

USB-MPC

with I2C and MDIO support

Software Developer's User Manual



Information in this document is provided solely to enable the use of Future Designs, Inc. products. FDI assumes no liability whatsoever, including infringement of any patent or copyright. FDI reserves the right to make changes to these specifications at any time, without notice. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Future Designs, Inc. 2702 Triana Blvd SW, Huntsville, AL 35805-4074.

© 2005 Future Designs, Inc. All rights reserved.

Microsoft, MS-DOS, Windows, Microsoft Word are registered trademarks of Microsoft Corporation. Other brand names are trademarks or registered trademarks of their respective owners.

P:\PC Products\USB-MPC\SW\Docs\USB-MPC Software Developer's User Manual.doc

Table of Contents

1.	USB-MPC Overview	1
2.	USB-MPC Installation	2
3.	Using the USB-MPC in a VC++ Project	3
4.	Two Example VC++ Projects	4
5.	General Purpose Routines.....	5
5.1.	SetupHardware.....	5
5.2.	DetectHardware	5
6.	I2C Routines	6
6.1.	I2cGenerateStartCondition	6
6.2.	I2cGenerateRepeatedStartCondition.....	6
6.3.	I2cGenerateStopCondition.....	6
6.4.	I2cWriteByte.....	6
6.5.	I2cReadByte.....	6
6.6.	I2cWriteDevice	7
6.7.	I2cReadDevice.....	7
6.8.	I2cReadMemory	8
6.9.	I2cWriteMemory.....	8
6.10.	I2cReadMemory16	9
6.11.	I2cWriteMemory16.....	9
7.	MDIO Routines.....	11
7.1.	Mdio22ReadWord.....	11
7.2.	Mdio22WriteWord.....	11
8.	System Definitions.....	12
8.1.	I2C Function Return Values	12
8.2.	MDIO Function Return Values.....	12

1. USB-MPC Overview

The USB-MPC Software Developer Kit contains all of the tools necessary to access and control the USB-MPC hardware from a custom application. The USB-MPC can be used with Visual C/C++ or any other programming language that supports the use of DLLs. (Note that the USB-MPC was generated and tested using Microsoft Visual C/C++ 6.0, but there is no reason that it will not work in all Microsoft Windows software development environments such as Borland C/C++, Microsoft Visual Studio, or any programming software which utilizes DLL.).

The USB-MPC consists of the following:

- An FDI Installation and Support CDROM
- USB-MPC Unit
- 4-pin connecting cable (18" length)
- Registration form

2. USB-MPC Installation

NOTE: Do not plug in the USB-MPC until you have installed the software!

Insert the USB-MPC Installation CDROM into the proper drive. If the setup program does not automatically run, open the USB-MPC Installation CDROM from “My Computer” and run “setup.exe” by double-clicking on it. Follow the instructions on the subsequent screens.

The Setup.exe program installs the files needed for the standard operation of the USB-MPC and those needed to add USB-MPC functionality to a custom application. The required .DLL and .SYS files will be copied to the appropriate system directories and the required modifications to the registry will be made. The remaining files will be copied into the following subdirectories (Assuming that the default directories were used during the installation procedure).

Directory (C:\Program Files\FDI\...)	Usage:
\USB-MPC	Standard USB-MPC application and help files.
\USB-MPC\DDF	Device Descriptor Files (DDF) used by the standard USB-MPC software. See the USB-MPC software on-line help for details.
\USB-MPC\SDF	Sequence Descriptor Files (DDF) used by the standard USB-MPC software. See the USB-MPC software on-line help for details.
\USB-MPC\Dev	Files that must be used by your custom application to access the USB-MPC hardware.
\USB-MPC\Driver	Files used for installing USB-MPC drivers
\USB-MPC \Examples	Microsoft VC++ example projects for I2C and MDIO.

