CF68KLib

68K Emulation Library for ColdFire

Important Note:

CF68KLib is a proprietary product of MicroAPL Ltd. MicroAPL makes no warranties in respect of the suitability of CF68KLib for any particular purpose, and accepts no liability for any loss arising out of the use of CF68KLib. The person or persons making use of this document and the CF68KLib emulation library must make the final evaluation as to their suitability and correctness for a particular application.

MicroAPL reserves the right to alter the specification of the product without warning.

MicroAPL welcomes comments and suggestions. These and any error reports should be sent using the feedback forms available on our web site:
http://www.microapl.co.uk

Version 2.0 December 2000

Updated versions of this manual, supporting documentation, and information on our range of porting tools and services can be downloaded from our web site:

http://www.microapl.co.uk

Introduction

What is the *CF68KLib* Emulation Library?

Although the ColdFire architecture is derived from and has much in common with the 68000 family, it is not possible simply to take a 680x0 binary and execute it on a ColdFire board. A large number of common 680x0 instructions are not supported by the ColdFire processor and would give rise to'Illegal Instruction' or 'Address Error' exceptions.

CF68KLib is a 680x0 emulation library which enables you to run 680x0 binaries *largely unmodified* under ColdFire. It achieves this by catching the exceptions caused by unimplemented 680x0 instructions, performing the same function as the instruction, and then adjusting the stacked program counter to resume execution at the following instruction.

Why 'largely unmodified' ? The assumption underlying the successful operation of CF68KLib is that all 680x0 instructions either behave exactly the same on ColdFire, or trap out so that the emulation library can deal with them. There are a very few 680x0 instructions for which this is not true, and you may have to make a few minor changes to your 680x0 code before it runs successfully on ColdFire. However you should find that the effort involved is very small.

Applications for CF68KLib

The CF68KLib emulation library can be used in one of two versions, User Mode and Supervisor Mode:

• You should use the User Mode form when your target system will basically be running native ColdFire code but you need to run an existing 680x0 application-level binary.

• You should use the Supervisor Mode form when you wish to run a complete 680x0 operating system under emulation.

User Mode Library

In this form CF68KLib will install its own handlers for the 'Illegal Instruction' and 'Address Error' exceptions which can occur when unimplemented 680x0 instructions are executed on ColdFire. These handlers will perform the same function as the missing instruction and then return execution to the instruction which follows. The existence of CF68KLib is thus transparent to your application code except that it can incur a substantial performance penalty.

It is occasionally necessary to make minor modifications to your 680x0 program before it can be run under CF68KLib. This is because there are a very small number of 680x0 instructions which are also legal in ColdFire (and hence do not cause an exception) but which do not behave identically. As an example, the MULS instruction executes identically under ColdFire except that it does not set the Overflow flag in the condition codes register.

Supervisor Mode Library

The ColdFire supervisor model is very different to the 680x0. For example there is only one stack pointer instead of separate User and Supervisor stacks, and the format of an exception stack frame is different. This means that although instructions such as TRAP and RTE are implemented in the ColdFire architecture, a 680x0 binary such as an operating system would be unlikely to work correctly. For this type of problem you can use the Supervisor Mode form of CF68KLib.

In the Supervisor Mode form, the CF68KLib library takes over the handling of all ColdFire exceptions in order to implement a complete 680x0 virtual machine. The presence of CF68KLib is transparent to your 680x0 code, which 'thinks' it is running on a real 680x0 processor, and you can run an entire 680x0 operating system including interrupt-driven hardware inside the virtual machine.

Because the Supervisor Mode form of CF68KLib has to take over the handling of all ColdFire exceptions it is generally not possible to use this form of the library alongside a native ColdFire operating system, although you can still execute application-level ColdFire code.

Supported Source Processors

A number of versions of CF68KLib are available which emulate different members of the 680x0 family:

68000
68010
68020
68030
68040
68060
CPU32
CPU32+

Each version is a very faithful emulation library of the particular 680x0 processor's instruction set, including both user- and optionally supervisor-mode instructions. Note however that CF68KLib does *not* include support for Floating Point (FPU) or Memory Management (MMU) instructions. Thus, for example, the 68040 version effectively emulates an LC68040 processor.

The 68000 and 68010 versions of CF68KLib depart slightly from the behaviour of the original processor when accessing memory. When the 68000-processor version of CF68KLib accesses memory it uses the full 32-bit 68K address (adding the 32-bit base address offset), rather than the 24-bit address which a real 68000 would use. In this respect, it is more like a later member of the 680x0 family than the original 68000 processor. In addition, the 68000 and 68010 versions of CF68KLib will not signal an exception on unaligned memory

accesses (e.g. reading a 16- or 32-bit value at an odd address). In this respect they behave like a 68020 or higher processor.

Supported Target Processors

CF68KLib is available for the ColdFire Version 3 and Version 4 cores. You must choose the appropriate version of the library for the ColdFire core you are using because the behavior of the two cores when trapping on certain unsupported 680x0 instructions is very slightly different

Building CF68KLib

Because so many variants of the CF68KLib emulation library exist, the library is not supplied in binary - or even source code - form. Instead a single 'embuild' executable application is provided which you run to *create* the source code for the library you require. For example, the command:

```
embuild -proc 68020 -a diab -core v4 -lib -user cf68klib_020.s
```

will create a source file 'cf68klib_020.s' containing the source of the User-Mode 68020 version of CF68KLib in a form that can be assembled with the Diab-Data ColdFire assembler. The version created is suitable for use with a Version 4 ColdFire core.

Customising CF68KLib

Functionally, each version of CF68KLib is divided into two parts - code which you may wish to customize for your particular application; and code which you will never normally need to touch. In order to achieve this separation, the source code is divided into two files which you generate and assemble separately and then link together.

The code which you may wish to customize is located in a 'front-end' source file which you first generate using embuild as follows, and then modify as required:

```
embuild -proc 68020 -core v3 -a diab -frontend -super frontend_020.s
```

This file contains sample code to initialize the CF68KLib library, together with a number of customizable routines…

As an example, consider the case where you are using the Supervisor-Mode version of CF68KLib to run an entire 680x0 operating system under emulation. The supervisor-mode version of the front-end file contains a customizable routine called 'cf68k_bus_error' which is called when a bus error exception occurs. You can either elect to handle the bus error in this routine, or pass it on to the 680x0 operating system's bus-error handler.

The main body of CF68KLib is contained in the back-end library file which you will not usually need to modify. It is also generated using embuild, e.g:

```
embuild -proc 68020 -a diab -lib -super cf68klib_020.s
```

CF68KLib requires Version 3 or later of the ColdFire core

Note that in order for the CF68KLib library to handle unimplemented 680x0 instructions they must cause an exception. For this reason CF68KLib can only be used with Version 3 or later of the ColdFire core, because earlier cores do not trap out on unimplemented instructions. In addition you must specify whether you are using the Version 3 or Version 4 core when generating the library by using the “-core” command-line option.

