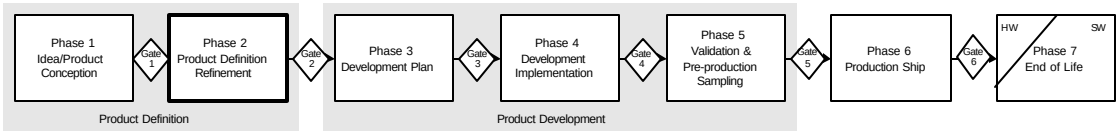
3dfx Interactive

Modularized Reference Implementation Document for I²C



0 Contents

0.1 Revision History

Revision	Description	Date
0.1	--Began initial draft.	990615
0.2	--Finished initial draft, and distributed for comments	990628
0.3	--Corrected various errata, modified InstantiateBusI2C, and added AssociateBusToMuxI2C and I2CBUS.	990629
0.4	--More errata, added a code conceptualization diagram, renamed i2c_releaseaccess to i2c_endaccess, and added an “Issues” section.	990629
0.5	--Changed InstantiateBusI2C, InstantiateMuxI2C, and AssociateBusToMuxI2C to use function pointers, to improve hardware independency. --Added I2C_PEDANTIC and related --Removed I2C_INPORTB, I2C_OUTPORTB --Effectively removed the “Device Specific Data Structures” section, since this document is targeted towards MRI clients. --Added lots of new optional defines (section 3.2) --Added I2C_ALLOCMEM --Removed InitializeI2C() and added i2c_initialize()	990630
0.6	--REALLY removed InitializeI2C() this time (no, really!) --Added the cool 3dfx product cycle timeline diagram to the front page	990630
0.99	--Removed I2C_IBMPCMACHINE --Updated some function definitions that changed during the course of implementation --Fixed various and sundry errata	990729
1.0	--Renamed “i2cconfig.h” to “i2cmacro.h”, and added “i2csetup.h” --Removed I2C_ALLOCMEM --Removed I2C_MICROSOFT --Removed section 2.12 (WDM interface) --Added I2CCONTEXT parameter to all applicable function prototypes --Updated section 3	991220

0.2 Issues

▪

1 Introduction

This document is written for software engineers who need to utilize this I²C MRI in their current project.

1.1 What is an MRI?

An MRI (Modularized Reference Implementation) is a piece of code that the author attempted to write in a sufficiently generic and robust manner, so as to maximize its portability and reusability across products and operating systems. By designing an implementation that can be used over and over again, developers can free up more of their future time to work on different, more interesting tasks.

1.2 Why an I²C MRI?

The impetus behind this I²C MRI effort is really secondary in nature. Our primary motive is to present a DDC MRI, for use in all of our display drivers. But since DDC is built on top of I²C, it became necessary to do this first. But beyond this motive, there exist many other important reasons why an I²C MRI is beneficial.

Many components and feature sets of our products rely upon the ability to communicate with ancillary, on-board devices using the industry standard I²C protocol. Examples include:

- Multimedia applications
- Monitor DDC communications
- Serial EEPROM access for changing boot-up configuration

History and experience has shown that many of these devices are very sensitive to timing issues. As such, it is important that we adhere to the timing requirements outlined in the I²C specification, and provide a mechanism to slow down gracefully, if a particular device proves to be sluggish. This requires our I²C implementation to have timing characteristics that are independent of CPU and system speed. Additionally, since I²C is inherently slow, it is important to yield wait times back to the scheduler of the operating system whenever possible, to avoid system stalls that are common with I²C implementations that "spin" while waiting on an event. *The more responsive a user's system is, the happier the user will be.*

Managing these types of issues across multiple products and multiple OSes has proven difficult, because each project's source base has had it's own implementation of I²C support—and sometimes even more than one. The goal of this project is to design and develop an I²C implementation that:

- Is reusable across products
- Is portable across operating systems
- Arbitrates access to each bus for multiple clients
- Exists in a centralized place in source control, that is shared by all products

This last feature affords us the ability to upgrade or fix the source once, and have all products benefit immediately. *Write once, reuse often.*

2 Functional Overview

Frequently, graphics hardware has the ability to control one or more I²C buses. Each bus can have its own, dedicated GPIO pins (as is the case with I²C bus #0, in the diagram below,) or they can share their connection with other buses, via an external mux. More than one I²C device may be attached to any given bus, and two or more buses may be attached to any given mux.

