

I2C2MEM Protocol and Firmware

1. Document

Document information

Info	Content
Keywords	I2C, Protocol, Monitor, Debug
Abstract	<p>This document describes a protocol (still under development) for passively reading from and writing to memory and registers of 16 bit and 32 bit address space electronic devices using I2C.</p> <p>The protocol can be considered to be an enhancement of normal protocols for communicating with eeprom devices with I2C.</p> <p>This document also describes specific target implementation of the protocol in device firmware for a 16 bit microcontroller and a 32 bit microcontroller as well as test software.</p>

Document Revisions

Rev	Date	Description
0.01	1 November, 2006	Initial version
0.20	11 November, 2006	Various additions. Rewrote NINCR bit definition and corrected NINCR bit definition order

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

2. Introduction

This document describes a protocol (still under development) for passively reading from and writing to memory and registers of 16 bit and 32 bit address space electronic devices using I2C.

The protocol can be considered an enhancement of normal protocols for communicating with eeprom devices with I2C.

This document also describes specific target implementation of the protocol in device firmware for a 16 bit microcontroller and a 32 bit microcontroller as well as test software.

A copy of test software and test results is included

The protocol name is provisionally named I2C2MEM. If the name is unacceptable for trade mark or other reasons then the alternative provisional name is TWI2MEM.

3. Authorship and Direction

© Auscyber Pty Ltd 2006. All rights reserved.

The protocol and sample firmware is a project of Auscyber Pty Ltd. Auscyber Pty Ltd is a minerals prospecting company in the state of Queensland, Australia. Auscyber Pty Ltd seeks to use advances in inexpensive electronics to improve the cost effectiveness and efficiency of its minerals prospecting activities and seeks to spin off electronic and software development activity arising from these prospecting goals and activity.

For electronic and software activity Auscyber Pty Ltd uses the name ZGUS. For prospecting activity Auscyber Pty Ltd uses the name Nuggety Gus.

The firmware can be downloaded subject to licence conditions below.

ZGUS activities include development of Hid and serial port communication PC software called Hidprobe. The protocol and firmware fits into Hidprobe improvements and enhancements currently well in progress.

Hidprobe is the freeware version of PC software. It is to be to be customised for specific projects. Information is provided at <http://www.zgus.com/hid/index.html>

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

Within Auscyber Pty Ltd, John Heenan is responsible for developing the protocol, the firmware, test software and Hidprobe

4. I2C2MEM Protocol

4.1 Protocol

The purpose of the protocol is to allow memory addresses and registers of target devices with 16 and 32 bit address spaces to be read from and written to with proper alignment and byte length access.

Write instructions include a complete write, a set, a clear and a xor.

The protocol also includes a facility to set or read values that cannot be made through a standard read or write instruction for architecture specific reasons.

The easiest way to get familiar with the protocol is to review the test software and the results for the target firmware implementation.

4.1.1 'Good Practice'

This protocol is a practical response to practical real world problems. The author has no problem with general principles espoused for decades indicated below. However the author regard these principles not as directives carved in stone but as nice guidelines free to be discarded in resource constrained systems. In fact just about any practical driver level code must be written with a 'resource constrained' attitude.

- Black box encapsulation is good (not embraced)
- Layered definitions that relate to hardware, functionality, data integrity and end use are good (avoided)
- Tying an information protocol unnecessarily to a hardware standard is bad (but not in a resource constrained system)
- Finished is good (not finished)
- Free of architecture specifics is good (the opposite is encouraged)
- Pointers are bad (shown in driver level tests)
- Casting is bad (shown in driver level tests)

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

One of the nice aspects of driver level code is that a nice wrapper to complete a 'black box' is always available. The Unix/Linux attitude is make all hardware access appear to be a file, including doing nothing (as in /dev/null)! Where the black box model does not fit perfectly there is always the ioctl escape!

In fact some operating systems are so concerned with efficiency at the driver level you cannot even pass information through normal call stacks once the driver has been entered! The Windows WDM is an excellent example. WDM uses the concept of an IRP as a 'hot potato' that must be accepted, processed if necessary and passed on as quickly as possible without expecting a reply through a custom designed information passing procedure that does not consume normal stack resources. Memory for passing parameters is protected by a higher level lock when the IRP is in transit! Eventually the IRP finds its way back up again, but not because of call stack returns.

4.1.2 Native carrier

I2C is defined to be the native carrier for the protocol.

The advantage of specifying one carrier as native is that it defines a definitive and complete testable protocol for implementation, test and comparative purposes.

The protocol does not define a series of abstract layers. However it is not difficult to envisage a series of abstract layers.

4.1.3 Meta Information provided by I2C

While the use of the terms 'native' and 'meta' is unusual in technical documents, the terms have well established context dependent usage.

The use of the term 'meta information' here is used in a practical sense (as opposed to an abstract philosophical sense, a 'directory of directories' sense or an 'entity properties' sense) to indicate ANY information that is relevant to making the protocol work (and has nothing to do with the information that is carried).

For the purpose of comparison with protocols that use embedded meta information such as control characters within a rigid and narrow clock speed, I2C can be considered to be standard that provided the following meta information

- Three hardware defined 'control characters': START, STOP and RESTART
- A defined way of specifying an address
- A way of specifying a data direction
- A way of providing a non rigid clock signal speed on the I2C bus
- A way of forcing the clock speed down to keep up with devices that may require a delay as they have work to do.

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

Extension to the protocol may be considered that allows (software) control characters and/or total packet lengths to be defined for non native or non I2C implantations.

With I2C there is no need for (software) control characters or packets that specify the length of the total packet. This simplifies implementation.

4.1.4 Protocol extensions and variations

4.1.4.1 Alternatives and Variations

There are alternatives to using I2C as a carrier. These alternatives include Memory Bus, RS232, LIN, CAN, Ethernet, TCP/IP, USB and Bluetooth. However none have the inherent meta information 'hardware control' simplicity of I2C that enables the I2C2MEM protocol to work so simply with I2C peripheral hardware.

4.1.4.2 Protocol Command Extensions

Architecture specific setup packets allow customised extensions. An obvious extension is to add more commands to allow interface to a complete debug protocol, such as GDB.

A protocol extension can either remap the interpretation of information following or define setup packets longer than one byte. In practice there may not be much difference.

4.2 Motivation for Protocol

The protocol arose from unreleased enhancements to Hidprobe (<http://www.zgus.com/hid/index.html>) which are currently well in progress. Planned enhancements include a capacity to customise Hidprobe to automate test, evaluation and diagnostic sequences with a lightweight protocol.

The closest existing lightweight protocol to read and write to memory locations is the simple protocol used by eeprom memory devices with I2C.