About this manual

This manual is divided into two parts. Part 1 discusses the principles behind CF68KLib's operation, and explains how it handles the issues which arise in executing 680x0 instructions on ColdFire. Part 2 details how to use CF68KLib in practise, how to build different versions of the library to emulate different 68000-family processors, and how to customise it to achieve different effects.

Part 1

Understanding CF68KLib

1 What is different in ColdFire?

1.1 Introduction

Although the ColdFire architecture is closely related to the 680x0, there are many simplifications to the instruction set which mean that 680x0 assembler code may require substantial modifications. In this chapter, we summarize the main differences between the 680x0 instruction set and ColdFire.

Nearly all of the differences are *omissions* from the 680x0 instruction set and addressing modes. This means that (with a few important exceptions detailed later), a 680x0 instruction which *is* implemented in ColdFire behaves in exactly the same way under the two architectures. In fact, almost all user-level (and much supervisor-level) ColdFire code can be run unchanged on a 68020 or later 680x0 processor. The converse, however, is not the case.

In outline, the main omissions fall into five categories:

- Missing addressing modes
- Missing instructions
- Non-availability of word- and byte-forms of nearly all arithmetic and logical instructions
- Many instructions act only on registers, not on memory
- Restrictions on available addressing modes for particular instructions
- Simplification of the supervisor-level programming model

In addition to these omissions, the ColdFire version 4 core includes some *new* instructions which CF68KLib optionally makes use of - in particular MVS (move-with-sign-extend) and MVZ (move-with-zero-extend).

1.2 Principles behind the differences

In order to understand the ColdFire instruction set in relation to that of the 680x0, it helps to have an appreciation of why the simplifications have been made. The philosophy behind ColdFire is influenced by the success of RISC processors in providing high performance - for a given degree of chip complexity - by eliminating seldom-used instructions and complex addressing modes, and by regularizing the instruction set to make it easier for the hardware to optimize despatch of the instruction stream.

However, standard RISC processors such as the PowerPC achieve high performance at the expense of low code density, in part because all instructions are the same width (generally 4 bytes) and also because only very simple addressing modes are available. In addition, RISC processors do not allow direct modification of memory locations; all memory reads and writes have to go via registers. This all means that programs compiled for RISC processors tend to be substantially larger than those compiled for CISC architectures such as the 680x0. This penalty does not greatly matter for powerful servers or workstations with 32MB or more of RAM, but for some embedded applications it can be a significant disadvantage, both in terms of system cost and power consumption.

The ColdFire architecture - which Motorola characterizes as “Variable-Length RISC” - aims to share many of the speed advantages of RISC, without losing too much of the code density advantages of the 680x0 family. Like most modern processor architectures, it is optimized for code written in C or C++, and instructions which are not frequently generated by compilers are amongst those removed from the instruction set. Some of the complex addressing modes - again not important for compilers - are eliminated, and the additional hardware complexities involved in supporting arithmetic operations on bytes and words also disappear. In order to regularize the instruction stream, **all ColdFire instructions are either 2, 4 or 6 bytes wide;** this is why certain combinations of source and destination operands are not available.

1.3 Missing addressing modes

The ColdFire addressing modes are quite similar to those of the original 68000, i.e. without the extensions introduced in the 68020 and later processors, but with some differences in indexed addressing. Compared with a 68020 or later processor, the comparison is as follows:

Fully supported:

Data Register Direct	D0	√
Address Register Direct	A3	√
Address Register Indirect	(A5)	√
Post-increment	(A1)+	√
Pre-decrement	-(A7)	√
Displacement (16-bit displacement)	100(A2)	√
PC Displacement (16-bit displacement)	100(PC)	√
Absolute Short	($100).W	√
Absolute Long	($220E0).L	√
Immediate	#3	√

Partially supported:

Indexed	(10,A2,D3.L*4)	(√)
PC Indexed	(0,PC,D2.L*2)	(√)

The restrictions on these two modes are:

(a) The displacement constant is 8-bit only;

(b) “Zero-suppressed” registers are not supported;

(c) The Index register can only be handled as a Long. Word-length index registers are not supported.

(d) The scale factor must be 1, 2, or 4. Scale factors of 8 are not supported.

Not implemented at all:

Memory-indirect post-indexed	([12,A3],D2*W,1000)	X
Memory-indirect pre-indexed	([12,A3,D2*W],1000)	X
PC-indirect post-indexed	([12,PC],D2*W,1000)	X
PC-indirect pre-indexed	([12,PC,D2*W],1000)	X

Note that further restrictions may be imposed on the addressing modes supported by particular instructions, even if a particular addressing mode is itself available on ColdFire.

1.4 Missing instructions

A number of instructions are not implemented at all under ColdFire. These include:

```
DBcc, EXG, RTR, RTD, CMPM,
ROL, ROR, ROXL, ROXR, MOVE16
ABCD, SBCD, NBCD
BFCHG, BFCLR, BFEXTS, BFEXTU
BFFFO, BFINS, BFSET, BFTST
CALLM, RTM, PACK, UNPK
CHK, CHK2, CMP2, CAS, CAS2, TAS (supported in V4 core),
BKPT, BGND, LPSTOP, TBLU, TBLS, TBLUN, TBLSN
TRAPV, TRAPcc, MOVEP, MOVES, RESET
ORI to CCR, EORI to CCR, ANDI to CCR
```

In addition, `DIVS` and `DIVU` (with some differences from the 680x0 equivalents) are available on some ColdFire processors but not others. `MULU` and `MULS` producing a 64-bit result are not implemented, but 16 x 16 producing 32-bit, and 32 x 32 producing (truncated) 32-bit, are available.

1.5 Long-word forms only

Most arithmetic and logical instructions can act on Long words only. This applies to:

```
ADD, ADDA, ADDI, ADDQ, ADDX, AND, ANDI, ASL, ASR
CMP*, CMPA, CMPI*, EOR, EORI, LSL, LSR,
NEG, NEGX, NOT, OR, ORI,
SUB, SUBA, SUBI, SUBQ, SUBX
```

*For the ColdFire Version 4 core the `CMP` and `CMPI` instructions are fully supported.

`MOVEM.W` has also been removed from the instruction set.

In fact, the only instructions which do act on the full set of byte, word and long operands are `CLR`, `MOVE` and `TST`. `EXT.W`, `EXTB.L` and `EXT.L` survive, as do `MULx.W` and `MULx.L`

1.6 Instructions which act only on registers, not on memory

Some arithmetic instructions cannot act directly on memory - the destination must be a register. This applies to:

```
ADDI, ADDX, ANDI, CMPI, ASL, ASR, LSL, LSR,
NEG, NEGX, NOT, EORI, ORI, SUBI, SUBX, Scc
```

Note that `ADDQ` and `SUBQ` can act directly on memory.