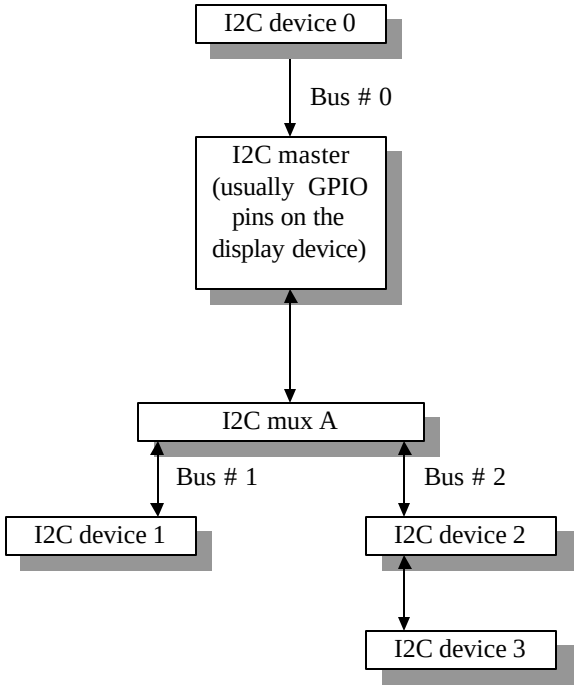


Figure 1, Conceptualization of Hardware

There are three sets of functions in the I²C MRI. Function names that follow the format "XxxxYyyyyI2C()" are used only for configuration and initialization of the I²C MRI code, at system init time. Function names that follow the "i2c_xxxyyyy()" naming convention are used when accessing the I²C buses. Finally, Function names that follow the "I2C_XXXYYYY" naming convention are used to denote functions that provide operating-system-specific and/or hardware-specific services.

A resource management problem arises when you have two or more I²C buses that are "joined at the hip" by a mux; you cannot allow both buses to be used simultaneously. In the example given in **figure 1**, Bus #1 and Bus #2 are mutually exclusive, due to their reliance on I²C mux A. This means that the code must provide resource management that is board specific, in addition to providing access to the registers that control the I²C hardware. This I²C MRI is designed with these issues in mind, and figure 2 gives a good visualization.