I2C eeprom memory devices use the first byte of a write following a start or restart condition to set a memory address for a read or write. If a write continues then subsequent data is written to memory locations. If instead of a further write there is a restart or stop/start with a read then the address set is used to read from instead.

I2CMEM extends on these concepts. Instead of a single byte to set an address for reading or writing, a setup packet of at least one byte is sent.

4.3 Endian Order

All information is sent and received in little endian order, irrespective of the endian order of stored information and the endian order of register information.

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

This includes the information in setup packets.

4.4 Setup Packet

A setup packet must always follow an I2C start with write.

A setup packet must have at least one byte and may be up to five bytes. A setup packet may be followed by additional setup packets unless an address only setup packet (hence an address setup packet must be the last setup packet)

The first byte is the command byte.

When a setup packet or series of setup packets is finished then subsequent bytes without a restart or stop are write bytes.

To start a read with the setup information there must be restart or stop/start with a read when there are no more setup packets.

This is the normal type of expected behavior for I2C eeprom

There are three types of setup packets. These are:

4.4.1 Address setup packet (three or five bytes)

This is three bytes long for 16 bit address space architectures and five bytes long for 32 bit address space architectures

If bit 0 of the command byte is 0 then this is an address setup packet. All other bits of the command byte are ignored. For 16 bit address space architectures the next two bytes is the address. For 32 bit address space architectures the next four bytes are the address.

Must always be the last setup packet if this type of packet is sent. It may be the only type of packet sent.

The address setup packet is not needed for 16 bit address space architectures as the two byte address_index of the parameter setup packet is sufficient to address the entire 16 bit address space using a default address of 0.

The address setup packet is intended to conform to a style of register addressing used in 32 bit ARM architectures where registers are addressed from an offset base. By not allowing the command byte bits any further interpretation it means earlier packet information is not overridden and a pool of common constant address setup packets can be maintained for register reading and writing purposes.

4.4.2 Architecture Specific Setup Packet

This can be one or more bytes.

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

An example is provided in the firmware section for and MSP430 device, which is one byte long.

While no architecture specific setup packets have so far being defined for NXP LPC2xxxx devices, there is discussion of several possibilities in the firmware section.

If bit 0 is 1 and bit 1 is 1 of the command byte then the setup packet is architecture specific.

There is an additional constraint: if bit 2 is 1 then an additional setup packet follows.

If the remaining five bits are sufficient to fully specify all architecture specific parameters then there is no need for an architecture specific set up packet to be longer than one byte.

An alternative approach is to consider an architecture packet to remap interpretation of packets following.

4.4.3 Parameter Setup Packet

A parameter setup packet is always five bytes long.

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

4.4.3.1 Command Byte (one byte)

The first byte is the command byte (Bit 0 must be 1 and bit 1 must be 0 of the command byte).

Bit	7	6	5	4	3	2	1	0
Name	WCMD		DTYPE		NINCR	ASF	ARCH	BM
Default	00		01(16 bit) 10(32 bit)		1	N/A	N/A	N/A

WCMD	Bits 7-6	Write Command 00 Write (default) 01 Set 10 Clear 11 Xor (an INVERT operation if all bits set to toggle)
DTYPE	Bits 5-4	Data Type (data length and alignment enforcer) 00 Byte 01 Word (two bytes-default for 16 bit address space) 10 DWord (four bytes-default for 32 bit address space) 11 Undefined: causes an I2C NAK
NINCR	Bit 3	No Read Post increment 0 Leave address_index incremented after a read 1 Reset address_index after an I2C read is over (default). A read is over if there is a stop or restart
ASF	Bit 2	Additional Setup Follows 0 Setup is over after the next four bytes 1 Another setup follows
ARCH	Bit 1	Interpret bit map as architecture specific: MUST BE 0 TO BE A PARAMETER SETUP PACKET 0 Interpret bit map as above 1 Interpret bit map as architecture specific and record ASF bit
BM	Bit 0	Interpret bit map: MUST BE 1 TO BE A PARAMETER SETUP PACKET 0 Ignore rest of bit map above. For 16 bit address space architecture next two bytes are the address. Setup is over. For 32 bit address space architectures next four bytes are the address. Setup is over 1 Interpret rest of bit map as above

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

4.4.3.2 address_index (two bytes)

Following the command byte is a two byte address_index.

The address_index is added to the address to form the address read from or written to, taking into account the data type.

4.4.3.3 read_rollover_length (two bytes)

Following the address_index is a two byte read_rollover_length.

Please see the section on address_index modifier operations in the reading section below for an explanation

4.5 Writing

The data type and the type of write operation are relevant. The write address used is obtained from adding the address and address_index together. The address_index is not used as an array index. It is used as a pure byte count and is incremented by one as each new byte is received. However a write will be delayed if the data type is not a byte. In addition there is automatic data alignment correction when writing.

For writing neither the read_rollover_length nor the no_read_post_increment setting has relevance

4.5.1 Data Types

16 bit address space architectures support separate aligned byte and word write or read-modify-write instructions.

32 bit address space architectures support separate aligned byte, word and dword write or read-modify-write instructions.

For a word a write occurs after every second byte is received to an address that is the sum of the address and index_address with the last bit forced to zero.

For a dword a write occurs after fourth byte is received to an address that is the sum of the address and address_index with the last two bits forced to zero.

This means that if a user has not taken alignment into consideration your data may not be written where expected.

It also means that if a user does not supply the full set of bytes for the data type expected then the information would not get written

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

4.5.2 Write Operations

The following type of write operations is supported. The test software and results provide practical examples of all of these operations

4.5.2.1 Write

A straight assignment of the value supplied

4.5.2.2 Set

Only high bits are set

4.5.2.3 Clear

Only high bits are used and they clear bits

4.5.2.4 Xor (eXclusive OR)

Only high bits are used and they toggle bits. If all the bits are set to toggle this becomes an INVERT operation.

4.6 Reading

Whereas there are four types of write operations there is only one type of read operation.

However unlike write operations both `read_rollover_length` and `no_read_post_increment` setting has relevance.

4.6.1 Data Types

16 bit address space architectures support separate aligned byte and word read instructions.

32 bit address space architectures support separate aligned byte, word and dword read instructions.

For a word a read occurs when the first byte is requested from an address that is the sum of the address and `index_address` with the last bit forced to zero. Subsequently a read occurs after second byte is requested

For a dword a write occurs when the first byte is requested from an address that is the sum of the address and `address_index` with the last two bits forced to zero. Subsequently a read occurs after every fourth byte is requested.

This means that if a user has not taken alignment into consideration your data may not be written where expected.

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

4.6.2 Read Operations

Unlike the multiple types of write operations there is only one type of read operation.

