Using CryptoMemory® in Full I²C Compliant Mode

1. Introduction

This application note describes how to communicate with CryptoMemory® devices in full I²C compliant mode. Full I²C compliance permits use of I²C hardware peripheral controllers within microcontrollers to communicate with CryptoMemory, thus eliminating the need for software drivers and General Purpose Input/Output (GPIO) pins. This leads to performance improvement by lowering CPU utilization and firmware footprint.

This application note comes with downloadable code examples for the Atmel AVR microcontroller platform. The download includes source and project Makefile files for user trials on any AVR based development platforms, including Atmel's Aris, Aris+, and Aris++ development kits

2. CryptoMemory Devices That Support Full I²C Compliance

Support for full compliance to I²C communication is available to newer generation CryptoMemory devices. These devices are identifiable by a "CA" termination of their base names as in AT88SC0104*CA*. They offer full backward compatibility to older generation CryptoMemory devices identifiable by a "C" termination in their base names as in AT88SC0104*C*.

3. CryptoMemory and I²C Compliance

All generations of CryptoMemory comply with I²C communications to some extent. The older generation devices only offer partial compliance. The entire command set for this generation requires the least significant bit of the command byte – also known as the *R/W* bit of an I²C command structure, is always at logic 0. In I²C communications, this signifies a *WRITE* operation. The implications of having the *R/W* bit always at logic 0 is that any microcontroller attempting to use its hardware I²C buss controller to implement the CryptoMemory command set will always try to send data to CryptoMemory irrespective of the command intent. When the command in question happens to be a *READ* command which requires transfer of data from CryptoMemory to the microcontroller, a buss contention will result as both microcontroller and CryptoMemory try to put information on the buss. Older generation CryptoMemory devices therefore require I²C buss controllers implemented entirely in firmware and using General Purpose Input/Output (GPIO) pins for physical connectivity.

Using CryptoMemory® in Full I²C Compliant Mode

AT88SC0104CA
AT88SC0204CA
AT88SC0404CA
AT88SC0808CA

Application Note

8662A–CryptoMemory–03/09

Figure 1. The R/W bit of the I^2C command byte is always logic 0 in the command set for older generation CryptoMemory.

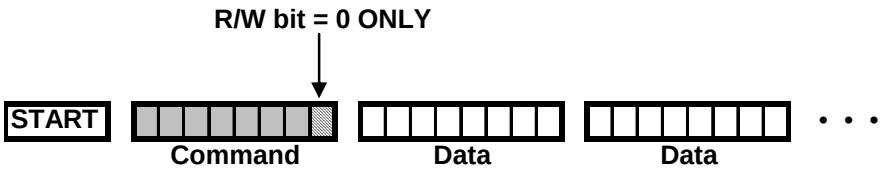
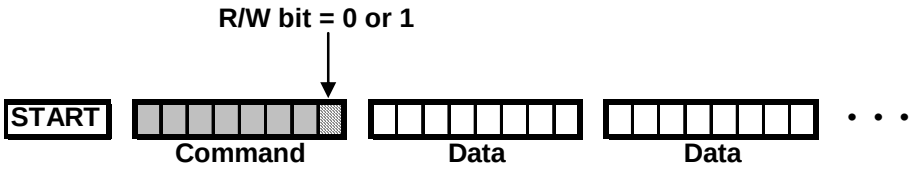


Figure 2. The R/W bit of the I^2C command byte is either logic 0 or 1 in the command set for the new generation CryptoMemory.



Newer generation CryptoMemory adds a new command, *Random Read*, to the existing CryptoMemory command set that has the least significant bit of the I^2C command byte set to logic 1.

While older generation CryptoMemory devices are only partially compliant to the I^2C communication protocol by supporting only *WRITE* type transactions, new generation devices are fully compliant and natively support both *READ* and *WRITE* transactions.

4. The CryptoMemory *Random Read* Command

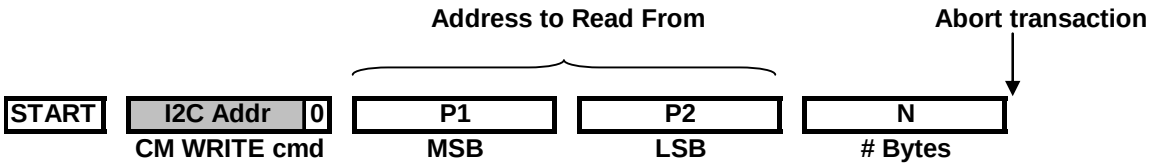
New generation CryptoMemory devices support the command, *Random Read*, in addition to the entire command set of older generation CryptoMemory. *Random Read* is the command that enables full compliance by encoding both the device address and setting the I^2C *R/W* bit to logic 1 to conform to the requirements of the I^2C command structure's *READ* command.

