Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

THIS SPEC IS OBSOLETE

Spec No: 001-14804

Spec Title: AN5074 - IMPLEMENTING INTER-PROCESSOR COMMUNICATION USING CYPRESS MOBL(R) DUAL-PORTS AND THE MAILBOX REGISTERS

Replaced by: NONE

AN5074

Implementing Inter-Processor Communication using Cypress MoBL® Dual-Ports and the Mailbox Registers

Author: Hingkwan Huen

Associated Project: No

Associated Part Family: MoBL® Dual-Port

Software Version: N/A

Related Application Notes: AN5055, AN5036

To get the latest version of this application note, or the associated project file, please visit http://www.cypress.com/go/AN5074.

Cypress MoBL dual port used to have effective communication between the two processors simultaneously using the mailbox registers. MoBL dual port provides a way to interconnect processing elements with different clock frequencies, bus widths, I/O voltages.

Contents

Introduction

The Cypress Semiconductor MoBL® dual-ports provide an ultra low-power, high-bandwidth, flexible solution for the intercommunication of two processing elements.

The MoBL dual-port removes the necessity for processing elements to communicate with a protocol such as I^2C, SPI or UART. In addition, the MoBL dual-port provides a way to interconnect processing elements with different clock frequencies, bus widths, I/O voltages, and at bandwidths in excess of 400 Mbit/s.

The MoBL dual-port is built with features such as power-down mode, upper-byte and lower-byte control, arbitration logic, and mailbox registers with interrupts.

There are several ways of implementing the MoBL dual-port effectively for communication between two processing elements. This application note describes how to interconnect two processing elements using the mailbox registers. Memory partitioning is shown and Pseudo-C code is given as an example of inter-processor communication with the mailbox registers.

Interconnecting Two Processing Elements

Figure 1 is a block diagram showing how to connect processing elements to a shared MoBL dual-port.

Figure 1. Connecting Two Processors Using a Dual Port

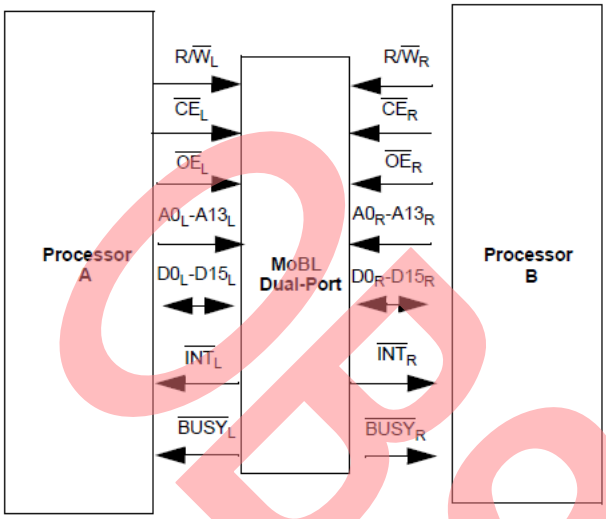
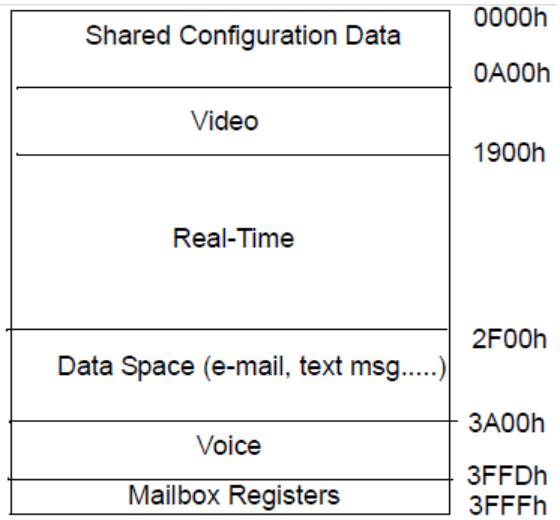


Figure 2. Partitioning of Memory

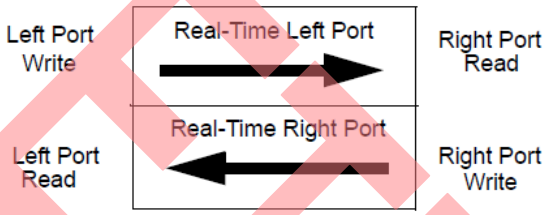


Partitioning Memory

One way to allow easy communication between two processing elements is to have both processing elements agree on a memory partitioning scheme. A memory partitioning scheme puts data into different categories. This is an easy way for both processors to organize memory. When the partitioning scheme has been agreed upon by both processing elements, the mailbox register can be used for message passing that typically includes a location and the length of memory to read. Once again, both processors must agree upon the meaning of the data written into the mailbox register. There are several ways to partition the MoBL dual-port. Figure 2 is an example using the Cypress CYDM256B16 (256k density, 16-bit bus) MoBL dual-port.

