

# Application Note



## I2C Programming (Scarlet)

© Fujitsu Microelectronics Europe GmbH

### History

<b>Date</b>	<b>Author</b>	<b>Version</b>	<b>Comment</b>
20.07.05	WG	V1.0	First version

## Warranty and Disclaimer

To the maximum extent permitted by applicable law, Fujitsu Mikroelektronik GmbH restricts its warranties and its liability for **all products delivered free of charge** (eg. software include or header files, application examples, application Notes, target boards, evaluation boards, engineering samples of IC's etc.), its performance and any consequential damages, on the use of the Product in accordance with (i) the terms of the License Agreement and the Sale and Purchase Agreement under which agreements the Product has been delivered, (ii) the technical descriptions and (iii) all accompanying written materials. In addition, to the maximum extent permitted by applicable law, Fujitsu Mikroelektronik GmbH disclaims all warranties and liabilities for the performance of the Product and any consequential damages in cases of unauthorised decompiling and/or reverse engineering and/or disassembling. **Note, all these products are intended and must only be used in an evaluation laboratory environment.**

1. Fujitsu Mikroelektronik GmbH warrants that the Product will perform substantially in accordance with the accompanying written materials for a period of 90 days form the date of receipt by the customer. Concerning the hardware components of the Product, Fujitsu Mikroelektronik GmbH warrants that the Product will be free from defects in material and workmanship under use and service as specified in the accompanying written materials for a duration of 1 year from the date of receipt by the customer.
2. Should a Product turn out to be defect, Fujitsu Mikroelektronik GmbH's entire liability and the customer's exclusive remedy shall be, at Fujitsu Mikroelektronik GmbH's sole discretion, either return of the purchase price and the license fee, or replacement of the Product or parts thereof, if the Product is returned to Fujitsu Mikroelektronik GmbH in original packing and without further defects resulting from the customer's use or the transport. However, this warranty is excluded if the defect has resulted from an accident not attributable to Fujitsu Mikroelektronik GmbH, or abuse or misapplication attributable to the customer or any other third party not relating to Fujitsu Mikroelektronik GmbH.
3. To the maximum extent permitted by applicable law Fujitsu Mikroelektronik GmbH disclaims all other warranties, whether expressed or implied, in particular, but not limited to, warranties of merchantability and fitness for a particular purpose for which the Product is not designated.
4. To the maximum extent permitted by applicable law, Fujitsu Mikroelektronik GmbH's and its suppliers' liability is restricted to intention and gross negligence.

### **NO LIABILITY FOR CONSEQUENTIAL DAMAGES**

**To the maximum extent permitted by applicable law, in no event shall Fujitsu Mikroelektronik GmbH and its suppliers be liable for any damages whatsoever (including but without limitation, consequential and/or indirect damages for personal injury, assets of substantial value, loss of profits, interruption of business operation, loss of information, or any other monetary or pecuniary loss) arising from the use of the Product.**

Should one of the above stipulations be or become invalid and/or unenforceable, the remaining stipulations shall stay in full effect.

# Contents

- 1 Overview..... 4
- 2 I2C sources ..... 5
- 3 Notes..... 9

# 1 Overview

**Note:** *it's required that the pin #186 of the Scarlet GDC is connected with GND, otherwise the I2C module is not enabled (within the GDC) and you can't use it.*

The following text describes the provided API functions.

The files `scarlet_i2c.h` and `scarlet_i2c.c` have the following functions:

```
void i2c_init(void);
```

Initialize the I2C module with the `i2c_init` function. This means that the internal I2C clock is programmed to a specific clock (100 kHz) and that the bus is set to a default state.

```
void i2c_reset(void);
```

Turn off the I2C clock and disable the I2C module. If you want to use the I2C module again, the `i2c_init` function must be called again.

```
void i2c_send(int addr, int saddr, int data);
```

Send a data byte over the I2C bus. Pass the (**read**) address of the I2C slave as first parameter. The subaddress (the register you want to write to) is written in the second parameter. The data to be transmitted is passed in the third parameter. Please note that only the lower 8 bit of the 'data' parameter are used. This is also the case for all other parameters in this function. In case of a transmission error (arbitration lost, bus error, etc.) the function determines the error and simply returns. Please update this function with the necessary error handling so that it fits to your needs. A simple error handling mechanism could be the following:

1. Send byte
2. Error occurred...
3. Call `i2c_reset`
4. Call `i2c_init`
5. Goto step 1

After a certain count of repetitions, abort the loop and inform the user about the failed transmission.

```
GDC_ULONG i2c_read(int addr, int saddr);
```

Receive a data byte over the I2C bus. Pass the (**read**) address of the I2C slave as first parameter. The subaddress (the register you want to read from) is written in the second parameter. The data to be received is returned. Please note that only the lower 8 bit of the return value are used. This is also the case for the parameters in this function. In case of a reception error (arbitration lost, bus error, etc.) the function determines the error and simply returns. Please update this function with the necessary error handling so that it fits to your needs. See the above description for a simplified error handling example.