3. Using the USB-MPC in a VC++ Project

In order to add the functionality of the USB-MPC to a custom application, you must perform the following steps. Note that you can use the example project located in the DLL_Example subdirectory as a reference for implementing these steps.

Copy the following files from the \Dev subdirectory (C:\Program Files\FDI\USB-MPC\Dev if the default installation location was used) to the directory that contains the source code for your custom application.

```
USBMPC.h  
USBMPC.lib
```

In the VC++ environment, add the .lib files to your project by opening the “Project” menu, selecting “Add to Project”, and clicking on “Files”. Select the following file from the dialog box and click “OK”.

```
USBMPC.lib
```

Add the following line to each of the files in your project that will access the USB-MPC software:

```
#include "USBMPC.h"
```

During your application’s initialization, add the following to setup communications to the USB-MPC:

```
SetupHardware(0, USBMPC_I2C_SPEED_100KHZ, 0);
```

4. Two Example VC++ Projects

The subdirectory “Examples” contains two example projects, one for I2C and one for MDIO.

The subdirectory “Examples\USBMPC_EEPROM” contains code to use I2C to read and write to a I2C EEPROM using 3 different methods (high, mid, and low level). A PCF8582C EEPROM was used in this project.

The subdirectory “Examples\USBMPC_MDIO” contains an example project that reads all 32 registers, increments one of them, writes it back, and then reads the 32 registers again. The output shows each step of the process.

5. General Purpose Routines

These routines are used to initialize, setup, and terminate the USB-MPC DLL functionality. They are usually called only once by the custom application.

5.1. SetupHardware

Prototype: int SetupHardware(
 int nReserved,
 int nSpeed,
 int nPort)

Function: This routine is called to set up the parameters to be used by the USB-MPC hardware. This routine must be executed once before calling the other commands.

Parameters: **nReserved** – Currently always 0.
 nSpeed – Use USBMPC_I2C_SPEED_100KHZ or USBMPC_I2C_SPEED_400KHZ to declare which speed to run the I2C bus.
 nPort – Usually 0 unless more than one USB-MPC device is used. Each device is sequentially numbered.

Returns: Always returns I2C_NO_ERROR.

5.2. DetectHardware

Prototype: int DetectHardware(void)

Function: This routine is called to detect the presence of the USB-MPC hardware. This routine can be called at any time.

Parameters: None

Returns: Returns TRUE if the board was detected and FALSE if it was not detected.

6. I2C Routines

These routines are used to support the I2C bus protocol. There are routines included to perform low level bit manipulation as well as routines to send entire messages across the I2C bus.

6.1. I2cGenerateStartCondition

Prototype: int I2cGenerateStartCondition(void)
Function: This routine generates a start condition on the I2C interface.
Parameters: None
Returns: Always returns I2C_NO_ERROR

6.2. I2cGenerateRepeatedStartCondition

Prototype: int I2cGenerateRepeatedStartCondition(void)
Function: This routine generates a repeated start condition on the I2C interface.
Parameters: None
Returns: Always returns I2C_NO_ERROR

6.3. I2cGenerateStopCondition

Prototype: int I2cGenerateStopCondition(void)
Function: This routine generates a stop condition on the I2C interface.
Parameters: None
Returns: Always returns I2C_NO_ERROR

6.4. I2cWriteByte

Prototype: int I2cWriteByte(
 int nByte)
Function: This function transmits a single byte to the I2C bus. It assumes that the bus is available and the proper Start Condition has previously been generated.
Parameters: **nByte** – The byte to write to the I2C interface.
Returns: Returns I2C_NO_ACK if the slave device does not acknowledge the byte. Otherwise it returns I2C_NO_ERROR.

6.5. I2cReadByte

Prototype: int I2cReadByte(
 int *nByte,
 int nLast)
Function: This function reads a single byte from the I2C bus. It assumes that the bus is available, that the proper Start Condition has previously been generated, and that the slave device has been properly addressed. If the parameter nLast does not have

I2C_READ_BYTE_LAST_READ set, an ACK is generated after the byte is transmitted. Otherwise, no ACK is generated. The result of the read is saved in *nByte.