Unlike the rest of CryptoMemory commands that are 4-bytes in length, for efficiency and throughput, the *Random Read* command is only one-byte long. This means quicker initiation of read operations. However, this also requires that the *READ* address would already have been defined within the CryptoMemory device prior to using this command. Initial usage for any given memory segment therefore entails two phases: an *address setup* and a *data read* phase.

4.1. The Address Setup Phase

The address setup phase allows passing of the start address of the CryptoMemory location to read from. It entails use of a CryptoMemory *WRITE* command with the desired *READ* address populated in the command's address fields. The microcontroller sends the four bytes of the command, but instead of following these with data bytes as would be expected, terminates the command. By so doing, the microcontroller has succeeded in clocking the desired address bytes into the CryptoMemory device and the CryptoMemory device's buffers now hold this address information.

Figure 3. The Random Read command address setup phase



The value of N (number of bytes) in this address setup phase is irrelevant for reasons that will be clear soon. The actual number of bytes received during a *random read* transaction depends on the number of bytes the microcontroller acknowledges during the *data read* phase.

Figure 4. Code snippet illustrating the address setup transaction for setting the address to 0x0A.

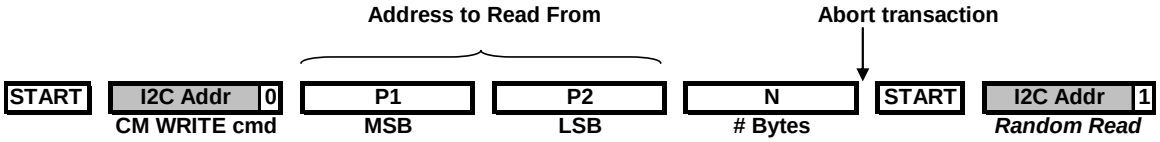
```
/* STEP 1: Setup the command to use a WRITE operation for setting the READ address */
/* Collect the four command bytes into pTxData  buffer*/
pTxData [0] = 0xB4;     // CryptoMemory SYSTEM_WRITE Command byte
pTxData [1] = 0x00;     // Most significant byte of address
pTxData [2] = 0x0A;     // Least significant byte of address (MTZ base in this example)
pTxData [3] = 0x01;     // 'don't care' byte; address has been catured by now

/* STEP 2: Send the command. */
twi_transmit(pTxData,0x04);
```

4.2. The *Data Read* Phase

This phase comprises actual extraction of data from the CryptoMemory device. It entails issuance of the single byte *Random Read* command. After aborting the WRITE transaction of the address setup phase, the microcontroller may immediately send a START signal followed by the *Random Read* command byte.

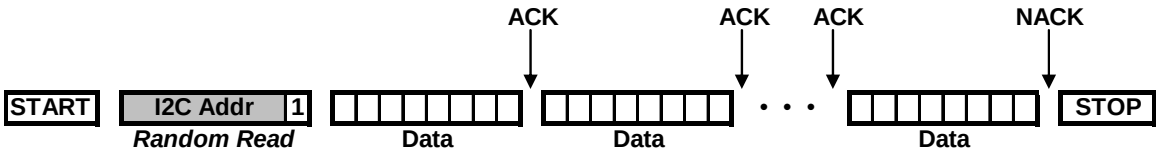
Figure 5. The data read phase of the Random Read command.

Figure 6. The Random Read command showing both address setup and data read phase initiation.



On receiving the *Random Read* command byte, the CryptoMemory device immediately clocks out the byte from the location defined by the address current in its address buffer. It then automatically increments the address value by one so as to point to the next data byte in anticipation of the next action from the microcontroller. If the microcontroller sends a receipt acknowledgement signal (ACK), CryptoMemory will clock out the next byte and again increment the address. CryptoMemory will continue to do so until the microcontroller sends a non-acknowledgement signal (NACK). Use of ACK and NACK signaling explains the non-relevance of the value of the byte count parameter, N, within the *address setup* phase. At this point, CryptoMemory expects the microcontroller to send a protocol STOP signal for formal termination of the transaction.

Figure 7. The Random Read command complete data read phase



It is therefore important that the internal address buffer contains the current address to read from prior to issuing the *Random Read* command. The *address setup* phase establishes this requirement.

Figure 8. Code snippet illustrating the data read transaction for reading two bytes.

```
/* Send CryptoMemory's I2C compliant RANDOM_READ READ command. */
/* Send 0xB1 command byte, receive two bytes and store them in pRxData buffer */
cm_twi_receive(0xB1,pRxData,0x2);
```

If subsequent READ operations are to continue from the last read location, then no additional *address setup* operations are necessary. This provides a big boost in transaction throughput.

The code snippets of figures 4 and 8 illustrate reading from the CryptoMemory device's configuration memory. Other read operations that pertain to CryptoMemory are fuse byte and user memory reads. The next two sections describe this in more detail.

5. Reading the Fuse Byte

The Random Read command takes care of all I2C compliant READ commands with the exception of the CryptoMemory Fuse Read command. To read the current value of the fuse, simply point to any address of the reserved section of the CryptoMemory device's configuration memory and read a single byte. This read operation will return the fuse byte.

