# 4FM

# Programmer's guide

4DSP LLC

Email: support@4dsp.com

# Table of Contents

## Revision History

| Date | Revision | Revision |
|---|---|---|
| 2013-12-10 | - Added FM780/VP780 information.<br>- Added information about _4FM_SetAPIOptions()<br>- Added information about offsetting the temperature reading. | 3.2 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# 1 Introduction

The programmer's guide describes the software layer part of 4DSP's "4FM Standard Development Kit".

The software layer is generic to most of the 4DSP hardware devices and able to communicate with several 4DSP hardware devices in your system.

The API (Application Programming Interface) is the key component allowing for communication with a 4DSP hardware device through its device driver. Some API functions are not documented because they are only providing the user with backward compatibility when using old 4DSP firmwares. Documentation about these old functions is available upon request.

The FPGA A firmware implemented by default on 4DSP PMC/XMC/cPCI devices allows the user to transfer data and commands back and forth between the host computer and FPGA B. The following diagram illustrates the architecture.

4DSP's CompactPCI and VPX card as FC6301, VP680 and VP780 do not have a FPGA B and only have an FPGA A

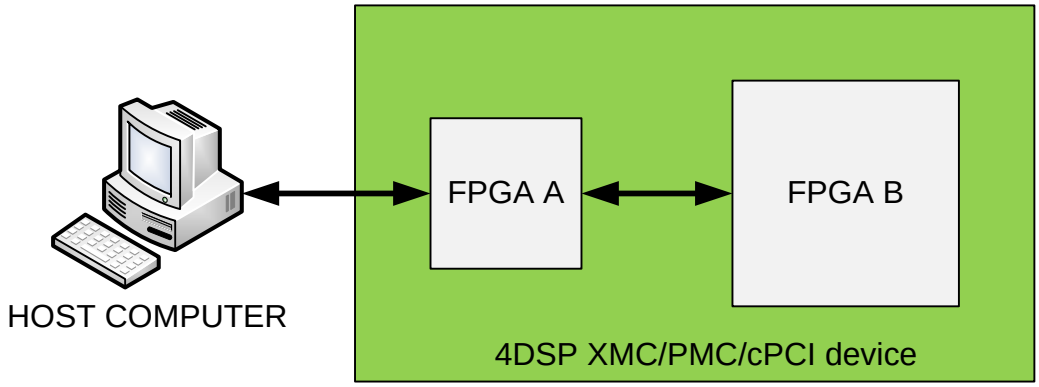


**Figure 1: PCI Data/Command flow in the system (FM482, FM489, FM577, FM680, FM780)**

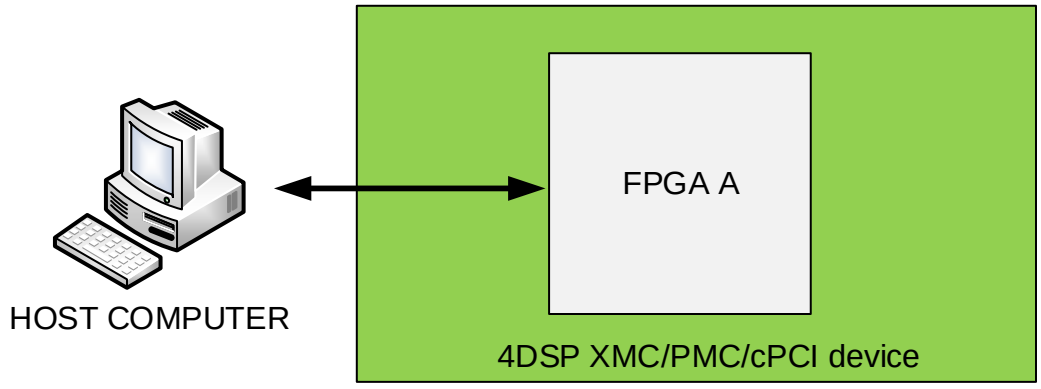


**Figure 2: PCI Data/Command flow in the system (FC6301, VP680, VP780)**

# 2 SDK Software Components

A few components are available after a successful SDK installation. This chapter describes these various components.

## 2.1 Application Programming Interface

The API is provided as a compiled Visual Studio 2012 library. A header (.h), a library configuration (.lib) and the library (.dll) are the components available for Windows users. The API is also available for Linux users where the API is delivered as a source package one can compile on most of the Linux distributions. The Linux Board Support Package (BSP) is available separately and need to be specifically requested.

## 2.2 Reference Applications

A few reference applications are part of the SDK and are available as source code. These sources are meant to be compiled and executed within Visual Studio 2012. These applications are the following:

| | |
|---|---|
| **4FM** | This application is a the simplest application that can control a 4DSP hardware device. This application is available under the "C Example" folder. |
| **SIPMEMTEST** | This application is able to test BLAST sites (memory banks) for DDR2, DDR3, QDR2, QDR+ and nand Flash. |

HINT: The 4FM API has to be properly referenced in Visual Studio to allow these applications to compile properly. Please refer to the "4FM Getting Started Guide" document for more information about this.

# 3 Common Software Operations

A few operations are available through the API. They are separated in two main categories; Information/Diagnostics and Communication. This chapter describes both of these categories and includes some example source code to illustrate the operations.

## 3.1 Information/Diagnostics

Temperatures, voltages, firmware IDs as well as firmware revisions are made available by the firmware through registers. It is not recommended to read these registers directly as a few functions in the API provide the required mechanism and data structures.

