

Application Note **98**

VFP Support Code

Document number: ARM DAI 0098

Issued: August 2002

Copyright ARM Limited 2002

ARM

Application Note 98 VFP Support Code

Copyright © 2002 ARM Limited. All rights reserved.

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
August 2002	A	First release

Proprietary notice

ARM, the ARM Powered logo, Thumb and StrongARM are registered trademarks of ARM Limited.

The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, ARM7TDMI, ARM9TDMI, TDMI and STRONG are trademarks of ARM Limited.

All other products, or services, mentioned herein may be trademarks of their respective owners.

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

<http://www.arm.com>

Table of Contents

1	Introduction	2
1.1	Floating-point support	2
1.2	VFP variants	2
1.3	What does the application note cover?	2
2	Producing Code to Run on a VFP System	4
2.1	Using the compiler	4
2.2	Using the assembler	4
2.3	Which options should I use?	5
3	VFP System Initialization	6
3.1	Provide the VFP Support Code	6
3.2	Install the VFP support code	6
3.3	Set up the undefined mode stack	7
3.4	Enable VFP hardware	7
3.5	Enable VFP internally	8
3.6	Installation routine	8
4	Debugger Setup for VFP	9
4.1	Configuring ARMulator for VFP	9
4.2	Configuration of AXD for VFP	9
4.3	Enabling VFP registers within the debugger	11
5	Example application using VFP	12
6	VFP Support Code	13
6.1	Features of the VFP Support Code	13
6.2	Overview of processing a bounced VFP instruction	15
6.3	Processing a bounced VFP instruction	16
6.4	Context switching VFP state	21
6.5	VFP Computation Engine	21
6.6	VFP Subarchitecture Support	23
7	References	25

1 Introduction

1.1 Floating-point support

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

- 1) The software floating-point library (fplib), supplied as part of the ARM Developer Suite C library, provides 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.
- 2) 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 architectures:
 - Floating-point Accelerator (FPA), as used for example in the ARM7500FE. This is now becoming obsolete.
 - Vector Floating-Point (VFP), which was originally developed as part of the ARM10 program. This implements IEEE floating-point and supports single and double precision, but not extended precision.
- 3) 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 an emulation of the coprocessor is provided which is attached to the ARM processor core's undefined instruction trap.

In reality floating-point arithmetic in both FPA and VFP is actually implemented using a combination of hardware (which executes the common cases) and software (which deals with the uncommon and exceptional cases).

1.2 VFP variants

VFP is a floating-point architecture which can provide both single and double precision operations. Many operations may also take place in scalar form or in vector form. At the time of writing two versions of the architecture have been implemented:

- VFPv1 was implemented in the VFP10 revision 0 silicon (as provided by the ARM10200).
- VFPv2 has been implemented in the VFP10 revision 1 (as provided by the ARM10200E) and in the VFP9-S.

In addition particular implementations may provide implementation-specific functionality. For example, the VFP coprocessor hardware can include extra registers describing an exceptional condition. These registers are not described in the VFP architecture, yet the operating system needs to know about them when handling the exception, and sometimes when saving VFP context.

This extra functionality is known as the *subarchitecture* of the implementation. This must be relied on only by system software, and only as described in this application note. Functions that depend on subarchitecture functionality should also be separated from the main body of the system software, so that it is easy to change to another VFP implementation. All other software must only rely upon the general architectural definition of the VFP architecture contained in the *ARM Architecture Reference Manual*.

1.3 What does the application note cover?

This application note contains the following:

1. An explanation of how to build floating-point code to run on a VFP-based system.

2. An explanation of how to set up a system so that the VFP can execute code.
3. A description of how to set up the debug tools so that VFP code can be loaded and executed.
4. A description of an implementation of the software components required to provide a working VFP system. These software components are commonly referred to as the VFP Support Code. These components are provided with this application note.
5. Application test code which can be used to check that the VFP Support Code has been successfully installed into a system.

All code provided with this application note should be built using either ADS 1.2 or RVCT 1.2. Use of the code with earlier versions of ADS is not supported.

2 Producing Code to Run on a VFP System

2.1 Using the compiler

By default the compiler generates code that makes calls to a software floating-point library routine in order to carry out floating-point operations. To make use of VFP instructions you must use appropriate compiler options to modify the code generated:

-fpu vfp	Selects hardware vector floating-point unit conforming to architecture VFPv1. This is a synonym for <code>-fpu vfpv1</code> . This option is not available for the Thumb compilers.
-fpu vfpv1	Selects hardware vector floating-point unit conforming to architecture VFPv1, such as the VFP10 rev 0. This option is not available for the Thumb compilers.
-fpu vfpv2	Selects hardware vector floating-point unit conforming to architecture VFPv2, such as the VFP10 rev 1. This option is not available for the Thumb compilers.
-fpu softvfp+vfp	Selects a floating-point library with software floating-point linkage that uses the VFPv1 hardware. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFPv1 unit. If you select this option: <ul style="list-style-type: none"> • <code>tcc</code> and <code>tcpp</code> behave exactly as for <code>-fpu softvfp</code> except that they link with VFP-optimized floating-point libraries. • <code>armcc</code> and <code>armcpp</code> behave the same as for <code>-fpu vfp</code> except that all functions are given software floating-point linkage. This means that ARM functions compiled with this option pass and return floating-point arguments and results as they would for <code>-fpu softvfp</code>, but use VFP instructions internally.
-fpu softvfp+vfpv2	Selects a floating-point library with software floating-point linkage that uses the VFPv2 hardware. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFPv2 unit.

Note *If you specify `-fpu softvfp+vfp` for both `armcc` and `tcc`, it ensures that your interworking floating-point code is compiled to use software floating-point linkage. If you specify `vfp`, `vfpv1`, or `vfpv2` for `armcc` you must use the `__softfp` keyword to ensure that your interworking ARM code is compiled to use software floating-point linkage. See the description of `__softfp` in the ADS 1.2 Compilers and Libraries Guide for more information.*

Note *The compiler only generates scalar floating-point operations. If you want to use the VFP's vector operations, then you must do this using assembly code.*

Note *Some of the compiler's `-cpu` options imply a floating-point unit. So, for example, if you select `-cpu ARM10200`, this implies `-fpu VFPv1`. If you also specify a specific `fpu` option, this overrides any implied `fpu` setting.*

2.2 Using the assembler

By default, the assembler faults the use of VFP instructions. To enable the assembly of such instructions, appropriate options need to be used. These options enable the use of VFP instructions in ARM code and modify the build attributes to enable the linker to determine the compatibility between object files, and to select appropriate libraries. They do not modify the code that is actually generated.