Parameters: ***nByte** – This is a pointer to the location that receives the byte read from the I2C bus.
nLast – This parameter determines if this is the last byte (I2C_READ_BYTE_LAST_READ) and needs a STOP after it or the second to last byte (I2C_READ_BYTE_SECOND_TO_LAST_READ) to float the SDA line to tell the I2C device to stop sending data.

Returns: Updates the value pointed to by nByte with the byte read from the I2C bus and returns any errors.

6.6. I2cWriteDevice

Prototype: `int I2cWriteDevice(
int nDeviceAddress,
int nCount,
int nBuffer[],
int nRegWidth = 1);`

Function: This function is used to write a complete message to the I2C bus. It handles generation of the Start and Stop Conditions as well as properly addressing the Slave device.

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nCount – The number of words to write to the slave device.
nBuffer[] – A buffer that contains the bytes to write to the slave device.
nRegWidth – The width of each register and thus each word.

Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge any of the bytes that are transmitted. Otherwise returns I2C_NO_ERROR.

6.7. I2cReadDevice

Prototype: `int I2cReadDevice(
int nDeviceAddress,
int nCount,
int nBuffer[],
int nRegWidth = 1);`

Function: This function is used to read a complete message from the I2C bus. It handles generation of the Start and Stop Conditions as well as properly addressing the Slave device. It also generates an ACK for every byte transmitted except for the final one. (This is a common method of terminating a read process on the I2C bus.)

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nCount – The number of words to read from the slave device.

nBuffer[] – A buffer that receives the bytes read from the slave device.
nRegWidth – The width of each register and thus each word.
Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge its address. Otherwise returns I2C_NO_ERROR. It also updates the contents of nBuffer with the read results.

6.8. I2cReadMemory

Prototype: `int I2cReadMemory(
int nDeviceAddress,
int nMemoryAddress,
int nCount,
int nBuffer[],
int nRegWidth = 1);`

Function: This function reads a block of memory from an I2C memory device using 11-bit internal addressing. It handles the generation of the Start and Stop Conditions as well as properly addressing the Slave device. It also handles setting up the proper address to read and the proper ACK sequence for the read procedure.

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nMemoryAddress – The address within the slave device to begin reading.
nCount – The number of words to read from the slave device. (A maximum of 0x100 bytes can be read at a time.)
nBuffer[] – A buffer that receives the bytes read from the slave device.

nRegWidth – The width of each register and thus each word.
Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge its address or the memory address to read. Returns I2C_COUNT_TOO_BIG if a number larger than 0x100 is read in nCount. Otherwise returns I2C_NO_ERROR. It also updates the contents of nBuffer with the read results.

6.9. I2cWriteMemory

Prototype: `int I2cWriteMemory(
int nDeviceAddress,
int nMemoryAddress,
int nCount, int nBuffer[],
int nRegWidth = 1)`

Function: This function writes a block of memory to an I2C memory device using 11-bit internal addressing. It handles the generation of the Start and Stop Conditions as well as properly addressing the Slave device. It also handles setting up the proper address to write.

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.

nMemoryAddress – The address within the slave device to begin writing.

nCount – The number of words to write to the slave device. (A maximum of 0x10 bytes can be written at a time.)

nBuffer[] – A buffer that contains the bytes to write to the slave device.

nRegWidth – The width of each register and thus each word.

Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge any of the bytes written to it. Returns I2C_COUNT_TOO_BIG if a number larger than 0x10 is read in nCount. Otherwise returns I2C_NO_ERROR.