1.7 Restrictions on addressing modes for particular instructions

Even where a particular memory addressing mode does exist in ColdFire, some instructions are subject to further restrictions. Often, this is because of the limit

of six bytes as the maximum length of a single instruction. Specific restrictions include:

(a) Some combinations of addressing modes for `MOVE` are disallowed:

• If the source addressing mode is Displacement or PC Displacement, the destination addressing mode cannot be Indexed or Absolute.

• If the source addressing mode is Indexed, PC–Indexed or Absolute, the destination addressing mode cannot be Displacement, Indexed or Absolute.

• For the Version 2 and Version 3 cores, if the source addressing mode is Immediate the destination addressing mode cannot be Displacement.

• For the Version 4 core, if the source addressing mode is Immediate and the operation is a 32-bit move, the destination addressing mode cannot be Displacement.

(b) The addressing modes for `MOVEM` are restricted to only Displacement and Indexed - no Pre-decrement or Post-increment!

(c) For `BTST`, `BSET`, `BCLR` and `BCHG`, if the source operand is a static bit number, the destination cannot be Indexed or Absolute memory.

1.8 Miscellaneous Omissions

There are a few miscellaneous omissions for specific instructions:

• `LINK.L` is not supported
• `MOVE to CCR/SR`: Source must be Immediate or Data Register
• `MOVE from CCR/SR`: Destination must be data register

• For the Version 2 and 3 cores, BSR and Bcc accept only an 8- or 16-bit displacement. In the Version 4 core, BSR and Bcc accept 8-, 16- or 32-bit displacement as in most 680x0 processors.

1.9 Instructions which behave differently from the 680x0 equivalent

In almost all cases, an instruction/addressing mode which *does* exist in ColdFire behaves exactly like its 680x0 equivalent, which makes it easy for experienced 680x0 programmers to understand ColdFire code. It also means that code written for ColdFire can generally run unchanged on a 680x0 processor. If the instruction/addressing mode *does not* exist in ColdFire the processor will normally take an illegal instruction or address error exception (or a line-F exception for a few instructions), and the CF68KLib emulation library will fix up the problem.

However, there are a few subtle cases where the ColdFire instruction is not exactly the same as its 680x0 counterpart and does not cause an exception.

68020 multiply/divide instructions don't trap out

The most significant difference between ColdFire and 680x0 is that some of the multiply/divide instructions introduced with the 68020 **do not behave the same and do not cause an exception.** The following instructions are affected:

MULS.L <ea>,Dh:Dl	(Signed multiply:	32x32 -> 64)
MULU.L <ea>,Dh:Dl	(Unsigned multiply:	32x32 -> 64)
DIVS.L <ea>,Dr:Dq	(Signed divide:	64/32 -> 32r:32q)
DIVSL.L <ea>,Dr:Dq	(Signed divide:	32/32 -> 32r:32q)
DIVU.L <ea>,Dr:Dq	(Unsigned divide:	64/32 -> 32r:32q)
DIVUL.L <ea>,Dr:Dq	(Unsigned divide:	32/32 -> 32r:32q)

If your code uses any of these you will have to make at least some changes to your code. Strategies for dealing with these instructions are examined in Section 7.

MULU and MULS do not set the overflow Bit

The multiply instructions (`MULU` and `MULS`) do not set the overflow bit. This means that a 680x0 code sequence which checks for overflow on multiply may run under ColdFire, but give incorrect results.

ASL and ASR do not set the overflow Bit

ASL and ASR also differ in that they do not set the overflow bit - but this is less likely to cause problems for real programs!

MOVE.B <ea>,-(A7) and MOVE.B (A7)+,<ea> only change the stack pointer by one

Another potential problem is that the instruction "MOVE.B <ea>,-(A7)" behaves differently under ColdFire. On a 680x0 chip, the A7 register is first decremented **by two** and the byte-sized operand is then pushed; in ColdFire the A7 register is only decremented by one. The "MOVE.B (A7)+,<ea>" instruction is similarly affected. This means that a 68000 code sequence such as the following would give incorrect results:

```
move.l      d0,-(a7)
move.b      d1,-(a7)
...
move.l      2(a7),d0    ; ColdFire gets wrong value
```

1.10 New Instructions in ColdFire Version 4 Architecture.

In Version 4 of the ColdFire architecture as well as re-introducing some instructions present in 680x0 but missing from earlier ColdFire cores (e.g. CMP.B/W, BRA.L, BSR.L, Bcc.L), there are two new instructions which CF68KLib can make use of:

`MVS`	`<ea>,Dn`	Move byte or word and sign-extend to 32-bits in Dn
`MVZ`	`<ea>,Dn`	Move byte or word and zero-extend to 32-bits in Dn

Use of the new instructions is enabled if you specify the command-line option **–core v4**

It is possible that you plan to use a standard 680x0 assembler to assemble the source of CF68KLib. In this case the use of the MVS and MVZ instructions will cause a problem since they are not part of the 680x0 instruction set. To solve this you can specify the command-line option **-mnem68k**, which causes the embuild utility to generate a library in which every MVS and MVZ has been replaced by a DC directive yielding the same opcode value, e.g.

```
dc.l  $77680002   ; mvs.w 2(a0),d3
```

1.11 Simplification of the supervisor programming model

Various members of the 68000 family have different register sets available at the supervisor level. The most important simplification in ColdFire's supervisor-level model is that there is only one stack pointer, shared for all code including interrupts, supervisor-level services, and user code. It follows from this that, on ColdFire, it is never safe to write below the stack, since any interrupt which occurs would overwrite the stored data. (Writing below the stack, though not recommended, is possible in some 680x0 systems in user mode, because interrupts cause a switch to the Interrupt or Supervisor Stack Pointer). A further issue is that ColdFire processors automatically align the stack to a four-byte

boundary when an exception occurs, which can cause problems if code is reading or writing at a fixed offset from the stack pointer. In fact, it is strongly recommended (for performance reasons) that the ColdFire stack should be kept long-word aligned at all times.

2 Principles of CF68KLib's Operation

2.1 Handling Unimplemented Instructions

Many 680x0 instructions also exist in ColdFire and use the same opcodes. However, certain opcodes corresponding to legal 680x0 instructions are not valid for ColdFire.

A 680x0 opcode may be invalid in ColdFire because it corresponds to an instruction which does not exist - for example ADD.B or RTD. Such opcodes will cause the ColdFire processor to take an 'Illegal Instruction' exception. (A few unimplemented instructions such as MOVE16 cause a line-F exception).

Other 680x0 opcodes are invalid because they correspond to instructions which exist in ColdFire but for which the addressing mode specified is invalid. For example the instruction "NEG.L (A0)" would be illegal because the ColdFire version of the NEG instruction only allows a data register as its operand. Such opcodes will cause the ColdFire processor to take an 'Address Error' exception.