-fpu vfp	This is a synonym for -fpu vfpv1.
-fpu vfpv1	Selects hardware vector floating-point unit conforming to architecture VFPv1.
-fpu vfpv2	Selects hardware vector floating-point unit conforming to architecture VFPv2.
-fpu softvfp+vfp	Selects hardware vector floating-point unit. To armasm, this is identical to -fpu vfpv1.
-fpu softvfp+vfpv2	Selects hardware vector floating-point unit. To armasm, this is identical to -fpu vfpv2.

2.3 Which options should I use?

The following rules can be used to help you select the most suitable floating-point build options to use for your application:

1. If all of your floating-point processing is done in ARM state code and none is done in Thumb state code then
 - for ARM code:
 - If you are using VFP10 rev 0 then build your ARM code with `-fpu vfpv1`
 - If you are using VFP10 rev 1 then build your ARM code with `-fpu vfpv2`
 - If you are using VFP9-S then build your ARM code with `-fpu vfpv2`.
 - for Thumb code build with `-fpu none`
 - This ensures that no floating-point operations take place in Thumb code.
2. If your floating-point processing is done in a mixture of ARM state code and Thumb state code then
 - If using VFP10 rev 0 then build your code using `-fpu softvfp+vfpv1`
 - If using VFP10 rev 1 then build your code using `-fpu softvfp+vfpv2`
 - If using VFP9-S then build your code using `-fpu softvfp+vfpv2`.

3 VFP System Initialization

To use VFP in your application, there are a number of steps that need to have been carried out before floating-point operations can be executed. These steps might need to be done in your initialization code, or may be done by your operating system.

1. Ensure the VFP Support Code is part of the system software.
2. Install the VFP Support Code on to the undefined instruction vector.
3. Ensure that there is a valid stack for undefined mode.
4. In some development systems, such as ARM's Integrator CM10200 board, you might have to enable the VFP unit in hardware.
5. Enable the VFP coprocessor internally by setting the VFPEnable bit in the VFP's FPEXC register. Note that at reset the VFP coprocessor will be disabled.

These steps are discussed in more detail in the following sections

3.1 Provide the VFP Support Code

Provided with this application note is an implementation of the VFP Support Code that can be used in your system. In the past, earlier implementations have been provided, for example as part of the ARM Firmware Suite. We now advise the use of the code provided with this application note.

In some applications VFP Support Code is linked in as part of the image. In others it is provided linked in as part of the system environment or operating system.

3.2 Install the VFP support code

Trying to execute VFP instructions not implemented by the VFP hardware or executing "exceptional cases" causes the ARM to take an undefined instruction exception. This is sometimes known as *bouncing* the instruction. The VFP support code must therefore be installed on to the undefined instruction vector before floating-point operations take place.

In an embedded application, initialization code installs the VFP Support Code into the vector table along with the other exception handlers (typically using scatterloading).

Alternatively during early development work, a simple patch function can be called to install an appropriate branch instruction into the undefined instruction vector table entry. The following assembler code example shows how this can be done

This example assumes that

- the VFP Support Code is located in memory within the first 32MB of memory so that a branch instruction can be used
- caches are disabled so that no cache flush/clean is required.
- high vectors are disabled
- page 0 is writable in the current mode

```

UNDEF_VECTOR    EQU    0x4            ; address of undefined instruction vector
                EXPORT  Install_VFPHandler
                IMPORT  TLUndef_Handler

Install_VFPHandler                ; function to install the VFP support code
                                ; on to undefined instruction vector

                LDR    r0, =TLUndef_Handler    ; get address of support code
                SUB    r0, r0, #UNDEF_VECTOR+8 ; allow for vector address and PC offset

```

```

MOV r0, r0, LSR #2
ORR r0, r0, #0xea000000 ; bit pattern for Branch always
MOV r1, #UNDEF_VECTOR
STR r0, [r1] ; store instruction to undefined vector
BX LR ; return from subroutine

```

3.3 Set up the undefined mode stack

As at least some of the VFP Support Code executes in undefined instruction mode, it is necessary to have an undefined mode stack set up. In a fully embedded system that runs from reset, then this is likely to have been done within the initialization code.

In the case of early development code downloaded via a debugger, then a simple function can be called to set up the stack. The following assembler code example shows how this can be done. Note that this must be executed in a privileged mode.

```

Mode_UNDEF EQU 0x1B ; bit pattern for undefined mode
UNDEF_Stack EQU 0x4000 ; location for undefined mode stack
EXPORT Setup_Undef_Stack
Setup_Undef_Stack ; function to set up a stack for undefined mode
MRS r0, CPSR ; get CPSR value
MOV r1, r0 ; take a working copy
BIC r1, r1, #0x1F ; clear mode bits of CPSR
ORR r1, r1, #Mode_UNDEF ; set mode bits for Undefined mode
MSR CPSR_c, r1 ; change to undefined mode
LDR SP, =UNDEF_Stack ; set up the stack pointer
MSR CPSR_c, r0 ; change back to the original mode
BX LR ; return from subroutine

```

Note *The support code provided with this application note runs largely in SVC mode and only a small amount of stack space is required for undefined mode operation. This means that you also have to ensure that you have the SVC mode stack set up (though this is normally the case). See section 6 for more details.*

3.4 Enable VFP hardware

In some systems the VFP might be disabled at reset by external hardware. Therefore you must turn the hardware on before any accesses at all to the VFP unit can take place.

For example with the ARM Integrator/CM10200 boards this is carried out by clearing the VFPTST bits of the CM_INIT register. The following assembler code example shows how this can be done:

```

CM_BASE EQU 0x10000000
CM_INIT EQU 0x24
CM_LOCK EQU 0x14
CM_LOCK_VALUE EQU 0xA05F
VFPTST EQU 2_11:SHL:12
EXPORT CM_VFP_enable
CM_VFP_enable
; Enable Integrator/CM10200 to allow use of VFP
MOV r0, #CM_BASE
LDR r2, =CM_LOCK_VALUE
STR r2, [r0,#CM_LOCK]
LDR r1, [r0,#CM_INIT]
BIC r1, r1,#VFPTST
STR r1, [r0,#CM_INIT]
MOV r2, #0
STR r2, [r0,#CM_LOCK]
BX LR

```

Note *This enabling of the VFP by external hardware is required by Integrator/CM10200 (ARM1020T (rev 0) based), but not by the later Integrator/CM10200E (ARM1020E (rev 1) based).*

3.5 Enable VFP internally

On all systems, it will be necessary to enable the VFP by setting the VFPEnable bit in the VFP's FPEXC register. Until this is done, the VFP coprocessor is disabled and any other access to the VFP causes an undefined instruction exception. When reinitializing the VFP it is also necessary to reset the EX bit in this register, to clear any pending exceptions. This operation must be carried out in a privileged mode.