6.10. I2cReadMemory16

Prototype: `int I2cReadMemory16(int nDeviceAddress, int nMemoryAddress, int nCount, int nBuffer[], int nRegWidth = 1)`

Function: This function reads a block of memory from an I2C memory device using 19-bit internal addressing. It handles the generation of the Start and Stop Conditions as well as properly addressing the Slave device. It also handles setting up the proper address to read and the proper ACK sequence for the read procedure.

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nMemoryAddress – The address within the slave device to begin reading.
nCount – The number of words to read from the slave device. (A maximum of 0x10000 bytes can be read at a time.)
nBuffer[] – A buffer that receives the bytes read from the slave device.
nRegWidth – The width of each register and thus each word.

Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge its address or the memory address to read. Returns I2C_COUNT_TOO_BIG if a number larger than 0x10000 is read in nCount. Otherwise returns I2C_NO_ERROR. It also updates the contents of nBuffer with the read results.

6.11. I2cWriteMemory16

Prototype: `int I2cWriteMemory16(int nDeviceAddress, int nMemoryAddress, int nCount, int nBuffer[], int nRegWidth = 1)`

Function: This function writes a block of memory to an I2C memory device using 19-bit internal addressing. It handles the generation of the Start and Stop Conditions as well as properly addressing the Slave device. It also handles setting up the proper address to write.

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nMemoryAddress – The address within the slave device to begin writing.
nCount – The number of words to write to the slave device. (A maximum of 0x10 bytes can be written at a time.)
nBuffer[] – A buffer that contains the bytes to write to the slave device.
nRegWidth – The width of each register and thus each word.

Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge any of the bytes written to it. Returns I2C_COUNT_TOO_BIG if a number larger than 0x10 is read in nCount. Otherwise returns I2C_NO_ERROR.

7. MDIO Routines

7.1. Mdio22ReadWord

Prototype: int Mdio22ReadWord(
 int nPhyAddr,
 int nRegAddr,
 int *nData)

Function: This routine reads a single 32bit word from the MDIO (Clause 22) interface. The resultant word is returned in the variable nData.

Parameters: **nPhyAddr** – This value is the address of the slave device on the MDIO bus.
 nRegAddr – This is the address of the register within the slave device that is to be read.
 nData – This is the location that will receive the results of the read operation.

Returns: Always returns MDIO_NO_ERROR. Also updates the value of nData with the value read from the bus.

7.2. Mdio22WriteWord

Prototype: int Mdio22WriteWord(
 int nPhyAddr,
 int nRegAddr,
 int nData)

Function: This routine writes a single 32bit word to the MDIO (Clause 22) interface.

Parameters: **nPhyAddr** – This value is the address of the slave device on the MDIO bus.
 nRegAddr – This is the address of the register within the slave device that is to be written.
 nData – This is the data that will be written to the slave device.

Returns: Always returns MDIO_NO_ERROR.

8. System Definitions

This section contains miscellaneous system definitions and notes for using the USB-MPC.

8.1. I2C Function Return Values

The following values can be returned by the Developer Kit routines. They are returned by the I2C routines. These values are declared in the USBMPC.h file.

I2C_NO_ERROR – Functions return this value if no error occurs. This is the standard return value.

I2C_NO_ACK – Functions return this value when the slave device fails to acknowledge a byte that was transferred over the bus.

I2C_COUNT_TOO_BIG – Functions return this value when the amount of data to move is too large for the internal buffers.

I2C_NO_HARDWARE – USB-MPC hardware was not found.

I2C_KA_ERROR – An internal error has occurred.

I2C_BUS_ERROR – An error has occurred when trying to manipulate the I2C bus.

8.2. MDIO Function Return Values

The following values can be returned by the Developer Kit routines. They are returned by the MDIO routines. These values are declared in the USBMPC.h file.

MDIO_NO_ERROR – MDIO functions return this value if no error occurs. This is the standard return value.

MDIO_USB_OPEN_FAILURE – An attempt to open communications to the USB-MPC device failed.

MDIO_USB_COMM_FAILURE – Communications to the USB-MPC device failed.