The following source code example illustrates how to obtain this information using the API

```
#include <stdio.h>
#include <stdlib.h>
#include <4FM.h>

int main(int argc, char *argv[])
{
  _4FM_DeviceContext ctx;
  _4FM_error_t rc;
  _4FMInformation info;
  _4FMDiagnostics diags;

  /* Open device and check handle is valid */
  rc = _4FM_OpenDevice(&ctx, "FM780", 0);
  if(rc != _4FM_OK){
        fprintf(stderr, "failed to open device\n");
        return EXIT_FAILURE;
  }

  /* Perform reset */
  rc = _4FM_ResetDevice(&ctx);
  if(rc != _4FM_OK) {
        fprintf(stderr, "failed to reset device\n");
        return EXIT_FAILURE;
  }

  /* Have to wait a little bit ( some diags require this ) */
  Sleep(100);

  /* obtain the information ( IDs and revisions ) */
  rc = _4FM_GetInformation(&ctx, &info);
  if(rc != _4FM_OK) {
         fprintf(stderr, "failed to obtain information\n");
         return EXIT_FAILURE;
  }

  /* obtain the diagnostics ( temperature and voltages) */
  rc = _4FM_GetDiagnostics(&ctx, &diags);
  if(rc != _4FM_OK) {
         fprintf(stderr, "failed to obtain information\n");
         return EXIT_FAILURE;
  }

  /* Close device */
  _4FM_CloseDevice(&ctx);

  return EXIT_SUCCESS;
}
```

### 3.1.1 The information structure (_4FMInformation)

This data structure is populated by **_4FM_GetInformation()** function. Some of the information is read using low speed communication busses in the firmware and it is recommended to **Sleep()** for a few hundred milliseconds after calling **_4FM_ResetDevice()** because of that**.**

Structure as defined in 4FM.h:

```
typedef struct _4FMInformation
{
  struct { unsigned hi, lo, cust, pci, type, ext; } fpga_A_revision;
  struct { unsigned hi, lo, cust; } fpga_B_revision;
  struct { unsigned hi, lo; } cpld_revision;
  unsigned board_serial;
  unsigned bankQDR;
  unsigned fpga_b_firmwaretype;
  unsigned fpga_b_guicompatible;
  unsigned user_value;
  char fpga_B_device_type[32];
} _4FMInformation;
```

Description of the items:

*fpga_A_revision* holds the information available about FPGA A firmware.

*fpga_B_revision* holds the revision information available about FPGA B firmware.

*cpld_revision* holds the information available about CPLD firmware.

*board_serial* holds the board's serial number programmed into the on board flash memory.

*bankQDR* only valid for FM482, the number and location of the QDR banks mounted on the printed circuit board.

*fpga_b_firmwaretype* holds FPGA B firmware type (firmware ID).

*fpga_b_guicompatible* tells if the current firmware is compatible with memory tests in the "4FM GUI Control Application". Note that this field is deprecated and will be removed without prior notice.

*user_value* holds the 32 bit value once can write to the onboard flash using "4FM GUI Control Application". This value is also called "User ROM" because it is non volatile. This information is actually read from the FLASH memory mounted on the 4DSP hardware devices.

*fpga_B_device_type* holds the Xilinx part number matching the FPGA B device mounted on the hardware. This information is read from the FLASH memory mounted on the 4DSP hardware devices, not read from the FPGA device itself.

### 3.1.2 The diagnostics structure (_4FMDiagnostics)

This data structure is populated by **_4FM_GetDiagnostics()** function. Some of the information is read using low speed communication buses in the firmware and it is recommended to **Sleep()** for a few hundred milliseconds after calling **_4FM_ResetDevice()** because of that**.** Note that only a subset of this data structure is available depending which 4DSP hardware device is currently used. You can see that as source commands on the view below.

Structure as defined in 4FM.h:

```
typedef struct _4FMDiagnostics
{
  float mgtvccaux;                    // VP780 + FM780
  float mgtavtt;                      // VP780 + FM780
  float mgtavcc;                      // VP780 + FM780
  float vccblast1;                    // C630 + VP780
  float vccblast2;                    // C630 + VP780
  float voltageADJ;                   // C630 + VP680 + VP780
  float currentV120;                  // C630
  float vccblast3;                    // VP680
  float vioblast1;                    // VP680 + VP780
  float vioblast2;                    // VP680 + VP780
  float vioblast3;                    // VP680
  float vttblast1;                    // VP680
  float temprefA;                     // FM577 + FM489 + FM780
  float temprefB;                     // FM577 + FM489 + FM780
  float voltage120;                   // FM577 + FM489 + FM780
  float voltageM120;                  // FM577 + FM489 + FM780
  float voltageIOPN4;                 // FM577 + FM489 + FM780
  float voltage50;                    // FM577 + FM489 + FM780
  float voltage25;                    // FM577 + FM489 + FM780
  float voltageMGT1V0;                // FM489 only
  float voltageMGT1V2;                // FM489 only
  float voltage10;                    // FM489 only + FM780
  float voltage33;                    // All devices + FM780
  float voltage18;                    // All devices + FM780
  float voltage12;                    // All devices
  float voltageVref;                  // All devices + FM780
  float voltageFP;                    // All devices + FM780
  float tempB;                        // All devices + FM780
  float tempA;                        // All devices + FM780
  float voltage15;                    // FC6603 + FM780
  float voltageS6vccint;              // FC6603
} _4FMDiagnostics;
```

*mgtvccaux, mgtavtt, mgtavcc* hold the voltage present on the MGT FPGA rails.

*vccblast1, vccblast2* hold the core voltage available on BLAST site1 and 2 respectively.

*voltageADJ* holds the voltage currently on the FMC connector's VADJ power rails.

*currentV120* holds the current actually present on the 12V power rail.

*vccblast3* holds the voltage on BLAST site 3.

*vioblast1*, *vioblast2* and *vioblast3* represent the voltage on a respecting IO bank.

