Application Note 133

Using VFP with RVDS

Document number: ARM DAI 0133C
Issued: July 2008

ARM

Application Note 133 Using VFP with RVDS

Release information

The following changes have been made to this Application Note:

Change history

Date	Issue	Change
September 2004	A	First release for RVDS 2.1
December 2005	B	Revised for RVDS 2.2 SP1 and RVDS 3.0: - Deprecated VFP1v1 and VFP10rev0 - Removed all references to controlbuffer_h.s - Updated examples for ARM1136JF-S - Updated RVD screenshots for ARM1136JF-S - Added references to ARMv7, VFPv3, Cortex-A8
July 2008	C	Revised for RVDS 3.1 and 4.0: - Added references to Cortex-A9, Cortex-R4F, VFPv3-D16 - Added references to --fp16_format switch and __fp16 data type - Removed references to obsolete CM10rev0

Proprietary notice

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.

Table of Contents

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 RealView 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 three such coprocessor architectures:

    - Floating-point Accelerator (FPA), as used for example in the ARM7500FE. This is now 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.

    - Floating Point Unit (FPU), introduced with the Cortex-A9 and Cortex-R4F. This implements support for single and double precision as well as half-precision conversion. Vector operations are not supported in hardware.

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 as system support code which is attached to the ARM processor core's undefined instruction trap.

In some cases, implementation of floating-point requires a combination of VFP/FPA/FPU hardware (to execute the common cases) and software (to deal with the uncommon and exceptional cases). This does not apply for VFPv3 or in "RunFast" mode where software (in the form of support code) is not required to handle uncommon or exceptional cases. See section 4 for more details on "RunFast" mode.

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 several versions of the architecture have been implemented:

- VFPv1 was implemented in the VFP10 revision 0 silicon (as provided by the ARM10200). Support for this was deprecated in RVDS 2.1 and removed from RVDS 2.2 onwards.

- VFPv2 has been implemented in the VFP10 revision 1 (as provided by the ARM10200E), the VFP9-S (as available as a separately licensable option for ARM926/946/966) and the VFP11 (as provided in the ARM1136JF-S and ARM1176JZF-S).

- VFPv3 is backwards compatible with VFPv2 except that VFPv3 cannot trap floating-point exceptions and therefore requires no software support code. VFPv3 is implemented on ARM architecture v7 and later (e.g. Cortex-A8). Some VFPv3 variants are:

    - VFPv3U is an implementation of VFPv3 that can trap floating-point exceptions and requires software support code.

- VFPv3-D32 is an implementation of VFPv3 that provides 32 double-precision registers. VFPv3-D32 is implied for NEON targets, e.g. Cortex-A8 and Cortex-A9.
- VFPv3-D16 is an implementation of VFPv3 that provides 16 double-precision registers rather than 32. VFPv3-D16 is implemented for ARM architecture v7 and later (e.g. Cortex-A9 and Cortex-R4F)
- VFPv3 can be extended by the half-precision extensions that provide conversion functions in both directions between half-precision floating-point (fp16 – see later) and single-precision floating-point.

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 upon 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 how to set up a system and software components to operate the VFP in RunFast mode.
5. Application test code which can be used to check that the VFP Support Code has been successfully installed into a system.
6. A description of an implementation of the software components required to provide a normal working VFP system. These software components are commonly referred to as the VFP Support Code. The modifiable portions of the support code are provided with this application note. Two unmodifiable libraries, vfpsupport.l and vfpsupport.b are supplied with RVDS.

This application note and the accompanying example code are written for use with RealView Developer Suite (RVDS) 2.1 and later. Differences which occur between RVDS 2.1, 2.2, 3.0, 3.1, and 4.0 are noted.

For information on using VFP with ADS 1.2 refer to Application Note 98, VFP Support Code, available for download from the ARM website.

1.4 Patches

It is advisable to update your tools to the latest versions as these updates will contain fixes to known problems. These updates are available from the downloads section of the Technical Support area on the ARM website.

2 Producing Code to Run on a VFP System

2.1 Using the compiler

2.1.1 --fpu *name*

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 instead you must use appropriate compiler options to modify the code generated:

--fpu vfp	This is a synonym for --fpu vfpv2.
--fpu vfpv1 (RVDS 2.1 only)	Selects hardware vector floating-point unit conforming to architecture VFPv1, such as the VFP10 rev 0. This option is deprecated in RVDS 2.1 and support was removed in RVDS 2.2.
--fpu vfpv2	Selects hardware vector floating-point unit conforming to architecture VFPv2, such as the VFP10 rev 1 or VFP11. Note that if you select this option and also compile with --thumb, then the compiler will actually generate ARM code for floating-point using functions.
--fpu vfpv3 (RVDS 3.0 and later)	Selects hardware vector floating-point unit conforming to architecture VFPv3 VFPv3 is backwards compatible with VFPv2 except that VFPv3 cannot trap floating-point exceptions.
--fpu vfpv3_d16 (RVDS 4.0 and later)	Selects hardware vector floating-point unit conforming to architecture VFPv3-D16.
--fpu vfpv3_fp16 (RVDS 4.0 and later)	Selects hardware vector floating-point unit that implements the VFPv3 half-precision architecture.
--fpu vfpv3_d16_fp16 (RVDS 4.0 and later)	Selects hardware vector floating-point unit conforming to architectures VFPv3 half-precision and VFPv3-D16.
--fpu softvfp+vfp	Synonym for --fpu softvfp+vfpv2
--fpu softvfp+vfpv2	Selects a floating-point library with software floating-point linkage that can use VFPv2 instructions. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFP unit. If you select this option: • Compiling with --thumb behaves in a similar way to --fpu softvfp except that it links with floating-point libraries that contain VFP instructions. • Compiling with --arm option behaves in a similar way to --fpu vfpv2 except that functions pass and return floating-point arguments and results in the same way as they would for --fpu softvfp, but use VFP instructions internally.
--fpu softvfp+vfpv3 (RVDS 3.0 and later)	Selects a floating-point library with software floating-point linkage that uses VFPv3 instructions.
--fpu softvfp+vfpv3_fp16 (RVDS 4.0 and later)	Selects a floating-point library with software floating-point linkage that uses VFPv3 instructions with half-precision floating-point extension support.

--fpu softvfp+vfpv3_d16 (RVDS 4.0 and later)	Selects a floating-point library with software floating-point linkage that uses VFPv3-D16 instructions.
--fpu softvfp+vfpv3_d16_fp16 (RVDS 4.0 and later)	Selects a floating-point library with software floating-point linkage that uses VFPv3-D16 instructions with half-precision floating extension support.

For more details of which compiler options to use in particular circumstances, please see section 2.3.

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 ARM1136JF-S, this implies --fpu vfpv2. For RVCT 3.0 and later, specifying a --fpu setting overrides the floating-point unit implied by --cpu.*

2.1.2 --fpmode *model*

Specifies the floating-point conformance, and sets library attributes and floating-point optimizations. Different libraries may be selected depending on the model chosen.