The following assembler code example shows how this can be done:

```
VFPEnable      EQU      0x40000000
                EXPORT  Enable_VFP
Enable_VFP     ; Enable the VFP itself
                MOV     r1, #VFPEnable
                FMXR   FPEXC, r1      ; FPEXC = r1
                BX     LR
```

3.6 Installation routine

One important point to note about the preceding steps is that they must all be carried out before the C library's floating-point initialization takes place. This is done by the library routine `_fp_init()`.

The easiest way to do this is to write a simple function that calls the required routines from sections 3.1-3.5 which armlink can automatically cause to be executed directly before `_fp_init()` is executed. This is done using the `$$Sub$$` and `$$Super$$` functionality, which is detailed in the *ADS 1.2 Linker and Utilities Guide*. The following C code example shows how this can be done:

```
extern void CM_VFP_enable (void);
extern void Setup_Undef_Stack (void);
extern void Enable_VFP (void);
extern void Install_VFPHandler (void);
extern void $$Super$$_fp_init(void);

// patch function to be called directly before _fp_init()
void $$Sub$$_fp_init(void)
{
    Install_VFPHandler();
    Setup_Undef_Stack();
    CM_VFP_enable();
    Enable_VFP();
    $$Super$$_fp_init(); // call real _fp_init() function
}
```

4 Debugger Setup for VFP

4.1 Configuring ARMulator for VFP

The ARMulator provides models of the VFP architecture that can run VFP instructions. However as these model the architecture rather than specific implementations, they cannot be used to accurately benchmark the floating-point performance that will be obtained in a real system

To configure the ARMulator to use the VFP models, within AXD select the “Options -> Configure Target” menu. Then ensure that “ARMUL” is the selected target and click on the Configure button. This displays the ARMulator Configuration dialog box.

You can then change the Processor and Floating-Point Coprocessor settings to match your requirements:

The screenshot shows the ARMulator Configuration dialog box with the following settings:

- Processor:** Variant: ARM1020T
- Clock:** Emulated (selected), Speed: []
- Options:** Floating Point Emulation (unchecked)
- Debug Endian:** Little (selected), Big (unchecked)
- Start target Endian:** Debug Endian (selected), Hardware Endian (unchecked)
- Memory Map File:** No Map File (selected), Map File (unchecked), [] Browse
- Floating Point Coprocessor:** FPU: VFP_NOSUP
- MMU/PU Initialization:** Pagetab: NO_PAGETABLES

Callouts provide the following instructions:

- Select an appropriate processor:** Note that if you select a processor that includes a built-in VFP coprocessor, such as ARM10200, you must leave the FPU setting below set to “NO_FPU”, otherwise the behavior of the ARMulator is unpredictable.
- Select an appropriate FPU setting:** If you wish to emulate a VFPv1 FPU, select VFP_NOSUP. If you wish to emulate a VFPv2 FPU, select VFPV2.
- Select appropriate MMU/PU settings:** Set to NO_PAGETABLES to disable ARMulator’s default cache initialization.

4.2 Configuration of AXD for VFP

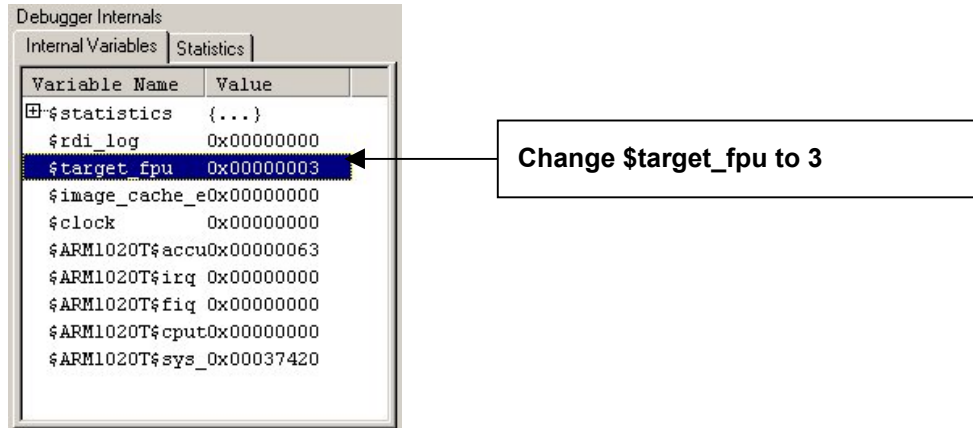
This section covers debugger internal variables that must be set correctly when debugging a system containing a VFP. More details are also in the *ADS 1.2 AXD and armsd Debuggers Guide*.

4.2.1 \$target_fpu

This debugger internal variable controls the way that floating-point numbers are displayed by the debugger. When using a VFP coprocessor (either within ARMulator or in real

hardware), it is important to set this to the correct value to ensure the correct display of float and double values in memory. If this is not done, then you get an error message from AXD when loading an image built for VFP.

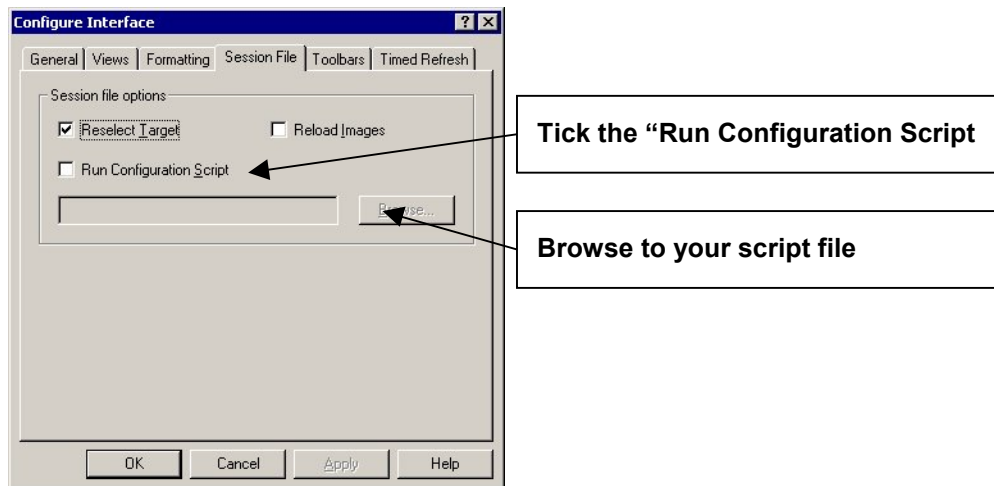
To change \$target_fpu from the AXD GUI select the menu “System Views -> Debugger Internals”:



This can also be done through the AXD command line interface using:

```
let $target_fpu,3
```

You might want to put this command into a script file, which can then be run automatically by AXD on starting up, using the Options -> Configure Interface (Session File tab) dialog:



An example script file, vfp_script.txt, is in the \utils subdirectory of the associated code suite.