*vttblast1* represents the voltage on the respecting IO termination bank.

*temprefA, temprefB* hold the temperature present on the monitoring devices.

*voltage120* holds the +12V voltage present on the printed circuit board.

*voltageM120* holds the -12V voltage present on the printed circuit board.

*voltageIOPN4* holds the voltage used as IO voltage on the PN4 connector on the printed circuit board.

*voltage50* holds the +5V voltage present on the printed circuit board.

*voltage25* holds the +2.5V voltage present on the printed circuit board.

*voltageMGT1V0* holds the +1V voltage present on the printed circuit board (MGTs).

*voltageMGT1V2* holds the +1.2V voltage present on the printed circuit board (MGTs).

*voltage10* holds the +1V voltage present on the printed circuit board.

*voltage33* holds the +3.3V voltage present on the printed circuit board.

*voltage18* holds the +1.8V voltage present on the printed circuit board.

*voltage12* holds the +1.2V voltage present on the printed circuit board.

*voltageVref* holds the reference voltage present on the printed circuit board.

*voltageFP* holds the voltage used as IO voltage for the front panel IOs on the printed circuit board.

*tempB* holds the temperature of FPGA B mounted on the printed circuit board.

*tempA* holds the temperature of FPGA B mounted on the printed circuit board.

*voltage15* holds the +1.5V voltage present on the printed circuit board.

*voltageS6vccint* holds the voltage of Spartan 6 device present on the printed circuit board.

## 3.2 Communication

### 3.2.1 Communication (hardware with two FPGAs; FM482, FM489, FM680 and FM780)

Two different communication interfaces are available from the API. The first way to communicate with FPGA B firmware is reading and writing firmware registers at a given address (communicate using the command bus). The second way to communicate with FPGA B firmware is reading and writing memory using DMA (Direct Memory Access) operations (communicate using the data bus).

FPGA A is a transparent interface between FPGA B firmware and the host computer and is able to forward both register and DMA requests to FPGA B firmware. The reference firmware have an "inter FPGA slave" block and this block has two distinct communication buses, one for the register operations (command bus) and the other one for DMA operations ( data bus )

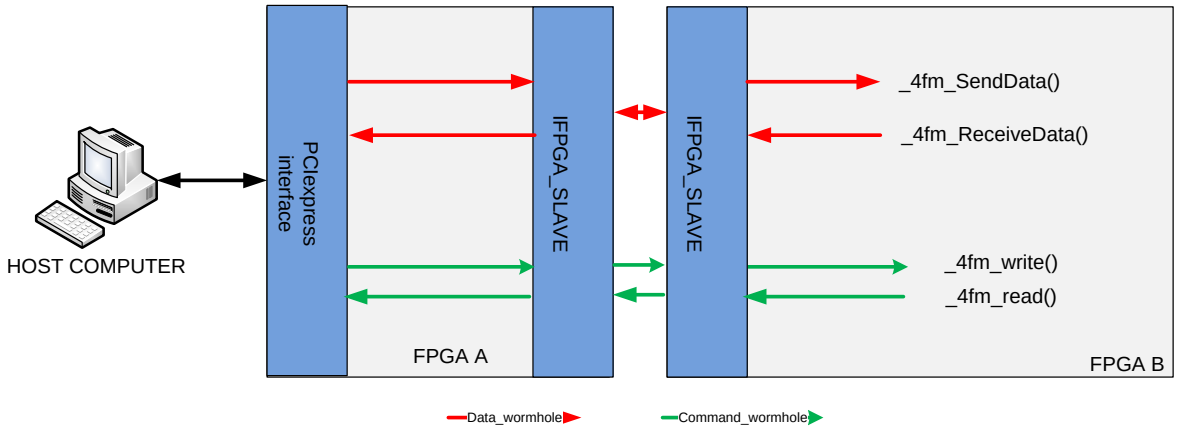


**Figure 3: Specific API functions generate traffic on two buses (cards with two FPGAs)**

### 3.2.2 Communication (hardware with one FPGA; FC6301, VP680 and VP780)

Two different communication interfaces are available from the API. The first way to communicate with the FPGA firmware is reading and writing firmware registers at a given address (communicate using the command bus). The second way to communicate with FPGA firmware is reading and writing memory using DMA (Direct Memory Access) operations (communicate using the data bus).

The reference firmware have an "HOST interface" block and this block has two distinct communication buses, one for the register operations (command bus) and the other one for DMA operations ( data bus )

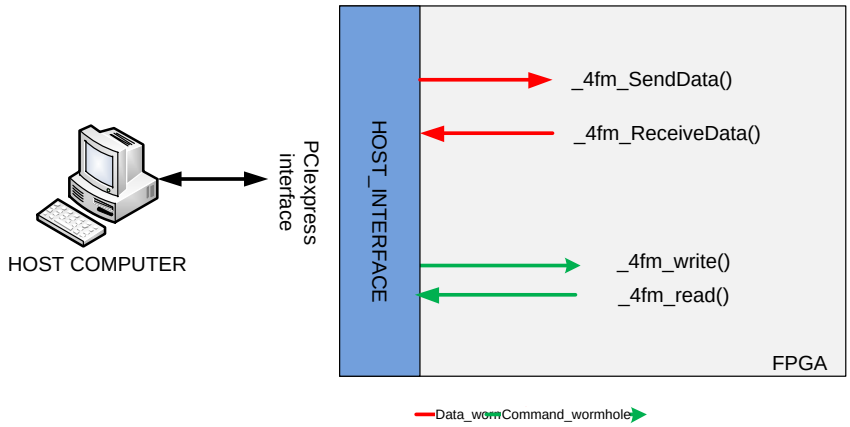


**Figure 4: Specific API functions generate traffic on two buses (cards with one FPGA)**

### 3.2.3 Communicating using the "Command Wormhole"

The command wormhole allows user to send and receive commands to/from FPGA B. These commands are routed to FPGA B transparently by FPGA A. In the current reference firmwares only two commands are implemented: read command and write command. These two commands allow reading and writing 32 bits wide values from/to stars in FPGA B. Each star is assigned to a specific memory range of the total 27 bits word address range that is available for register access.