--fpmode std	IEEE finite values with denormals flushed to zero, round-to-nearest and no exceptions. It is C and C++ compatible. This is the default option.
--fpmode ieee_full	All facilities, operations and representations guaranteed by the IEEE standard are available in single and double precision. Modes of operation can also be selected dynamically at run-time.
--fpmode ieee_fixed	IEEE standard with round-to-nearest and no inexact exception.
--fpmode ieee_no_fenv	IEEE standard with round-to-nearest and no exceptions. This mode is compatible with the Java floating-point arithmetic model.
--fpmode fast	Perform more aggressive floating-point optimizations that might cause a small loss of accuracy to provide a significant performance increase. This option results in behavior that is not fully ISO C and C++ standard-compliant, however numerically robust floating-point programs will behave correctly.

Note *"--fpmode std" and "--fpmode fast" set the Flush-to-Zero and Default-NaN bits and clears the exceptions in the FPSCR (see section 3.6 for more details). These settings conform to RunFast mode. However, initialization code is still required to enable the VFP. See section 4 for more details.*

2.1.3 --fp16_format

Introduced in RVDS 4.0, this option enables the use of half-precision (16-bit) floating-point numbers and sets the format of variables marked with the new __fp16 data type.

Half-precision floating-point numbers are provided as an optional extension to the VFPv3 architecture. If the VFPv3 coprocessor is not available, or if a VFPv3 coprocessor is used that does not have this extension, they are supported through the floating-point library fplib.

--fp16_format=none	This is the default setting. It is equivalent to not specifying a format and means that the compiler will fault use of the __fp16 data type.
--fp16_format=ieee	Half-precision binary floating-point format defined by IEEE 745r, a revision to the IEEE 754 standard.
--fp16_format=alternative	An alternative to --fp16_format=ieee that provides additional range, but has no NaN or infinity values.

Note Also introduced in RVDS 4.0 are a set of fp16-based intrinsics that implement common NEON operations with half-precision floating-point values. A full listing of the available fp16-based intrinsics can be found in the RVCT Compiler Reference Guide.

2.2 Using the assembler

By default, the assembler faults the use of VFP instructions. To enable the assembly of VFP instructions, appropriate options need to be used. These options enable the use of VFP instructions in ARM or Thumb-2 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.

2.2.1 --fpu *name*

This option determines the target FPU architecture. *name* options are the same as for the compiler.

2.2.2 --fpmode *model*

This option selects the target floating-point model. Different libraries may be selected depending on the model chosen. *model* options are the same as for the compiler.

2.3 Floating point linkage and the --fpu option

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

2.3.1 Floating point linkage

How floating point arguments are passed into (and returned from) functions is called the "floating point linkage". There are two different types of floating point linkage available in RVDS : “hardware” and “software” floating point linkage.

With hardware floating point linkage, floating point arguments and return values are passed in VFP coprocessor registers. For this to be possible the target must have a VFP coprocessor present and be executing an instruction set that supports coprocessor instructions (ARM or Thumb-2).

With software floating point linkage, floating point arguments and return values are passed in ARM integer registers. This means it can be used on any ARM, with or without a VFP coprocessor. It can also be used from any instruction set (ARM, Thumb or Thumb-2) as it does not require access to a VFP coprocessor to pass arguments.

It is not possible to mix functions built with different floating point linkage because the arguments will be held in different physical registers.

You must choose the most appropriate linkage based on your needs. Hardware linkage offers improved performance as arguments and return values do no need to be

transferred between the VFP coprocessor register and the ARM registers on function calls, but can only be exploited when explicitly targeting a device known to contain VFP hardware. Software linkage (all code built with --fpu softvfp, --fpu softvfp+vfpv2 or --fpu softvfp+vfpv3) is more commonly used for code not specifically targeted at a particular processor.

Note *Note: If software floating point linkage is being used, both the calling function and the called function must either be compiled using --fpu softvfp+vfpv2 or --fpu softvfp+vfpv3 or declared using the __softfp keyword. The __softfp keyword allows software floating-point linkage to be specified on a function by function basis rather than across the whole file as is the case with --fpu softvfp+vfpv2 or --fpu softvfp+vfpv3. See the description of __softfp in the RVCT Compilers and Libraries Guide or the RVCT Compiler Reference Guide for more information.*

2.3.2 ARM state only floating-point (ARMv6 and earlier)

If all of your floating-point processing is done in ARM state code and none is done in Thumb state code then:

- Build your ARM code with --fpu vfpv2 (if you are using VFP9-S, VFP10 rev 1 or VFP11)
- Build your Thumb code with --fpu none. This ensures that no floating-point operations take place in Thumb code.

2.3.3 ARM and Thumb floating-point (ARMv6 and earlier)

The Thumb instruction set in ARMv6 and earlier does not support coprocessor instructions and this means you cannot access the VFP coprocessor register in code compiled for Thumb. Therefore you must either compile for the ARM instruction set or use software floating point linkage.

- Compiling with --fpu vfpv2 will use hardware VFP linkage and therefore any functions that use floating point types will be compiled as ARM code to allow access to the VFP registers.
- Compiling with --fpu softvfp+vfpv2 will use software floating point linkage. When compiling for the ARM instruction set the compiler will generate VFP instructions inline and pass parameters in ARM registers. When compiling for Thumb (--thumb) the compiler will generate Thumb code with calls to floating point support libraries to perform the floating point operations. At link time, the linker will normally then link-in a version of the floating point support library that contains hardware VFP instructions to perform the floating point operation.

The choice of options that provides the best code size/performance depends upon the code being compiled. For example, there is a trade-off between compiling --fpu softvfp+vfpv2 (as opposed to --fpu vfpv2) to obtain Thumb code and the corresponding inclusion of library code. It is best to experiment with different options to find the combination which provides the required code size/performance attributes.

2.3.4 ARM and/or Thumb-2 floating-point (ARMv7, RVDS 3.0 and later)

VFP is directly accessible from both the ARM and Thumb-2 instruction set on ARMv7 processors with VFPv3 such as Cortex-A8. This allows you to use Thumb-2 for your entire application, without needing to switch to ARM state to perform VFP operations. To do this, build your code with --thumb and one of the following options:

- --fpu vfpv3
- --fpu vfpv3_fp16
- --fpu vfpv3_d16

- --fpu vfpv3_d16_fp16

Alternatively, you can build your code for ARM only with --arm and one of the above options.

To build code that is compatible with other code that specifies software floating-point linkage, use one of the following options:

- --fpu softvfp+vfpv3
- --fpu softvfp+vfpv3_fp16
- --fpu softvfp+vfpv3_d16
- --fpu softvfp+vfpv3_d16_fp16

3 VFP System Initialization

To use VFP in your application, there are a number of steps that must be 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. Enable the VFP coprocessor by setting the VFPEnable bit in the VFP's FPEXC register. For architecture V6 processors, coprocessor bits CP10 and CP11 in the Coprocessor Access Control Register also need setting. Note that at reset the VFP coprocessor will be disabled.

These steps are discussed in more detail in the following subsections.

Note If you wish to operate the VFP coprocessor in RunFast mode the initialization required is different, see section 4 for more details.

3.1 Provide the VFP Support Code

Provided with this application note is an implementation of the modifiable portions of the RVDS 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 conjunction with the libraries vfpsupport.l and vfpsupport.b supplied in the library directory of RVDS. In some applications VFP Support Code is linked-in as part of the image. In other applications it may be provided linked-in as part of the system environment or operating system.