CF68KLib installs its own routines to handle these exceptions. When an unimplemented 680x0 instruction causes an exception the handler will decode the opcode of the instruction and dispatch to a small subroutine where a series of legal ColdFire instructions achieve an equivalent effect.

In the example of "NEG.L (A0)" above, CF68KLib would first decode the opcode to determine that it was a NEG.L instruction. It would then decode the addressing mode to determine that it was (A0). Finally, it would read the operand from (A0), negate it, and write it back, setting the condition bits appropriately before returning to the instruction following the NEG.

2.2 Handling Supervisor Mode Instructions

The ColdFire chip's Supervisor-level architecture is simpler than 680x0. For example, ColdFire only has a single stack pointer instead of two for the 68000, (and three for the 68020). Exception stack frames have a different format, and the stack pointer is always aligned to a four-byte boundary before an exception frame is created. These differences mean that without the aid of CF68KLib you could not simply take a 680x0 Operating System - even if it contains only legal ColdFire instructions - and expect it to run on a ColdFire board.

The Supervisor Mode form of CF68KLib allows you to run a whole 680x0 operating system under emulation. To do this it surrounds the 680x0 code with an entire 'virtual machine' which hides the differences in architecture. To achieve this CF68KLib uses a number of techniques:

• CF68KLib takes over the 'Illegal Instruction', 'Address Error', 'Line-A' and 'Line-F' exceptions and fixes up unimplemented user-mode 680x0 instructions in the manner described above.

• In order to coerce exception stack frames into 680x0 format, CF68KLib installs its own handlers for all ColdFire exceptions. When an exception such as a TRAP is taken the CF68KLib handler will modify the exception stack frame so that it is in the correct format and then pass control to the original 680x0 trap handler.

• Since exception frames are now in 680x0 format, CF68KLib needs to take steps to regain control before an RTE instruction (which is also legal in ColdFire) tries to execute with a frame format that would be invalid for ColdFire. To do this it runs the 680x0 program in ColdFire *User Mode.* In user mode, supervisor instructions like RTE will cause a privilege violation exception. By catching this exception CF68KLib can unpick the 680x0 exception frame and hence pass control back to the appropriate address.

• Certain 680x0 registers have no equivalent in ColdFire - for example the 68020 has three stack pointers, the USP, the ISP and the MSP. Some instructions such as RTE may cause a change to the active stack pointer; others such as "MOVEC to USP" may alter an inactive stack pointer. CF68KLib keeps track of the current contents of these unmapped registers in its private data area, and when a privileged instruction such as MOVEC or RTE is executed, the library will use/update its private copies. For example, when the 680x0 code switches stack from the SSP to the USP, the CF68KLib library saves away the current state of A7 into the location it uses to track the SSP, and reloads A7 from the location where it stored the USP value.

2.3 Handling Supervisor Mode Instructions - an annotated example

As a demonstration of how CF68KLib runs 680x0 operating-system code, consider the following annotated example which is intended to represent a real 68020 operating system in miniature…

On a real 68020 processor, this code would be called in Supervisor Mode. It sets up a user stack, switches to user mode, and then enters a loop to display the string "Hello world" using the services of a TRAP #0 handler:

```
; Set up user stack
      lea     user_stack_top,a0          ; [1]
      movec   a0,usp                     ; [2]

; Switch to user mode
      move    #0,sr                      ; [3]
```

```
; Now in 68020 user mode. Loop to write string
; continuously

loop:
      pea     hello_string                ; [4]
      trap    #0                          ; [5]
      dc.w    $1     ; In line selector: Write String
      addq.l  #4,a7                       ; [6]
      bra     loop                        ; [7]

hello_string:
      .byte   "Hello world", 13, 10, 0

user_stack:
      .space   10000
user_stack_top:


; TRAP #0 handler
trap0:
      movem.l  a0-a1,-(a7)                ; [8]
      move.l   10(a7),a1                  ; [9]
      cmp.w    #$1,(a1)+                  ; [10]
      bne.s    trap0_done                 ; [11]
      movec.l   usp,a0                    ; [12]
      move.l   (a0),a0                    ; [13]
      bsr      write_string               ; [14]
trap0_done:
      move.l   a1,10(a7)                  ; [15]
      movem.l  (a7)+,a0-a1                ; [16]
      rte                                 ; [17]
```

Under ColdFire you would call CF68KLib to begin executing this 'Operating System'. CF68KLib will set its internal representation of the 680x0 Status Register to Supervisor Mode, then pass control to the instruction at line [1] with the ColdFire processor in real user mode.

Because the ColdFire processor is in user mode, line [2] will cause a privilege violation trap. CF68KLib stores the value in A0 into its internal representation of the USP and then resumes execution.

Line [3] also causes a privilege violation trap. CF68KLib will store '0' into its internal copy of the 680x0 Status Register. It will detect that a change from supervisor mode to user mode has occurred, store the current A7 in its internal SSP value, and reload A7 from the internal representation of USP that was set by line [2].

Line [5] executes a TRAP instruction. This is a legal ColdFire instruction, but the TRAP exception frame created is different to the 68020. Notice that the TRAP is followed by an in-line selector representing the service requested from the operating system - in our case a request to write a string.

When Line [5] executes the TRAP, the exception is initially handled by CF68KLib. This converts the ColdFire-format exception frame into the format used by the 68020, and then passes control to the first instruction of the 68020 code's trap handler.

Within the trap handler itself, line [9] is picking up the program counter value from the 68020 exception stack frame - a pointer to the trap selector word. In our simple example it just checks for selector=1, but a real operating system would be much more complex. The program counter is updated to skip the selector at line [15].

At lines [12] and [13] the trap handler picks up the pointer to the string to write, which was pushed onto the user stack. Line [12] will cause a privilege violation trap which CF68KLib handles by copying its internal copy of USP into A0.

At line [17] the 68020 trap handler finishes by executing an RTE. Since this is a

privileged instruction it will cause a privilege violation trap because the ColdFire processor is in real User Mode. The CF68KLib exception handler will unpick the 68020-format exception frame and emulate an RTE in order to pass control back to the example code at line [7].

Finally, note that lines [8] and [16] are variants of the MOVEM instruction which are not legal in ColdFire. These cause an address error exception and the CF68KLib library handles this in order to emulate the behavior of the MOVEM using only legal instructions.

Part 2

Using CF68KLib

3 Creating the Source of CF68KLib

3.1 The 'embuild' application

There are a large number of potential variants of the source code of CF68KLib:

• Different versions of the library exist to support different members of the 680x0 family: 68000, 68010, 68020, 68040, 68060, CPU32 and CPU32+.

• Different versions exist to support the Version 3 and Version 4 ColdFire cores which behave slightly differently when trapping on certain illegal 680x0 instructions.

• The library can exist in one of two forms - User Mode (for running 680x0 applications in a ColdFire envionment), and Supervisor Mode (for running a complete 680x0 operating system under emulation).