A read command is initiated by the host application by calling the function **_4fm_read().** This function is part of the 4FM API. Note that **_4fm_read()** does not read directly from the hardware device. This function reads data from a software FIFO (First In First Out ) memory implemented in the device driver. After calling this function the driver will send a read command packet to the command bus. The Star that decodes the address as a valid address will return the requested data to the driver who will place the data into the software FIFO.

In the other hand, a write command is initiated by the host application by calling the function **_4fm_write().** This function is also part of the 4FM API. There is no FIFO buffering when writing from host to firmware. The driver will send a write command packet to the command bus when this function is called.

Following is a source code actually writing and reading to/from FPGA B's command wormhole:

```
#include <stdio.h>
#include <stdlib.h>
#include <4FM.h>

int main(int argc, char *argv[])
{
  _4FM_DeviceContext ctx;
  _4FM_error_t rc;
  unsigned long valout, valin;

  /* Open device and check handle is valid */
  rc = _4FM_OpenDevice(&ctx, "FM780", 0);
  if(rc != _4FM_OK){
        fprintf(stderr, "failed to open device\n");
        return EXIT_FAILURE;
  }

  /* Perform reset */
  rc = _4FM_ResetDevice(&ctx);
  if(rc != _4FM_OK) {
        fprintf(stderr, "failed to reset device\n");
        return EXIT_FAILURE;
  }

  /* Write a value to a read/write register (0x4) */
  valout =  0x87564312;
  rc = _4FM_Write(&ctx, 0x4, valout, 0);
  if(rc != _4FM_OK) {
       fprintf(stderr, "failed to write\n");
       return EXIT_FAILURE;
  }

  /* Read a value from a read/write register (0x4) */
  rc = _4FM_Read(&ctx, 0x4, &valin, 2000);
  if(rc != _4FM_OK) {
         fprintf(stderr, "failed to read\n");
         return EXIT_FAILURE;
  }

  /* Both value should be equal */
  if(valin!=valout) {
         fprintf(stderr, "Comparison error (0x%8.8x!=0x%8.8x\n",
                valin, valout);
         return EXIT_FAILURE;
  }

  /* Close device */
  _4FM_CloseDevice(&ctx);

  return EXIT_SUCCESS;
}
```

### 3.2.4 Communicating using the "Data Wormhole"

The data wormhole allows user to send/receive larger quantities of data at higher transfer rates to/from FPGA B. The operations are using DMA (Direct Memory Access). 4DSP has chosen a "scatter and gather" implementation also known as "chained list" implementation.

Two functions are available in the API, they are **_4fm_ReceiveData()** and **_4fm_SendData()**. These functions both take a pointer to a previously allocated memory buffer. The buffer should be allocated in an aligned manner where the buffer is PAGE aligned (4096 bytes in most cases). On the Windows operating system this is done by using the function **_aligned_malloc()**. This function can be used after including the "malloc.h" header in your source code. On Linux operating systems the function **posix_memalign()** can be used for this purpose.

The data wormhole and DMA has a big advantage over the command wormhole, it is fast and well optimized. This method allows reading and writing data at transfer speeds over 600MB/s depending on the PCI bus topology and performances.

Following is a source code actually writing and reading to/from FPGA B's data wormhole:

*This source code is an abstract only. Configuration of firmware's data routers is not covered by this code*

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <4FM.h>

int main(int argc, char *argv[])
{
  _4FM_DeviceContext ctx;
  _4FM_error_t rc;
  char *bufin, *bufout;

  /* allocate the buffers, aligned manner, 4kB aligned on 4kB. */
  bufin = (char *)_aligned_malloc(4096, 4096);
  bufout = (char *)_aligned_malloc(4096, 4096);

  /* Open device and check handle is valid */
  rc = _4FM_OpenDevice(&ctx, "FM780", 0);
  if(rc != _4FM_OK){
       fprintf(stderr, "failed to open device\n");
       return EXIT_FAILURE;
  }

  /* Perform reset */
  rc = _4FM_ResetDevice(&ctx);
  if(rc != _4FM_OK) {
       fprintf(stderr, "failed to reset device\n");
       return EXIT_FAILURE;
  }

  /* tell FPGA A firmware to route DMAs to FPGA B */
  rc = _4FM_SelectTarget(&ctx, tgMainFPGA);
  if(rc != _4FM_OK) {
        fprintf(stderr, "failed to set DMA targets\n");
        return EXIT_FAILURE;
  }
```

```
  /* Send data over the data wormhole */
  rc = _4FM_SendData(&ctx, bufout, 4096);
  if(rc != _4FM_OK) {
      fprintf(stderr, "failed to send data\n");
      return EXIT_FAILURE;
  }

  /* Receive data from the data wormhole */
  rc = _4FM_ReceiveData(&ctx, bufin, 4096);
  if(rc != _4FM_OK) {
        fprintf(stderr, "failed to receive data\n");
        return EXIT_FAILURE;
  }

  /* Both buffers should be equal */
  if(memset(bufin, bufout, 4096)) {
        fprintf(stderr, "Comparison error\n");
        return EXIT_FAILURE;
  }