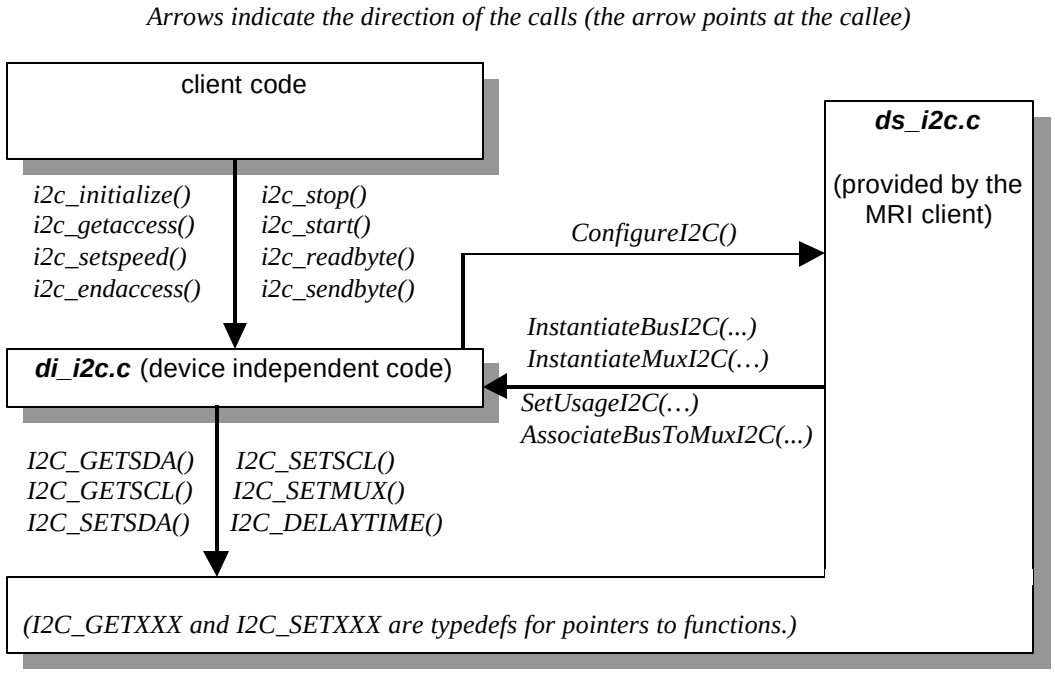


Figure 2, Conceptualization of Code

In the example above, ***di_i2c.c*** provides resource management in addition to implementing the I²C protocol. But to do this, it must know which I²C buses are exclusive to others, and how. This is the point of the *AssociateBusToMux()* calls that the client-supplied ***ds_i2c.c*** file makes.

2.1 Initializing the I²C MRI Code

Before it does anything else I²C-relate, the client code must call *i2c_initialize()*.

```
int i2c_initialize(I2CCONTEXT pContext);
```

This instructs the I²C MRI code to initialize its internal data structures, and to call the *ConfigureI2C()* function, located in the ***ds_i2c.c*** file.

This function returns **I2C_SUCCESS** or **I2C_FAILURE**. If it fails, no I²C support will be available.

The ***pContext*** parameter is a generic handle. Exactly what it is will vary of OS to OS, and will be defined by the particular I2C MRI being used. All that is required is for this value to map 1-to-1 with a data structure associated with particular I2C host device, and for that data structure to contain (possibly in addition to other, non-related things,) both a set of state information for each I2C bus, and a set of register base addresses for each I2C bus. The I2C MRI will use the os-specific **I2C_DEVINFOPTR** macro to obtain the I2C state information, using this generic handle.

Exempli gratia: In the case of Microsoft operating systems, **I2C_DEVINFOPTR** expects it to be the address of the current DEVTABLE structure, which contains both the register base address(es) of the I2C control registers, and the i2cinfo structure that contains the state information.

All of the i2c_xxxyyyyy() and XxxYyyyyI2C() functions require this parameter, and assume it is valid. If pContext is invalid, the results are undefined, and most likely undesireable.

2.2 Obtaining a handle to an I²C bus

Ownership of an I²C bus can be obtained by calling *i2c_getaccess()*.

```
I2CKEY i2c_getaccess(I2CCONTEXT pContext, FxU32 usage, FxU8 speed);

I2CKEY tvout;
I2CCONTEXT pContext;
tvout = i2c_getaccess(pContext, I2C_TVENCODER, I2C_NORMALSPEED);
```

This function gives the owner of the handle (also called the "key") exclusive rights to the I²C bus in question. If this I²C bus is attached to a mux, then all other buses attached to that same mux will be unavailable while this bus is locked. Likewise, this function will refuse to assign ownership if this bus or another bus on the same mux is already in use. The key returned by this function serves two purposes; it simultaneously acts as a handle identifying which bus you are referring to, and as a unique lock that guarantees you exclusive access to the I²C bus in question.

The ***pContext*** parameter has the usual meaning.

The ***usage*** parameter indicates which I²C bus you are interested in. If a bus that supports the given usage is present, and is available for use, *i2c_getaccess()* will return a handle for that I²C bus. Otherwise, the value returned is **I2C_NOTAKEY**.

The ***speed*** parameter allows the code to slow down the I²C activity. At full speed, the I²C code is timed to operate at between 90 and 95% of the 100 kHz maximum allowed by the I²C specification. This parameter can be useful for I²C devices that have difficulty operating at that speed. This parameter is interpreted as a delay multiplier (i.e., the lower the number, the faster the bit rate.) A value of zero is understood to mean the same thing as a value of 1.

You should be aware that the actual maximum bus speed will be closer to 90 or 95 kHz, due to both overhead in the code, and built in "fudge factors" that ensure adherence to the I²C specification. This margin reduces in percentage for lower specified bus frequencies.

