Application Report
SLUA758 – October 2015

UCD3138 – Responding to Multiple PMBus Slave Addresses

Jack Tan, Ian Bower *High Voltage Power Solution*

ABSTRACT

The use of digital power controllers is becoming mainstream in isolated power supplies for achieving higher system integration, enabling advanced control methods, and implementing real-time metrology and communication. The Power Management Bus (PMBus) is a commonly used protocol which supports communication between power converters, host processors and other related peripherals in a digital power-management ecosystem. The UCD3138 family of digital controllers feature a flexible and powerful PMBus interface integrated in the device for this purpose. While the controller is designed to respond to a single PMBus address, some applications have a need for the controller to support more than one PMBus address. This application note provides a firmware solution to make a single UCD3138 device support multiple PMBus addressees. The document covers the high-level approach and provides a suggested firmware structure, along with the detailed firmware code changes list. This approach also works on other devices in the UCD3138 family of products such as UCD3138A, UCD3138064, UCD3138128 and so on.

Contents

Figures

1 Introduction

The Power Management Bus (PMBus) is an open standard protocol that defines a means of communicating with power conversion and other devices.

In some power supply applications, there are 2 or more digital controllers or MCUs. One of them is used to control the power converter, and others are added for monitoring or housekeeping. All the digital controllers are able to communicate with host via PMBus or I2C, and each of devices has its own slave address.

The UCD3138 has powerful peripherals, and it can do both monitoring and controlling of the power converter. Please see Figure 1, here is the example of using a single UCD3xxx to replace 2 or more digital controllers. In order to be compatible with original PMBus or I2C host, UCD3xxx should be able to respond at least more than 2 slave addresses.

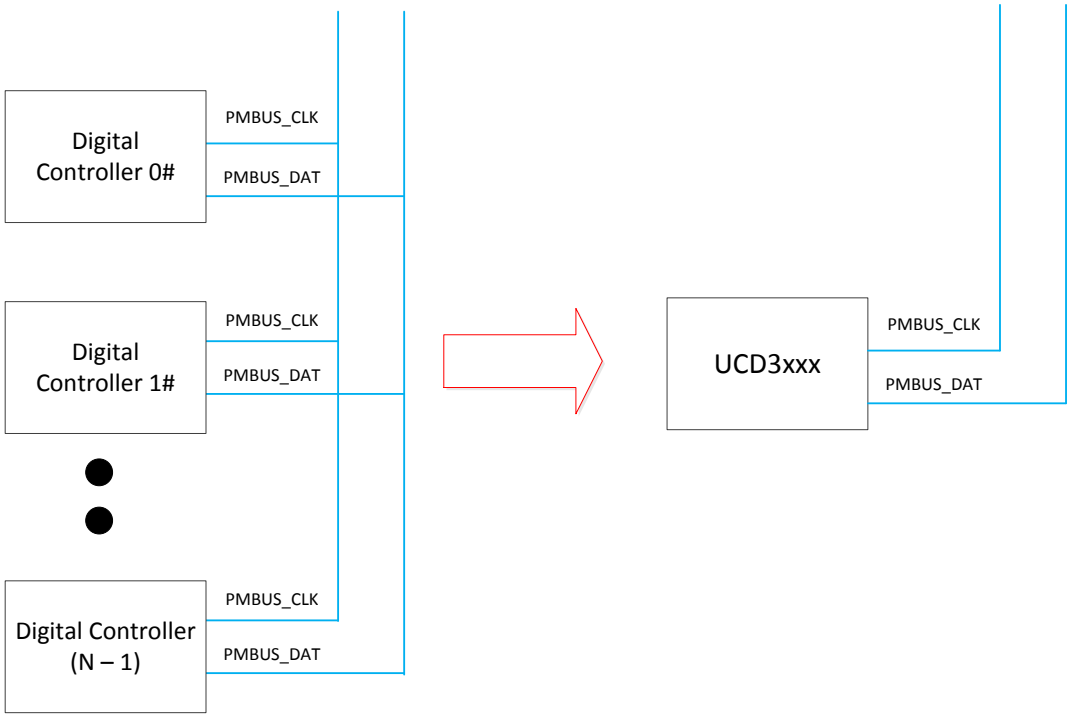

Figure 1. Use Single UCD3138 to Replace 2 or More Devices

The current PMBus solution only supports response to a single address. This application note describes how to use the UCD3138 to respond to multiple addresses and introduce the PMBus firmware structure. This solution can be also applicable to UCD30xx and UCD3138 refresh silicon.

2 Overview of the Approach

As previously described, a single UCD3138 should respond to multiple addresses, and the addresses may not sequential, so the PMBus hardware could not meet this requirement. Here we propose using manual acknowledge to each PMBus address and manual acknowledge to command. Manual acknowledge to address means that firmware were used to ack valid PMBus address and command instead of using hardware to do automatic ack.

2.1 Handling Write Sequence

For most standard write sequences, there is only one address, and others are data. Code is implemented to identify the valid PMBus address when the address ready bit was set. So it is easy to implement in firmware.