  /* Close device */
  _4FM_CloseDevice(&ctx);

  return EXIT_SUCCESS;
}
```

# 4 4FM Tools Guide

A number of command line tools are provided with the 4FM SDK. These tools allow a variety of tasks , which are:

- Get Diagnostics
- Read / Write Mailbox (Not supported by StellarIP based firmware, ie FM680)
- Read / Write Custom Register (Not supported by StellarIP based firmware, ie FM680)
- Read / Write System Registers (StellarIP based firmware, ie FM680)
- Reset
- Upload firmware
- Write user ROM

## 4.1 _4FM_get_diagnostics

Displays diagnostics information from the specified card. This contains the voltage on 3.3, 1.8, 1.2, 0.85 and Front Panel rails, plus the main FPGA junction temperature and other hardware specific voltagees.

**Usage**

```
_4FM_get_diagnostics <devtype><devno>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

## 4.2 _4FM_get_information

Displays information regarding the specified card. This contains the driver and library versions, Device A revision, CPLD revision, board serial number, user ROM value, and Device B device type.

**Usage**

```
_4FM_get_information <devtype><devno>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

## 4.3 _4FM_write_mailbox

Writes a value to the mailbox of the specified card.

**Usage**

```
_4FM_write_mailbox <devtype><devno><value>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

## 4.4 _4FM_read_mailbox

Reads the mailbox of the specified card.

**Usage**

```
_4FM_write_mailbox <devtype><devno>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

## 4.5 _4FM_read_creg (deprecated)

Reads the specified custom register of the specified card.

**Usage**

```
_4FM_read_creg <devtype><devno><regno>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

## 4.6 _4FM_write_creg (deprecated)

Writes a value to the specified custom register of the specified card.

**Usage**

```
_4FM_write_creg <devtype><devno><regno> <value>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

## 4.7 _4FM_read_anyreg

Reads the specified register of the specified card. This function reads a register directly mapped to the PCI bus address range.

**Usage**

```
_4FM_read_anyreg <devtype><devno><regno>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

## 4.8 _4FM_write_anyreg

Writes a value to the specified register of the specified card. This function writes directly to a register mapped to the PCI bus address range.

**Warning**: This operation can possibly crash your machine! Use with caution.

**Usage**

```
_4FM_write_anyreg <devtype><devno><regno> <value>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

## 4.9 _4FM_reset

Resets the specified card.

**Usage**

```
_4FM_reset <devtype><devno>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

## 4.10 _4FM_upload_firmware

Upload a firmware configuration to the FPGA Device A or Device B of the specified card. Can optionally be written to flash or direct to FPGA. FM680 firmware does not support direct FPGA uploading. FC6301 and VP680 devices do not have a FPGA B!

**Usage**

```
_4FM_upload_firmware <devtype><devno><-a|-b[,direct]> <file>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

**Example**

To configure Device B without writing to the flash of the first FM482 in the system.

```
_4FM_upload_firmware FM482 0 -b,direct "DeviceBconfig.hex"
```

To configure Device A and the flash of the first FM482 in the system.

```
_4FM_upload_firmware FM482 0 –a "DeviceAConfig.hex"
```

### 4.11 _4FM_upload_safety_firmware

IMPORTANT: Use only under instructions from 4DSP.

**Usage**

```
_4FM_upload_safety_firmware <devtype><devno><file>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

### 4.12 _4FM_write_user_rom

Writes a value – usually representing a version number - to the flash.

**Usage**

```
_4FM_write_user_rom <devtype><devno><value>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

### 4.13 _4FM_read_user_rom

Reads the user ROM value from the flash. This value can also be retrieved with _4FM_get_information.

**Usage**

```
_4FM_read_user_rom <devtype><devno>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

### 4.14 _4FM_read_sysreg

Read the specified system register of the specified card. Only present in linux board support packages. This function reads a register mapped to the command wormhole address range.

**Usage**

```
_4FM_read_sysreg <devtype><devno><regno>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

### 4.15 _4FM_write_sysreg

Write a value to the specified system register of the specified card. Only present in linux board support packages. This function writes to a register mapped to the command wormhole address range.

**Usage**

```
_4FM_write_sysreg <devtype><devno><regno> <value>
devtype = "FM480", "FM482", "FM485", "FM486", "FM489", "FM577", "FM680" and
"FC6301", "VP680"
```

# 5 Annex 1 - Structures