4.6.3 address_index modifier operations

After every byte is sent the address_index increases.

For a continuous read without stops or restarts the read_rollover_length specifies how much the address_index increased before it rolls back over to the original address index when reading commenced after a start. This is useful to read from the same location. An example is provided the test software. If the read_rollover_length is zero or one then the same byte location is read over and over again

If no_read_post_increment is set then when a read finished (through a stop or restart) the address_index is reset to the value it had upon starting.

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

5. Sample Target and Test Firmware

5.1 Firmware

Four firmware files are available from <http://www.zgus.com/i2c2mem/index.html>.

Two are protocol target firmware files and two are protocol test firmware files.

5.1.1 Sample Target Firmware

These firmware files include sample implementations of the protocol in firmware. They don't do much more than implement the protocol for test and evaluation purposes. Please note the larger size of the firmware for the NXP is not a reflection of differences due to architecture: all the NXP firmware includes console related and printf functionality which 'bloats' firmware.

- I2c2mem_tia.hex packaged in i2c2mem_tia.zip. Example of I2C2MEM implementation for a TI MSP430F2013 microcontroller, a 16 bit address space device
- I2c2mem_nxa.hex, packaged in i2c2mem_nxa.zip. Example of I2C2MEM implementation for an NXP LPC2148 microcontroller, a 32 bit address space device.

5.1.2 Sample Test Firmware and Software

These firmware files test the protocol on each of the firmware files above and generate test results through UART0

- I2c2mem_nxt_tia.hex, packaged in i2c2mem_nxt_tia.zip. This firmware runs on a NXP LPC2148 to test target firmware running on a TI MSP430F2013 i2c2mem_tia.hex firmware.
- I2c2mem_nxt_nxa.hex, packaged in i2c2mem_nxt_nxa.zip. This firmware runs on a NXP LPC2148 to test target firmware running on another NXP LPC2148 with i2c2mem_nxa.hex firmware.

Relevant test source code of the test software is included below.

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

5.2 Test Code

5.2.1 Test code is 'driver' not 'black box' code

The test software is written as 'driver level' code as opposed to an 'OS API black box' code. An OS API will provide a familiar 'open/read/write/close/ioctl' file interface to driver code.

The purpose of the test software is not to provide a 'black box' interface. The purpose is to help illuminate the inner workings of the protocol, to test its functionality and to provide practical usage information.

The usage information is a help in writing an API file wrapper.

5.2.2 i2c_transfer function

The i2c_transfer function in the test code has the following prototype

```
void i2c_transfer(tU16 i2c_address, tU8 memory_address, tU8 count,...);
```

It is a typical prototype for i2c operations with i2c eeproms. The i2c address is doubled.

5.2.2.1 Write

If the doubled address is left as even then the i2c operation is a master transmit operation. The i2c_address is sent followed by the memory_address. If count is greater than zero then a pointer to a char buffer must be included to enable the specified number of count characters to be read from the buffer for writing. If count is 0 then no pointer to a buffer need follow and only the i2c_address and the memory_address is sent (this can be used to set a memory address for a subsequent read operation).

5.2.2.2 Read

If the doubled address as 1 added to it then then the i2c operation is a master receive operation. The memory_address IS NOT USED FOR ANYTHING, it can be any value as it will be ignored. The count specifies the number of characters to read into a char buffer following.

5.3 Download and Licence

Please see <http://www.zgus.com/i2c2mem/index.html> for a download link to the firmware files.

5.3.1 Licence

The firmware is under the copyright ownership of Auscyber Pty Ltd and is the work of Auscyber Pty Ltd.

A licence to use the firmware is free. Grant of a licence is conditional on accepting that the ownership and work of the firmware must not be

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

misrepresented. Grant of licence is also conditional on accepting the firmware has no warranty as per paragraph below

No license is granted to alter the firmware other than the flash byte I2C addresses. No part of 'About' or help information encoded in the firmware may be removed, altered or added to.

5.3.2 Distribution

The firmware may only be distributed from the <http://www.zgus.com> website. If you got the firmware other than <http://www.zgus.com> website then permission was not granted for this.

5.3.3 Warranty

No Warranty: No warranty of any kind, such as no warranty of fitness for any purpose, including no warranty fitness for merchantability.

No Liability: Auscyber Pty Ltd and its employees shall not be liable for any damage or harm the software may cause by its use, whether from faults or not, - including direct, indirect, incidental, consequential, loss of business profits or special damages

5.4 TI MSP430F2013

5.4.1 Protocol Firmware

Firmware file i2c2mem_tia.hex for an MSP2F2013 is packaged in zip file i2c2mem_tia.zip. The firmware file is in Intel hex format. This is a standard format for uploading through programmers. The I2c address is 0xFE>>1 (0x7F).

5.4.1.1 Firmware upload with ez430 and IAR

A programmer software tool does not appear to accompany the TI ez430 tool. In addition the MSP430F2013 does not support BSL. Hence upload through the Spy-Bi-Wire or JTAG interfaces are the only practical means for generic upload (provided the JTAG fuse has not been blown).

IAR Workbench, which accompanies the ez430 tool, is a development tool, not a programmer tool.

If you are using the TI ez430 tool with IAR Workbench then the following workaround may be used to upload the i2c2mem_tia.hex firmware.

- Create a simple C default project with IAR default main.c contents

```
int main( void ){return 0;}
```

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

- Ensure you can upload the default RELEASE version through the ez430 tool and IAR Workbench, which uses output format msp430-txt.
- Change the output format to Intel-Standard in the linker options.
- Ensure the Intel-Standard hex format version can be uploaded. Stop the session.
- Without deleting the Intel hex format file, which has an extension of a43 in IAR, open this file in notepad and replace its contents through a clipboard copy and paste of contents of i2c2mem_tia.hex file. Close the .a43 file.
- Start a session, end it straight away, detach the ez430 tool and reset or repower the detached MSP430F2013.

5.4.1.2 Firmware Description and Operation

By default the firmware keeps the microcontroller in its lowest power LPM4 mode, which is the RAM retention off mode, consuming typically one ten millionth of an amp (0.1 micro amps).

Activity on the I2C bus will wake up the microcontroller. All servicing of I2C is performed inside the USI I2C interrupt. The microcontroller CPUOFF bit is left permanently enabled so the microcontroller cannot run outside interrupts.

However the OSCOFF, SCG0 and SCG1 bits can be modified with the architecture specific setup packet.

In addition to the USI interrupt the NMI interrupt for a flash access violation (ACCVIE) is set up to cause a PUC reset by forcing a watchdog security key violation. The reason for this is because the protocol will allow flash to be modified.