Figure 2. Standard PMBus Write Sequence

2.2 Handling Read Sequence

There is a little bit more difficulty for reading PMBus command sequence than write sequence, since the command is in the middle of PMBus address write and PMBus address read. In the UCD3xxx family, the polling method is recommended since it is simple and leaves more time available for interrupt. Suppose that using automatic ack to command, and further suppose that the background loop is not fast enough, there is a possibility that the PMBus handler is executed after both command and address are ready. That makes it hard to identify which is command and which one is address. In order to handle the read sequence, manual acknowledge to command is used. That means only the slaver got the command, then master can send out the slave address plus read.

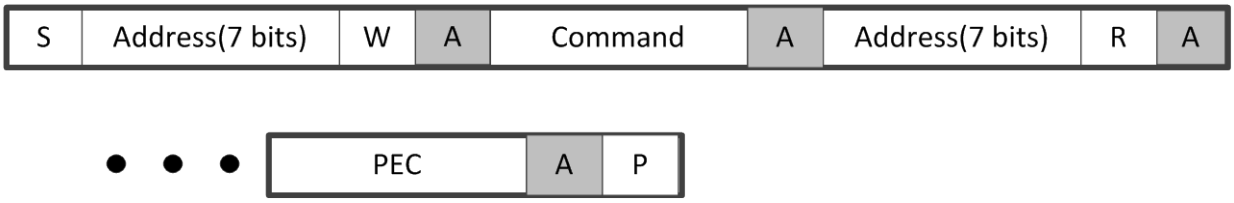

Figure 3. Standard PMBus Read Sequence

3 PMBus Low-Level Firmware Structure

The PMBus Low-Level firmware was targeted to handle data from or to host. It will depend on PMBus status Register, then change state machine and how to process data. The firmware, supported multi-address responds, does contain 7 states. Each state has different responsibility.

- **Idle state** – When does the STOP condition occur, or some fault which causes PMBus hardware reset, the firmware will be returned to this state. This is beginning of PMBus handle data. And the valid address is acked in this state.
- **Command ack state** - If the address + write is valid in idle state, then the state will change to this one, and the valid command will be acked.
- **Wait for state** - This state is from the original idle state, due to the manual ack to slave address, this handler has a tiny difference with the original idle state code.
- **Read state** - If the wait for state find there is address + read, this state will be the next subroutine in PMBus state machine. The data will be transferred from UCD to host in this state.
- **Read Block state** – This state is similar with read state. But the condition of entering into this state is that the host request more data than the read handler can operate, it just continues to send data to host.
- **Write block state** – If the write number of bytes is more than wait for state can handle, this state is executed, and it continues to handle the write data from host.
- **Wait for EOM state** – This sate handler is the continuous state for read sequence. The firmware should wait for the host sends the stop condition before resetting the state to idle state.

3.1 PMBus Idle State Structure

In this state, just check if there is a valid address ready or not.

- If the slave address ready bit was set
  - If it is the valid address, ACK it and change the state to command ack handler.
  - If there is an invalid address ready, NACK it and stay in idle state handler.
- Otherwise, do nothing, just keep staying in this sate

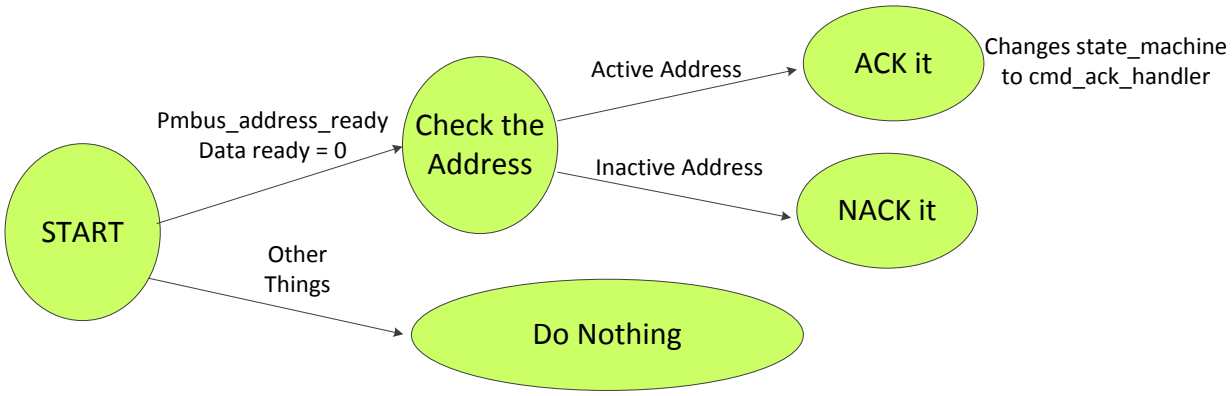

Figure 4. Idle State Handler Structure

3.2 PMBus Command ack State Structure

In this state:

- If nothing happened, still stay in this handler
- If there is a data ready, ack to the command and change the state to wait handler
- If the stop condition happened, that must be a quick command, then go back to idle state directly
- If none of the previous 3 conditions, goes back to idle state