```
#include <4FM.h>
```

## 5.1 _4FMDiagnostics Struct Reference

_4FMDiagnostics

Data Fields

float mgtvccaux;
float mgtavtt;
float mgtavcc;
float vccblast1;
float vccblast2;
float voltageADJ;
float currentV120;
float vccblast3;
float vioblast1;
float vioblast2;
float vioblast3;
float vttblast1;
float tempref A;
float tempref B;
float voltage120;
float voltageM120;
float voltageIOPN4;
float voltage50;
float voltage25;
float voltageMGT1V0;
float voltageMGT1V2;
float voltage10;
float voltage33;
float voltage18;
float voltage12;
float voltageVref;
float voltageFP;
float tempB;
float tempA;
float voltage15;
float voltageS6vccint;

## 5.2 _4FMInformation Struct Reference

_4FMInformation

Data Fields

struct {
  unsigned **hi**
  unsigned **lo**

unsigned **cust**
unsigned **pci**
unsigned **type**
unsigned **ext**
} **fpga_A_revision**

struct {
unsigned **hi**
unsigned **lo**
unsigned **cust**
} **fpga_B_revision**
struct {
unsigned **hi**
unsigned **lo**
} **cpld_revision**
unsigned **board_serial**
unsigned **bankQDR**
unsigned **fpga_b_firmware_type**
unsigned **fpga_b_guicompatible**
unsigned **user_value**
char **fpga_B_device_type** [32]

# 6 Annex 2 - API functions

This chapter lists the functions part of the 4FM API.

## 6.1 _4FM_CloseDevice

_4FM_error_t _4FM_CALL _4FM_CloseDevice (_4FM_DeviceContext * *ctx*)

Close an 4FM family device.

Parameters:
*ctx* the device context to operate on.

Returns:
_4FM_OK on success.

## 6.2 _4FM_DeviceSupported

int _4FM_CALL _4FM_DeviceSupported (const char * *type*)

See if this library supports a particular device type.

Parameters:
*type* the type of device to check for.

Returns:
non-zero if supported, zero otherwise.

## 6.3 _4FM_GetClockFrequency

_4FM_error_t _4FM_GetClockFrequency (_4FM_DeviceContext * *ctx*, unsigned *clockSelect*, double * *clockFreqMHz*)

Get the clock frequency of a particular clock.

Parameters:
*ctx* the device context to operate on.
*clockSelect* the clock to get the frequency from.
*clockFreqMHz* pointer to a doulbe that will contain the clock frequency in MHz on success.

Returns:
_4FM_OK on success.

### 6.4 _4FM_GetCurrentTimeout

_4FM_error_t _4FM_CALL _4FM_GetCurrentTimeout (_4FM_DeviceContext * *ctx*, unsigned int * *time*)

Get the current timeout. This can either equal the default timeout or the timeout set by _4FM_SetTimeoutOnce.

Parameters:

*ctx* the device context to operate on.

*time* pointer to the variable that will contain the current timeout in ms on success.

Returns:

_4FM_OK on success.

### 6.5 _4FM_GetDiagnostics

_4FM_error_t _4FM_CALL _4FM_GetDiagnostics (_4FM_DeviceContext * *ctx*, _4FMDiagnostics * *diagnostics*)

Get diagnostics for this board.

Parameters:

*ctx* the device context to operate on.

*diagnostics* pointer to a **_4FMDiagnostics** structure that will be populated on success.

Returns:

_4FM_OK on success.

See also:

**_4FMDiagnostics**.

## 6.6 _4FM_GetDriverVersion

_4FM_error_t _4FM_CALL _4FM_GetDriverVersion (_4FM_DeviceContext * *ctx*, char * *buffer*, int *size*)

Get the driver version.

Parameters:

*ctx* the device context to operate on.

*buffer* buffer in which the version string will be stored.

*size* the size of the buffer in bytes.

Returns:

_4FM_OK on success.

Remarks:

Library version will typically comprise 3 characters except for customized versions. It is recommended to set size to 16.