In fact the I2C address (0x7F, stored as 0xFE) was deliberately chosen so it could be modified though a byte write flash operation to any chosen valid I2C address.

The address can also be modified in the firmware file before upload by modifying the first occurrence of 'FE' in the first line of firmware file (i2c2mem_tia.hex) to the desired address AND also modifying the two hex character checksum for the first line at the end of the line. The checksum is the least significant byte formed from the two's complement sum of the bytes in the line (encoded as hex pair digits) not including the colon or checksum.

The I2C address is stored at flash address 0xF800, which is the first flash address used

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

Bytes 0xF801 to 0xF804 are encoded with the firmware version number (0010).

The start of flash is coded in the firmware similarly to:

```
const char message[] @SLAVE_ADDRESS_NUMERIC =
"\xFE\x00\x11i2c2mem_tia "
"FRM:i2c2mem for MSP430F2013 by Auscyber Pty Ltd\n"
"Firmware for Memory and Register Read/Write via i2c\n"
"Version 0.11, 11 November 2006.\n"
"Reference http://www.zgus.com/i2c2mem/index.html\n"
"This message is at Flash location 0xF810.\n"
"A 20 byte buffer of uninitialised RAM is available\n"
"for private use at address 0x0200.\n"
"The (alterable) slave address is at 0xF800 (start of flash).\n";
```

The 256 byte Information Memory Section was not chosen to store any information in order to avoid upload complications

5.4.1.3 Architecture Specific Setup Packet

The architecture specific setup packet is one byte long. Bits 0, 1 and 2 have significance as discussed in the protocol. Of the five remaining bits only bits 5, 6 and 7 have significance. They are respectively the interrupt on exit bit values of OSCOFF, SCG0 and SCG1 in the status register (R2).

The test software uses the architecture specific setup packet to enable the Sigma-Delta ADC16 to run from the SMCLK and so read values.

5.4.1.4 Firmware Uses

Although the firmware is passive it does have practical uses. It can be used to enable and use the Sigma-Delta ADC16 for real and evaluation purposes.

The firmware can be used to set port values and read port values,

The firmware can be used to enable an ultra low power timekeeper while more power consuming devices are turned off or set to RAM retention mode.

The firmware can store and recall up to 20 bytes in RAM. Please see the test software for more information.

None of the information memory has been used, with the exception of the top eight calibration bytes., hence the information memory can be used as eeprom.

The firmware was tested with a Texas Instruments ez430 development tool, which includes an MS4430F2013. A 2x7 header socket was soldered in and three wires (ground, SCL and SDA) were jumpered with wires to the board with an NXP LPC2148. The I2C frequency was set at 100Khz except for the Sigma-Delta ADC16 tests.

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

5.4.2 Test Software

Firmware file `i2c2mem_nxt_tia.hex` for an LPC2148 is packaged in zip file `i2c2mem_nxt_tia.zip`. The firmware file is in Intel hex format. This is a standard format for uploading through programmers.

The NXP LPC2148 microcontroller requires a 12MHz external crystal. ISP upload speed should not be above 38000 baud. The UART0 ISP port is also used to report results at 38400 baud (8N1, no flow control). UART0 will also report unexpected exceptions.

5.4.2.1 Test Software Extract

```
tU8 buffer[25] __attribute__((aligned(4))); //not necessary to force alignment on
a four byte boundary, however emphasises the issues...
//four setup bytes (the five bytes are made up of one byte specified separately in
a variable and the first four bytes of the buffer)
//20 bytes follow. last byte set to 0 to allow information to be interpreted as
null terminated: it is not sent or received

//tested at 100KHz with a 2x7 header socket soldered onto ez430
//and with wire connections to i2c bus.
//i2c system as used here does not guarantee message delivery
//and does not even wait for a return code
//it allows i2c do what it has to do through processing i2c status codes
//with minimum overhead in an interrupt handler
//it is up to you to decide from reading back if communication is
//taking place successfully (for example reading a RAM location, changing it,
//and reading back to see if successfully changed)
//this allows message integrity to be kept tracked of more independently of the
delivery protocol
//without concern about whether the 'details are right' during development
phases
//U8 wait_for_unlock(tU8 *lock, tU16 timeout_ms) allows you to know if i2c is
finished with the buffer and
//will take drastic i2c reset action if there is a severe problem requiring
likely hardware action
//due to say an SCL line being held low too long by a hung i2c bus peripheral
//communication failure occurs if there strings of X in reply, eg
"XXXXXXXXXXXXXXXXXX"
//as this indicates nothing changed

int i2c2mem_tia_sample(struct s_i2c_params *pi2c_params, tU8 *i2c_buffer, tU8
i2c_buffer_length)
{
//next three variables make up the values of the five byte setup packet
tU8 command_byte;
tU16 *address_index=(tU16*)(void*)i2c_buffer;//overrides alignment warnings
tU16 *read_rollover_length=(tU16*)(void*)(i2c_buffer+2);//overrides alignment
warnings

tU8 *reg8 =(tU8*) (void*)(i2c_buffer+4);
tU16 *reg16=(tU16*)(void*)(i2c_buffer+4);//alignment is OK

const tU8 copy_message[]="RAM: ";
tU i,j;//counters
tU16 *psdval;//moving pointer to a transferred sigma delta value
```

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

```

tU8 test_SCG;//variable to alter saved clock control status register bits in the
MSP430
tU16 prior_freq_kHz;//preserve old i2c frequency for a test

if(i2c_buffer_length<25 || sizeof(copy_message)>i2c_buffer_length-8)//buffer is
too short
return 0;
i2c_buffer_length-=1;//reduce length of buffer by one
i2c_buffer[i2c_buffer_length]=0;//and provide a null terminator where buffer
has been chopped

command_byte=0x09;//BYTE, no_read_post_incr+interpret_bitmap
*address_index=DEVICE_FLASH_MESSAGE_ADDRESS;//first byte is the i2c address,
skip it
*read_rollover_length=i2c_buffer_length+1;//make sure there is no read rollover
memset(i2c_buffer+4,'X',i2c_buffer_length-4);//fill buffer+4 with X to ensure
i2c changes buffer

i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 4, i2c_buffer);//set address to
start of message in flash
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
i2c_transfer(DEVICE_I2C_ADDRESS+1, 0, i2c_buffer_length-4, i2c_buffer+4);//i2c
read so setup_code not sent, read into non setup part of buffer
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
printf("\nstart of device message is: \"%s\"", i2c_buffer+4);

memcpy(i2c_buffer+4, copy_message, sizeof(copy_message)-1);//do not copy over
the null terminator of the overwrite copy_message
*address_index=DEVICE_RAM_ADDRESS;
//done: *read_rollover_length=i2c_buffer_length+1;//make the rollover length
larger than buffer length
printf("\nwriting %d bytes to RAM address 0x0200 \"%s\"",
DEVICE_RAM_ADDRESS_FREELength, i2c_buffer+4);
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, DEVICE_RAM_ADDRESS_FREELength+4,
i2c_buffer);//write 20 bytes to RAM what just read, overwriting the start
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

memset(i2c_buffer+4,'X',i2c_buffer_length-4);
//command_byte=0x09;//write+BYTE, no_read_post_incr+interpret_bitmap
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 4, i2c_buffer);//set address to
start again
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
i2c_transfer(DEVICE_I2C_ADDRESS+1, 0, i2c_buffer_length-4, i2c_buffer+4);//i2c
read so setup_code not sent, read into non setup part of buffer
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
printf("\nreading from RAM: \"%s\"", i2c_buffer+4);

memset(i2c_buffer+4,'X',i2c_buffer_length-4);
i2c_transfer(DEVICE_I2C_ADDRESS+1, 0, i2c_buffer_length-4, i2c_buffer+4);//i2c
read so setup_code not sent, read into non setup part of buffer
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
printf("\ntesting no_read_post_incr, reading from RAM again, expect same as
previous: \"%s\"", i2c_buffer+4);

*read_rollover_length=4;//test read rollover length
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 4, i2c_buffer);//set address to
start again
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
memset(i2c_buffer,'X',i2c_buffer_length);