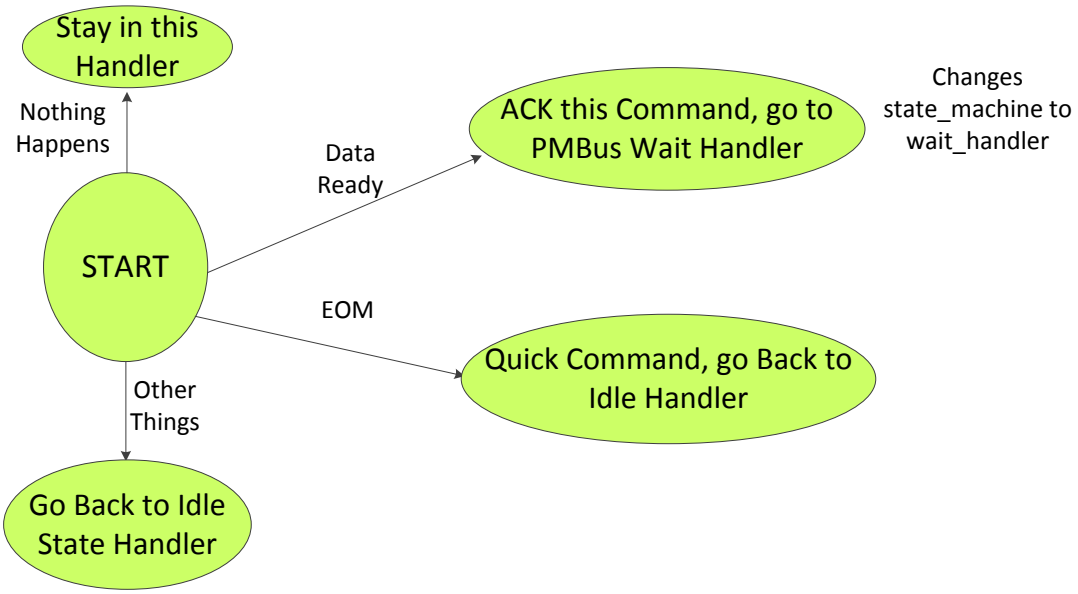

Figure 5. Command ack State Structure

3.3 PMBus Wait for Handler

This state is from the original idle state, due to the manual ack to slave address, this handler has a tiny difference with the original idle state code:

- If nothing happened, just stay in this handler
- If the PEC and stop condition bit were set, that must be write byte or word. Process it directly and goes back to idle state.
- If only the data ready bit is set, and the byte counter is 4, that means the buffer is full and it needs to transfer multiple times. That must be a write block, change the state machine to write block state.
- If the slave address bit is set, that must be a read sequence.
  - If there is a valid address, ack to this address and it goes to PMBus read handler
  - Otherwise, NACK to this address and it goes back to idle state
- If none of the previous things happen, reset firmware and it goes back to idle directly.

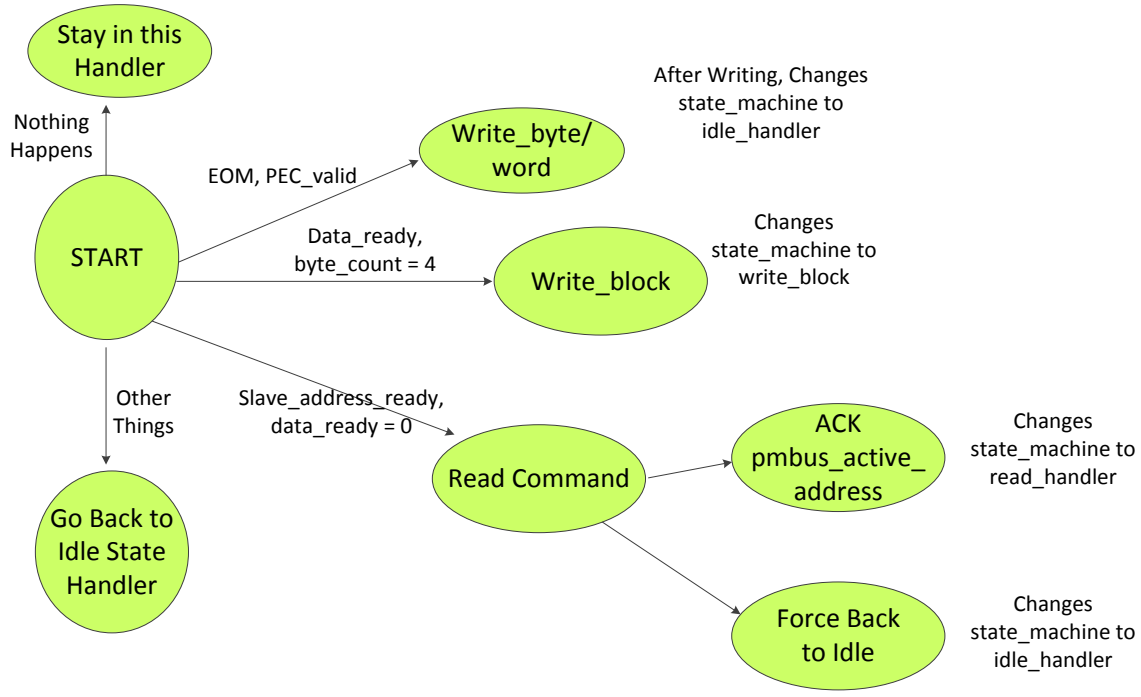

Figure 6. Wait for State Handler Structure

3.4 PMBus Read Handler Structure

In this state:

- If nothing happened, just stay in this state.
- If Data request bit was set and the number of bytes of this command is less than 5, then transmit the data to master directly, and change the state to wait of end of message state.
- If Data request bit was set and the number of bytes of this command is larger than 5, transmit the first 4 bytes and change the state to read block state for continuing transfer.
- If none of the previous things happened, then it goes back to idle state

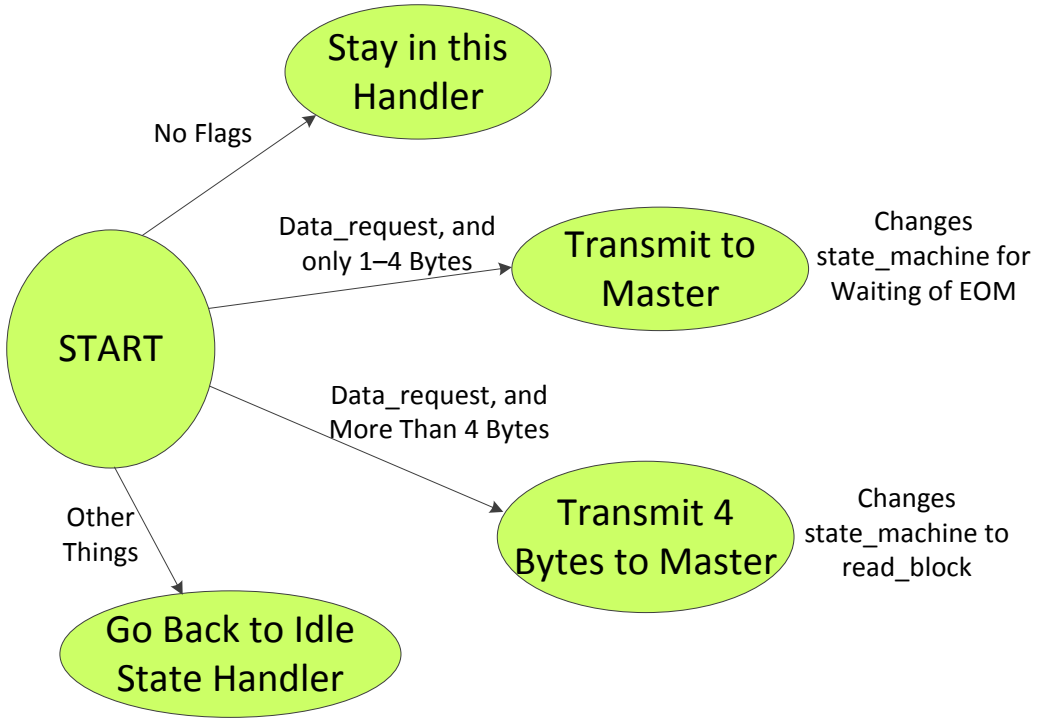

Figure 7. Read Handler Structure

3.5 PMBus Read Block Handler

The read block handler is very similar to the read handler, it continues to transfer the data to master.

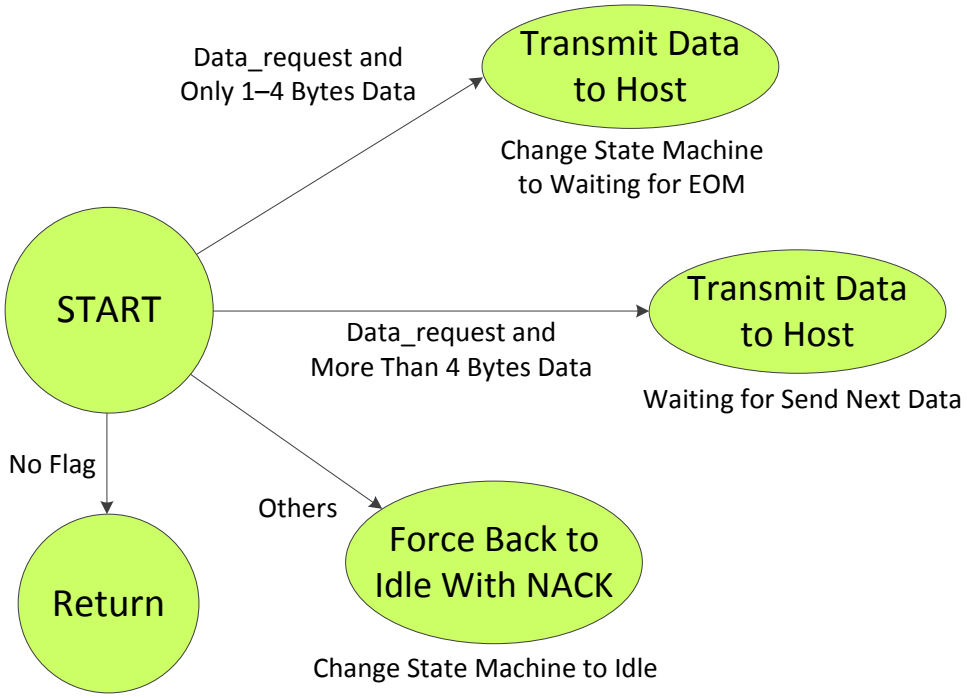

Figure 8. Read Block Handler Structure

3.6 PMBus Write Block Handler

If there are more than 4 bytes for write sequence, the state will enter into this handler:

- No flag was set, just stay in this handler.
- If EOM, Data ready and PEC valid bit are set, that means write block is finished, process this command and it goes back to idle.
- If EOM was set, process this command and go back to idle.
- If data ready was set, RD_BYTE_COUND is 4, and NO EOM, that means the receive data from master is still required.
  - If the total number of bytes is less than the max buffer size, stay in this state and continuing to receive data
  - Otherwise, stop to receive the data and force the sate machine back to idle
- If none of the previous things occurred, then force the state machine back to idle state.

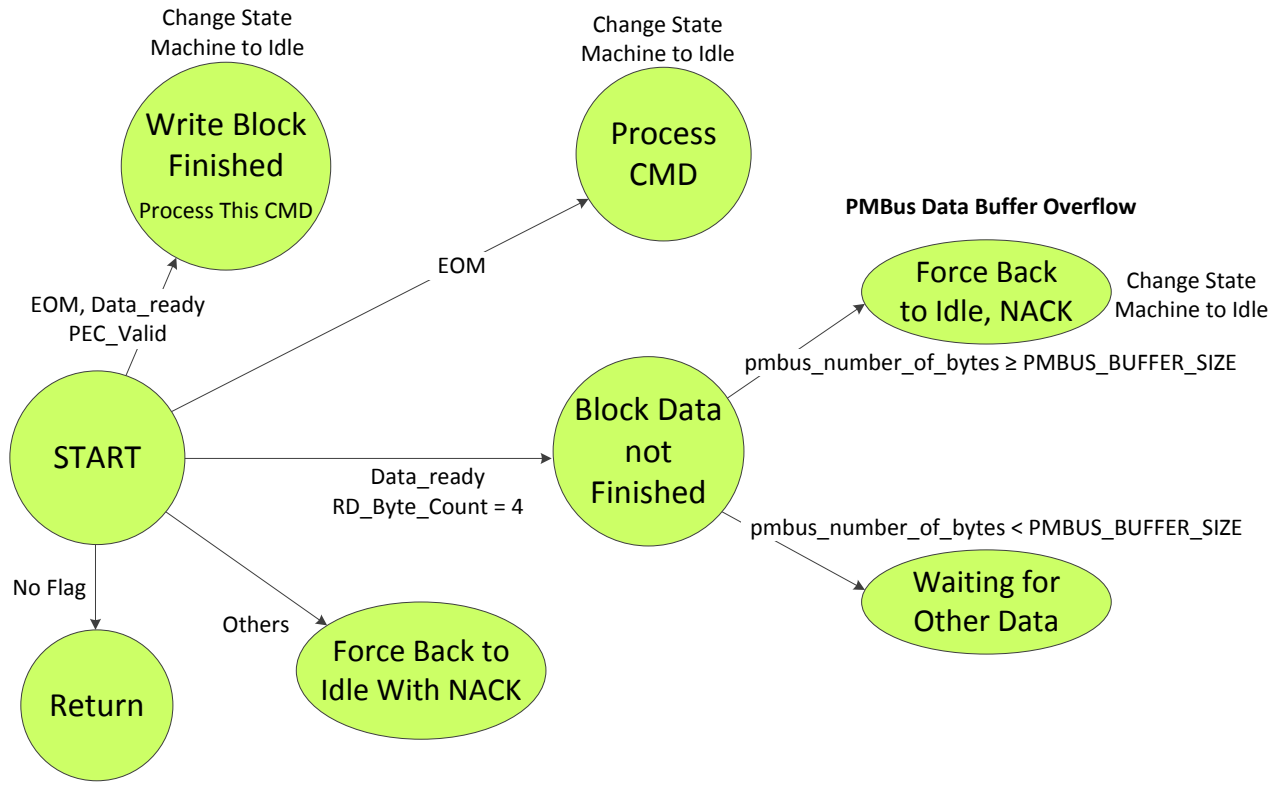

Figure 9. Write Block Handler Structure

3.7 PMBus Wait for EOM Handler

This state handler is the continuous state for read sequence:

- If the stop condition bit is set, then goes back to idle state
- If nothing happened, stay in this state
- If other things, such as clock low time out, goes back to idle state