To allow for bidirectional communication between the processors, each different partition can also be broken into sections. For instance, each section can be dedicated to a specific direction of transfer. There are several ways to divide a partition of memory and Figure 3 is only an example. In this case there is a half-half split for each port.

Figure 3. Dividing a Partition For Bidirectional Communication



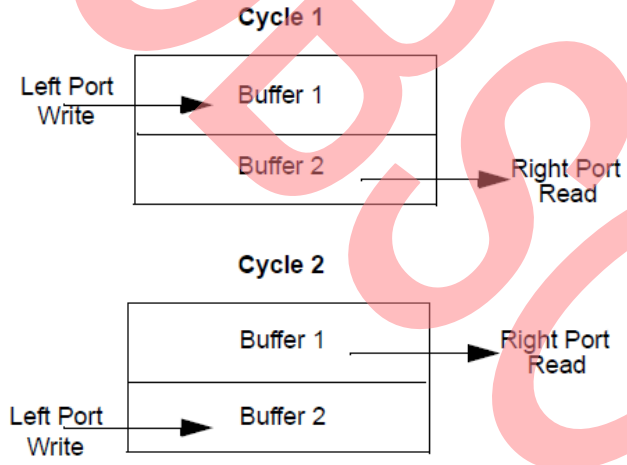
Often times partition size may need to change based on the size of incoming data in order to prevent latency. The mailbox registers can be used to change the partitioning scheme of memory dynamically. As before, the processing elements must agree in software on how to change the partitioning scheme based on the data present in the mailbox register.

Priority can be assigned to different partitions as well. In an application where a cellular phone receives an incoming phone call while a user is sending an email, priority needs to be given to the phone call rather than the email. Sections such as real-time data can be given a higher priority while other sections can have a lower priority.

Additionally, using rotating buffers can further maximize the efficiency of inter-processor communication. By dividing the partitioned memory into two buffers, one processor can write into one buffer while the other processor reads the other buffer. For example, if the left processor has just completed writing into the first buffer, the right processor is eligible to begin reading the data placed in the first buffer. At the same time, the left processor begins writing data into the second buffer. This maximizes the bandwidth since the left processor is constantly writing data at high speed.

Figure 4 is a two cycle example showing a rotating buffer.

Figure 4. Rotating Buffer Example



How to Use the Mailbox Register

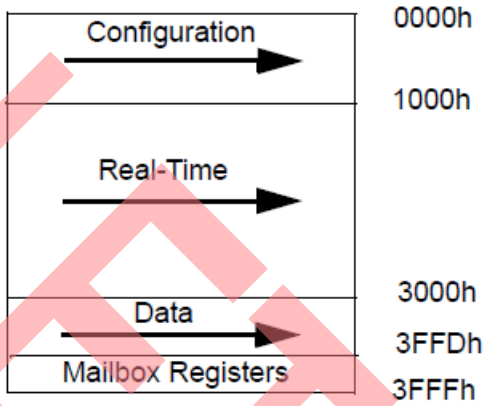
The two highest memory locations of the dual-port are called mailbox registers. The highest memory location is reserved as the mailbox register for the right port, and the second highest memory location is the mailbox register for the left port. When one port writes data to the other port's mailbox register, an interrupt is generated to the owner of that mailbox. After the owner of that mailbox register reads that data, the interrupt is then cleared. Since the mailbox register uses interrupts, both processors must have one of their interrupt request input pins connected to the dual-port. One thing to note is when the dual-port powers up, both interrupts are asserted, so an initialization program must reset both interrupts. The mailbox register is important for message passing from one processor to another.

The mailbox register can be used for efficient passing of bytes/words, and it can be used for more extensive data transfers. For instance, if the left processor needs to send 100 bytes of data to the right processor immediately, the left processor can write into the right processor's mailbox that tells the right processor to read 100 bytes as well as the starting address of this data in the dual-port. In this application, the left processor and right processor must have an agreed mapping scheme of how data written to the mailbox is translated into the starting address, and the length of data (or "buffer size"). Sample programs are written in this application note to show various methods of using the mailbox registers for inter-processor communication.