Figure 9. Code snippet illustrating a complete Fuse Read transaction.

```
/* STEP 1: Setup the command to use a WRITE operation for setting the READ address */
pTxData [0] = 0xB4;     // CryptoMemory SYSTEM_WRITE command byte
pTxData [1] = 0x00;     // MSB of address
pTxData [2] = 0xFF;     // LSB of address (Any address from the RESERVED memory)
pTxData [3] = 0x00;     // 'don't care' byte; address has been captured by now

/* STEP 2: Send the command. */
twi_transmit(pTxData,0x04);

/* STEP 3: Send CryptoMemory's I2C compliant RANDOM_READ READ command. */
/* Receive the fuse byte and put it in pRxData buffer */
cm_twi_receive(0xB1,pRxData,0x1);
```

6. Reading the User Memory

The Random Read command allows for reading of the configuration memory, fuses, and user memory. Previous sections show how to read configuration memory and the fuse byte. Usage of the Random Read command fundamentally remains the same in both cases and so equally applies to reading of the user memory. One just has to point to the right user memory zone. The following example illustrates this.

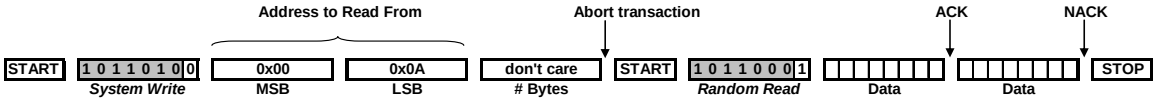
Figure 10. Code snippet illustrating a read of user memory

```
/* STEP 1: Issue a SetUserZone command for zone */
pTxData [0] = 0xB4;          // SYSTEM_WRITE command byte
pTxData [1] = 0x03;          // Set User Zone command code
pTxData [2] = 0x00;          // Zone ID
pTxData [3] = 0x00;          // 'don't care'

/* STEP 2: Send the command */
twi_transmit(pTxData,0x04);

/* STEP 3: Setup the READ address */
pTxData [0] = 0xB0;          // WRITE_USER_ZONE command byte
pTxData [1] = 0x00;          // MSB of address
pTxData [2] = 0x00;          // LSB of address
pTxData [3] = 0x00;          // 'don't care'

/* STEP 4: Send the command. */
twi_transmit(pTxData,0x04);

/* STEP 5: Send CryptoMemory's I2C compliant RANDOM_READ READ command. */
cm_twi_receive(RANDOM_READ, pRxData,0x4);
```

7. Example

As an example, a complete transaction to setup the address and read two bytes from CryptoMemory device's Memory Test Zone (MTZ) having a starting address of 0x0A would look thus:

Figure 11. A complete Random Read command address setup and data read transaction example.



Although all the commands of the command set inherited from the older generation CryptoMemory devices look like I^2C *WRITE* commands and potentially are viable for use in the *address setup* phase, it is important to use *WRITE* rather than *READ* commands for setting up the address. Using a *READ* command will succeed in setting the address but runs the risk of buss contention as CryptoMemory may legitimately also begin to respond.

Figure 12. Code snippet illustrating a complete transaction for reading 10 bytes from CryptoMemory device's configuration memory.

```
/* Read 10 bytes of Configuration Memory starting from address 0x02 */
/* ------------------------------------------------------------------- */
/* STEP 1: Setup the command to use a WRITE operation for setting the READ address */
/* Collect the four command bytes into pTxData  buffer*/
pTxData [0] = 0xB4;     // CryptoMemory SYSTEM_WRITE Command byte
pTxData [1] = 0x00;     // Most significant byte of address
pTxData [2] = 0x02;     // Least significant byte of address
pTxData [3] = 0x00;     // 'don't care' byte; address has been captured by now

/* STEP 2: Send the command. */
twi_transmit(pTxData,0x04);

/* STEP 3: Send CryptoMemory's I2C compliant RANDOM_READ READ command. */
/* Send 0xB1 command byte, receive two bytes and store them in pRxData buffer  */
cm_twi_receive(0xB1,pRxData,0x0A);
```

8. Conclusion

This application note demonstrates how to use CryptoMemory in full I^2C compliant mode. Full I^2C compliance permits use of hardware I^2C buss controllers within any microcontroller for CPU offloading and low firmware footprint advantages. It examines the degree of compliancy by older generation CryptoMemory devices in contrast with full compliance by new generation devices. The application note comes with separate downloadable code examples using the AVR microcontroller hardware I^2C controllers complete with source code and project Makefile files for any AVR microcontroller platforms including Atmel's Aris, Aris+, and Aris++ development kits.

9. Revision History

Doc. Rev.	Date	Comments
8662A	03/2009	Initial document release

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
CryptoMemory@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Requests
www.atmel.com/literature