Also note that the write address of the I2C slave is used internally – you don't have to provide it explicitly.

For more hardware-related I2C info, please refer to the 'I2C Specification of Scarlet/Orchid' Application Note. You can find this document in your 'Graphic Controller CD-ROM' (select the link 'Application Notes').

## 2 I2C sources

The contents of the file `scarlet_i2c.h`:

```
// -----  
// -  
// - I2C support code for the Scarlet GDC.  
// -  
// - v1.0 WG 08.07.05  
// - Tested data transmission and  
// - reception successfully.  
// -  
// - walantis.giosis@fme.fujitsu.com  
// -  
// -----  
#ifndef __SCARLET_I2C_H__  
#define __SCARLET_I2C_H__  
  
#include "gdc.h"  
  
//! Initialize I2C module.  
/**  
    The GDC I2C module must be setup before using it.  
    This function sets the bus control register and  
    initializes/enables the I2C clock.  
*/  
void i2c_init(void);  
  
//! Turn off I2C module.  
/**  
    Disable the I2C clock and reset  
    all I2C registers to 0.  
*/  
void i2c_reset(void);  
  
//! Send a byte via I2C.  
/**  
    Send a byte to a specific I2C module. All necessary I2C flags  
    are checked during transmission. The arbitration lost error  
    can be determined but not handled (because the GDC is no more  
    the I2C master).  
  
    @param addr The slave-write address (this address is always even).  
    @param saddr The slave subaddress to write within.  
    @param data The data byte to write.  
*/  
void i2c_send(int addr, int saddr, int data);  
  
//! Receive a byte via I2C.  
/**  
    Receive a data byte from a specific I2C module. All necessary  
    I2C flags are checked during transmission / reception. The  
    slave-read address is used internally in this function.  
  
    @param addr The slave-write address (this address is always even).  
    @param saddr The slave subaddress to read from.  
  
    @return GDC_ULONG The lower 8 bit contain the received data byte.  
*/  
GDC_ULONG i2c_read(int addr, int saddr);  
  
#endif // #ifndef __SCARLET_I2C_H__
```