Note You will need to select \$target_fpu each time you start AXD unless you automatically run a script file on start up.

4.2.2 vector_catch

As the VFP Support Code is installed on to the undefined instruction vector table entry, it is also necessary to ensure that the debugger is not trapping undefined instructions using its “vector_catch” function. To ensure that this is disabled, either use the “Options-> Configure Processor” menu, or use the following AXD CLI command:

```
spp vector_catch u
```

Again, this can also be placed in an AXD script file, as in the example script file, vfp_script.txt, which can be found in the \utils subdirectory of the associated code suite.

4.3 Enabling VFP registers within the debugger

AXD displays the VFP registers only when it believes that the target system contains a VFP unit. If ARMulator has been configured to use a VFP (as described above), then this will be handled automatically.

However if you are connecting to real hardware via, for example, Multi-ICE, then it is possible that the VFP unit might not get detected. This happens, for example, when autoconfiguring Multi-ICE to connect to an Integrator/CM10200. As described in section 3.4, the VFP is disabled on this board at powerup. Multi-ICE only detects an ARM1020T. This means that the VFP registers will not be displayed in AXD. There are three possible mechanisms to enable the VFP registers to be displayed in this case:

1. When the VFP is turned on (for example by running the `CM_VFP_Enable()` function), if you then autoconfigure again, Multi-ICE is able to detect the VFP and identifies an ARM10200T processor, and the VFP registers can then be seen in AXD.
2. You can manually configure Multi-ICE to use an appropriate processor which does contain a VFP unit (for example ARM10200T). An example manual configuration file is provided in the `\utils` subdirectory of the associated example code. Details of how to use manual configuration files are in the *Multi-ICE 2.2 User Guide*.
3. You can tell AXD that if it connects to a particular processor (for example ARM1020T), then there is actually a VFP unit in the system too. This can be done by providing an appropriate `.xml` file. An example one for the ARM1020T processor is provided in the `\utils` subdirectory of the associated example code. To use this, copy it into the `\bin` directory of your ADS installation.

5 Example application using VFP

A simple application is provided as part of the associated code suite, as main.c. This carries out two floating-point operations, one of which is executed by the VFP hardware, the other of which uses a denormalized number and therefore bounces to the VFP Support Code to handle. This provides a simple test that the VFP support code has been successfully integrated into a system.

```
#include <stdio.h>

double VFPTest(double x, double y)
{
    return x + y;
}

int main(void)
{
    double val1 = 1.0;
    double val2 = 2.0;
    double val3 = 3.1234e-322; // denormalized number
    double res1;
    double res2;
    printf("VFP Support Code Test\n");
    printf("=====\n\n");
    printf("Non-bouncing calculation of %f + %f\n", val1, val2);
    printf("The result should be : 3.000000\n");
    printf("The result is          : %f\n\n", res1=VFPTest(val1, val2));
    printf("Bouncing calculation of %e + %e\n", val3, val3);
    printf("The result should be : 6.225227e-322\n");
    printf("The result is          : %e\n\n", res2=VFPTest(val3, val3));
    return 0;
}
```

Details of how to build and run this code can be found in the readme.txt of the associated example code.

- Note** *In fact, the code above should bounce to the support code four times, twice for the printf which displays val3, once for the denormalized addition itself in VFPTest(), and once for the printf that displays the result of the denormalized addition.*
- Note** *The result 6.225227e-322 is correct, even though the mathematical error is large. Calculations using small denormalized numbers are significantly less accurate than those using normalized arithmetic.*

6 VFP Support Code

This section describes the VFP support code in detail.

Read this section if you have to integrate the VFP support code with an operating system.

6.1 Features of the VFP Support Code

The VFP Support Code provides a VFP system with a mechanism of dealing with uncommon and exceptional instructions that are not dealt with directly by the VFP coprocessor hardware.

The support code provided with this application note supports a fully IEEE 754 compliant floating-point model when used in conjunction with a VFP coprocessor. The support code does not provide a complete software implementation of the VFP architecture (VFP emulation software).

New floating-point models introduced by enabling the VFP architecture options Flush-to-Zero and DefaultNaN mode are not supported in the support code provided. However, support for these options is provided entirely in hardware by the RunFast mode of the VFP9-S and VFP10 rev 1 implementations of the VFP architecture. Software support code is not required in RunFast mode.

6.1.1 VFP Support Code files

The support code provided with this application note is made up of a number of files.

The following files are provided in source form in the `\vfp_support` subdirectory of the associated example code. These files might have to be modified when integrating with an operating system.

controlbuffer.c	Buffer used to transfer information from the top-level handler to the computation engine.
controlbuffer.h	C header for controlbuffer.c
controlbuffer_h.s	Assembler header for controlbuffer.c
slundef.h	Interface to second level undef handlers
tlundef.s	Top-level handler which identifies the cause of the undefined instruction exception and takes the appropriate action
sldummy.s	Dummy coprocessor and undef handlers for use in example code
vfpfptrap.s	Provides a wrapper around the standard <code>_fp_trap</code> handler in the C Library for the computation engine to call
vfpundef.c	Called from top-level undef handler once a CP10,CP11 bounce is identified, registers are saved and mode is switched
vfpwrapper.s	Provides a wrapper around the Computation Engine for the top level handler to call