The value **I2C_NORMALSPEED** equates to the value 0, which is a synonym for 1, or full speed. The given speed will remain in effect until ownership of the bus is relinquished, or the function *i2c_setspeed()* is called. The I²C MRI may arbitrarily impose a practical limitation on the slowest allowed bus speed, to protect system performance.

2.3 Changing the I²C bus speed

If you find yourself needing to change the bus speed on a bus your code already has ownership of, you do not have to relinquish ownership to obtain a new handle with the speed set differently. You can change the bus speed on the fly by calling *i2c_setspeed()*.

```
void i2c_setspeed(I2CCONTEXT pContext, I2CKEY key, FxU8 speed);

i2c_setspeed(pContext, tvout, 2);  /* reduce bus speed to half of maximum */
```

The ***pContext*** and ***key*** parameters have their usual meaning.

The ***speed*** parameter has the exact meaning that it has in *i2c_getaccess()*.

If ***key*** is invalid, this function does nothing. If ***pContext*** is invalid, the results are undefined.

2.4 Relinquishing ownership of an I²C bus

Because a bus (and all other buses related to it via a mux) becomes unusable for other threads when owned, your code should release bus ownership when it is finished with the immediate task at hand. Your code should not obtain a bus handle, and stow it away for future use. The *i2c_endaccess()* function is provided for this purpose

```
void i2c_endaccess(I2CCONTEXT pContext, I2CKEY key);  /* "key" must be a valid I2CKEY */

/*
 * the following relinquishes control  of the I2C bus.
 * (this example assumes tvout is a valid I2CKEY, and that pContext is a valid I2CCONTEXT*
 */
i2c_endaccess(pContext, tvout);
```

The ***pContext*** and ***key*** parameters have their usual meaning.

If ***key*** is invalid, this function does nothing. If ***pContext*** is invalid, the results are undefined.

2.5 Generating a Start Condition

The *i2c_start()* function is used to generate a start condition on a specified I²C bus.

```
int i2c_start(I2CCONTEXT pContext,  I2CKEY key);
```

This function returns **I2C_SUCCESS** or **I2C_FAILURE**. Possible reasons for failure:

- Invalid **I2CKEY**
- Unable to generate start condition (possible bus contention)

If ***key*** is invalid, this function returns **I2C_FAILURE**. If ***pContext*** is invalid, the results are undefined.

2.6 Generating a Stop Condition

The *i2c_stop()* function is used to generate a stop condition on a specified I²C bus.

```
int i2c_stop(I2CCONTEXT pContext, I2CKEY key);
```

This function returns **I2C_SUCCESS** or **I2C_FAILURE**. Possible reasons for failure:

- Invalid **I2CKEY**
- Bus contention

If ***key*** is invalid, this function returns **I2C_FAILURE**. If ***pContext*** is invalid, the results are undefined.

2.7 Sending a Byte on the I²C bus

The *i2c_sendbyte()* function is used to send a byte across a specified I²C bus.

```
int i2c_sendbyte(I2CCONTEXT pContext, I2CKEY key, FxU8 byte);
```

This function returns **I2C_SUCCESS** or **I2C_FAILURE**. Possible reasons for failure:

- Invalid **I2CKEY**
- Device did not respond (failure to ACKnowledge)
- Bus contention

If ***key*** is invalid, this function returns **I2C_FAILURE**. If ***pContext*** is invalid, the results are undefined.

2.8 Sending a Series of Bytes on the I²C bus

The *i2c_sendbytes()* function is used to send a series of bytes on a specified I²C bus.

```
int i2c_sendbytes(I2CCONTEXT pContext, I2CKEY key, FxU16 numbytes, FxU8* pbytes);
```

This function returns **I2C_SUCCESS** or **I2C_FAILURE**. Possible reasons for failure:

- Invalid **I2CKEY**
- *pbytes* is **NULL**
- Bus contention
- Device did not respond (failure to ACKnowledge)

If ***key*** is invalid, this function returns **I2C_FAILURE**. If ***pContext*** is invalid, the results are undefined.