## The contents of the file scarlet\_i2c.c:

```
// -----
// -
// - I2C support code for the Scarlet GDC.
// -
// - V1.0 WG 08.07.05
// - Tested data transmission and
// - reception successfully.
// -
// - walantis.giosis@fme.fujitsu.com
// -
// -----
#include "scarlet_i2c.h"

void i2c_init(void)
{
    // Setup bus control register (set BEIE and INTE)
    // and clock (set EN and frequency to 100 kHz).
    GdcI2CSetClock(0x31);
    GdcI2CSetBusControl(0x42);
}

void i2c_reset(void)
{
    // Reset the I2C interface.
    GdcI2CSetClock(0x00);
    GdcI2CSetBusControl(0x00);
    GdcI2CSetData(0x00);
    GdcI2CSetAddress(0x00);
}

void i2c_send(int addr, int saddr, int data)
{
    // Load slave-write address.
    GdcI2CSetData(addr);
    // Generate START sequence and send slave-write address by enabling the MSS bit.
    GdcI2CSetBusControl(GdcI2CGetBusControlStatus() | 0x10);
    // Check bus status.
    while (1) {
        // BCR.Bit #0 -> 1 (Check that the bus is interrupted).
        // BSR.Bit #7 -> 1 (Check for bus busy).
        // BSR.Bit #5 -> 0 (Check for lost arbitration not occurred).
        // BSR.Bit #3 -> 1 (Indicate data transmission).
        // BSR.Bit #0 -> 1 (Make sure it was the first byte after START sequence).
        if (((GdcI2CGetBusControlStatus() & 0x01) == 0x01) && ((GdcI2CGetBusStatus() &
0xA9) == 0x89)) {
            break;
        }
    }
    // Load slave subaddress.
    GdcI2CSetData(saddr);
    // Send subaddress after clearing the interrupt.
    GdcI2CSetBusControl(GdcI2CGetBusControlStatus() & ~0x01);
    // Check bus status.
    while (1) {
        // BCR.Bit #0 -> 1 (Check that the bus is interrupted).
        // BSR.Bit #7 -> 1 (Check for bus busy).
        // BSR.Bit #5 -> 0 (Check for lost arbitration not occurred).
        // BSR.Bit #3 -> 1 (Indicate data transmission).
        if (((GdcI2CGetBusControlStatus() & 0x01) == 0x01) && ((GdcI2CGetBusStatus() &
0xA8) == 0x88)) {
            break;
        }
    }
    // Load data byte.
    GdcI2CSetData(data);
    // Send data byte after clearing the interrupt.
    GdcI2CSetBusControl(GdcI2CGetBusControlStatus() & ~0x01);
    // Check bus status.
    while (1) {
        // BCR.Bit #0 -> 1 (Check that the bus is interrupted).
```

```

        // BSR.Bit #7 -> 1 (Check for bus busy).
        // BSR.Bit #5 -> 0 (Check for lost arbitration not occurred).
        // BSR.Bit #3 -> 1 (Indicate data transmission).
        if (((GdcI2CGetBusControlStatus() & 0x01) == 0x01) && ((GdcI2CGetBusStatus() &
0xA8) == 0x88)) {
                break;
        }
    }
    // Generate STOP sequence after clearing the MSS bit (SCC bit is actually not needed).
    // Btw. INT bit will be cleared also during the MSS reset.
    GdcI2CSetBusControl(GdcI2CGetBusControlStatus() & ~0x30);
}

GDC_ULONG i2c_read(int addr, int saddr)
{
    GDC_ULONG val;

    // Load slave-write address.
    GdcI2CSetData(addr);
    // Generate START sequence and send slave-write address by enabling the MSS bit.
    GdcI2CSetBusControl(GdcI2CGetBusControlStatus() | 0x10);
    // Check bus status.
    while (1) {
        // BCR.Bit #0 -> 1 (Check that the bus is interrupted).
        // BSR.Bit #7 -> 1 (Check for bus busy).
        // BSR.Bit #5 -> 0 (Check for lost arbitration not occurred).
        // BSR.Bit #3 -> 1 (Indicate data transmission).
        // BSR.Bit #0 -> 1 (Make sure it was the first byte after START sequence).
        if (((GdcI2CGetBusControlStatus() & 0x01) == 0x01) && ((GdcI2CGetBusStatus() &
0xA9) == 0x89)) {
                break;
        }
    }
    // Load subaddress.
    GdcI2CSetData(saddr);
    // Send subaddress after clearing the interrupt.
    GdcI2CSetBusControl(GdcI2CGetBusControlStatus() & ~0x01);
    // Check bus status.
    while (1) {
        // BCR.Bit #0 -> 1 (Check that the bus is interrupted).
        // BSR.Bit #7 -> 1 (Check for bus busy).
        // BSR.Bit #5 -> 0 (Check for lost arbitration not occurred).
        // BSR.Bit #3 -> 1 (Indicate data transmission).
        if (((GdcI2CGetBusControlStatus() & 0x01) == 0x01) && ((GdcI2CGetBusStatus() &
0xA8) == 0x88)) {
                break;
        }
    }
    // Load slave-read address, by setting the R/W bit.
    GdcI2CSetData(addr | 0x01);
    // Send slave-read address after generating the repeated START sequence (INT flag will
be cleared automatically).
    GdcI2CSetBusControl(GdcI2CGetBusControlStatus() | 0x20);
    // Check bus status.
    while (1) {
        // BCR.Bit #0 -> 1 (Check that the bus is interrupted).
        // BSR.Bit #7 -> 1 (Check for bus busy).
        // BSR.Bit #5 -> 0 (Check for lost arbitration not occurred).
        // BSR.Bit #3 -> 0 (Indicate data reception).
        if (((GdcI2CGetBusControlStatus() & 0x01) == 0x01) && ((GdcI2CGetBusStatus() &
0xA8) == 0x80)) {
                break;
        }
    }
    // Read DAR register after clearing interrupt flag.
    GdcI2CSetBusControl(GdcI2CGetBusControlStatus() & ~0x01);
    // Check bus status.
    while (1) {
        // BSR.Bit #7 -> 1 (Check for bus busy).
        // BSR.Bit #5 -> 0 (Check for lost arbitration not occurred).
        // BSR.Bit #4 -> 1 (Check that the last bit is transferred).
        // BSR.Bit #3 -> 0 (Indicate data reception).
        if (((GdcI2CGetBusStatus() & 0xB8) == 0x90)) {
                break;
        }
    }
}

```

```

    }
    val = GdcI2CGetData();
    // Check bus status.
    while (1) {
        // BCR.Bit #0 -> 1 (Check that the bus is interrupted).
        // BSR.Bit #7 -> 1 (Check for bus busy).
        // BSR.Bit #5 -> 0 (Check for lost arbitration not occurred).
        // BSR.Bit #3 -> 0 (Indicate data reception).
        if (((GdcI2CGetBusControlStatus() & 0x01) == 0x01) && ((GdcI2CGetBusStatus() &
0xA8) == 0x80)) {
            break;
        }
    }
    // Generate STOP sequence after clearing the MSS bit.
    // Btw. INT bit will be cleared also during the MSS reset.
    GdcI2CSetBusControl(GdcI2CGetBusControlStatus() & ~0x30);
    return(val);
}

```

### **3 Notes**

Use the latest GDC API (at least V0109) for proper operation with the I2C module.