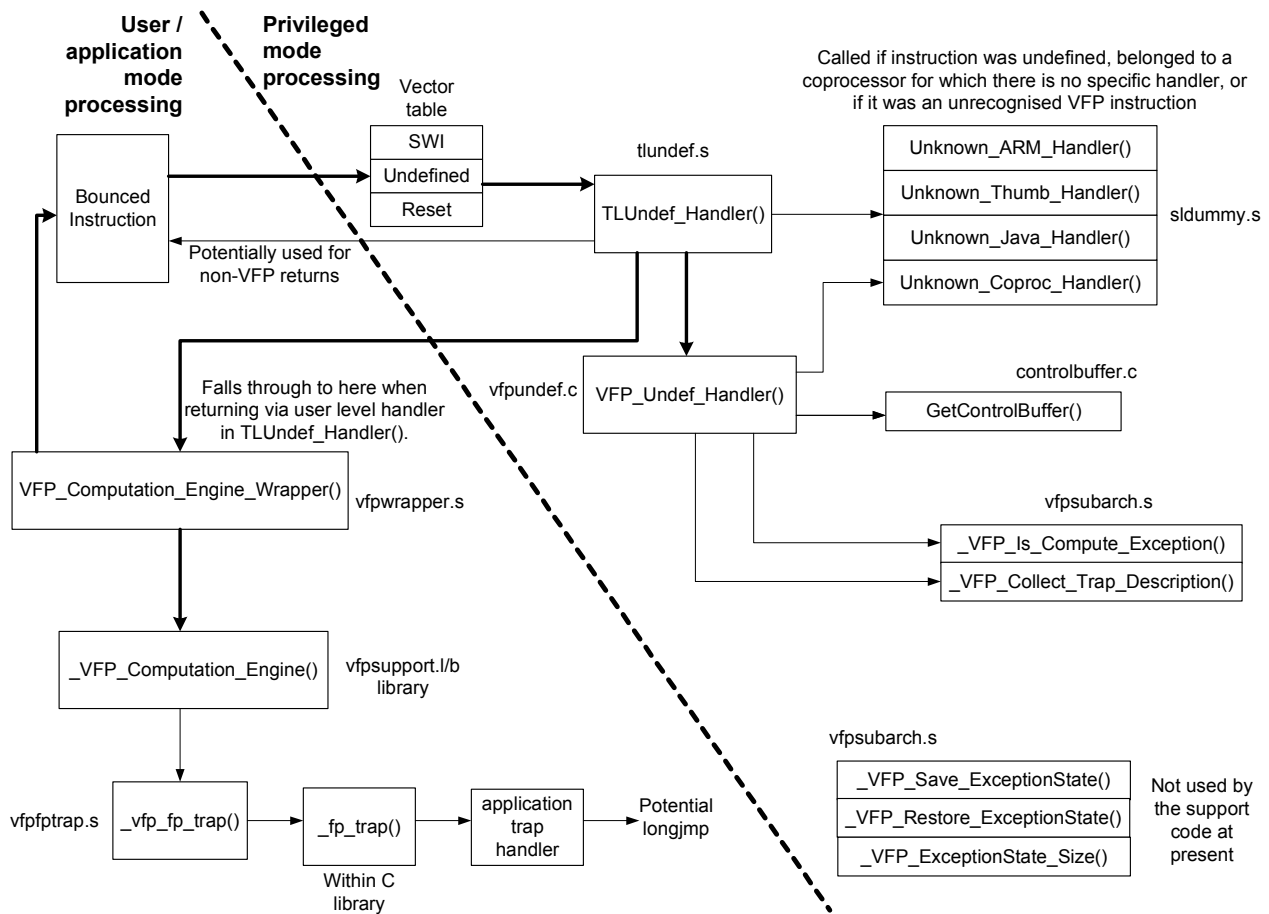
The following files are provided in source format in the `\vfp_support` subdirectory of the associated example code. These files contain functions to access the subarchitecture defined functionality of the VFP9/VFP10 family of VFP coprocessors.

vfpsubarch.h	Header for vfpsubarch.s
vfpsubarch.s	Subarchitecture support.

The following files are provided in library form, and should be installed in the lib\arm\lib subdirectory of your ADS or RVCT installation.

vfpsupport.b	Computation Engine of the VFP Support Code (big-endian version). The computation engine emulates VFP operations that the hardware has bounced.
vfpsupport.l	Computation Engine of the VFP Support Code (little-endian version)

The relationships between these files are shown in the following diagram:



6.2 Overview of processing a bounced VFP instruction

The VFP support code is entered when the VFP coprocessor bounces an instruction. This causes the processor to signal an Undefined Instruction exception.

6.2.1 Top-level Undefined Instruction handler

The top-level Undefined Instruction handler `TLUndef_Handler()` saves the processor state, calls an appropriate second-level handler, and then restores the saved state. There are separate second-level handlers for each coprocessor and for each type of undefined instruction.

6.2.2 Second-level VFP Undefined Instruction handler

The VFP second-level handler `VFP_Undef_Handler()` first checks the VFP is enabled and the exception was not caused by an illegal instruction. If either check fails the handler calls `Undef_Coproc_Handler()` to process the bounce.

Otherwise the handler reads exception information out of the coprocessor to construct a list of operations that must be processed in software, and resets the VFP coprocessor's exception mechanism. It then prepares arguments for a user-level exception handler which will process the instruction list, and returns control to the top-level handler.

Note *VFP_Undef_Handler() uses SUBARCHITECTURE DEFINED functions to access and interpret the exception information in the coprocessor. Suitable functions for VFP9 and VFP10 are provided in `vfpsubarch.s`. See Section 6.6 "VFP Subarchitecture Support".*

6.2.3 Returning to the top-level handler

The second-level handler returns to the top-level handler, where the application's context is restored.

The second-level handler might raise a user-level exception. The second-level VFP handler does this, and the top-level handler returns by transferring control and parameters to the user-level VFP handler instead of returning directly to the bounced instruction.

6.2.4 User-level VFP handler

The user-level VFP handler processes bounced operations in the application's context using the VFP computation engine software provided.

As floating-point exceptions are encountered calls can be made to application trap handlers, as specified by *ANSI / IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*. The application trap handlers can either return substitute results, or abort processing with `longjmp`. The support code uses the floating-point trap handler mechanism in the ARM C library, as described in *ADS 1.2 Compiler and Libraries Guide*.

Note *A single VFP instruction can generate multiple floating-point traps. This is possible only if it is made up of multiple operations, as are vector instructions and multiply-accumulate instructions.*

When all bounced operations have been processed the user-level exception handler returns to an address provided by the top-level Undefined Instruction handler. For imprecise exceptions this is the address of the instruction that bounced. For precise exceptions this is the address of the following instruction.

6.3 Processing a bounced VFP instruction

6.3.1 Top-level Undefined Instruction handler

The top-level Undefined Instruction handler `TLUndef_Handler()` saves the processor state, calls an appropriate second-level handler, and then restores the saved state. There are separate second-level handlers for each coprocessor and for each type of undefined instruction.

This section describes example top-level handler code provided as `TLUndef_Handler()` in `tlundef.s`.

Overview

After doing some initial state saving, `TLUndef_Handler()` first of all checks whether the bounced instruction was a coprocessor instruction or not.

If the instruction that bounced was not a coprocessor instruction then one of `Undef_ARM_Handler()`, `Undef_Thumb_Handler()`, or `Undef_Java_Handler()` is called, depending on the type of instruction that bounced. In the supplied code these are simply dummy routines that go in to an infinite loop.

It then checks which coprocessor the instruction belongs to. The coprocessor number is then used in a table lookup to find the address of a suitable second-level handler. In the case of VFP the number will be decimal 10 or 11 and the appropriate function is `VFP_Undef_Handler()`. By default all other coproc numbers are vectored on to `Unknown_Coproc_Handler()`.

On return from the second-level handler the saved state is restored, and a user-level exception handler may be called. This is described in Section 6.6.3.

Initial state saving

When the ARM core signals an Undefined Instruction exception it copies the current execution state to the `R14_undef` and `SPSR_undef` registers, disables IRQ interrupts, and starts executing the code at the UNDEF vector in ARM state.

The UNDEF vector contains a jump to the top-level handler code.