2.9 Reading a Byte from the I²C bus

The *i2c_readbyte()* function is used to read a byte on a specified I²C bus.

```
int i2c_readbyte(I2CCONTEXT pContext, I2CKEY key, FxU8* pbyte, int ack);
```

This function returns **I2C_SUCCESS** or **I2C_FAILURE**. Possible reasons for failure:

- Invalid **I2CKEY**
- *pbyte* is **NULL**
- Bus contention

If ***key*** is invalid, this function returns **I2C_FAILURE**. If ***pContext*** is invalid, the results are undefined.

2.10 Reading a Series of Bytes from the I²C bus

The *i2c_readbytes()* function is used to read a series of bytes from an I²C bus.

```
int i2c_readbytes(I2CCONTEXT pContext, I2CKEY key, FxU16 numbytes, FxU8* pbytes, int acklastbyte);
```

This function returns **I2C_SUCCESS** or **I2C_FAILURE**. Possible reasons for failure:

- Invalid **I2CKEY**
- *pbytes* is **NULL**
- Bus contention

If ***key*** is invalid, this function returns **I2C_FAILURE**. If ***pContext*** is invalid, the results are undefined.

2.11 I²C Primitives

The following functions, while available for use, are deprecated; you should use them only if absolutely needed (such as with a broken I²C device that requires special attention, like the LCDfx chip.)

```
int i2c_getsda(I2CCONTEXT pContext, I2CKEY key);
int i2c_getscl(I2CCONTEXT pContext, I2CKEY key);
int i2c_setsda(I2CCONTEXT pContext, I2CKEY key, int state, FxU8 delay);
int i2c_setscl(I2CCONTEXT pContext, I2CKEY key, int state, FxU8 delay);
```

The ***pContext*** and ***key*** parameters have their usual meaning.

The ***state*** parameters express the desired state of the SDA or SCL line. A value of 0 means low; any other value is interpreted to mean high.

The ***delay*** parameters express the amount of time, in microseconds, the code should pause after performing the designated action. These values will be multiplied by the delay multiplier specified in the call to *i2c_getaccess()* or *i2c_setspeed()*.

The only reason these functions would fail is if **I2CKEY** is the wrong key. In the case of failure, the "get" functions return **HIGH**, (as this will keep the guilty code from thinking that something has ACKed its transaction,) and the "set" functions return **I2C_FAILURE**. Otherwise, the "get" functions return the correct value (**LOW** or **HIGH**,) and the "set" functions return **I2C_SUCCESS**.

3 Configuration and Initialization

Configuration is supported via the files ***i2cmacro.h***, ***i2csetup.h***, and ***ds_i2c.c***. The ***i2cmacro.h*** file is used to define the following macros, that hide OS dependencies

NOTE: This section is only necessary reading for those who need to implement the DS side of the I2C code.

3.1 Required Entries in *i2cmacro.h*

This file contains the compile-time portion of the I²C MRI configuration process.

I2C_DEVINFOPTR(pcontext)

This should equate to the address of an **I2C_DEVINFO** structure. How this is implemented is up to the client code. For an operating system that doesn't provide multi-monitor support, the following may suffice,

```
I2C_DEVINFO DevInfoI2C;
#define I2C_DEVINFOPTR(foo) &DevInfoI2C  /* foo parameter not used */
```

whereas a multi-monitor operating system will need to provide something more substantial. The following is an example for a Windows98 display driver:

```
#define I2C_DEVINFOPTR(pcontext)  ((PI2C_DEVINFO)&((PDEVTABLE)(pcontext))->i2cInfo)
```

Note that it is OK for such a function to return **NULL**, on error; the I²C MRI code checks the address that **I2C_DEVINFOPTR** equates to, to make sure it is not null before proceeding. Also note that in this last example, **I2C_DEVINFOPTR** equates to a function call, not just a function name. It is only used by the ***ds_i2c.c*** file. Client code should not use this macro.

I2C_GETBUSINFO(pcontext, busid)

This macro maps the given **I2CCONTEXT** and **BUSID** handles to an **I2C_BUSINFO** structure. It can be (and usually is) implemented using the **I2C_DEVINFOPTR** macro. It is only used by the ***ds_i2c.c*** file. Client code should not use this macro.

I2C_DELAYTIME(usecs)

This macro accepts, as its only parameter, a 32-bit unsigned value expressing the *minimum* delay time in microseconds. It should map to a OS-specific function that will delay the current thread for at least the amount of time specified. It is highly recommended that the specified function yield the time back to the operating system's scheduler, rather than spin in a delay loop. It is only used by the ***ds_i2c.c*** file. Client code should not use this macro.