One-way Communication using Mailbox Registers

Figure 5 is a specific partitioning scheme that will be used in the Pseudo-C code for left to right port communication with the CYDM256B16.

Figure 5. Partition for One-Way Communication



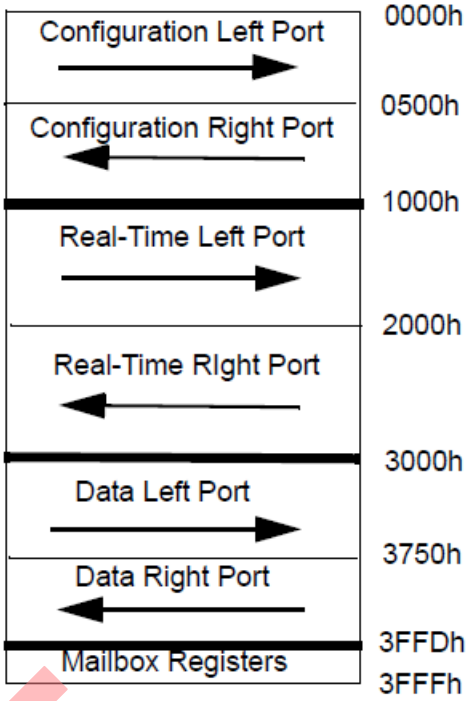
```
//****************************************
//Pseudo-C Code Written for CYDM256B16
//Assume Processor A = left processor
//Assume Processor B = right processor
//This code shows the Processor A writing
30 words to the data buffer/partition and
then Processor B will fetch those words.
//Processor A writes all its data to a
buffer, then write to the mailbox register
which generates an interrupt. Processor B
then reads this data in the mailbox
register which clears the interrupt. The
mailbox data that Processor B reads is
mapped to an address at which to read the
data, and the length of data to read. This
is agreed upon by both processors in
software.
//****************************************
//PROTOTYPES
int mapping_scheme(int address_start, int
buffer_length);
int unmapping_scheme_address(int
mapped_word);
int unmapping_scheme_length(int
mapped_word);

//MAIN PROGRAM
void main()
{
//First write data to allocated buffer,
write using CE controlled
//timing
//Write into data buffer which starts at
0x3000h shown in
//partition, then write until the user
specified buffer length

int address_start = 0x3000; // this
corresponds to the data buffer starting
add.
int buffer_length = 30; //arbitrary 30
words
int mapped_word; // will be used later to
read in word from mailbox register
//Processor A code
for(i = address_start; i < address_start +
buffer_length; i++)
{
          if(busy_flagL = 1) // check busy
flag before start writing
{
Write_Address(i);
R/W_L = 0;
CE_L = 0;
delay(15 ns); //make sure address is
present 35 ns
Write_Word(data);//strobe in data of users
choice
delay(20ns);//make sure data present for 20
ns w/ CE asserted
CE_L = 1;
R/W_L = 1;
}
}
//Now Have Processor A write data to
mailbox register
//To generate interrupt, and let Processor
B know data is ready
if(busy_flag_L = 1) // check busy flag for
mailbox register(should not be 0)
{
   Write_Address(0x3FFF);//Mailbox
location for right port CYDM256B16
R/W_L = 0;
CE_L = 0;
delay(15ns);
Write_Word(mapping_scheme(address_start,
buffer_length);
//INTERRUPT(IR2) is now generated
delay(20ns);
CE_L = 1;
R/W_L = 1;
}
//Processor B Interrupt Routine:
//Sample ISR Routine
CE_R = 0;
RW_R = 1;
OE_R = 0;
Write_Address(0x3FFF); //location of right
mailbox register for CYDM256B16
mapped_word = Read_Word(); //INTERRUPT(IR2)
is now cleared
address_start =
unmapping_scheme_address(mapped_word);
buffer_length =
unmapping_scheme_length(mapped_word);
for(i = address_start; i < address_start +
buffer_length; i++)
{
Write_Address(i);
Read_Data(); //store in register/use at
user's discretion
}
} // end of main

//FUNCTIONS
int mapping_scheme(int address_start, int
buffer_length)
```

```
{
//User's mapping scheme of word for both
processors
//should include starting location and
length of data to read
return mapped_word;
}

int unmapping_scheme_address(int
mapped_word)
{
//Take mapped word and unmap into address
based on user scheme
return address_start;
}

int unmapping_scheme_length(int
mapped_word)
{
//Take mapped word and unmap into buffer
length based on user scheme
return buffer_length;
}
```