Note VFP Support Code is not needed in VFPv3-based systems, except for VFPv3U or when performing vector operations on a VFPv3-D16 - contact your supplier for more information.

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 assumes that

- the VFP Support Code is located in memory within the first 32MB of memory so that a branch instruction can be used
- high vectors are disabled
- address 0 is writable in the current mode (determined by MMU access permissions)

If caches are enabled the predefine WANT_CACHE_FLUSH should be used to flush the caches after the patching of the vector table.

```
UNDEF_VECTOR    EQU     0x4 ; address of undefined instruction vector
                            ; (hivecs not handled)
Install_VFPHandler FUNCTION
        ; Install VFP handler onto undefined instruction

; TLUndef_Handler must be reachable via BL from UNDEF_VECTOR.
        ADR     r0, TLUndef_Handler_Offset
        LDR     r1, [r0]
        ADD     r0, r0, r1
        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
  IF ARCH_V6_OR_LATER :LAND: {ENDIAN} = "big" :LAND: :LNOT: :DEF: ENDIAN_BE_32
        REV     r0, r0
  ENDIF
        STR     r0, [r1]

; If we have separate data and instruction caches then we need to clean the
; data cache and invalidate the instruction cache.
; If we have a branch target cache we need to invalidate that as well.

  IF :DEF: WANT_CACHE_FLUSH

; This code is known to work with ARM926, ARM946, ARM1020, ARM1022
; ARM1026 and ARM1136 cores.
;
; For other cores the following may need to be modified.
; Please check the TRM for your core.

CACHE_ADDR_SBZ EQU 0x7  ; these bits "should be zero" when cleaning/invalidating
                        ; by virtual address, v6 doesn't have SBZ bits

UNDEF_VECTOR_CACHE_LINE EQU (UNDEF_VECTOR :AND: :NOT:CACHE_ADDR_SBZ)
        MOV r1, #UNDEF_VECTOR_CACHE_LINE

        MCR p15, 0, r1, c7, c10, 1 ; clean D cache at 'r1'
        MCR p15, 0, r1, c7, c5,  1 ; invalidate I cache at 'r1'
        MOV r0,#0
        MCR p15, 0, r0, c7, c10, 4 ; drain write buffer

    IF ARCH_V6_OR_LATER
        MOV r1, #UNDEF_VECTOR
        MCR p15, 0, r1, c7, c5,  7 ; invalidate branch target cache at 'r1'
    ENDIF

  ENDIF

        BX      LR                          ; return from subroutine
TLUndef_Handler_Offset
        DCD     TLUndef_Handler - TLUndef_Handler_Offset
        ENDFUNC
```

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
   IF :DEF: SETUP_UNDEF_STACK

        EXPORT  Setup_Undef_Stack

Setup_Undef_Stack FUNCTION
        ; Now set up a stack for undefined mode
        MRS     r0, CPSR                  ; get CPSR value
        MOV     r1, r0                    ; take a working copy
        ORR     r1,r1, #Mode_UNDEF        ; set mode bits for Undefined mode
        MSR     CPSR_c, r1                ; change to undefined mode

      IF :DEF: RWPI
        LDR     r2, UNDEF_Stack_Offset
        ADD     SP, r2, r9
      ELSE
        LDR     SP, =UNDEF_Stack          ; set up the stack pointer
      ENDIF

        MSR     CPSR_c, r0                ; change back to the original mode
        BX      LR                        ; return from subroutine
        ENDFUNC

      IF :DEF: RWPI
UNDEF_Stack_Offset DCDO UNDEF_Stack
      ENDIF

       ENDIF

    IF :DEF: SETUP_UNDEF_STACK
; Location for undefined-mode stack
        AREA   UNDEF_STACK, NOINIT, ALIGN=3
        %       8               ; Only two words will be used
UNDEF_Stack   EQU .
        ENDIF

        END
```

Note *The support code 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 7 for more details.*

3.4 Enable VFP

On all systems, it will be necessary to enable the VFP by setting the VFPEnable (EN) 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. On pre-v7 cores, you will also need 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

Enable_VFP FUNCTION
        ; Enable VFP itself
        MOV     r0,#VFPEnable
        FMXR    FPEXC, r0       ; FPEXC = r0
        BX      LR
        ENDFUNC
```

Note The new UAL equivalent of FMXR is VMSR.

3.4.1 Architecture V6 and later

Due to changes made in V6 and later architectures in the way in which the VFP coprocessor interfaces with the core processor, the VFP support code has some conditional actions which apply only to architecture V6 or later (such as VFP11). Coprocessors CP10 and CP11 must be enabled by setting bits 20-23 (b1111 gives full read/write access) of the Coprocessor Access Control Register. For more information please refer to the technical reference manual of the ARM processor core that you are using.

```
VFPEnable       EQU     0x40000000

       GBLL ARCH_V6_OR_LATER   ;Create global variable
  IF "6" <= {ARCHITECTURE} ; ok until architecture 10
ARCH_V6_OR_LATER SETL {TRUE}
  ELSE
ARCH_V6_OR_LATER SETL {FALSE}
  ENDIF

Enable_VFP FUNCTION

  IF ARCH_V6_OR_LATER
        MRC p15, 0, r1, c1, c0, 2 ; r1 = Access Control Register
        ORR r1, r1, #(0xf << 20)  ; enable full access for p10,11
        MCR p15, 0, r1, c1, c0, 2 ; Access Control Register = r1
        MOV r1, #0
        MCR p15, 0, r1, c7, c5, 4 ; flush prefetch buffer because of FMXR below
                                  ; and CP 10 & 11 were only just enabled
  ENDIF

        ; Enable VFP itself
        MOV     r0,#VFPEnable
        FMXR    FPEXC, r0       ; FPEXC = r0
        BX      LR
        ENDFUNC
```

Note The new UAL equivalent of FMXR is VMSR.

3.5 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 described in sections 3.1-3.5 *before* _fp_init() is executed. This is done using the linker's *$Sub$$* and *$Super$$* functionality, which is detailed in the *RVCT Linker Guide*. The following C code example shows how this can be done:

```
#ifdef __cplusplus
 #define EXTERN_C extern "C"
#else
 #define EXTERN_C extern
#endif

#ifdef SETUP_UNDEF_STACK
        EXTERN_C void Setup_Undef_Stack (void);
#endif
#ifdef PATCH_UNDEF_VECTOR
        EXTERN_C void Install_VFPHandler (void);
#endif

EXTERN_C void Enable_VFP (void);
EXTERN_C void $Super$$_fp_init(void);

// Call $Sub$$_fp_init() in place of original _fp_init()
EXTERN_C void $Sub$$_fp_init(void)
{

#ifdef PATCH_UNDEF_VECTOR
        Install_VFPHandler();
#endif
#ifdef SETUP_UNDEF_STACK
        Setup_Undef_Stack();
#endif
        Enable_VFP();
        $Super$$_fp_init();   // Call original _fp_init()
}
```

3.6 FPSCR settings

The Floating-Point Status and Control Register (FPSCR) holds control bits that affect the way the VFP operates. The FPSCR is usually set by the C library initialization code depending on the compilation options selected. However you may wish to modify the FPSCR contents within your code (for example to change the vector length if you are using vector floating-point calculations).