```

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

```

    i2c_transfer(DEVICE_I2C_ADDRESS+1, 0, i2c_buffer_length, i2c_buffer); //use
    entire buffer this time for reading, not saving setup.
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
    printf("\ndevice address loop back test without resetting address, expect repeat
    of four character sequence \"%s\"", i2c_buffer);

//read all of the help message
    command_byte=0x01; //write+BYTE(write irrelevant), interpret_bitmap (read will
    increment)
    *address_index=DEVICE_FLASH_MESSAGE_ADDRESS;
    *read_rollover_length=i2c_buffer_length+1; //make the rollover length larger
    than buffer length to
    //avoid rollovers during read
    i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 4, i2c_buffer); //set the address
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
    printf("\nPrinting the entire device message:\n");
    i=DEVICE_MAX_LENGTH_BEFORE_NULL_TERMINATOR; //max to read if no null terminator
    while(1)
    {
        i2c_transfer(DEVICE_I2C_ADDRESS+1, 0, i2c_buffer_length, i2c_buffer); //read
        continuously until null terminator found
        if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
        char ch=0;
        for(j=0;j<i2c_buffer_length;j++)
        {
            i--;
            ch=i2c_buffer[j];
            if(!ch || !i)
                break;
            else
                putchar(ch);
        }
        if(!ch || !i)
            break;
    }
    if(i)
        printf("\nfinished printing the device message");
    else
        printf("\nnull terminator of device message not found within %d characters",
        DEVICE_MAX_LENGTH_BEFORE_NULL_TERMINATOR);

    command_byte=0x11; //write+WORD, interpret_bitmap
    *address_index=0x0120; //address of WDTCTL
    *read_rollover_length=i2c_buffer_length;
    //WDTCTL = 0; //force a PUC reset by writing to the watchdog timer control
    register with a security key violation
    *reg16= 0;
    printf("\nabout to force a device reset: device pull up resistors on i2c bus
    are not enabled during reset");
    i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 6, i2c_buffer); //write to the
    register
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
    delay_ms(100);
    printf("\nforced a PUC reset by writing to the watchdog timer control register
    with a security key violation");
    memset(i2c_buffer, 'X', i2c_buffer_length);
    i2c_transfer(DEVICE_I2C_ADDRESS+1, 0, i2c_buffer_length, i2c_buffer); //read
    from the default reset location
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

```

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

```

printf("\nread from default read location on PUC reset: \"%s\"", i2c_buffer);

/*address_index=0xF000;//invalid flash address
//not tested reset with invalid flash access write
//as may occur when changing the i2c slave address at address 0xF800
//instead of changing before upload in the i2c2mem_msp430f201x.hex file
//through ACCVIE NMI interrupt set to force a PUC reset

//only changing the address_index tested
//changing the default address from 0 will be tested whetn testing on a 32 bit
architerture

command_byte=0xC9;//xor+BYTE, no_read_post_incr+interpret_bitmap,
no_read_post_incr is irrelevant for a write
*address_index=0x0021;//address of P1OUT byte register
/*read_rollover_length is irrelevant for a write
buffer[4]=0x01;//bit 0 is the bit to XOR, SET and CLEAR in P1OUT

for(i=0;i<5;i++)
{
printf("\ntoggling TI ez430 led with an xor write");
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 5, i2c_buffer);
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
delay_ms(2000);
}
command_byte=0x49;//set+BYTE, no_read_post_incr+interpret_bitmap
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 5, i2c_buffer);
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
printf("\nensuring the TI ez430 led is turned on");
delay_ms(2000);
command_byte=0x89;//clear+BYTE, no_read_post_incr+interpret_bitmap
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 5, i2c_buffer);
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
printf("\nensuring the TI ez430 led is turned off");
delay_ms(2000);

for(test_SCG=0;test_SCG<8;test_SCG++)
{
printf("\nsetting for 2 seconds OSCOFF bit to %d, SCG0 bit to %d, SCG1 bit
to %d", test_SCG&0x01?1:0, test_SCG&0x02?1:0, test_SCG&0x04?1:0);
command_byte=0x03 | (test_SCG<<5);//adjust saved SR with no addiitonal setup
following
i2c_transfer(DEVICE_I2C_ADDRESS,command_byte,0);
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
delay_ms(2000);// power consumption can be measured during each 10 second
test!
}
printf("\nlast setting was (OSCOFF,SCG0,SCG1)=(1,1,1). Hence MSP430 im LPM4
sleep state consuming less than 10 millionth of an amp");

//setting up the Sigma-Delta ADC16
//A continuous single-ended sample is made on A1+ (P1.2) using internal VRef with
Unipolar output format
//turn on SMCLK
printf("\nreturning on the SD16");
command_byte=0x03 | 0x20;//turn on SMCLK with SCG0=0 and SCG1=0 to enable SMCLK
for SD16
i2c_transfer(DEVICE_I2C_ADDRESS,command_byte,0);
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