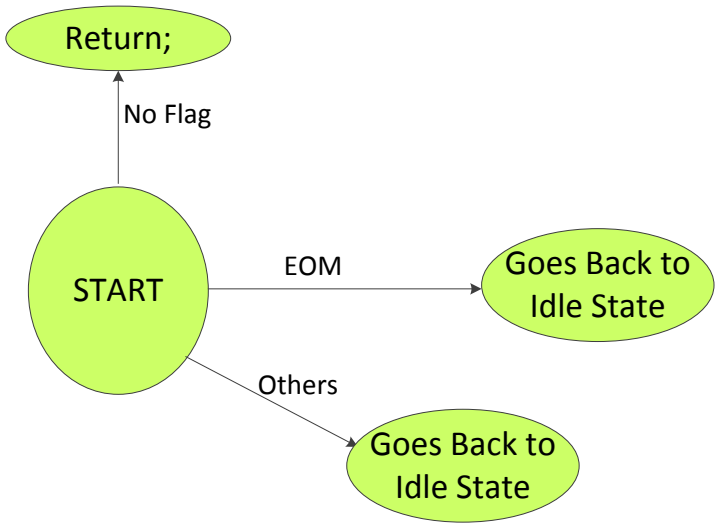

Figure 10. End of Message Handler Structure

4 Changes to the Existing Code to Make it Respond to Multiple Slave Addresses

4.1 Implement Other 3 State Machines in the Existing Code

- #define PMBUS_STATE_WAIT 4
- #define **PMBUS_STATE_CMD_ACK** 5
- **#define PMBUS_STATE_READ 6**

4.2 Implement the Address You Would Like to Respond

- #define PMBUS_ADDRESS1 0x01
- #define PMBUS_ADDRESS2 0x22
- #define PMBUS_ADDRESS3 0x59
- #define PMBUS_ADDRESS4 0x7b

4.3 Define a New Variable to Store the PMBus Address and PMBus Status Registers

```
EXTERN Uint32 pmbus_active_address;
EXTERN union PMBST_REG pmbst_shadow;
```

4.4 Changes to Initialization PMBus

UCD should be able to manually acknowledge to multiple addresses, the initialization code should be changed from auto acknowledge address to manual acknowledge address. And considering the PMBus read sequence, UCD should be also configured as manual acknowledge to command.

```
PMBusRegs.PMBCTRL2.all = PMBCTRL2_HALF0_PEC_ENA + pmbus_address
                       + PMBCTRL2_HALF0_SLAVE_ADDRESS_MASK_DISABLE
                       + PMBCTRL2_ALL_RX_BYTE_ACK_CNT
                       + PMBCTRL2_ALL_MAN_SLAVE_ACK
```

```
+ PMBCTRL2_ALL_MAN_CMD; // for Manual slave address ack
```

4.5 Changes to pmbus_handler.c.

As previously described, the manual acknowledge address and command will be used in this part. So the related PMBus handler part also should be modified.

a. Implement 3 subroutines at PMBus low-level routine.

```
else if (pmbus_state == PMBUS_STATE_CMD_ACK)
{
                pmbus_cmd_ack_handler();
                return ;

}
else if (pmbus_state == PMBUS_STATE_WAIT)
{
                pmbus_wait_handler();
                return ;

}
else if (pmbus_state == PMBUS_STATE_READ)
{
                pmbus_read_handler();
                return ;
}
else
{
                pmbus_state = PMBUS_STATE_IDLE;
                pmbus_idle_handler();
                return ;
}
```

b. For pmbus_idle_handler, the main job is to check the valid PMBus address if there is an address ready.

```
void pmbus_idle_handler(void)
{
        pmbst_shadow.all=PMBusRegs.PMBST.all;

        if( (pmbst_shadow.bit.SLAVE_ADDR_READY == 1) && (pmbst_shadow.bit.DATA_READY == 0) )
                {
                        pmbus_active_address = (PMBusRegs.PMBRXBUF.byte.BYTE0 & 0x7F);
                        if((pmbus_active_address == PMBUS_ADDRESS1)
                                        || (pmbus_active_address == PMBUS_ADDRESS2)
                                        ||(pmbus_active_address  == PMBUS_ADDRESS3)
                                        ||(pmbus_active_address  == PMBUS_ADDRESS4))
                        {
```

```
                                    PMBusRegs.PMBACK.bit.ACK=1;
                                    pmbus_state= PMBUS_STATE_CMD_ACK;

                        }
                        else
                        {
                                    PMBusRegs.PMBACK.bit.ACK=0;
                                    pmbus_state = PMBUS_STATE_IDLE;

                        }
            }
            return;
}
```

c. Add ***void pmbus_cmd_ack_handler(void)*** function to acknowledge command manually.

```
void pmbus_cmd_ack_handler(void)
    {
            pmbst_shadow.all=PMBusRegs.PMBST.all;

            if( (pmbst_shadow.all & (PMBST_HALF0_CHECK_BITS +
        PMBST_BYTE0_RD_BYTE_COUNT)) == 0) //if there is nothing happen
            {
                    return;
            }

            if(pmbst_shadow.bit.DATA_READY==1)
            {
                pmbus_buffer[0] = PMBusRegs.PMBRXBUF.byte.BYTE0;// copy command

                PMBusRegs.PMBACK.byte.BYTE0 = 1;  //ack them
                pmbus_state = PMBUS_STATE_WAIT ;
            }

            else if (pmbst_shadow.all & (PMBST_HALF0_CHECK_BITS+
        PMBST_BYTE0_RD_BYTE_COUNT) == PMBST_BYTE0_EOM) //quick command
            {
                    pmbus_state= PMBUS_STATE_IDLE ;
            }

            else
            {
                    pmbus_state= PMBUS_STATE_IDLE ;
            }
            return ;

    }
```