The FPSCR contains:

- N, Z, C and V condition flags resulting from the most recent FP comparison
- Alternative Half-Precision (VFPv3-D16 only)
- Default NaN mode control
- Flush-to-zero mode control
- Rounding mode control
- Vector length/stride control
- Exception status and control.

For more information on these options please refer to the Technical Reference Manual of the VFP unit that you are using. The following assembler code example shows how to set the rounding mode, it is not included in the VFP example code provided by this application note but illustrates how VFP options can be set.

```
RZ_Enable       EQU     2_11:SHL:22
        ; Bit pattern to enable Round towards zero mode

        EXPORT  Round_Towards_Zero

Round_Towards_Zero FUNCTION

        FMRX   r0, FPSCR   ; Read current status
        ORR    r0, r0, #RZ_Enable
        FMXR    FPSCR, r0       ; FPSCR = r0
        BX      LR
        ENDFUNC
```

Note The new UAL equivalents of FMRX and FMXR are VMRS and VMSR respectively.

4 RunFast Mode Initialisation

This mode is applicable to VFP9-S, VFP10 rev1,VFP11 and VFPv3-based coprocessors.

Hardware floating-point calculations are considerably faster than software calculations. However, VFP coprocessors still require support code for handling of exceptional cases (such as subnormal numbers). In many applications the additional accuracy and IEEE 754 standard compliance provided by the support code are unimportant. In these applications execution speed can be increased and program size reduced by configuring the VFP in RunFast mode.

RunFast mode is not configured by setting a single register bit. It is the combination of the following conditions:

- The VFP coprocessor is in flush-to-zero mode.
- The VFP coprocessor is in default NaN mode.
- All exception bits are cleared.

In RunFast mode the VFP coprocessor:

- Processes subnormal operands as positive zeros
- Processes input NaNs as default NaNs
- Processes results that are tiny before rounding, that is, between the positive and negative minimum normal values for the destination precision, as positive zeros.
- Returns the IEEE 754 standard for operations that overflow, operations which are considered as invalid, and for divide-by-zero cases, fully in hardware and without additional latency.
- Process all operations in hardware without trapping to support code.

In order to activate RunFast mode, all that is required is to set bits 24 and 25 of the FPSCR, clear the exception bits and activate the VFP. If the code is compiled using "--fpmode std" or "--fpmode fast" bits 24 and 25 of the FPSCR are automatically set and the exceptions cleared by the C runtime library. The RunFast_Enable function is included for completeness should you wish to enable RunFast mode when using a different --fpmode option.

```
        AREA RunFast, CODE, READONLY

VFPEnable       EQU     0x40000000
RF_Enable       EQU     2_11:SHL:24
        ; Bit pattern to enable RunFast mode
        ; FPSCR [24] - Flush to Zero mode
        ; FPSCR [25] - Default NaN mode

        EXPORT  RunFast_Enable

RunFast_Enable FUNCTION

        MOV     r0,#RF_Enable
        FMXR    FPSCR, r0       ; FPSCR = r0
        BX      LR
        ENDFUNC
```

```
        EXPORT  Enable_VFP

Enable_VFP FUNCTION

        ; Enable VFP itself
        MOV     r0,#VFPEnable
        FMXR    FPEXC, r0       ; FPEXC = r0
        BX      LR
        ENDFUNC

        END
```

Note The new UAL equivalent of FMXR is VMSR.

As with normal VFP support when using V6 or later coprocessors, RunFast mode requires extra code for Architecture V6 or later (such as coprocessor VFP11) because of the changes to the coprocessor interface. Coprocessors cp10 and cp11 must be enabled by setting bits 20-23 (b1111 gives full read/write access) of the Coprocessor Access Control Register. For more information please refer to the technical reference manual of the ARM11 or Cortex processor that you are using. The example code below shows how this may be done.

```
Enable_VFP FUNCTION

  if "6" <={ARCHITECTURE}
        MRC p15, 0, r1, c1, c0, 2 ; r1 = Access Control Register
        ORR r1, r1, #(0xf << 20)  ; enable full access for p10,11
        MCR p15, 0, r1, c1, c0, 2 ; Access Control Register = r1
        MOV r1, #0
        MCR p15, 0, r1, c7, c5, 4 ; flush prefetch buffer because of FMXR below
                                  ; and CP 10 & 11 were only just enabled
  endif

        ; Enable VFP itself
        MOV     r0,#VFPEnable
        FMXR    FPEXC, r0       ; FPEXC = r0
        BX      LR
        ENDFUNC
```

Note The new UAL equivalent of FMXR is VMSR.

5 Debugger Setup for VFP

5.1 Configuring RVISS (ARMulator) for VFP in RVD

The RVISS (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.

In RVD an ARMulator connection can be established in two different ways, either by using the RealView connection broker (recommended) or the Remote Debug Interface (RDI). If you have a multi-core license, the connection broker allows more than one ARMulator to operate in a single instantiation of RVD (See the RVISS User Guide provided with RVDS).

5.1.1 Connecting to RVISS via RealView Connection Broker

To do this navigate to:

RVDS 2.1 **File → Connection → Connect to target**

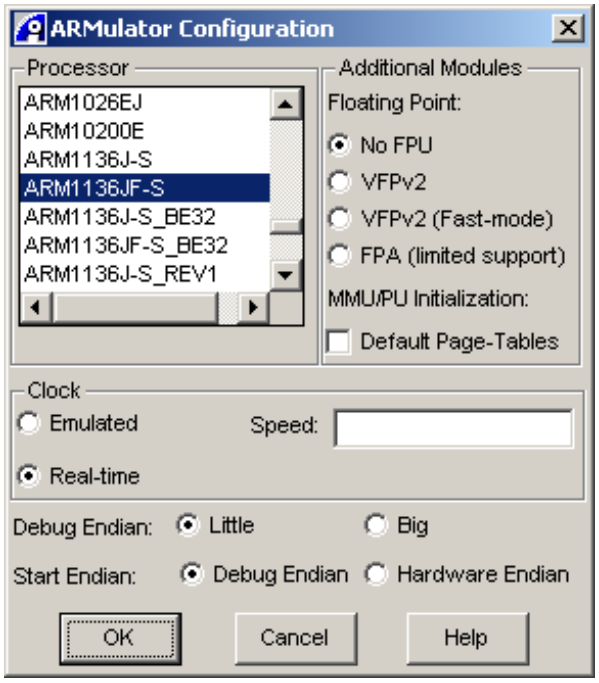
RVDS 2.2: **Target → Connect to Target**

This opens the Connection Control window, you must start a new ARMulator Simulator by clicking on the crosses navigate to: **+ Server + localhost.**

To configure the new device right click on the new_ARM option and select “Configure Device Info”. Select the required Processor and VFP type and click “OK”. Then, in the Connection Control Window, place a tick by “new arm” to start the simulator.

RVDS 3.0: **Target → Connect to Target**

Click on localhost in the Connection Control Window, then click "Open Target Access". Right click on new_arm which should now have appeared in the expanded tree and select Configure. In the Configuration Window select the required Processor and VFP type and click OK. Re-open the Connection Control Window, select “new_arm” and click Connect to start the simulator.