```

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

```

    command_byte=0x11;//write+WORD, interpret_bitmap
    *address_index=0x0100;//address of SD16CTL
    //SD16CTL = SD16REFON + SD16SSEL_1 + SD16XDIV_2; // 1.2V ref, SMCLK / 16
    *reg16= (1<<2) | (1<<4) | (1<<10);
    i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 6, i2c_buffer);//write to the
register
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

    command_byte=0x01;//write+BYTE, interpret_bitmap
    *address_index=0x00B0;//address of SD16INCTL0
    //SD16INCTL0 = SD16INCH_1; // A1+/-
    *reg8= 0x01;
    i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 5, i2c_buffer);//write to the
register
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

    command_byte=0x11;//write+WORD, interpret_bitmap
    *address_index=0x0102;//address of SD16CCTL0
    //SD16CCTL0 = SD16UNI; // 256OSR, unipolar
    *reg16= (1<<12);
    i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 6, i2c_buffer);//write to the
register
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

    command_byte=0x01;//write+BYTE, interpret_bitmap
    *address_index=0x00B7;//address of SD16AE
    //SD16AE = SD16AE2; // P1.2 A1+, A1- = VSS
    *reg8= (1<<2);
    i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 5, i2c_buffer);//write to the
register
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

    command_byte=0x51;//set+WORD, interpret_bitmap
    *address_index=0x0102;//address of SD16CCTL0
    //SD16CCTL0 |= SD16SC; // Set bit to start conversion
    *reg16= (1<<1);
    i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 6, i2c_buffer);//write to the
register
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

    command_byte=0x19;//write+WORD(write irrelevant),
no_read_post_incr+interpret_bitmap
    *address_index=0x0112;//address of SD16MEM0
    *read_rollover_length=2;
    i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 4, i2c_buffer);//set the address
of the register to read
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
    memset(i2c_buffer, 'X', i2c_buffer_length);
    delay_ms(100);
    prior_freq_kHz=pi2c_params->freq_kHz;
    pi2c_params->freq_kHz=20;
    i2c_init(pi2c_params);//slow down i2c to 20Khz as new result frequency is now
1Mhz/256 approx 4KHz
    //i2c byte transfer rate becomes approx 2KHz
    //NB to eliminate noise i2c should not be run while sampling and conversions are
taking place
    i2c_transfer(DEVICE_I2C_ADDRESS+1, 0, i2c_buffer_length, i2c_buffer);//read
result continuously
    if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

```

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

```

printf("SD16MEM0 read with i2c bus slowed to 20KHz to allow continuous
conversion results between reads:");
printf("\n(note: for high resolution accuracy results, i2c should not be
running during sampling and conversion)\n");

i=i2c_buffer_length>>1;
psdval=(tU16*)(void*)i2c_buffer;
while(i--)
    printf("0x%04x ",*psdval++);

pi2c_params->freq_kHz=prior_freq_kHz;
i2c_init(pi2c_params); //reset i2c speed to prior i2c speed
printf("\ni2c bus now at 100KHz again");

command_byte=0x11;//write+WORD, interpret_bitmap
*address_index=0x0120;//address of WDTCTL
*read_rollover_length=i2c_buffer_length;
//WDTCTL = 0; //force a PUC reset by writing to the watchdog timer control
register with a security key violation
*reg16= 0;
printf("\nabout to force a device reset: device pull up resistors on i2c bus
are not enabled during rest");
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 6, i2c_buffer);//write to the
register
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
delay_ms(100);
printf("\nfinished forcing a PUC reset by writing to the watchdog timer control
register with a security key violation");
memset(i2c_buffer,'X',i2c_buffer_length);
i2c_transfer(DEVICE_I2C_ADDRESS+1, 0, i2c_buffer_length, i2c_buffer);//read
from the default reset location
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
printf("\nread from default read location on PUC reset: \"%s\"", i2c_buffer);

return TRUE;
}

```

5.4.2.2 Test Results

demo

start of device message is: "FRM:i2c2mem for MSP4"

writing 20 bytes to RAM address 0x0200 "RAM:i2c2mem for MSP4"

reading from RAM: "RAM:i2c2mem for MSP4"

testing no_read_post_incr, reading from RAM again, expect same as previous:

"RAM:i2c2mem for MSP4"

device address loop back test without resetting address, expect repeat of four character sequence "RAM:RAM:RAM:RAM:RAM:RAM:"

Printing the entire device message:

FRM:i2c2mem for MSP430F2013 by Auscyber Pty Ltd

Firmware for Memory and Register Read/Write via i2c

Version 0.11, 11 November 2006.

Reference <http://www.zgus.com/i2c2mem/index.html>

This message is at Flash location 0xF810.

A 20 byte buffer of uninitialized RAM is available

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

for private use at address 0x0200.

The (alterable) slave address is at 0xF800 (start of flash).

finished printing the device message

about to force a device reset: device pull up resistors on i2c bus are not enabled during reset

forced a PUC reset by writing to the watchdog timer control register with a security key violation

read from default read location on PUC reset: "FRM:i2c2mem for MSP430F2"

toggling TI ez430 led with an xor write

toggling TI ez430 led with an xor write

toggling TI ez430 led with an xor write

toggling TI ez430 led with an xor write

toggling TI ez430 led with an xor write

ensuring the TI ez430 led is turned on

ensuring the TI ez430 led is turned off

setting for 2 seconds OSCOFF bit to 0, SCG0 bit to 0, SCG1 bit to 0

setting for 2 seconds OSCOFF bit to 1, SCG0 bit to 0, SCG1 bit to 0

setting for 2 seconds OSCOFF bit to 0, SCG0 bit to 1, SCG1 bit to 0

setting for 2 seconds OSCOFF bit to 1, SCG0 bit to 1, SCG1 bit to 0

setting for 2 seconds OSCOFF bit to 0, SCG0 bit to 0, SCG1 bit to 1

setting for 2 seconds OSCOFF bit to 1, SCG0 bit to 0, SCG1 bit to 1

setting for 2 seconds OSCOFF bit to 0, SCG0 bit to 1, SCG1 bit to 1

setting for 2 seconds OSCOFF bit to 1, SCG0 bit to 1, SCG1 bit to 1

last setting was (OSCOFF,SCG0,SCG1)=(1,1,1). Hence MSP430 im LPM4 sleep state

consuming less than 10 millionth of an amp

turning on the SD16SD16MEM0 read with i2c bus slowed to 20KHz to allow continuous conversion results between reads:

(note: for high resolution accuracy results, i2c should not be running during sampling and conversion)

0x0bb1 0x0baf 0x0bb6 0x0ba2 0x0bb1 0x0bb5 0x0bb9 0x0bb0 0x0bc2 0x0bbe 0x0bbd

0x0bb6

i2c bus now at 100KHz again

about to force a device reset: device pull up resistors on i2c bus are not enabled during rest

finished forcing a PUC reset by writing to the watchdog timer control register with a security key violation

read from default read location on PUC reset: "FRM:i2c2mem for MSP430F2"

i2c2mem tests to MSP430F2 complete

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

5.5 NXP LPC2148

5.5.1 Protocol Firmware

Firmware file i2c2mem_nxa.hex for an LPC2148 is packaged in zip file i2c2mem_nxa.zip and is in Intel hex format. This is a standard format for uploading through programmers. The I2c address is 0xFC>>1 (0x7E)

5.5.1.1 Firmware Upload

The file format is suitable for upload through ISP loaders or through JTAG programmers.