Simultaneous Bidirectional Communication using Mailbox Registers

There are several methods of implementing bidirectional communication between two processing elements using the MoBL dual-port. Figure 6 is one example that shows Processor A writing to the dual-port while Processor B is simultaneously writing to another location in the dual-port. When either Processor A or B is first done writing, it writes to the mailbox register and let the other processor know that data is ready to be read. The previous partition scheme is used in this example with the CYDM256B16.

Figure 6. Partition for Bidirectional Communication



```
/*****************************************
//Pseudo-C Code Written for CYDM256B16
//Interfacing processors with MoBL dual-
port.
//Assume Processor A = left processor.
//Assume Processor B = right processor.
//There is bidirectional communication
between Processor A and Processor B.
//Processors A and B both begins by writing
words into their respective buffers within
the Real-Time partition. After each
Processor finishes their write operation,
it generates an interrupt to let the other
Processor know where the data is stored.
The interrupted Processor halts all current
operations and read that data in. Only
after reading this data will the Processor
return to its previous operations.

//*************************************

//PROTOTYPES

int mapping_scheme(int address_start, int
buffer_length);
int unmapping_scheme_address(int
mapped_word);
int unmapping_scheme_length(int
mapped_word);
//MAIN PROGRAM

void main()
{
//Processor A code:
int address_start_L = 0x1000;
int buffer_length_L = 50;

//These values are for reading dating from
mailbox, and the mapped addresses and
buffer lengths
int read_buffer_length_L;
int read_address_start_L;
int mapped_word;

for(i = address_start_L; i < buffer_length_L
+ address_start_L; i++)
{
if(busy_flag_L = 1)
{
Write_Address(i);
R/W_L = 0;
CE_L = 0;
delay(15 ns); //make sure address is
present 35 ns
Write_Word(data);//strobe in data of users
choice
delay(20 ns);//make sure data present for
20 ns w/ CE asserted
CE_L = 1;
R/W_L = 1;
}
}

//After Processor A has finished writing
its code, it sends a mapped word to the
mailbox register. This will generate an
interrupt(IR2) and in this case Processor B
stops whatever function it is currently
executing
if(busy_flagL = 1) // check busy flag for
mailbox register(should not be 0)
{
   Write_Address(0x3FFF);//Mailbox
location for CYDM256B16
R/W_L = 0;
CE_L = 0;
delay(15 ns);
Write_Word(mapping_scheme(address_start_L,
buffer_length_L);
//INTERRUPT(IR2) is now generated
delay(20 ns);
CE_L = 1;
R/W_L = 1;
}

Processor A Interrupt Routine:

//Sample ISR Routine occurs when
INTERRUPT(IR1) is generated. Processor A
will halt current operations until the ISR
routine is completed. Make sure to "push"
onto the stack all registers that related
to the code that was being written before
entering ISR. The following is a sample ISR
Routine w/o push and pop
CE_L = 0;
RW_L = 1;
OE_L = 0;
Write_Address(0x1FFE); //Mailbox register
for left port of CYDM256B16
mapped_word = Read_Word(); //INTERRUPT(IR1)
is now cleared
read_address_start_L =
unmapping_scheme_address(mapped_word);
read_buffer_length_L =
unmapping_scheme_length(mapped_word);
//Now Processor A can read the mailbox
register, and finish reading in the data
```

```
//Sent from Processor B
for(i = read_address_start_L; i <
read_buffer_length_L + read_address_start_L;
i++)
{
Write_Address(i);
Read_Data(); //store in register/use at
user's discretion
}
//make sure to "pop" all values off the
stack
//RETI (Return Interrupt)
//Processor A will now return to its
previous actions before it was interrupted


//Processor B code

int address_start_R = 0x2000;
int buffer_length_R = 100;

//These values are for reading dating from
mailbox, and the mapped addresses
//and buffer lengths
int read_buffer_length_R;
int read_address_start_R;
int mapped_word;

for(i = address_start_R; i < buffer_length_R
+ address_start_R; i++)
{
if(busy_flag_R = 1)
{
Write_Address(i);
R/W_R = 0;
CE_R = 0;
delay(15 ns); //make sure address is
present 35 ns
Write_Word(data);//strobe in data of users
choice
delay(20 ns);//make sure data present for
20 ns w/ CE asserted
CE_R = 1;
R/W_R = 1;
}
}

// After Processor B has finished writing
its code, it sends a mapped word to the
mailbox register. This will generate an
interrupt(IR1) and in this case Processor A
stops whatever function it is currently
executing

if(busy_flag_R = 1) // check busy flag for
mailbox register(should not be 0)
{
   Write_Address(0x1FFE);//Mailbox
register for left port of CYDM256B16
R/W_R = 0;
CE_R = 0;
delay(15 ns);
Write_Word(mapping_scheme(address_start,buf
fer_length);
//INTERRUPT(IR1) is now generated
delay(20 ns);
CE_R = 1;
R/W_R = 1;
}

Processor B Interrupt Routine:

//Sample ISR Routine occurs when
INTERRUPT(IR2) is generated. Processor B
halts current operations until the ISR
routine is completed. Make sure to "push"
onto the stack all registers that related
to the code that was being written before
entering ISR. The following is a sample ISR
Routine w/o push and pop
CE_R = 0;
RW_R = 1;
OE_R = 0;
Write_Address(0x3FFF); //location of right
mailbox register for CYDM256B16
mapped_word = Read_Word(); //INTERRUPT(IR2)
is now cleared
read_address_start_R =
unmapping_scheme_address(mapped_word);
read_buffer_length_R =
unmapping_scheme_length(mapped_word);

for(i = read_address_start_R; i < read_
buffer_length_R + read_address_start_R; i++)
{
Write_Address(i);
Read_Data(); //store in register/use at
user's discretion
}
//make sure to "pop" all values off the
stack
//RETI (Return Interrupt)
//Processor B will now return to its
previous actions before it was interrupted


}//end of main

//FUNCTIONS
```

```
int mapping_scheme(int address_start, int
buffer_length)
{
//User's mapping scheme of word for both
processors
//should include starting location and
length of data to read
return mapped_word;
}
int unmapping_scheme_address(int
mapped_word)
{
//Take mapped word and unmap into address
based on user scheme
return address_start;
}
int unmapping_scheme_length(int
mapped_word)
{
//Take mapped word and unmap into buffer
length based on user scheme
return buffer_length;
}
```