The top-level handler immediately re-enables IRQ interrupts.

As a consequence interrupt handlers that wishes to use VFP must save the contents of `R14_und` and `SPSR_und` to avoid corrupting return information stored there, and possibly also save the contents of the Undef stack if it is small.

This code effectively relies on the banked register state obeying the following hierarchy:

Highest priority	Fiq, Abort, Irq
	Undef
	Svc
Lowest priority	User

At any point where an exception corresponding to banked registers can occur the registers must not contain live values. Thus Undef exceptions must not be allowed when in Fiq, Abort, Irq or Undef mode.

After enabling interrupts the top-level handler switches to Supervisor mode. It constructs a complete dump of the application's register state on the Supervisor stack, and copies the return information out of `R14_und` and `SPSR_und`, switching modes where necessary.

Note *If the exception occurred in Supervisor mode the `R13_svc` register is modified to allocated stack space for the register dump, before it is saved in to the register dump. In this case second-level handlers may not read or modify the saved value of `R13`. As a consequence if VFP instructions are used by Supervisor mode code, then this handler is not suitable for*

building a software implementation of the VFP architecture, where instructions that copy data to and from the VFP are emulated.

The top-level handler makes temporary use of two words of Undef stack, but mostly runs in Supervisor mode with the Undef stack empty. This reduces the need for a true Undef stack, and so reduces the total amount of stack space required by the system.

The top-level handler should be modified on systems that do not use the Supervisor stack for handling Undef exceptions.

Calling a second-level handler

The saved PSR is checked to find out if the processor was in ARM or Thumb state, and the instruction that caused the exception is loaded. If Thumb, `Unknown_Thumb_Handler()` is called. If ARM, it is checked for a coprocessor instruction encoding, otherwise `Unknown_ARM_Handler()` is called.

A function `Unknown_Java_Handler()` is provided for exceptions in Java state (on processors conforming to ARMv5TEJ and above). However, the processor never generates Undefined Instruction exceptions due to Java execution.

For coprocessor instructions the coprocessor number is used to index in to a table of second-level handlers, `TLUndef_CpTable`. Unused coprocessor entries use `Unknown_Coproc_Handler()`.

The location, initialization and use of this table might be modified for systems that support other coprocessors, or dynamically load coprocessor second-level handlers.

Suitable implementations of `Unknown_ARM_Handler()`, `Unknown_Thumb_Handler()` and `Unknown_Coproc_Handler()` must be provided by the operating system. These are called for illegal instructions. They usually signal a fatal exception in the offending application.

The behavior of `TLUndef_Handler` when the second-level handler returns is described in Section 6.3.2.

Second-level handler prototype

All handlers have this ATPCS-compliant prototype:

```
__value_in_regs HandlerReturnType handler_function(
    uint32 instr, ARM_RegDump *regdump)
```

Where:

`instr` is the undefined instruction, where known.

`regdump` points to the register dump giving the processor state when the instruction bounced, including the CPSR. The handler might modify the contents of the register dump.

```
struct ARM_RegDump {
    uint32 reg[16];
    uint32 cpsr;
};
```

`HandlerReturnType` describes the two return values. These are returned in registers due to the “`__value_in_regs`” qualifier on the function definition.

```
struct HandlerReturnType {
    uint32 skip_instr;
    ControlBuffer *controlbuffer;
};
```

The `skip_instr` flag (r0) is usually set, to indicate that the bounced instruction has been processed. If the `skip_instr` flag is clear the top-level handler returns to and retries the bounced instruction. The VFP handler uses this for handling imprecise exceptions. When

an internal exception condition caused by one coprocessor instruction is signaled imprecisely, by refusing to respond to a later coprocessor instruction, the handler takes the action necessary to clear the exception condition, and then returns to the refused instruction.

The controlbuffer pointer (r1), if valid, points to a ControlBuffer describing a user-level exception. Instead of returning to the application the top-level handler continues application execution in a user-level exception handler as defined in section 6.3.3.

A NULL controlbuffer pointer indicates exception processing is complete, and a standard return to the application code can be carried out.

6.3.2 Second-level VFP Undefined Instruction handler

An example VFP second-level handler is VFP_Undef_Handler in vfpundef.c.

This should need no modification for systems that can process VFP bounces in a user-level exception handler. For systems that process all VFP bounces on the kernel stack, the creation of a user-level exception control buffer can be replaced with a simple call to `_VFP_Computation_Engine()`.

Overview

`VFP_Undef_Handler()` first checks the VFP is enabled, and if so it calls `_VFP_Is_Compute_Exception()` to check if it is a legal VFP instruction. If either check fails the handler calls `Undef_Coproc_Handler()` to process the bounce and `VFP_Undef_Handler()` will complete if it returns.

At this point, the Support Code knows the instruction is one that it needs to handle. The Support Code needs to handle it in the application's context so that the applications' floating-point trap handlers can be called when necessary. It is therefore necessary to create a data structure describing the exception that can be passed to the computation engine running in the same mode as the application. The Support Code also needs to arrange for `TLUndef_Handler()` to return to `VFP_Computation_Engine_Wrapper()` rather than directly back to the application code. The Support Code calls out to `GetControlBuffer` to allocate the necessary space. Note that this space must be thread local.

The Support Code then calls the `_VFP_Collect_Trap_Description()` to copy the exception description from the VFP coprocessor into the allocated space, and to reset the VFP coprocessor's exception mechanism. The exception description is a list of operations that must be processed in software.

Checking for illegal instructions

If any of the following conditions are true when an instruction is passed to the VFP, then the VFP bounces that instruction:

- the instruction is illegal, or
- the EN bit in the FPEXC is clear, and the instruction is not a privileged access to an exception register, (for example FPEXC or FPSID).
- there is a pending exception (the EX bit in the FPEXC is set) and the instruction is not a privileged access to an exception register.
- the VFP is signaling a precise exception for the current instruction.

The Support Code first checks that the VFP is enabled. If not the faulting instruction is passed on to the system's undefined coprocessor instruction handler.

Then the support code checks if the only reason for the bounce was an illegal instruction. If there is a pending exception and another reason for the bounce, then the pending exception must be dealt with first. Subsequently retrying the bounced instruction then triggers illegal handling.

The EX bit in the FPEXC signals a pending exception. If the EX bit is clear then illegal instruction bounces can be identified by checking the validity of the encoding of the instruction that bounced.

The subarchitecture support library provides a subarchitecture optimized function `_VFP_Is_Compute_Exception()` for recognizing illegal instruction bounces. This is described in Section 6.6.1.

If illegal handling is chosen, then VFP bounce processing is finished, and the faulting instruction is passed on to the operating system's undefined instruction handler.

Collecting exception information

When the reason for the bounce has been confirmed as a floating-point operation that needs software involvement, then the details of the bounce are collected and stored. These details include both the bounced instruction encoding and, if the EX bit is set, private exception information in the VFP.