A 12MHz external crystal is required. ISP upload speed should not be above 38000 baud.

Two tested ISP uploaders took longer to upload than expected as there is a large and unused gap between firmware in lower parts of flash and message firmware in upper parts of flash that there is no need to fill which the ISP uploaders insisted on writing to. A JTAG programmer (using IAP) did not have this problem. Future versions will take this into consideration.

The UART0 ISP port is also used to report a reset message (as well as unexpected exceptions).

5.5.1.2 Firmware Description and Operation

Currently for the i2c2mem_nxa.hex firmware no architecture specific setup packets have been defined.

The firmware currently runs in idle mode, not power down mode.

It should be possible to perform the same types of port and peripheral actions as for the MSP4320F2013.

However there is one crucial difference. If an exception arises because of

- values provided by I2CMEM or
- through faults on the I2C bus result that corrupt data

then the firmware will report the exception through UART0 and enter an infinite loop. A reset will be required. Hence a reported exception should not be seen as a bug in the firmware

5.5.1.3 Architecture Specific Setup Packet

Use of an architecture specific setup packet will currently be ignored, except for the 'additional setup follows' bit.

There are several candidates for architecture specific setup information.

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

One is a one byte setup package to bring the microcontroller out of power down mode (EINT1 can be made level sensitive to SDA0). If the microcontroller was already out of power down mode it can ignore the packet. If it was in power down mode it should be fully awake by the time another i2c start is sent.

Another candidate is to use the architecture specific setup packet for protocol extension. A command can be issued to remap interpretation of information following the setup packet to suit

- IAP commands
- Use of microcontroller as an eeprom device using the LPC2K_ee source code provided by Philips/NXP.
- A GDB stub for debugging

Alternatively information can be passed in completely in setup packets. It is a matter of definition. In practice there may be little practical difference between remapping meaning of packets following setup packets or defining new setup packets.

5.5.2 Test Software

Firmware file i2c2mem_nxt_nxa.hex for an LPC2148 is packaged in zip file i2c2mem_nxt_nxa.zip. The firmware file is in Intel hex format. This is a standard format for uploading through programmers.

The NXP LPC2148 microcontroller requires a 12MHz external crystal. ISP upload speed should not be above 38000 baud. The UART0 ISP port is also used to report results at 38400 baud (8N1, no flow control). UART0 will also report unexpected exceptions.

The test firmware is simply used as a 'proof of concept' for the protocol for 32 bit address space architectures with DWORD writing and reading to RAM.).

The MSP430F2013 sample test firmware demonstrates a more sophisticated use of the protocol.

5.5.2.1 Test Software Extract

```
int i2c2mem_nxpa_sample(struct s_i2c_params *pi2c_params, tU8 *i2c_buffer, tU8
i2c_buffer_length)
{
    //next three variables make up the values of the five byte setup packet
    tU8 command_byte;
    tU16 *address_index=(tU16*)(void*)i2c_buffer;//overrides alignment warnings
    tU16 *read_rollover_length=(tU16*)(void*)(i2c_buffer+2);//overrides alignment
warnings
    tU32 *address =(tU32*)(void*)i2c_buffer;//alignment is OK

    //tU8 *reg8 =(tU8*) (void*)(i2c_buffer+4);
```

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

```

//tU16 *reg16=(tU16*)(void*)(i2c_buffer+4);//alignment is OK
//tU32 *reg32=(tU32*)(void*)(i2c_buffer+4);//alignment is OK

const tU8 copy_message[]="RAM: ";
tU i,j;//counters

if(i2c_buffer_length<25 || sizeof(copy_message)>i2c_buffer_length-8)//buffer is
too short
return 0;
i2c_buffer_length-=1;//reduce length of buffer by one
i2c_buffer[i2c_buffer_length]=0;//and provide a null terminator where buffer
has been chopped

//command_byte=0x0D;//BYTE,
no_read_post_incr+additional_setup_follows+interpret_bitmap
command_byte=0x05;//BYTE, additional_setup_follows+interpret_bitmap
*address_index=0;//first byte is the i2c address, skip it
*read_rollover_length=i2c_buffer_length+1;//make sure there is no read rollover
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 4, i2c_buffer);//set address to
start of message in flash
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
*address=DEVICE_FLASH_MESSAGE_ADDRESS;
i2c_transfer(DEVICE_I2C_ADDRESS, 0, 4, i2c_buffer);//set address to start of
message in flash
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

memset(i2c_buffer,'X',i2c_buffer_length);//fill buffer with X to ensure data
read changes
printf("\nPrinting the entire device message:\n");
i=DEVICE_MAX_LENGTH_BEFORE_NULL_TERMINATOR;//max to read if no null terminator
while(1)
{
i2c_transfer(DEVICE_I2C_ADDRESS+1, 0, i2c_buffer_length, i2c_buffer);//read
continuously until null terminator found
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
char ch=0;
for(j=0;j<i2c_buffer_length;j++)
{
i--;
ch=i2c_buffer[j];
if(!ch || !i)
break;
else
putchar(ch);
}
if(!ch || !i)
break;
}
if(i)
printf("\nfinished printing the device message");
else
printf("\nnull terminator of device message not found within %d characters",
DEVICE_MAX_LENGTH_BEFORE_NULL_TERMINATOR);

command_byte=0x2D;//write+DWORD,
no_read_post_incr+additional_setup_follows+interpret_bitmap
*address_index=0;//the address_index
*read_rollover_length=i2c_buffer_length+1;//make sure there is no read rollover
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 4, i2c_buffer);//set address to
start of message in flash
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;