d. In pmbus_wait_handler() is from the original pmbus_idle_handler()

i. Use pmbst_shadow.all to replace int32 pmbus_status = PMBusRegs.PMBST.all

```
pmbst_shadow.all=PMBusRegs.PMBST.all;
pmbus_status_half_word_0_value = pmbst_shadow.all & 0xFFFF ;
```

ii. Write command changes

```
if((pmbus_status_half_word_0_value & (PMBST_HALF0_CHECK_BITS )) ==
            (PMBST_BYTE0_EOM + PMBST_BYTE0_DATA_READY +
PMBST_BYTE0_PEC_VALID))
            //end of message, good data ready, pec valid, must be 2 to 4 bytes
    {
        //copy all 4 over just to make sure
        pmbus_buffer[1] = PMBusRegs.PMBRXBUF.byte.BYTE0;
```

```
                pmbus_buffer[2] = PMBusRegs.PMBRXBUF.byte.BYTE1;
                pmbus_buffer[3] = PMBusRegs.PMBRXBUF.byte.BYTE2;
                pmbus_buffer[4] = PMBusRegs.PMBRXBUF.byte.BYTE3;
        pmbus_number_of_bytes = ( (pmbus_status_half_word_0_value &
PMBST_BYTE0_RD_BYTE_COUNT) + 1 );
                pmbus_write_message();
                //PMBusRegs.PMBACK.byte.BYTE0 = 1;  //ack them
                pmbus_state = PMBUS_STATE_IDLE;
                return;
        }
        iii. Write block request changes
                else if((pmbus_status_half_word_0_value & (PMBST_HALF0_CHECK_BITS +
PMBST_BYTE0_RD_BYTE_COUNT)) ==
                        (4 //4 is read byte count
                         + PMBST_BYTE0_DATA_READY))
                        //no end of message, good data ready, 4 bytes of data - must be a send block
        {
                //copy data into buffer, set up to receive more.
                pmbus_buffer[1] = PMBusRegs.PMBRXBUF.byte.BYTE0;
                pmbus_buffer[2] = PMBusRegs.PMBRXBUF.byte.BYTE1;
                pmbus_buffer[3] = PMBusRegs.PMBRXBUF.byte.BYTE2;
                pmbus_buffer[4] = PMBusRegs.PMBRXBUF.byte.BYTE3;
                pmbus_number_of_bytes = 5 ; //start counting bytes.

                PMBusRegs.PMBACK.byte.BYTE0 = 1;  //ack first 4 bytes;
                pmbus_state = PMBUS_STATE_WRITE_BLOCK;
        }
    iv. Delete read request and add ack to valid address to this function
    else if( (pmbst_shadow.bit.SLAVE_ADDR_READY == 1) && (pmbst_shadow.bit.DATA_READY==0) ) //ack
pmbus address
    {
        pmbus_active_address = (PMBusRegs.PMBRXBUF.byte.BYTE0 & 0x7F);

        if(         (pmbus_active_address == PMBUS_ADDRESS1)
                        ||(pmbus_active_address == PMBUS_ADDRESS2)
                        ||(pmbus_active_address == PMBUS_ADDRESS3)
                        ||(pmbus_active_address == PMBUS_ADDRESS4) )
                {

                        PMBusRegs.PMBACK.bit.ACK=1;
                        pmbus_state = PMBUS_STATE_READ;
                        return;


                }
                else
                {
```

```
                        PMBusRegs.PMBACK.bit.ACK=0;
                        pmbus_state = PMBUS_STATE_IDLE;
                        return;
                }
```