The subarchitecture support library provides the `_VFP_Collect_Trap_Description()` function to gather and record this information. This is described in Section 6.6.1. This function writes a bounce description to the data block provided by the caller. The data block describes the VFP operations that must be emulated before continuing normal execution.

When the bounce details have been saved the support library clears the EX bit in the FPEXC.

This function returns a flag that indicates whether or not the return address should be incremented, so that the bounced instruction is not repeated.

The VFP second-level handler returns a ControlBuffer to the top-level handler, indicating that exception handling must continue in the application context, transferring control to the `VFP_Computation_Engine_Wrapper()` user-level exception handler.

The top-level handler restores the user context before continuing with VFP bounce processing. This is useful if FP traps are enabled, and full application-level IEEE Floating-point trap support is required, because it enables direct calls from the `_VFP_Computation_Engine()` to the application's `_fp_trap()` exception handler.

6.3.3 Returning to the top-level handler

The second-level handler returns to the top-level handler, where the application's context is restored.

The second-level handler might raise a user-level exception. In this case the top-level handler returns by transferring control and parameters to a user-level exception handler, instead of returning directly to the bounced instruction.

Returning directly to the application

On return, if the controlbuffer pointer is NULL, the top level handler restores saved registers, (as modified by the second-level handler), and switches back to the application, continuing execution either by retrying the refused instruction or from the following instruction, as specified by `skip_instr`.

Returning via a user-level exception handler

Alternatively the second handler might forward an exception on to an exception handler operating in the application context, that uses the application stack.

The handler can then make calls to user-defined exception handlers, which can in turn use `longjmp()` to change execution flow. In VFP this is necessary to support user-level trap handlers.

These can return a result for a trapped operation, and because a single instruction can cause multiple exceptions, there might be multiple entries to user-level handlers for a single instruction.

The mechanism used for this is likely to be operating system dependent, so this code and the associated ControlBuffer functions might need to be replaced for your system.

The handler provided calls GetControlBuffer() to get the address of a buffer for data that is to be passed to the user-level handler. This is likely to require modification for use with a multiple process operating system.

This has the prototype:

```
ControlBuffer *GetControlBuffer(
    ARM_RegDump *regdump, exception_id, data_size)
```

Where:

regdump is the register dump pointer, as passed to the handler.

exception_id identifies the entry point of the user-level exception handler.

data_size gives the size of the exception description data that the handler wishes to pass to the user-level handler.

ControlBufferData(controlbuffer) returns a pointer to a buffer of data_size bytes that is copied somewhere the user-level handler can read it.

The top-level handler can be modified to copy the ControlBuffer information if appropriate.

User-level exception handler prototype

The Undef handler passes control to code that receives the description data and resaves user state as necessary, and then calls an appropriate handler in an ATPCS-compliant way.

On return from the handler the wrapper code must completely restore the application state, including the flags in the CPSR, and then return to an address passed in by the top-level Undefined Instruction handler.

6.3.4 User-level VFP handler

The VFP user-level exception handler is _VFP_Computation_Engine_Wrapper() in "vfpwrapper.s". This code runs in the application's context.

The user-level VFP handler processes bounced operations in the application's context using the VFP computation engine software. This emulates the instructions described in the bounce description data using software floating-point, and triggers floating-point traps as appropriate.

Saving state

The _VFP_Computation_Engine_Wrapper() first saves some registers and copies the ControlBuffer information on to the application's stack. This enables the use of VFP in application trap handlers, which might cause recursive bounces to the Support Code.

Calling the VFP Computation Engine

The _VFP_Computation_Engine() function is called to emulate the bounced instructions, and signal floating-point traps as appropriate. This is described in detail in Section 6.5.

As floating-point traps are encountered calls can be made to application trap handlers as specified by *ANSI / IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*. The application trap handlers can either return substitute results, or abort processing with longjmp. The Support Code uses the floating-point trap handler mechanism in the ARM C library, as described in *ADS 1.2 Compiler and Libraries Guide*.

Note *A single VFP instruction can generate multiple floating-point traps. This is possible only if it is made up of multiple operations, as are vector instructions and multiply-accumulate instructions.*

Returning to the application

When all bounced operations have been processed the user-level exception handler restores registers and returns directly to the application, using the return address provided by the top-level handler.

6.4 Context switching VFP state

To context switch VFP state you must save and restore the main VFP register bank using FLDMX and FSTMX instructions. The FPSCR must also be preserved.

If the EX bit is set in the FPEXC register, then there is additional state describing a pending exception that must also be saved. The subarchitecture support library provides functions for this. These are described in Section 6.6.2.

Context switching of VFP state can be reduced by only switching when an application uses the VFP. You can achieve this by disabling the VFP in ARM context switch code, (by clearing the EN bit in the FPEXC register), and then only switching VFP registers when the VFP Undef handler is entered, (as the result of an attempt to use the VFP).

6.5 VFP Computation Engine

The computation engine is provided by the vfpsupport libraries.

All objects are Read-Only Position Independent. They make no use of static data, and they are therefore compatible with both Read-Write Position Independent (RWPI) and non-RWPI applications.

The computation engine neither performs a stack check nor corrupts the SL register, so applications using stack checking must be sure to allocate sufficient stack before calling the computation engine. At the time of writing ARM's implementation of the VFP Computation Engine can use up to 132 bytes stack, and can make nested calls to `_fp_trap()` from this maximum stack depth. The library contains sufficient information for the linker to check its stack depth.

6.5.1 `_VFP_Computation_Engine` function

```
void _VFP_Computation_Engine(_VFP_Computation_Description *cdesc)
```

This function accepts a list of VFP computation operations in `cdesc`, and performs the given transformations on the hardware VFP registers. Operands are read from the real hardware VFP registers using FMRS, FMRDH, and FMRDL, and written back to the hardware VFP registers using FMSR, FMDHR, and FMDLR.

The operations that can be specified in `cdesc` are exactly those that are encoded in the VFP instruction set as a CDP instruction. This includes trivial operations such as VFP-to-VFP register moves (FCPY), even though all implementations of VFP coprocessors must implement these in hardware. This is necessary for supporting sequences of bounced operations, where preceding operations caused a bounce after copy operations had been accepted by the VFP coprocessor.

In the course of fulfilling the request, the only hardware VFP instructions that the provided computation engine uses are copy operations between the ARM and VFP register bank:

- FMRS, FMDLR, FMDHR, FMDRR, FMSRR, FMXR (write to VFP regs from ARM regs)
- FMRS, FMRDL, FMRDH, FMRRD, FMRRS, FMRX (read from VFP regs to ARM regs)

In other words, it reads operands out of VFP registers, performs all the data processing in integer registers, and then writes results back to the target VFP registers. It does not appeal to the VFP to do any part of its thinking for it, and it is therefore compatible with any VFP coprocessor.