DEBUGI2C(expr, args)

This macro calls "printf##args"if (***expr***) evaluates to true. Note that it is necessary to pass ***args*** to this macro enclosed in parenthesis, i.e., "DEBUGI2C(error, ("Error #%lu occurred.\n", error))"

3.2 Optional Entries in *i2csetup.h*

I2C_PEDANTIC

If this macro is defined and nonzero, the I²C MRI will be more picky about bus errors (i.e., returning **I2C_FAILURE** for more reasons,) and will output more debug spew, if debugging output is turned on. It is generally best to have this turned on. It is turned off by default in ***ds_i2c.h***.

I2C_DEBUGTEXT

If this macro is defined, the I²C MRI code will output debug text using the **I2CDEBUG** macro.

I2C_PRINTERPORT

If this macro is defined, the hardware debugging feature of the I²C MRI will be enabled, if implemented. If it is not implemented, a compilation error will indicate this.

I2C_MAXBUSES

This macro should equate to the maximum number of I2C buses that will be controlled by a single I2C master. If it is not overridden in i2csetup.h, it will default to a value of 10 in ***ds_i2c.h***

I2C_MAXMUXEN

This macro should equate to the maximum number of I2C muxen that will be controlled by a single I2C master. If it is not overridden in i2csetup.h, it will default to a value of 10 in ***ds_i2c.h***

I2C_MAXPURPOSES

This macro should equate to the maximum number of purpose Ids that each I2C bus can be associated with. If it is not overridden in i2csetup.h, it will default to a value of 10 in ***ds_i2c.h***

I2C_STRETCH

This macro indicates how long the I2C code should wait before timing out on a slave device that is stretching the clock. The total time waiting for a stretched clock to go high will be approximately equal to

delay*speed*I2C_STRETCH

If it is not overridden in i2csetup.h, it will default to a value of 50 in ds_i2c.h

3.3 Initialization

The following describes what your *ConfigureI2C()* function (located in ***ds_i2c.c***,) should do to initialize the I²C MRI:

- Instantiate all buses by calling *InstantiateBusI2C()*, once for each bus.
- Call *SetUsageI2C()* for each bus (using the **I2CBUS** value returned by *InstantiateBusI2C()*,) to specify what each bus is used for. If a particular bus has more than one applicable usage ID, you will need to call this function more than once—but you cannot call this function more than **I2C_MAXPURPOSES** times per bus. This maximum value can be overridden in the ***i2csetup.h*** file, by defining **I2C_MAXPURPOSES** as a macro that equates to a number.
- Instantiate all muxen by calling *InstantiateMuxI2C()*, once for each mux.
- Define how some (or all) of the I²C buses bolt up to the muxen, using *AssociateBusToMuxI2C()*. As many as **I2C_MAXBUSES** may be connected to a single mux, but no bus may be connected to more than one mux. Also, You may call this function as many times as you need, so long as you do not exceed these limits.

Once you have completed these steps, you will have no more need for the I2CBUS and I2CMUX values that were returned by *InstantiateBusI2C()* and *InstantiateMuxI2C()*, respectively.

The following is a detailed description of the functions mentioned above:

```
I2CBUS InstantiateBusI2C(I2CCONTEXT pContext, I2C_GETSDA pfGetSDA,
                         I2C_GETSCL pfGetSCL, I2C_SETSDA pfSetSDA,
                         I2C_SETSCL pfSetSCL)
```

This function is used to provide the I²C MRI with 4 functions that control a particular I²C bus.

The ***pContext*** parameter has the usual meaning.