• The library is divided into two source files - the main body of the emulation code, and a customizable front end.

• Source code may need to be in a form suitable for one of a number of supported ColdFire assemblers - Diab-Data, Microtec , GNU or Metrowerks.

Because so many variants of the CF68KLib emulation library exist, the library is not supplied in binary - or even source code - form. Instead a single 'embuild' executable application is provided which you run to *create* the source code for the library and front end you require. For example, the command:

```
embuild -core v3 -user -proc 68020 -lib -a diab cf68klib_020.s
```

will create a source file 'cf68klib_020.s' containing the source of the User-Mode 68020 version of the main CF68KLib emulation library in a form that can be assembled with the Diab-Data ColdFire assembler. The version generated is suitable for use with the ColdFire version 4 core.

3.2 Creating the Source Code of CF68KLib

To create the source code of CF68KLib you should take the following steps: (The command line options are explained in detail in the next section)

• Determine whether you're planning to use the ColdFire version 3 or version 4 core, because you'll need to generate the appropriate library (either the `–core v3` or `–core v4` option).

• Decide whether you need the User Mode or Supervisor Mode form of the emulation library (either the `-user` or `-super` option).

• Decide which member of the 680x0 family you wish to emulate (the `-proc` option).

• Use 'embuild' to create a file containing the main body of the library code (by specifying the `–lib` option).

• Use 'embuild' again to create a customizable front-end file (by specifying the `–frontend` option).

• Edit the front-end file to modify the initialization code and customize CF68KLib's behavior as described below.

• Assemble the two files and link them together with your ColdFire application. If you are using a standard 680x0 assembler (as opposed to a true ColdFire assembler) to build the Version 4 core library it will fail on the MVS and MVZ

instructions. See page 27 for use of the `-mnem68k` option to circumvemt this.

3.3 Invoking the 'embuild' application

The embuild executable is used to generate the source code of CF68KLib. The command line syntax is:

```
embuild [options...] filename
```

where `filename` is the name of the source file to be created. The command-line options are as follows:

-a <assembler>

Specifies which ColdFire assembler syntax to use for the generated code. The valid options are:

diab Diab Data's *das* (default)
gnu Gnu *gas*
mri Microtec Research

You may also need to specify one of the other output-syntax options such as **–out_cmp_reversed** or **–out_syntax**.

-core <v3 | v4>

Specfy whether you wish to generate a version of CF68KLib for use with the version 3 or version 4 ColdFire core.

-frontend

Generate front-end code. The file created will contain sample source code for the customizable front-end to CF68KLib. (You may only specify one of **-frontend** or **-lib**).

-lib

Generate source code for the main CF68KLib emulation library. (You may only specify one of **-frontend** or **-lib**).

-mnem68k

Use only valid 680x0 mnemonics. This option is useful if you are assembling the source code using a 68K assembler which doesn not support ColdFire instructions such as MAC or MVS. These will be converted into hexadecimal form, e.g. DC.L $nnnnnnnn

-omit <list>

Omit unwanted instructions. This option is useful if you are tailoring CF68KLib to have a small footprint and you know that the 680x0 code you are running does not make use of certain instructions. For example if you know that your code does not use the ABCD, NBCD or SBCD instructions you can omit the library code to handle these - they will be treated as illegal instructions if they are encountered.

The <list> parameter allows you to specify a comma-separated list of one or more of the following values:

bcd - Omit BCD instructions & lookup tables

bitfield - Omit 68020 bitfield instructions (BFCHG, etc) & lookup tables

64bit - Omit 68020 64-bit multiply/divide instructions & support routines

-out_cmp_reversed

Some 680x0/ColdFire assemblers swap round the operands to compare (`cmp cmpa cmpi`) instructions, i.e. they expect

```
cmp Dn,<ea>
```

rather than the

```
cmp <ea>,Dn
```

form specified in the Motorola 680x0 documentation. This option causes 'embuild' to reverse the operands to `CMP` instructions in the output file to be compatible with this idiosyncracy. It is typically used in conjunction with the **–out_syntax munix** option.

-out_reg_prefix

Causes 'embuild' to place a % prefix in front of register names in the generated code. The default is **off** unless you specify **–a gnu**. Register names are always prefixed with a % character if you specify **–out_syntax munix**.

-out_syntax <standard|mit|munix>

This option allows you to specify that the output syntax for ColdFire instructions should be modifed to use the ‘MIT’ or ‘Motorola Unix’ conventions rather than the standard Motorola mnemonics and syntax.

-proc <type>

Specify the processor you wish to emulate. The valid options are:

68000 (default)
68010
68020
68030
68040
68060
cpu32
cpu32+

Note that CF68KLib does not include support for Floating Point (FPU)

or Memory Management (MMU) instructions. Thus, for example, the 68040 version effectively emulates an LC68040 processor.

-super

Generate the Supervisor-Mode form of CF68KLib with support for operating system code. (You may only specify one of **–supe**r or **–user**).

-user

Generate the User-Mode form of CF68KLib (default). (You may only specify one of **–supe**r or **–user**).

4 Using CF68KLib

4.1 Initializing CF68KLib

Before running any 680x0 code, you must initialize CF68KLib by calling its `cf68k_initialize` routine. This routine takes one parameter: A0 must point to an area of 1024 bytes of memory which CF68KLib can use as its private data area.

The front-end file generated by 'embuild' contains some sample initialization code which demonstrates how this may be done:

```
      .text
sample_initialisation_code:
      lea     library_data_area,a0
      jsr     cf68k_initialize
      ...


; Memory for library's private use
      .data
library_data_area:
      .space      regList_size
```

(Note that the front-end defines an equate `regList_size` for the size of the area required) .

As part of its initialization process, CF68KLib needs to install its own exception handlers. For the User-Mode version it only needs to install handlers for the 'Illegal Instruction', 'Address Error', 'Line-A' and 'Line-F' exceptions; for the Supervisor-Mode version it install handlers for every type of exception.

In order to install a handler, CF68KLib does not directly patch the ColdFire exception vector table. Instead it calls a routine called `cf68k_install_vector` in the front-end file for every vector it wishes to

install. This is useful for two reasons.

• In the User-Mode version of CF68KLib it is expected that the library and the 680x0 application will be running under a native ColdFire operating system. The OS may disallow the practice of patching directly into the exception vector table, requiring you to make a system call in order to install vectors.

• The Supervisor-Mode version of CF68KLib will try to install its own handlers for all types of exception. This includes the interrupt handlers, so that an interrupt-driven 680x0 operating system can be run under emulation. However, your ColdFire system may include interrupt sources which you wish to handle with native ColdFire code. For these interrupt vectors you can ignore CF68KLib's request to install its own handler.