For further clarification, assume the following scenario occurs. Processor A and Processor B are both writing data in to their respective buffers at the same time. The following string of events would occur if Processor A completed its Write Operation before Processor B:

1. Processor A writes data to Mailbox Register B, and in this process, an interrupt (IR2) is generated.
2. Processor B receives the interrupt and halts all current operations.
3. Processor B reads the data at Mailbox Register B and clears the interrupt. Processor B now reads the data sent from Processor A.
4. Processor B now continues its Write Operation, and once it finishes, it writes data to Mailbox Register A. An interrupt (IR1) will be generated.
5. Processor A receives the interrupt and halts its current operations.
6. Processor A reads the data at Mailbox Register A and clears the interrupt. Processor A now reads the data sent from Processor B.

Summary

Using the mailbox registers of a MoBL dual-port is an effective, high-bandwidth, lower-power method of implementing inter-processor communication. Partitioning can be done within the dual-port memory as a method of organization, and each partition can be separated into different sections to allow for simultaneous bidirectional communication

For further information, please visit the Cypress web site at www.cypress.com. The web site also provides the latest data sheets, models and any related documentation.

References

1. CYM256B16, CYDM128B16, CYDM064B16, CYDM128B08, CYDM064B08, 1.8 V 4K/8K/16K x 16 and 8K/16K MoBL® Dual-Port Static RAM

Document History

Document Title: AN5074 - Implementing Inter-processor Communication using Cypress MoBL® Dual-Ports and the Mailbox Registers

Document Number: 001-14804

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	989740	ODC	04/26/2007	New Application Note.
*A	3160456	ODC	02/11/2011	Added Additional Resources (Added application notes and provided links to application notes). Added Document History Page. Updated to new template.
*B	3233251	ODC	04/18/2011	No technical updates. Completing Sunset Review.
*C	4393161	PRVE	05/29/2014	Updated to new template. Completing Sunset Review.
*D	5849670	RAJV	08/10/2017	Obsolete document. Completing Sunset Review.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com