A subarchitecture optimized computation engine could use the VFP coprocessor for some of its calculations, but it must never cause a bounce.

6.5.2 `_VFP_Computation_Description` structure

A `_VFP_Computation_Description` object is used to pass information from the subarchitecture specific exception decoding code to the VFP computation engine.

```
struct _VFP_Computation_Description {
    uint32 count;
    uint32 flags;
    struct {
        uint32 op;
        uint32 op_dbg;
    } desc[count];
};
```

This structure contains a variable length array of `desc` entries, where each entry represents an operation. The operation is encoded in `op` just like a VFP CDP instruction word, except that bits [26..24] denote the vector length that is used for issuing the instruction, (encoded minus 1 mod 8, as the `LEN` field of the `FPSCR`). Bits [31..27], [11..9], and [4] are ignored. The `op_dbg` field is also ignored.

In other words, the caller must ensure all `op` entries represent a valid kind of VFP operation. The caller must replace bits [26..24] with the vector length minus one.

The vector length is interpreted as if it had been in the `FPSCR` when the equivalent CDP instruction was executed, so the operation uses the vector addressing modes as defined in the ARM ARM. For instructions that are always scalar, the value of the iterations field is ignored.

The vector stride and rounding mode for all operations are taken straight from the hardware `FPSCR`, because it is impossible for these to change in the middle of the instruction sequence. (It is also impossible for the vector length in the `FPSCR` to change; the vector length is specified individually in order to be able to resume a partially executed vector instruction.)

Bits [7..0] of the `count` word give the length of the array. The array has a SUBARCHITECTURE DEFINED maximum length. In current implementations this is only 2 entries. This is expected to be no greater than 16 entries in future VFP implementations. Bits [31..8] of the `count` word are ignored.

The `flags` word is currently ignored.

Subarchitecture note

For forward compatibility all unused fields are cleared to zero by the subarchitecture support code that creates this structure.

The iterations field might contain any value for instruction encodings that are always scalar. This allows code to copy the vector length directly out of the `FPSCR` for any operation that has not yet begun execution in the coprocessor.

6.5.3 Returns from `_VFP_Computation_Engine`

There is no explicit return value.

Results of the VFP operations are returned by being written back to the VFP register bank, as specified in the instruction patterns passed in.

If the computation engine receives an instruction pattern it does not recognize, it signals an error by calling `_VFP_Computation_Error()`.

```
void _VFP_Computation_Error(
    _VFP_Computation_Description *cdesc, uint32 index)
```

The `index` argument identifies the entry of `cdesc` that is invalid. If no implementation of `_VFP_Computation_Error` is provided then errors are ignored.

`_VFP_Computation_Engine()` signals floating-point traps where necessary. Trap handlers can either produce a result, or cause a `longjmp` out of `_VFP_Computation_Engine()`. Traps are signaled by calling `_vfp_fp_trap()`. This function takes the same arguments as the `_fp_trap()` function that forms part of ARM floating-point library support.

The `_vfp_fp_trap()` wrapper provided in `vfpfptrap.s` prevents the corruption of VFP register state by saving callee-save VFP registers, and then calls the standard `_fp_trap()` handler. The implementation of `_vfp_fp_trap()` provided performs no stack checking.

6.6 VFP Subarchitecture Support

The subarchitecture support library provides functions to access `SUBARCHITECTURE_DEFINED` state as required for exception processing and context switching. An implementation is provided in `vfpsubarch.s`. This implementation is for use only with the VFP9 and VFP10 implementations provided by ARM.

These functions are used by the support code supplied in source form with this document. They are not used by the VFP Computation Engine library code.

None of these functions require user modification or configuration. However, some form of dynamic linking of these functions might be required in a system that must work with a variety of VFP implementations.

6.6.1 Support for handling bounces

These functions support VFP Undef exception handling. They must be called from a privileged processor mode.

```
bool _VFP_Is_Compute_Exception(uint32 instr)
```

This function must be called after the VFP has bounced an instruction, with the encoding of the instruction bounced as an argument.

If the bounce was caused by an illegal instruction, this function returns false. If any other exceptional condition exists this takes priority, and this function returns true.

If the `FPEXC EX` bit is set, then there is some sort of VFP exception pending, and this function returns true for all subarchitectures.

```
bool _VFP_Collect_Trap_Description(
    _VFP_Computation_Description *cdesc, uint32 instr)
```

Return value:

- false indicates `instr` must be retried
- true indicates that `instr` has been included in `cdesc`, and therefore must not be retried.

This function must be called either:

- after the VFP has bounced a legal instruction. In this case the encoding of the bounced instruction should be passed as the `instr` argument.

or

- when the `FPEXC EX` bit indicates an exception is pending. In this case the `instr` argument should be zero.

This function writes a `_VFP_Computation_Description` for use by the `_VFP_Computation_Engine()`, as described above.

6.6.2 Support for context switching exception state

The subarchitecture support library provides the following functions to support context switching of the subarchitecture state that describes a pending exception. These functions must be called in a privileged processor mode.

```
uint32 _VFP_ExceptionState_Size(void)
```

This function returns the size of the buffer required to store SUBARCHITECTURE DEFINED exception state. This is expected to be no more than 132 bytes. For current implementations this is 12 bytes.

```
void _VFP_Save_ExceptionState(void *state)
```

This function saves SUBARCHITECTURE DEFINED exception state to the buffer pointed to by `state`.

```
void _VFP_Restore_ExceptionState(void *state)
```

This function restores SUBARCHITECTURE DEFINED exception state from the buffer pointed to by `state`.

7 References

For details of the VFP instruction set, refer to:

- *ARM Architecture Reference Manual*, second edition, ARM DDI 0100E, ISBN 0-201-73719-1, published by Addison-Wesley

For details of the floating-point results that VFP should produce, refer to:

- *ANSI / IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*

For details of implementations of VFP coprocessor hardware, refer to:

- *VFP10 Vector Floating-Point Coprocessor (Rev 0) Technical Reference*
- *VFP10 Vector Floating-Point Coprocessor (Rev 1) Technical Reference Manual*
- *VFP9-S Vector Floating-Point Coprocessor Technical Reference Manual*

For details on floating-point support within the tools, refer to:

- *ADS 1.2 Compiler and Libraries Guide*

For details on AXD and Multi-ICE configuration, refer to

- *ADS 1.2 AXD and armsd Debuggers Guide*
- *Multi-ICE 2.2 User Guide* – for manual configuration files.

For details on Writing Code for ROM, refer to

- *ADS 1.2 Developers Guide*

For details of linker configuration, refer to

- *ADS 1.2 Linker and Utilities Guide*