The routine `cf68k_install_vector` is called with two parameters - A0 contains the vector to install, and D0 contains the vector number. A typical implementation is included in the sample front-end file:

```
;   On entry:
;       A0 -> exception handler
;       D0 =  exception vector number

cf68k_install_vector:
      move.l  a1,-(a7)
      move.l  #coldfire_vector_base,a1
      move.l  a0,(a1,d0.l*4)
      move.l  (a7)+,a1
      rts
```

4.2 The Register Parameter Area

The block of private storage that you pass to CF68KLib during initialization includes an area used to communicate register values. When the library calls one of your front-end routines (see below) it will pass you a pointer to this area, in which it has stored the current contents of the 680x0 registers.

The generated sample front-end file will contain a series of equates which give offsets into this area for each of the registers:

```
reg_d0      .equ  0          Data Register D0
reg_d1      .equ  4                       D1
reg_d2      .equ  8                       D2
reg_d3      .equ  12                      D3
reg_d4      .equ  16                      D4
reg_d5      .equ  20                      D5
reg_d6      .equ  24                      D6
reg_d7      .equ  28                      D7
reg_a0      .equ  32         Address Register A0
reg_a1      .equ  36                      A1
reg_a2      .equ  40                      A2
reg_a3      .equ  44                      A3
reg_a4      .equ  48                      A4
reg_a5      .equ  52                      A5
reg_a6      .equ  56                      A6
reg_a7      .equ  60         Current (active) Stack Pointer
reg_pc      .equ  64         PC - Program Counter
reg_sr      .equ  68         SR - Status Register
reg_ccr     .equ  reg_sr+1   CCR - Condition Codes Register
```

The remaining fields are only included for the Supervisor-Mode version of CF68KLib. The exact format depends on *which* 680x0 processor the library is emulating; the following example is for the 68020:

```
reg_usp     .equ  72          USP - User Stack Pointer
reg_ssp     .equ  76          SSP - Supervisor Stack Pointer
reg_isp     .equ  reg_ssp     (Also known as ISP - Interrupt Stack Pointer)
reg_msp     .equ  80          MSP - Master Stack Pointer
reg_vbr     .equ  84          VBR - Vector Base Register
reg_sfc     .equ  88          SFC - Source Function Code Register
reg_dfc     .equ  92          DFC - Destination Function Code Register
reg_cacr    .equ  96          CACR - Cache Control  Register
reg_caar    .equ  100         CAAR - Cache Address Register
```

All the fields are four bytes, except `reg_sr` which is two bytes, and `reg_ccr` which is a single byte.

4.3 Running 680x0 Code

For the User-Mode version of CF68KLib, once the library has been initialized and has installed its own handlers for the 'Illegal Instruction' , 'Address Error', 'Line-A' and 'Line-F' exceptions, no further steps are required before you can begin executing 680x0 code. Any time that you have a section of 680x0 code you need to run you can just call it directly.

For the Supervisor-Mode version you must run the 680x0 operating system inside CF68KLib's 'virtual machine' - the wrapper it puts around your code to make the ColdFire chip look like a 680x0. You begin executing the 680x0 code by jumping to the library's '`cf68k_execute`' entry point. (NB This is not a subroutine, and control never returns directly to the caller).

Before jumping to `cf68k_execute` you must set the initial states that certain 680x0 registers will have once the code starts executing. This is done by

initializing the corresponding fields in the Register Parameter Area.

For example, a 68K operating system is typically started from a reset. When a real 680x0 processor is reset it will set the Status Register to supervisor state with the interrupt priority mask set at level seven. The Vector Base Register is forced to zero, and then the initial values of the Supervisor Stack Pointer and Program Counter are loaded from address zero in the operating system image. To simulate this you might perform the following initialization:

```
sample_initialization_code:
; First initialize the library
      lea     library_data_area,a0
      jsr     cf68k_initialize

; Now set the initial state of the 680x0 model

      lea     operating_system_image,a1
      move.l  a1,reg_vbr(a0)          ; Initialize VBR
      move.l  0(a1),reg_ssp(a0)       ; Initialize SSP
      move.l  4(a1),reg_pc(a0)        ; Initialize PC
      move.w  #0x2700,d0              ; Initialize SR
      move.w  d0,reg_sr(a0)

; Now go execute the code
      jmp     cf68k_execute
```

4.4 Callbacks to the front-end during 680x0 code execution

In order to allow you to customize the behavior of CF68KLib, the library will call routines in your front end file during the execution of 680x0 code. The callback routines fit into four categories:

• Routines which are called when CF68KLib emulates a 680x0 instruction which would cause an exception - for example a divide-by-zero.

• Routines which are called when CF68KLib encounters a 680x0 instruction which cannot sensibly be emulated because it is hardware-dependent - for example the CAS instruction.

• Routines which are called just before a 680x0 control register is read, and just after it is changed.

• A routine which is called when CF68KLib detects an error condition which means that the emulated code is unlikely to work correctly on ColdFire.

The exact routines depend on which member of the 680x0 family you are emulating, and on whether you are using the user-mode or supervisor-mode version of the library. In all cases the generated sample front-end file will contain sample routines for you to adapt.

CF68KLib will call your front-end routines in real Supervisor Mode with all interrupts fenced (i.e. with the ColdFire Status Register = 0x2700). You must make sure that you **do not lower the interrupt fence** if any of your interrupt handlers contain 680x0 instructions which are not also legal in ColdFire. Such instructions would cause an exception, but CF68KLib is still processing the previous exception and it is **not re-entrant**.

4.5 Callbacks to handle 680x0 Exceptions

Because the details of how to handle exceptions differ for user-mode and supervisor-mode versions of CF68KLib, the question is considered separately for each in turn below:

Handling Exception Callbacks in Supervisor Mode

The Supervisor-Mode form of CF68KLib emulates a complete 680x0 processor, including support for 680x0-style exception processing. Before the library begins emulating the processing of most 680x0 exceptions it will call one of the following routines in your front-end.

Routine	Vector Number	Exception
`cf68k_bus_error`	2	Bus Error
`cf68k_address_error`	3	Address Error
`cf68k_illegal_instruction`	4	Illegal Instruction
`cf68k_zero_divide`	5	Zero Divide
`cf68k_chk_exception`	6	CHK Instruction
`cf68k_trapv_exception`	7	TRAPV Instruction
`cf68k_privilege_violation`	8	Privilege Violation
`cf68k_trace`	9	Trace
`cf68k_line_a`	10	Line 1010 Emulator
`cf68k_line_f`	11	Line 1111 Emulator
`cf68k_format_error`	12	Format Error (68010 and higher)
`cf68k_trap0`	32	Trap #0 Instruction
`cf68k_trap1`	33	Trap #1 Instruction
`cf68k_trap2`	34	Trap #2 Instruction
`cf68k_trap3`	35	Trap #3 Instruction
`cf68k_trap4`	36	Trap #4 Instruction
`cf68k_trap5`	37	Trap #5 Instruction
`cf68k_trap6`	38	Trap #6 Instruction
`cf68k_trap7`	39	Trap #7 Instruction
`cf68k_trap8`	40	Trap #8 Instruction
`cf68k_trap9`	41	Trap #9 Instruction
`cf68k_trap10`	42	Trap #10 Instruction
`cf68k_trap11`	43	Trap #11 nstruction
`cf68k_trap12`	44	Trap #12 Instruction
`cf68k_trap13`	45	Trap #13 Instruction
`cf68k_trap14`	46	Trap #14 Instruction
`cf68k_trap15`	47	Trap #15 Instruction

For each of these exceptions you can elect to do nothing, in which case

CF68KLib will begin emulating normal exception processing for the exception. Alternatively, you can elect to handle the exception in your front-end routine and have CF68KLib resume execution at the instruction after the one which caused the exception.