Select an appropriate Processor and FPU setting

To emulate a vfpv2 FPU on an ARM9, first select the processor needed (e.g. ARM926EJ-S) and then select VFPv2. VFPv2 (Fast-mode) is a VFP model that does not bounce to support code.

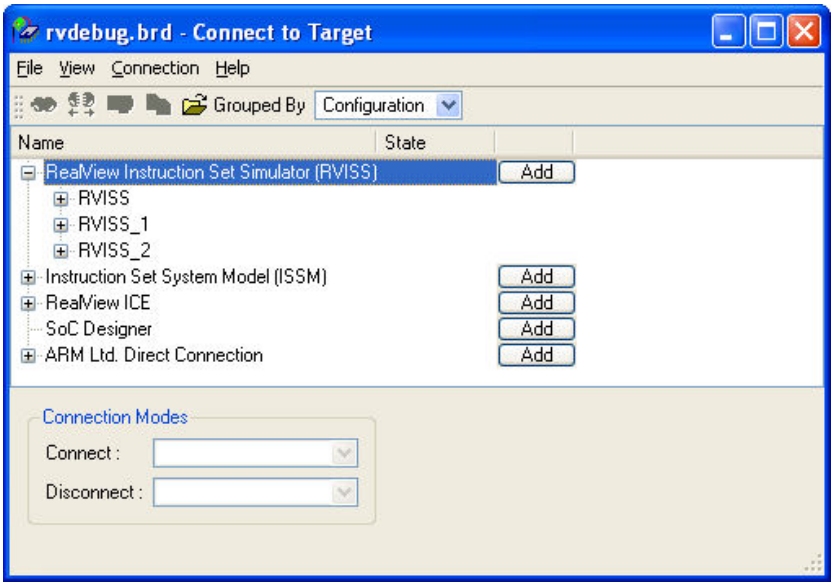
If a vfpv2 FPU is to be included for an ARM10, for example the ARM1020E, the variant including the VFP10, e.g. ARM10200E should be chosen and the "No FPU" setting should be selected.

ARM11 processors have variants including an "F" in the name symbolising that it includes an FPU. Once again when choosing one of these processors, for example ARM1136JF-S, the "No FPU" setting should be selected.

If you select a processor containing an FPU and then an FPU model is chosen explicitly, the behaviour of the ARMulator can be unpredictable.

RVDS 3.1 and onward: **Target → Connect to Target**

Expand RealView Instruction Set Simulator (RVISS) in the Connection Control Window, then right click on one of the RVISS connections and select Configure. In the Configuration Window select the required Processor and VFP type and click OK. Go back to the Connection Control Window, select the RVISS connection you configured and click Connect to start the simulator.



5.1.2 Connecting to RVISS via Remote Debug Interface

To do this within RVD navigate to:

RVDS 2.1: **File → Connection → Connect to target**

RVDS 2.2: **Target → Connect to Target**

This will open the Connection Control window. Click on the cross next to the ARM-A-RR item to expand the RDI targets tree.

If an ARMulator item does not appear, right click on the ARM-A-RR item and select “Add/Edit/Remove devices”. If an ARMulator option appears in the window that opens ensure that the corresponding check box is ticked. If ARMulator does not appear click “Add DLL…” and open:

“*Install_directory*\RVARMulator\ARMulator*vers**build**platform*\ARMulate.dll”

where *Install_directory, vers, build,*and *platform* are specific to your installation (e.g. "C:\Program Files\ARM", vers 1.41, build 253 and win_32-pentium).

To configure the ARMulator right click on the ARMulator item and select “Configure Device Info…”. The configuration dialogue is the same as that used in AXD, see section 5.3 below.

RVDS 3.0 onwards:

RDI has been deprecated in RVDS 3.0, use connection broker instead.

5.1.3 Vector Catch

The VFP Support Code is installed on to the undefined instruction vector table entry, so unless you are using RunFast mode, it is also necessary to ensure that the debugger is not trapping undefined instructions using its “vector_catch” function.

To do this navigate to:

RVDS 2.1: **Debug → Simple Breakpoints → Processor events**

RVDS 2.2 onwards: **Debug → Processor Exceptions**

and clear the “Undefined” tick box.

To alter vector_catch on ARMulator connected via RDI or Multi-ICE navigate to:

RVDS 2.1: **View → Pane Views → Registers.**

RVDS 2.2: **View → Registers.**

RVDS 3.0 onwards: RDI has been deprecated in RVDS 3.0, use connection broker instead.

In the register pane select the debug tab. Vector_catch default value is 0x13B to disable “Undefined” set the value to 0x139 (i.e. clear the 0x002 bit).

5.2 Configuring ISSM for VFP in RVD

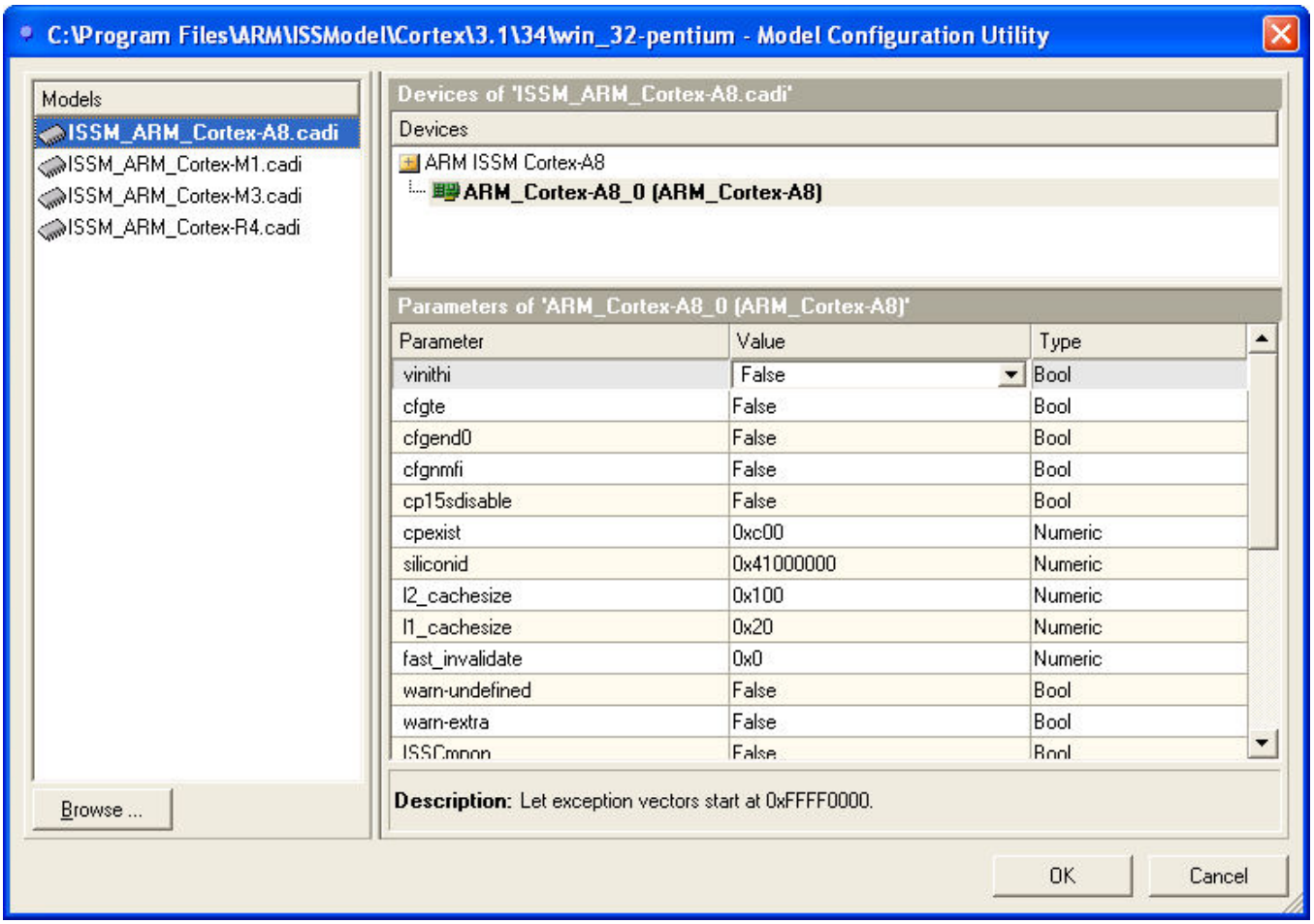
Introduced in RVD 3.0, ISSM provides models for the Cortex family of processors. When connected to a ISSM that supports VFP, a VFP tab will be displayed in the Register window.