```
char version[16];
_4FM_DeviceContext ctx;
_4FM_OpenDevice(&ctx, "FM482", 0); // Error check omitted.
_4FM_error_t rc = _4FM_GetDriverVersion(&ctx, sizeof(version));
```

## 6.7 _4FM_GetInformation

_4FM_error_t _4FM_CALL _4FM_GetInformation (_4FM_DeviceContext * *ctx*, _4FMInformation * *information*)

Get information about this board.

Parameters:

*ctx* the device context to operate on.

*information* pointer to an **_4FMInformation** structure that will be populated on success.

Returns:

_4FM_OK on success. /see **_4FMInformation**.

## 6.8 _4FM_GetLibraryVersion

_4FM_error_t _4FM_CALL _4FM_GetLibraryVersion (char * *buffer*, int *size*)

Get the library version.

Parameters:

*buffer* buffer in which the version string will be stored.

*size* the size of the buffer in bytes.

Returns:

_4FM_OK on success.

Remarks:

Library version will typically comprise 3 characters except for customized versions. It is recommended to set size to 16.

char version[16];

_4FM_error_t rc = _4FM_GetLibraryVersion(version, sizeof(version));

## 6.9 _4FM_GetTransferTimeout

_4FM_error_t _4FM_CALL _4FM_GetTransferTimeout (_4FM_DeviceContext * *ctx*, unsigned int * *time*)

Get the transfer timeout.

Parameters:

*ctx* the device context to operate on.

*time* pointer to the variable that will contain the current timeout in ms on success.

Returns:

_4FM_OK on success.

### 6.10 _4FM_OpenDevice

_4FM_error_t _4FM_CALL _4FM_OpenDevice (_4FM_DeviceContext * *ctx*, const char * *type*, int *devno*)

Open an 4FM family device.

Parameters:

*ctx* the device context to operate on.

*type* the type name of the device to open.

*devno* the device number to open for the device specified by type.

Returns:

_4FM_OK on success.

Remarks:

Gets a handle from the OS that can be used for communicating with the 4FM board. The parameter devno refers to the board that we wish to obtain the handle for. The first board is '0' and typically this is the board nearest the CPU. Pass '1' in order to get the handle to the second board and so forth.

### 6.11 _4FM_OpenDeviceEx

_4FM_error_t _4FM_CALL _4FM_OpenDeviceEx(_4FM_DeviceContext *ctx, const char *type, int *devno, enum _4FM_OpenModes mode);

Extended _4FM_OpenDevice function. This function provides the user with legacy operations as well as a few new operation modes.

Parameters:

*ctx* the device context to operate on.

*type* pointer to a string, the type name of the device to open.

*devno* pointer a device number to open for the device specified by type.

*mode* decide which opening mode the function should use.

Mode descriptions:

- OPEN_MODE_COMPATIBILITY . This mode transparently calls _4FM_OpenDevice. In this mode *ctx*, *type* and *devno* cannot be NULL. All the arguments are mandatory. Please look at _4FM_OpenDevice for more information.
- OPEN_MODE_FIRST_DEVICE_FOUND. This mode opens the first device found in the system. In this mode *ctx* cannot be null. *devno* and *type* arguments are optional. If a valid pointer (not NULL, pointing to previously allocated memory) is passed as argument then the function provide the caller with the device type as well as device number. The maximum device type returned by the function is MAX_DEVTYPE_LEN.
- OPEN_MODE_FIRST_TYPE_FOUND This mode opens the first device found of a given type in the system. In this mode *ctx* and *type* cannot be null. *devno* argument is optional. If a valid pointer (not NULL, pointing to previously allocated memory) is passed as argument then the function provides the caller with the device number.

Returns:

_4FM_OK on success.

Remarks:

Gets a handle from the OS that can be used for communicating with the 4FM board. The parameter devno refers to the board that we wish to obtain the handle for. The first board is '0' and typically this is the board nearest the CPU. Pass '1' in order to get the handle to the second board and so forth.

### 6.12 _4FM_ReadCustomRegister

_4FM_error_t _4FM_CALL _4FM_ReadCustomRegister (_4FM_DeviceContext * *ctx*, unsigned long *regno*, unsigned long * *value*)

Note: Only available on old firmware (FM482, FM577, ... )

Read from a custom register on an 4FM family device.

Parameters:
*ctx* the device context to operate on.
*regno* register number to read from.
*value* pointer to the memory where the result should be stored.

Returns:
_4FM_OK on success.

### 6.13 _4FM_ReadMailbox

_4FM_error_t _4FM_CALL _4FM_ReadMailbox (_4FM_DeviceContext * *ctx*, unsigned long * *value*, unsigned long *timeout*)

Read from an 4FM family device's mailbox.

Parameters:
*ctx* the device context to operate on.
*value* to the memory location to store the value.
*timeout* timeout in milliseconds.

Returns:
_4FM_OK on success.

### 6.14 _4FM_ReadUserROM

_4FM_error_t _4FM_CALL _4FM_ReadUserROM (_4FM_DeviceContext * *ctx*, unsigned int * *value*)

Read the user rom register.

Parameters:

*ctx* the device context to operate on.

*value* pointer to the variable that will contain the user rom value on success.

Returns:

_4FM_OK on success.

### 6.15 _4FM_ReceiveData

_4FM_error_t _4FM_CALL _4FM_ReceiveData (_4FM_DeviceContext * *ctx*, void * *buffer*, unsigned long *count*)

Receive data from an 4FM family device.

Parameters:

*ctx* the device context to operate on.

*buffer* pointer to the memory where to store the data.

*count* number of bytes to receive.

Returns:

_4FM_OK on success.

Remarks:

Performs a DMA transfer. Count must be a multiple of 32 bytes (128 for FM680, FC6301 and VP680 devices). The data must be aligned to 4KB.

### 6.16 _4FM_ResetDevice

_4FM_error_t _4FM_CALL _4FM_ResetDevice (_4FM_DeviceContext * *ctx*)

Reset an 4FM family device.

Parameters:

*ctx* the device context to operate on.

Returns:

_4FM_OK on success.

### 6.17 _4FM_ResetDriverQueues

_4FM_error_t _4FM_CALL _4FM_ResetDriverQueues(_4FM_DeviceContext *ctx);

This function reset the software queues in the device driver. This API function does not interact with the firmware but only with the driver. This function is not required as the software queues are reset implicitly by _4FM_ResetDevice. However in some circumstance it might be required to empty/reset the software queues. A good example would be a link resynchronization in case if a major host<>firmware communication issue.

Parameters:

*ctx* the device context to operate on.

Returns:

_4FM_OK on success.

### 6.18 _4FM_SelectTarget

_4FM_error_t _4FM_CALL _4FM_SelectTarget (_4FM_DeviceContext * *ctx*, unsigned long *target*)

Select the data transfer target on an 4FM family device.

Parameters:

*ctx* the device context to operate on.

*target* target to send to or receive from.

Returns:

_4FM_OK on success.

## 6.19 _4FM_SendData

_4FM_error_t _4FM_CALL _4FM_SendData (_4FM_DeviceContext * *ctx*, const void * *buffer*, unsigned long *count*)