All four remaining parameters are pointers to functions that perform the desired actions. The prototypes for the functions referenced by these parameters must match the following typedefs:

```
typedef uchar (*I2C_GETSDA)();
typedef uchar (*I2C_GETSCL)();
typedef void  (*I2C_SETSDA)(uchar state);
typedef void  (*I2C_SETSCL)(uchar state);
```

On failure, this function returns **I2C_NOTABUS**. Otherwise, it returns a handle for use with *SetUsageI2C()* and *AssociateBusToMuxI2C()*.

int SetUsageI2C(I2CCONTEXT pContext, I2CBUS busid, FxU32 usage)

This function is used to describe to the I²C MRI what the I²C bus (represented by ***busid***) is used for. This function may be called no more than **I2C_MAXPURPOSES** times per bus; otherwise it will fail with the return value **I2C_FAILURE**. (If you need more per bus, you may override this value in your ***i2csetup.h*** file.) You should view the ***di_i2c.h*** file for a complete list of supported usage IDs.

The ***pContext*** parameter has the usual meaning.

The ***busid*** parameter is a nonzero value returned by the *InstantiateBusI2C()* function.

The ***usage*** parameter is one of the usage ID values documented in section 4.1

This function returns **I2C_SUCCESS** or **I2C_FAILURE**.

I2CMUX InstantiateMuxI2C(I2CCONTEXT pContext, I2C_SETMUX pfMux)

This function is used to provide the I²C MRI with a function that controls a particular mux.

The ***pContext*** parameter has the usual meaning.

The ***pfMux*** parameter is a pointer to a function that perform the desired action. The prototype for the function referenced by this parameter must match the following typedef:

```
typedef void  (*I2C_SETMUX)(FxU32);
```

On failure, this function returns **I2C_NOTAMUX**. Otherwise, it returns a handle for use with *AssociateBusToMuxI2C()*.

int AssociateBusToMuxI2C(I2CCONTEXT pContext, I2CBUS busid, I2CMUX muxid, FxU32 muxtoken)

This function is used to describe to the I²C MRI how an I²C bus relates to a mux. It is not necessary to call this function if the I²C bus is not controlled via a mux.

The ***pContext*** parameter has the usual meaning.

The ***busid*** parameter is a nonzero value returned by the *InstantiateBusI2C()* function.

The ***muxid*** parameter is a nonzero value returned by the *InstantiateBusI2C()* function.

The ***muxtoken*** parameter is a nonzero value that instructs the mux function (corresponding to ***muxid***,) how to switch to the appropriate position for this bus.

This function returns **I2C_SUCCESS** or **I2C_FAILURE**.

4 Data Structures, Types and Constants

4.1 Device-Independent Data Structures, Types, and Constants

I2CBUS (di_i2c.h)

This is a 32-bit value indicating the ID of the bus being referenced. It is returned by *InstantiateBusI2C()*, for use with *SetUsageI2C() and AssociateBusToMuxI2C()*.

I2CMUX (di_i2c.h)

This is a 32-bit value indicating the ID of the mux being referenced. It is returned by *InstantiateMuxI2C()*, for use with *AssociateBusToMuxI2C()*.

I2CKEY (di_i2c.h)

This is a 32-bit value that specifies which I²C bus is being used, and also serves as a lock for that particular bus (and, if applicable, the associated mux). The most significant byte of this value indicates the ID of the bus being used (essentially an I2CBUS value.) The remaining bits are the lock value for the bus. A value of **I2C_NOTAKEY** indicates an invalid value.

I2C_NOTAPURPOSE

Used internally.

I2C_TVENCODER

This is a usage ID, indicating that a TV encoder is attached to the bus.

I2C_MONITORDDC

This is a usage ID, indicating that a particular bus is used for DDC communications with a CRT.

I2C_MULTIMEDIA

This is a usage ID, indicating that bus contains WDM-supported devices, such as tuners and video decoders.

I2C_FLATPANELDDC

This is a usage ID, indicating that a particular bus is used for DDC communications with a digital flat panel.

I2C_SERIALEEPROM

This is a usage ID, indicating that a serial EEPROM is attached to the bus.

I2C_LPTDEBUGGING
This is a usage ID, indicating that this bus is actually the system's parallel port. This can be useful for using a secondary machine to poke I²C commands into an I²C bus on another machine. A custom cable must be made to support this feature. A bus claiming this usage ID will only be present if **I2C_PRINTERPORT** is defined.

More usage IDs will be assigned as needed.

4.2 Device-Specific Data Structures, Types, and Constants

It should not be necessary for an MRI client to access any of the data structures, types, or constants defined in the ***ds_i2c.h*** file.