5.2.1 Connecting to ISSM

To do this within RVD navigate to:

Target → Connect to target

to open the Connection Control window and start a new ISSM configuration by clicking on the "Add" button. This will open the Model Configuration Utility window.



Select which core you want to connect to from the "Models" menu on the left side of the configuration utility and click "OK". In the Connection Control window, double click on the ISSM configuration you created to connect to the ISSM.

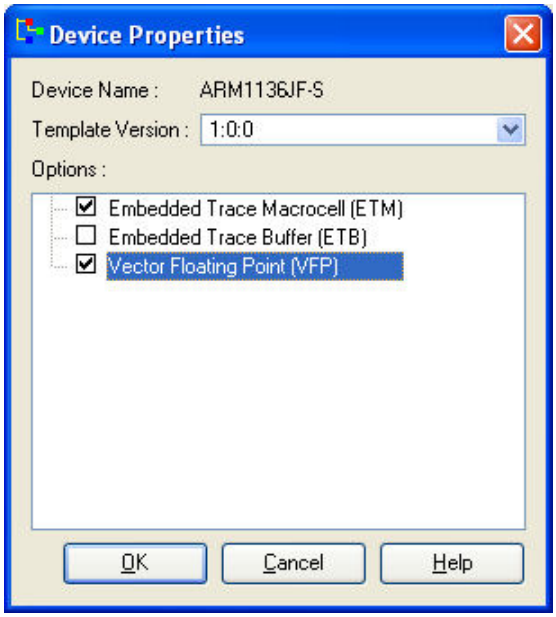
Note Some ISSMs such as the Cortex-A8 allow you to disable the VFP unit from the Model Configuration Utility window. Make sure that the VFP unit is enabled.

The ISSMs do not support vector catch, so you will not need to disable the vector catch logic before running your application.

5.3 Enabling VFP registers in RVD

Normally RVD will automatically display the registers present in the target to which you have connected. However, if connecting to a target processor using RealView ICE or Multi-ICE, For certain targets RVD may occasionally fail to display the VFP registers.

If connecting with RealView ICE, RVD can be manually told that the target processor has a VFP unit. To do this right click on the RealView ICE entry in connection control, then click Configure Device Info. This opens the RVConfig window. Select the core you wish to connect to, then click on Device Properties. In the Device Properties window, select the VFP item as shown below.



5.3.1 Enabling VFP registers in RVD when connected to a Versatile PB926EJ-S using Multi-ICE

RVD does not display VFP registers when Multi-ICE is connected to a Versatile PB926EJ-S. When Multi-ICE is auto-configured an ARM926EJ-S is detected however this definition does not contain a VFP as standard (the Integrator/CM926EJ-S does not have a VFP).

A workaround for this problem is to use the vfp.xml provided in the \util directory of the VFP examples. When copied into the same directory as the version of armperip.xml that is used by RVD, vfp.xml overrides the existing definition of the ARM926EJ-S to include a VFPv2.

Copy vfp.xml to the \RVD\Core\version#\release#\platform\bin directory of your ARM installation. If the above is not successful you may need to copy vfp.xml to an alternative directory containing armperip.xml

5.4 Configuring ARMulator for VFP in AXD

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.

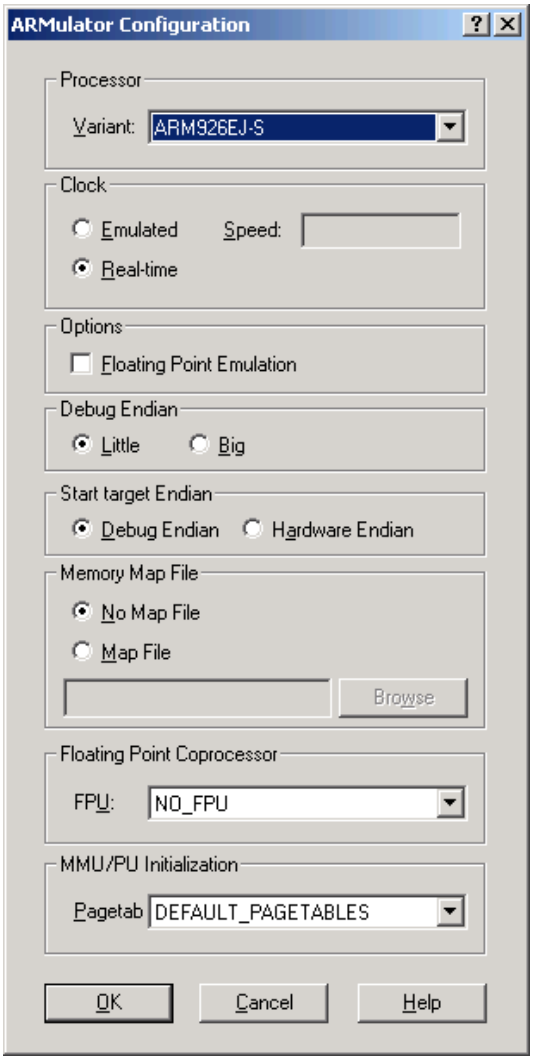
AXD supports the debugging of ARM7, 9 and 10 processors only, not ARM11 or later (use RVD instead).

AXD was deprecated in RVDS 3.0,and removed from RVDS 3.1..

AXD supports debugging with DWARF2 debug tables only, not DWARF3. The RVCT 3.0 compilation tools generate debug images with DWARF3 debug tables by default. To use AXD, compile with --dwarf2.