```

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

```

*address=DEVICE_RAM_ADDRESS;//only needs to be set once
*((tU32*)(void*)(i2c_buffer+4))=0x01234567;
*((tU32*)(void*)(i2c_buffer+8))=0x456789AB;
*((tU32*)(void*)(i2c_buffer+12))=0x89ABCDEF;
i2c_transfer(DEVICE_I2C_ADDRESS, 0, 16, i2c_buffer);//set address to start of
message in flash
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
printf("\nwriting to RAM address 0x%08x three 32 bit values 0x%08x, 0x%08x,
0x%08x", DEVICE_RAM_ADDRESS, *((tU32*)(void*)(i2c_buffer+4)),
*((tU32*)(void*)(i2c_buffer+8)), *((tU32*)(void*)(i2c_buffer+12)));

//address has already been set
command_byte=0xE9;//xor+DWORD, no_read_post_incr+interpret_bitmap
//next two need to be reset because overwrote the buffer with *address
*address_index=0;//reset address_index to 0
*read_rollover_length=i2c_buffer_length+1;//make sure there is no read rollover
*((tU32*)(void*)(i2c_buffer+4))=0xFFFFFFFF;
*((tU32*)(void*)(i2c_buffer+8))=0xFFFFFFFF;
*((tU32*)(void*)(i2c_buffer+12))=0xFFFFFFFF;
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 16, i2c_buffer);//set address to
start of message in flash
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
printf("\nInverting the values passed into RAM using an XOR with all bits set
to toggle");

//address has already been set. also all the prior codes can be used as only
reading
i2c_transfer(DEVICE_I2C_ADDRESS, command_byte, 4, i2c_buffer);//set address to
start of message in flash
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
printf("\nresetting address+address_index to start of RAM");
i2c_transfer(DEVICE_I2C_ADDRESS+1, 0, 12, i2c_buffer+4);
if(!wait_for_unlock(&i2c_lock, I2C_WAITFORLOCK_TIMEOUT_MS)) return FALSE;
printf("\nread from RAM address 0x%08x three inverted 32 bit values 0x%08x,
0x%08x, 0x%08x", DEVICE_RAM_ADDRESS, *((tU32*)(void*)(i2c_buffer+4)),
*((tU32*)(void*)(i2c_buffer+8)), *((tU32*)(void*)(i2c_buffer+12)));

return TRUE;
}

```

5.5.2.2 Test Results

Printing the entire device message:

FRM:i2c2mem for LPC2148 by Auscyber Pty Ltd

Firmware for Memory and Register Read/Write via i2c

Version 0.11, 9 November 2006.

Reference <http://www.zgus.com/i2c2mem/index.html>

This message is at Flash location 0x0007B010.

A 256 (or 0x100) byte buffer of uninitialised RAM is available

for private use at address 0x40000080.

The (alterable) slave address is at 0x0007B000.

finished printing the device message

writing to RAM address 0x40000080 three 32 bit values 0x01234567, 0x456789ab,
0x89abcdef

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

Inverting the values passed into RAM using an XOR with all bits set to toggle
resetting address+address_index to start of RAM
read from RAM address 0x40000080 three inverted 32 bit values 0xfedcba98,
0xba987654, 0x76543210

i2c2mem tests to LPC2148 complete
Printing the entire device message:
FRM:i2c2mem for LPC2148 by Auscyber Pty Ltd
Firmware for Memory and Register Read/Write via i2c
Version 0.11, 9 November 2006.
Reference <http://www.zgus.com/i2c2mem/index.html>
This message is at Flash location 0x0007B010.
A 256 (or 0x100) byte buffer of uninitialised RAM is available
for private use at address 0x40000080.
The (alterable) slave address is at 0x0007B000.

finished printing the device message
writing to RAM address 0x40000080 three 32 bit values 0x01234567, 0x456789ab,
0x89abcdef
Inverting the values passed into RAM using an XOR with all bits set to toggle
resetting address+address_index to start of RAM
read from RAM address 0x40000080 three inverted 32 bit values 0xfedcba98,
0xba987654, 0x76543210

i2c2mem tests to LPC2148 complete

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

6. Disclaimers

No Warranty: No warranty of any kind, such as no warranty of fitness for any purpose, including no warranty fitness for merchantability.

No Liability: Auscyber Pty Ltd and its employees shall not be liable for any damage or harm the software may cause by its use, whether from faults or not, - including direct, indirect, incidental, consequential, loss of business profits or special damages

7. Trademarks

All referenced brands, product names, service names and trademarks are the property of their respective owners.

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com

8. Contents

1.	Document	1	5.1	Firmware	12
2.	Introduction	2	5.1.1	Sample Target Firmware	12
3.	Authorship and Direction	2	5.1.2	Sample Test Firmware and Software	12
4.	I2C2MEM Protocol	3	5.2	Test Code	13
4.1	Protocol.....	3	5.2.1	Test code is 'driver' not 'black box' code	13
4.1.1	'Good Practice'.....	3	5.2.2	i2c_transfer function.....	13
4.1.2	Native carrier.....	4	5.2.2.1	Write.....	13
4.1.3	Meta Information provided by I2C	4	5.2.2.2	Read	13
4.1.4	Protocol extensions and variations	5	5.3	Download and Licence	13
4.1.4.1	Alternatives and Variations	5	5.3.1	Licence.....	13
4.1.4.2	Protocol Command Extensions.....	5	5.3.2	Distribution	14
4.2	Motivation for Protocol	5	5.3.3	Warranty.....	14
4.3	Endian Order.....	5	5.4	TI MSP430F2013	14
4.4	Setup Packet.....	6	5.4.1	Protocol Firmware	14
4.4.1	Address setup packet (three or five bytes).....	6	5.4.1.1	Firmware upload with ez430 and IAR	14
4.4.2	Architecture Specific Setup Packet.....	6	5.4.1.2	Firmware Description and Operation.....	15
4.4.3	Parameter Setup Packet.....	7	5.4.1.3	Architecture Specific Setup Packet	16
4.4.3.1	Command Byte (one byte)	8	5.4.1.4	Firmware Uses	16
4.4.3.2	address_index (two bytes)	9	5.4.2	Test Software	17
4.4.3.3	read_rollover_length (two bytes).....	9	5.4.2.1	Test Software Extract.....	17
4.5	Writing.....	9	5.4.2.2	Test Results	22
4.5.1	Data Types.....	9	5.5	NXP LPC2148	24
4.5.2	Write Operations	10	5.5.1	Protocol Firmware	24
4.5.2.1	Write	10	5.5.1.1	Firmware Upload	24
4.5.2.2	Set	10	5.5.1.2	Firmware Description and Operation.....	24
4.5.2.3	Clear	10	5.5.1.3	Architecture Specific Setup Packet	24
4.5.2.4	Xor (eXclusive OR)	10	5.5.2	Test Software	25
4.6	Reading	10	5.5.2.1	Test Software Extract	25
4.6.1	Data Types.....	10	5.5.2.2	Test Results	27
4.6.2	Read Operations.....	11	6.	Disclaimers	29
4.6.3	address_index modifier operations	11	7.	Trademarks	29
5.	Sample Target and Test Firmware	12	8.	Contents	30

Contact information

For up to date or additional information, please visit: <http://www.zgus.com/i2c2mem/index.html>

Email i2c2mem@zgus.com