the return link.

On exit:

All registers except R14 are preserved.

`core_initws`

This is a call to initialise the FPASC/FPE workspace, and to determine what floating-point hardware is present, if any.

On Entry:

r1 is the address of the undefined instruction handler to pass control to if the instruction encountered is an unallocated coprocessor 1 floating-point instruction;

r2 is the address of the undefined instruction handler to pass control to if the instruction encountered is an unallocated coprocessor 2 floating-point instruction;

r10 is the pointer to workspace/context;

r13 is the stack pointer;

r14 is the return link.

On Exit:

r0 is SysId of hardware found (in bottom byte), or -1 if no hardware found; All other registers except r14 are preserved.

Typically, `core_initws` is called during initialisation with r1 and r2 set to the current contents of the addresses `BranchTable+4` and `BranchTable+8` respectively, and `veneer_newhandlers` with r1and r2 set to the desired new contents of these locations.

Note also that the routines `veneer_preservehandlers` and `veneer_restorehandlers` described above will typically need to be changed to act on both the coprocessor 1 and coprocessor 2 undefined instruction handlers.

7 What this Application Note does not cover

Use of the FPASC/FPE with ARMs in 26-bit configuration is not covered. These are:

- ARM2 and ARM3 processors;
- ARM6- and ARM7-based processors (but not ARM7T-based processors) with the PROG32 configuration input (or bit 4 of coprocessor 15's register1) set to 0.

Details of this can be found in the comments in the `main.s` source file, but note that use of ARMs in 26-bit configuration is only recommended when it is needed for backwards compatibility reasons. New systems should use ARMs in 32-bit configuration.

Details of the trap handler routines.

These can be found in the `main.s` source file comments