Send data to an 4FM family device.

Parameters:

*ctx* the device context to operate on.

*buffer* pointer to the memory that contains the data.

*count* number of bytes to send.

Returns:

_4FM_OK on success.

Remarks:

Performs a DMA transfer. Count must be a multiple of 32 bytes (128 for FM680, FC6301 and VP680 devices). The data must be aligned to 4KB.

## 6.20 _4FM_SetAPIOptions

_4FM_error_t _4FM_CALL _4FM_SetAPIOptions(unsigned int optionidx, unsigned int optionval);

Set an API option. Most of options are not documented and reserved for 4DSP internal use. A set of options is available for the user and can be changed using this function.

**Valid option indexes (defined in 4FM.h):**

*OPTION_OFFSET_TEMPERATURE*

Configure an offset to be added (if the offset is positive) or subtracted (if the offset is negative). This can be change to compensate for physical differences between the external diode measurement on the FPGA and the chipscope measurement. If the user finds a difference of -10, then _4FM_SetAPIOptions() should be called with -10 as second argument.

Parameters:

*optionidx* index to the option we are trying to modify

*optionval value to configure the option with*.

Returns:

_4FM_OK on success.

### 6.21 _4FM_SetClockSynth

_4FM_error_t _4FM_SetClockSynth (_4FM_DeviceContext * *ctx*, unsigned *M*, unsigned *N*)

Set the clock synthesizer.

Parameters:
*ctx* the device context to operate on.
*M* the multiplier value (250-500).
*N* the divider value (0-7)

### 6.22 _4FM_SetCDCEParameters

_4FM_error_t _4FM_CALL _4FM_SetCDCEParameters(_4FM_DeviceContext *ctx, unsigned M, unsigned N);

Configure the clock synthesizer (CDCE925) for a given M and N parameters. The actual frequency is dictated as the following: fSynth[MHz] = M/N.

A valid fSynth range for the CDCE925 is between 81[Mhz] to 230[Mhz].

Note: This function first sets the synthesizer frequency and then copies the parameters into the chip's EEPROM memory.

### 6.23 _4FM_SetDDRAMOffset

_4FM_error_t _4FM_CALL _4FM_SetDDRAMOffset (_4FM_DeviceContext * *ctx*, unsigned long *offset*)

Set the DDR offset.

Note : This function only applies to FM482 with DDR2 memory attached on FPGA A

Parameters:

*ctx* the device context to operate on.

*offset* the offset in bytes in the DDR. The data will be read or written from or to this location.

Returns:

_4FM_OK on success.

Remarks:

The offset needs to be a multiple of 32 bytes. The address used in _4FM_SendData, _4FM_SendDataA, _4FM_ReceiveData and _4FM_ReceiveDataA must be 32 byte aligned when data is transferred to or from the DDR.

### 6.24 _4FM_SetTimeoutOnce

_4FM_error_t _4FM_CALL _4FM_SetTimeoutOnce (_4FM_DeviceContext * *ctx*, unsigned int *time*)

Set the timeout for the next transfer only.

Parameters:

*ctx* the device context to operate on.

*time* the timeout in ms.

Returns:

_4FM_OK on success.

### 6.25 _4FM_SetTransferTimeout

_4FM_error_t _4FM_CALL _4FM_SetTransferTimeout (_4FM_DeviceContext * *ctx*, unsigned int *time*)

Set the transfer timeout.

Parameters:
*ctx* the device context to operate on.
*time* the timeout in ms.

Returns:
_4FM_OK on success.

### 6.26 _4FM_UploadFirmware

_4FM_error_t _4FM_CALL _4FM_UploadFirmware (_4FM_DeviceContext * *ctx*, const char * *filename*, enum ufTarget *target*)

Upload firmware to the board.

Parameters:
*ctx* the device context to operate on.
*filename* name of the image file.
*target* the target on the board that will receive the image.

Returns:
_4FM_OK on success.

### 6.27 _4FM_WriteCustomRegister

_4FM_error_t _4FM_CALL _4FM_WriteCustomRegister (_4FM_DeviceContext * *ctx*, unsigned long *regno*, unsigned long *value*)

Write to a custom register on an 4FM family device.

Note: Only available on old firmware (FM482, FM577, ... )

Parameters:
*ctx* the device context to operate on.
*regno* register number to write to.
*value* to write.
Returns:
_4FM_OK on success.

### 6.28 _4FM_WriteMailbox

_4FM_error_t _4FM_CALL _4FM_WriteMailbox (_4FM_DeviceContext * *ctx*, unsigned long *value*)

Write to an 4FM family device's mailbox.

Parameters:
*ctx* the device context to operate on.
*value* to write.

Returns:
_4FM_OK on success.

### 6.29 _4FM_WriteUserROM

_4FM_error_t _4FM_CALL _4FM_WriteUserROM (_4FM_DeviceContext * *ctx*, unsigned *value*)

Write the user rom register.

Parameters:
*ctx* the device context to operate on.
*value* the value to write.

Returns:
_4FM_OK on success.

### 6.30 _4FM_WriteBurst (FM577 ONLY) Remove from list, only for internal usage

_4FM_error_t _4FM_CALL _4FM_WriteBurst (_4FM_DeviceContext * *ctx*, unsigned char **buf*, unsigned long *size*, unsigned long *addr*)

Write to an 4FM family memory space.

Parameters:
*ctx* the device context to operate on.
*buf* pointer to a buffer containing data to be sent to a card.
*size* size of the data to be sent to the card (in bytes).
*addr* address to write to on the card (!!! Need to be a multiple of 4 bytes).

Returns:
_4FM_OK on success.

### 6.31 _4FM_ReadBurst (FM577 ONLY)

_4FM_error_t _4FM_CALL _4FM_ReadBurst (_4FM_DeviceContext * *ctx*, unsigned char **buf*, unsigned long *size*, unsigned long *addr*)

Remove from list, only for internal usage

Read from an 4FM family memory space.

Parameters:

*ctx* the device context to operate on.

*buf* pointer to a buffer where data has to read from the card.

*size* size of the data to be read from the card (in bytes).

*addr* address to read to on the card (!!! Need to be a multiple of 4 bytes).

Returns:

_4FM_OK on success.

### 6.32 _4FM_Read

_4FM_error_t _4FM_CALL _4FM_Read(_4FM_DeviceContext *ctx, unsigned long addr, unsigned long *value, unsigned long timeout);

Read a 32 bit value from an address in command wormhole address space. A specific firmware is required to use the function.

Parameters:

*ctx* the device context to operate on.

*addr* 28 bits address value corresponding to a 32 bits word address.

*value* pointer to a 32 bit variable. This variable receives the value at *addr*.

*timeout* the maximum time the function wait in millisecond before to time out.

Returns:

_4FM_OK on success or any other error in case of an error has been encountered.

### 6.33 _4FM_Write

_4FM_error_t _4FM_CALL _4FM_Write(_4FM_DeviceContext *ctx, unsigned long addr, unsigned long value, unsigned long ack);

Write a 32 bit value at a given address in the command wormhole address space. A specific firmware is required to use this function.

Parameters:

*ctx* the device context to operate on.

*addr* 28 bits address value corresponding to a 32 bits word address.

*value* a 32 bit variable. This value is written at *addr*.

*ack* tells the function to wait for a "Write Ack". The timeout is hardcoded to two seconds in this case