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:



Select an appropriate Processor and FPU setting

To emulate a vfpv2 FPU on an ARM9, first select the processor needed (e.g. ARM926EJ-S) and then select VFPv2. VFPv2 (Fast-mode) is a VFP model that does not bounce to support code.

If a vfpv2 FPU is to be included for an ARM10, for example the ARM1020E, the variant including the VFP10, e.g. ARM10200E should be chosen and the "No FPU" setting should be selected.

If you select a processor containing an FPU and then an FPU model is chosen explicitly, the behaviour of the ARMulator can be unpredictable.

An error will be given if an incompatible combination of processor and fpu are chosen.

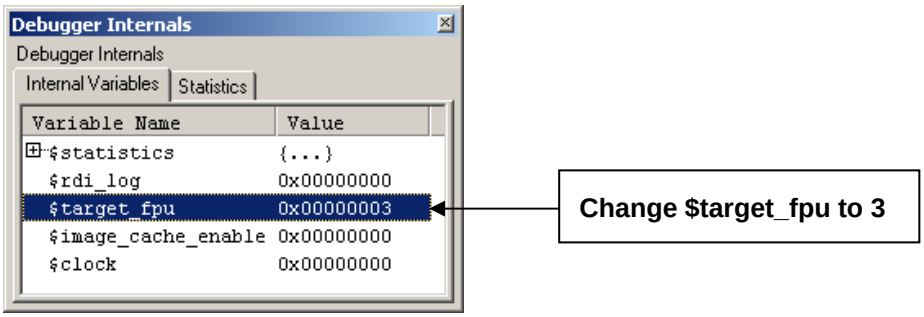
5.5 Configuration of AXD for VFP

This section covers debugger internal variables that must be set correctly when debugging a system containing a VFP. More details are also in the *RVDS 2.2 (or later) AXD and armsd Debuggers Guide*.

5.5.1 $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 before loading your image 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.

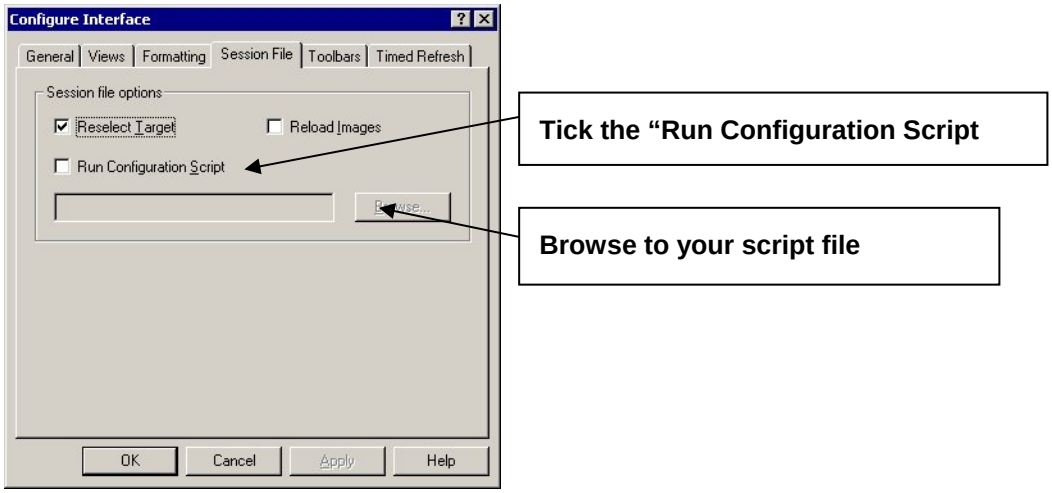
To change $target_fpu from the AXD GUI select the menu “System Views -> Debugger Internals”:



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.*

5.5.2 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

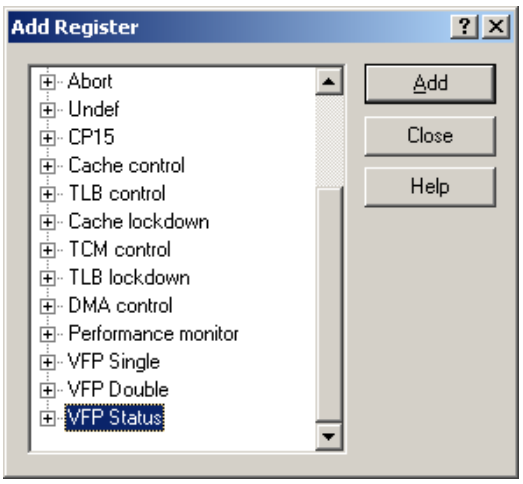
its “vector_catch” function. To ensure that this is disabled, either use the “Options-> Configure Processor” menu, or use the following AXD CLI command:

```
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.

5.6 Enabling VFP registers in AXD

In AXD the VFP registers can be viewed by right clicking in the register pane and selecting “Add register…”. The dialogue shown below opens, allowing the desired registers to be selected and added to the register pane.

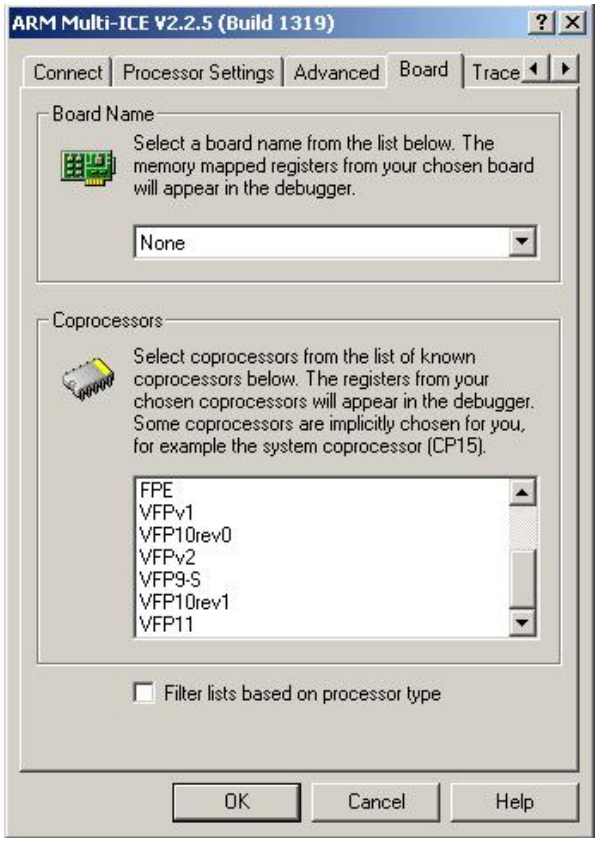


When using Multi-ICE the VFP registers occasionally may not be shown in the “Add Register” pane. To rectify this navigate to:

Options → Configure Target

Select the Multi-ICE entry and press “Configure”.

The Multi-ICE configuration dialogue which appears (shown below) has a “Board” tab where the relevant coprocessor can be selected.



6 Example application using VFP

A simple application is provided as part of the associated code suite, as main.c. This carries out three floating-point operations, the first two of which are executed by the VFP hardware, the third uses a denormalized number and therefore bounces to the VFP Support Code for handling (except if RunFast mode is enabled). 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 = 2.22524e-308; //Small, just normalized number
    double val4 = 3.1234e-322; // denormalized number
    double res1;
    double res2;
    double res3;
    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");
    res1=VFPtest(val1,val2);
    printf("The result is        : %f\n\n", res1);
    printf("Non-bouncing calculation of %e + %e\n",val3,val3);
    printf("The result should be : 4.45048e-308\n");
    res2=VFPtest(val3,val3);
    printf("The result is        : %e\n\n", res2);
    printf("Bouncing calculation of %e + %e\n", val4, val4);
    printf("The result should be : 6.225227e-322\n");
    res3=VFPtest(val4,val4);
    printf("The result is        : %e\n\n", res3);
    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 val4, 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.*

7 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.

7.1 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).

7.1.1 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
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 _fp_trap 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

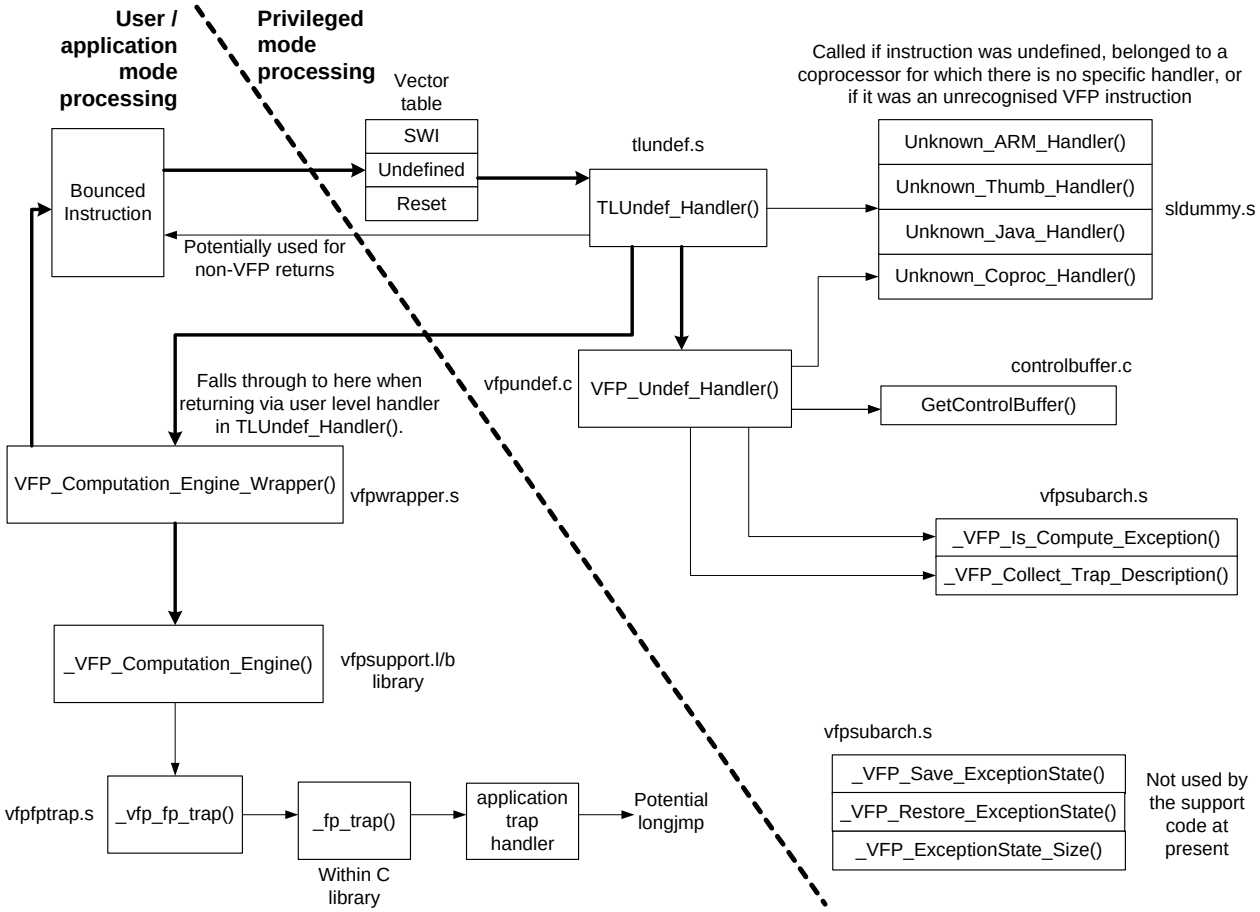
The following files are provided in source format in the \vfp_support subdirectory of the associated example code. These files contain functions to access the subarchitecture defined functionality of the VFP9/VFP10/VFP11 family of VFP coprocessors.

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

The following files are provided in library form, they are installed as part of RVDS in the \lib\armlib subdirectory of the RVCT directory.

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:



7.2 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.

7.2.1 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.

7.2.2 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, VFP10 and VFP11 are provided in vfpsubarch.s. See Section 7.6 "VFP Subarchitecture Support".*

7.2.3 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.

7.2.4 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 *RVCT 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.

7.3 Processing a bounced VFP instruction

7.3.1 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 7.3.3.

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_undef and SPSR_undef 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_undef and SPSR_undef, 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 7.3.2.

Second-level handler prototype

All handlers have this AAPCS-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 7.3.3.

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

7.3.2 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 7.6.1.

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 7.6.1. 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.

7.3.3 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, ReturnAddress target, unsigned size)
```

Where:

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

`ReturnAddress target` identifies the entry point of the user-level exception handler.

`Unsigned 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 AAPCS-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.

7.3.4 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 7.5.

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 *RVCT 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.

7.4 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 7.6.2.

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).

7.5 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.

7.5.1 _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 (VMOV, pre-UAL 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:

- VMOV (pre-UAL FMSR, FMDLR, FMDHR, FMDRR, FMSRR) and VMSR (pre-UAL FMXR) to write to VFP regs from ARM regs, and

- VMOV (pre-UAL FMRS, FMRDL, FMRDH, FMRRD, FMRRS) and VMRS (pre-UAL FMRX) to 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.

7.5.2 _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[MAXCOUNT];
};
```

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 Architecture Reference Manual. 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.

7.5.3 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.

7.6 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, VFP10 and VFP11 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.

7.6.1 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 *desc, 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.

7.6.2 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`.

8 References

For details of the VFP instruction set, refer to:

- *ARM Architecture v7-AR Reference Manual*
- *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:

- Cortex-A9 Floating-Point Unit Technical Reference Manual
- *VFP11 Vector Floating-Point Coprocessor (Rev r1p0) Technical Reference Manual*
- *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:

- RVDS 3.1 and later: *RVCT Compiler User Guide*
- RVDS 3.1 and later: *RVCT Compiler Reference Guide*
- RVDS 2.1 - 3.0: *RVCT Compiler and Libraries Guide*

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

- *RVD Target Configuration Guide*
- *RVDS 2.2 (or later) AXD and armsd Debuggers Guide*
- *RVI User Guide v1.4 (or later)*
- *Multi-ICE 2.2 User Guide* – for manual configuration files.

For details on Embedded Software Development, refer to

- *RVCT Developers Guide*

For details of linker configuration, refer to

- *RVCT Linker and Utilities Guide*