The calling conventions are the same for each of these routines:

Parameters:

D0 = 16-bit opcode of instruction which caused exception

A0 -> register parameter area

Return:

D0 = 0 if you wish CF68KLib to begin emulating normal exception processing

D0 = 1 if you handled the exception in the front-end routine

You do not need to preserve any of the ColdFire registers D1–D7/A0–A6 in your front-end routine.

The ability to override CF68K's default processing of exceptions can be very useful. For example you might wish to extend the 680x0 instruction set by re-using unassigned opcodes. The following version of `cf68k_line_f` would output the character in D0 to the console every time the processor executed the opcode 0xF000:

```
;   On entry:
;       D0 =  opcode which caused exception
;       A0 -> register parameter area
;   On return:
;       D0 = 1 if we handled exception, else 0
cf68k_line_f:
      cmp.l     #$f000,d0          ; Output char?
      bne.s     not_char_output
```

```
      move.l    reg_d0(a0),d0      ; Get character
      bsr       output_character   ; Go output it
      addq.l    #2,reg_pc(a0)      ; Bump PC
      moveq.l   #1,d0              ; Say we handled
      rts                          ; exception
not_char_output:
      moveq.l   #0,d0              ; Say we didn't handle
      rts                          ; exception
```

Handling Exception Callbacks in User Mode

When the user-mode form of CF68KLib emulates certain instructions, it is possible that the instruction being executed would have caused a 680x0 exception. For example, the library emulates theTRAPV instruction which would cause an exception on a 680x0 processor if the V flag in the Condition Codes register is set.

If CF68KLib determines that an exception would occur during emulation of a 680x0 instruction, it calls one of the following front-end routines:

Routine	Vector Number	Exception
`cf68k_address_error`	3	Address Error
`cf68k_illegal_instruction`	4	Illegal Instruction
`cf68k_zero_divide`	5	Zero Divide
`cf68k_chk_exception`	6	CHK Instruction
`cf68k_trapv_exception`	7	TRAPV Instruction
`cf68k_line_a`	10	Line 1010 Emulator
`cf68k_line_f`	11	Line 1111 Emulator

Which of these routines is called is determined as follows:

• If the 680x0 opcode is a CHK or CHK2 instruction, the routine `cf68k_chk_exception` is called.

• If the 680x0 opcode is a TRAPV or TRAPCC instruction, the routine `cf68k_trapv_exception` is called.

• If the 680x0 opcode is a divide instruction *which is not legal in ColdFire*, `cf68k_zero_divide` is called. Note that certain instructions such as 'DIVS.W D0,D1' are also legal in ColdFire. They do not cause an illegal instruction trap, and hence are not handled by CF68KLib. If one of these instructions does a divide-by-zero, the native ColdFire exception is called.

• If the 680x0 opcode is a Line-F instruction (0xF000 - 0xFFFF), `cf68k_line_f` is called.

• If the 680x0 opcode is a Line-A instruction (0xA000 - 0xAFFF), `cf68k_line_a` is called. Note however that ColdFire reuses some of the line-A opcodes for ColdFire-specific instructions such as MAC.

• If the 680x0 opcode is some other opcode that is not legal in ColdFire, `cf68k_illegal_instruction` is called. Note however that ColdFire reuses some illegal 680x0 opcodes for ColdFire-specific instructions.

• If the library catches an 'Address Error' exception and it is genuine (not caused simply by a 680x0 instruction using an addressing mode that's not supported in ColdFire), `cf68k_address_error` is called.

The calling conventions are the same for each of these routines:

Parameters:

D0 = 16-bit opcode of instruction which caused exception

A0 -> register parameter area

Return:

D0 = 1 to say that you handled the exception in the front-end routine

You do not need to preserve any of the ColdFire registers D1–D7/A0–A6 in your front-end routine.

Unlike the Supervisor-Mode version of CF68KLib, you **must** handle each of these exceptions in your front-end, for example by printing an error message and quitting the application. If your front-end routine is able to fix up the error condition that caused the exception, you can return with D0 = 1, and CF68KLib will resume execution at the next instruction.

Modifying Registers during a Callback Routine

When one of the front-end routines such as `cf68k_trap0` is called, A0 points to the register parameter area in which the current values of all the 680x0 registers are stored. You can modify these values as required - for example:

```
move.l  reg_a7(a0),a1     ; Get A7 before TRAP
move.l  (a1)+,d0          ; Pop parameter to TRAP
move.l  d0,reg_a7(a0)     ; Update A7
```

You should note the following:

Stack Pointers

There are multiple entries connected with stack pointers:

`reg_usp` - value of the User Stack Pointer (USP)
`reg_ssp` - value of the Supervisor Stack Pointer (SSP). For the 68020/030/040 version this corresponds to the Interrupt Stack Pointer (ISP).
`reg_msp` - value of the 68020/030/040 Master Stack Processor (MSP).
`reg_a7` - value of the *active* stack pointer (USP, SSP or MSP)

The value passed in each of these fields is the value *before* emulation of 680x0 exception processing begins.

To modify the *active* stack pointer you should change `reg_a7` rather than one of the other fields.

Program Counter

The value of `reg_pc` depends on the cause of the exception:

Group 2 Exceptions: CHK, CHK2, Divide-by-zero, TRAP, TRAPV, TRAPcc
`reg_pc` points to the instruction *after* the one which caused the exception.

Group 3 Exceptions: Illegal Instruction, Line–A, Line–F, Privilege Violation.
`reg_pc` points to the instruction which caused the exception.

Group 4 Exceptions: Trace
`reg_pc` points to the instruction *after* the one which caused the exception.

If you elect to handle the exception within your front-end routine, CF68KLib will resume execution of the 680x0 code at the instruction whose address is in `reg_pc`. It follows that for Group 3 exceptions you must modify this value before returning.

4.6 Callbacks to Handle Problem Instructions

Certain 680x0 instructions which have no equivalent in ColdFire cannot sensibly be handled by CF68KLib. These instructions are

• TAS: Test-and-Set instruction used to synchronize several processors (Not supported for Version 3 core; fully supported for Version 4 core).
• CAS: Compare-and-Swap instruction used in to implement semaphores in a multi-processor environment.
• CAS2 (68020 and higher): Similar to CAS
• MOVES (68010 and higher): Move Address Space instruction
• BKPT (68020 and higher): Hardware breakpoint instruction
• CALLM (68020 only): Call Module instruction uses external hardware for access control.
• RTM (68020 only): Similar to CALLM

For each of these instructions, CF68KLib includes a front-end routine:

```
cf68k_tas (version 3 core only)
cf68k_cas
cf68k_cas2
cf68k_moves
cf68k_bkpt
cf68k_callm
cf68k_rtm
```

The calling conventions are similar to the exception-handling routines described above:

Parameters:

D0 = 16-bit opcode of instruction which caused exception

A0 -> register parameter area

Return:

D0 = 1 to say that you handled the exception in the front-end routine

You do not need to preserve any of the ColdFire registers D1–D7/A0–A6 in your front-end routine.

4.7 Callbacks when a Control Register is Read or Modified

Many of the control registers used in the 680x0 architecture have no direct equivalent in ColdFire. For example, there is no Source Function Code Register (SFC).

The supervisor-mode form of CF68KLib has partial support for handling these registers - it reserves a 32-bit field in the register parameter area for each of them, and faithfully reads or updates it when emulating MOVEC instructions such as 'MOVEC D0,SFC' and 'MOVEC SFC,D0'. However, CF68KLib never modifies any control registers in the real ColdFire processor

The 680x0 control registers which are only partially supported include the following: SFC, DFC, CACR, CAAR, TC, ITT0, ITT1, DTT0, DTT1, MMUSR, URP, SRP.

Registers which are *fully supported* are the User Stack Pointer (USP), Interrupt Stack Pointer (ISP), Master Stack Pointer (MSP) and Vector Base Register (VBR).

If you need to do additional processing when the 680x0 control registers are

used or changed, you can do so in a callback routine:

`cf68k_read_control_register` is called just before a control register is read via the 'MOVEC <control-register>,Rn' instruction.

`cf68k_write_control_register` is called just after a control register is written via the 'MOVEC Rn,<control-register>' instruction.

The parameters to these two routines are the same:

Parameters:

D0 = 32-bit opcode of MOVEC instruction
A0 -> register parameter area

Return:

None.

Note that the parameter passed in D0 is the full 32-bit opcode of the MOVEC instruction, from which you can determine which control register is affected.

After the front-end routine has been called, CF68KLib will complete the emulation of the MOVEC instruction.

4.8 Callback to handle problems during emulation

There are a few conditions under which CF68KLib detects that it cannot successfully emulate a 680x0 instruction. For these instructions the front-end routine `cf68k_emulation_error` is called.

Parameters:

D0 = opcode which caused the poblem

D2 = error code (see below)

A0 -> register parameter area

Return:

D0 = 1 to say that you handled the exception in the front-end routine

The possible conditions under which CF68KLib detects an emulation error are as follows:

• Error code = -1

This occurs if the 680x0 instruction was something like:

```
move.l -4(a7),(a0,d0.w)
```

The problem with this instruction is that it is not legal in ColdFire and hence causes an exception, but by the time CF68KLib is called to begin emulating the instruction the exception stack frame has over-written the data at -4(a7)

• Error code = -2

This occurs if the 680x0 instruction was something like:

```
move.l (a7)+,(a0,d0.w)
```

The problem with this instruction is that the exception does not occur until the ColdFire processor is half-way through processing it - it succesfully fetches the source operand from (a7)+ but then discovers that the destination operand is not legal for ColdFire. It then proceeds to take an exception, but the exception stack frame over-writes the source operand.

• Error code = -3

This is generated by the User-Mode version of CF68KLib if it calls one of your front-end routines such as `cf68k_zero_divide`, but you do not handle the exception (You don't return with D0 = 1).

4.9 Miscellaneous

You can find out the version number of CF68KLib at any time by calling the library routine `cf68k_get_version:`

Parameters:

None

Returns:

D0.L Major version number
D1.L Minor version number
D2.L Revision number
D3.L Processor id:

1 - 68000
2 - 68010
3 - CPU32
4 - CPU32+
5 - 68020
6 - 68030
7 - 68040
8 - 68060

You can also determine this information by looking at the headers of the source files generated by 'embuild'.

5 Handling Problematic Instructions

5.1 Instructions which can cause problems

The principle behind the successful operation of CF68KLib is that all 680x0 instructions are either legal in ColdFire - and behave identically - or cause an exception which CF68KLib can catch to handle the differences. Unfortunately, as described in Section 1.9, there are a very few 680x0 instructions for which this is not the case. To recap:

1. Certain 68020 multiply/divide instructions don't trap out and don't give the same result:

MULS.L <ea>,Dh:Dl	(Signed multiply:	32x32 -> 64)
MULU.L <ea>,Dh:Dl	(Unsigned multiply:	32x32 -> 64)
DIVS.L <ea>,Dr:Dq	(Signed divide:	64/32 -> 32r:32q)
DIVSL.L <ea>,Dr:Dq	(Signed divide:	32/32 -> 32r:32q)
DIVU.L <ea>,Dr:Dq	(Unsigned divide:	64/32 -> 32r:32q)
DIVUL.L <ea>,Dr:Dq	(Unsigned divide:	32/32 -> 32r:32q)

2. The multiply instructions (MULU and MULS) do not set the overflow bit. This means that a 680x0 code sequence which checks for overflow on multiply may run under ColdFire, but give incorrect results.

3. The arithmetic shift instructions (ASL and ASR) also differ in that they do not set the overflow bit

4. The instructions "MOVE.B <ea>,-(A7)" and "MOVE.B (A7)+,<ea>" only change the stack pointer by one - on 680x0 the stack pointer would change by two.

If any of these differences affect the correct operation of your 680x0 program you will need to make changes to the source code.

To handle 2. or 3. or 4. you need to recode the source to avoid using the problem instruction. CF68KLib does provide support for handling problem 1 - the wrong behavior of the 68020 multiply/divide routines.

5.2 Handling 68020 Multiply/Divide Instructions

CF68KLib provides full support for the 68020 multiply and divide instructions. However under normal circumstances these routines would not be called because the ColdFire processor does not trap out when it encounters the instructions. What is needed is a way of *forcing* the ColdFire processor to trap so that CF68KLib can handle the instruction.

CF68KLib reassigns one of the 16-bit opcodes, 0x4E00, which is not used in 680x0 or ColdFire and which causes an exception. This is used as an 'escape' telling CF68KLib to emulate the next instruction. For example if your source code contains:

```
.short  0x4E00
divsl.l d0,d1:d2
```

…then CF68KLib will catch the exception caused by the 0x4E00 opcode and emulate the DIVSL instruction which follows as though it had itself caused the exception.

Index