e. Add new ***void pmbus_read_handler(void),*** in addition, please also set the manual command bit in PMBCTRL2 while writing to the register for sending data to host.

```
void pmbus_read_handler(void)
{
        int32 i; //loop counter
        union
        {
                Uint32 pmbus_transmit_load;
                Uint8 pmbus_transmit_bytes[4];
        }
         u;

        pmbst_shadow.all=PMBusRegs.PMBST.all;

        pmbus_status_half_word_0_value = pmbst_shadow.all & 0xFFFF ; //assign to temporary value
        pmbus_status_half_word_0_value_ored = pmbus_status_half_word_0_value_ored |
pmbus_status_half_word_0_value;

        if((pmbus_status_half_word_0_value & (PMBST_HALF0_CHECK_BITS +
PMBST_BYTE0_RD_BYTE_COUNT)) == 0)
        {//if no activity on PMBus, do nothing

                return ;
        }

        else if ( (pmbus_status_half_word_0_value & (PMBST_HALF0_CHECK_BITS +
PMBST_BYTE0_RD_BYTE_COUNT)) ==
                PMBST_BYTE0_DATA_REQUEST)//
        {

                pmbus_read_message();

                if(pmbus_number_of_bytes == 0) //0 shows it is not a valid command, so
                {
                        PMBusRegs.PMBACK.byte.BYTE0 = 1;  //nack them
                        pmbus_state= PMBUS_STATE_IDLE ;
                        return;
                }

                else if(pmbus_number_of_bytes < 5) //here if it all fits in one rxbuf
```

```
                        {
//                                  PMBusRegs.PMBACK.byte.BYTE0 = 1;  //ack command;

                                    PMBusRegs.PMBCTRL2.byte.BYTE2 = PMBCTRL2_BYTE2_TX_PEC +
pmbus_number_of_bytes + PMBCTRL2_BYTE2_MAN_CMD +
PMBCTRL2_BYTE2_RX_BYTE_ACK_CNT;                 //transmit with pec, desired number of bytes.

                                        for(i = 0;i < pmbus_number_of_bytes;i++)
                                        {
                                                u.pmbus_transmit_bytes[3-i] = pmbus_buffer[i]; //big
endian stuff - great fun
                                        }

                                        PMBusRegs.PMBTXBUF.all = u.pmbus_transmit_load; //this
sends message

                                        PMBusRegs.PMBACK.byte.BYTE0 = 1;  //ack command;

                                        pmbus_state= PMBUS_STATE_READ_WAIT_FOR_EOM;

                                        //pmbus_state = PMBUS_STATE_ACK;

                                        return ;
                        }
                        else //here if we have to do multiple rxbuf loads - do first one, set up for more
                        {

                                        PMBusRegs.PMBCTRL2.byte.BYTE2 =  4 +
PMBCTRL2_BYTE2_RX_BYTE_ACK_CNT + PMBCTRL2_BYTE2_MAN_CMD;

                                        for(i = 0;i < 4;i++)
                                        {
                                                u.pmbus_transmit_bytes[3-i] = pmbus_buffer[i]; //big
endian stuff - great fun
                                        }

                                        PMBusRegs.PMBTXBUF.all = u.pmbus_transmit_load; //this
sends message

                                        PMBusRegs.PMBACK.byte.BYTE0 = 1;  //ack command;
                                        pmbus_buffer_position = 4;

                                        pmbus_state = PMBUS_STATE_READ_BLOCK;
                                        return ;
                        }

        }
```

```
}
```

f. Changes to void **pmbus_read_wait_for_eom_handler(void)**:

```
else if( (pmbus_status_half_word_0_value & (PMBST_HALF0_CHECK_BITS+
PMBST_BYTE0_RD_BYTE_COUNT )) == PMBST_BYTE0_EOM)
    {//if eom, done
        pmbus_state = PMBUS_STATE_IDLE;
        //PMBusRegs.PMBACK.byte.BYTE0 = 1;  //ack them
        return ;
    }
    else //some other error, still go back to idle.
    {
        //PMBusRegs.PMBACK.byte.BYTE0 = 0;  //NACK
        pmbus_state = PMBUS_STATE_IDLE;
        return;
    }
```

g. Changes to void **pmbus_write_block_handler(void)**:

Delete the ack since we have manual ack after receiving address in pmbus_idle_handler

```
else if((pmbus_status_half_word_0_value & (PMBST_HALF0_CHECK_BITS )) ==
                (PMBST_BYTE0_EOM + PMBST_BYTE0_DATA_READY +
PMBST_BYTE0_PEC_VALID))
                //end of message, good data ready, pec valid, must be 2 to 4 bytes
    {
        //copy all 4 over just to make sure
        pmbus_buffer[pmbus_number_of_bytes] = PmbusRegs.PMBRXBUF.byte.BYTE0;
        pmbus_buffer[pmbus_number_of_bytes + 1] = PmbusRegs.PMBRXBUF.byte.BYTE1;
        pmbus_buffer[pmbus_number_of_bytes + 2] = PmbusRegs.PMBRXBUF.byte.BYTE2;
        pmbus_buffer[pmbus_number_of_bytes + 3] = PmbusRegs.PMBRXBUF.byte.BYTE3;
        pmbus_number_of_bytes = pmbus_number_of_bytes +

    (pmbus_status_half_word_0_value & PMBST_BYTE0_RD_BYTE_COUNT);
        pmbus_write_message();
        //PmbusRegs.PMBACK.byte.BYTE0 = 1;  //ack these 4 bytes;
        pmbus_state = PMBUS_STATE_IDLE;
        return;
    }

else if((pmbus_status_half_word_0_value & (PMBST_HALF0_CHECK_BITS )) ==
         (PMBST_BYTE0_EOM + PMBST_BYTE0_PEC_VALID))
        //end of message, no new data ready, pec valid; was exact multiple of 4 bytes, already
acked last time
    {
        pmbus_write_message();
```

```
            //PmbusRegs.PMBACK.byte.BYTE0 = 1;  //ack the message;
            pmbus_state = PMBUS_STATE_IDLE;
            return;
    }
    else //here if something has happened on bus, not covered by anything above
    {
            //PmbusRegs.PMBACK.byte.BYTE0 = 1;  //nack them
            pmbus_state = PMBUS_STATE_IDLE;
            return;
    }
```

5 Test Result

5.1 Multiple Addresses Shown in Device GUI

The device GUI shows there are multiple slavers on the PMBus interface, but it is actually a single device.

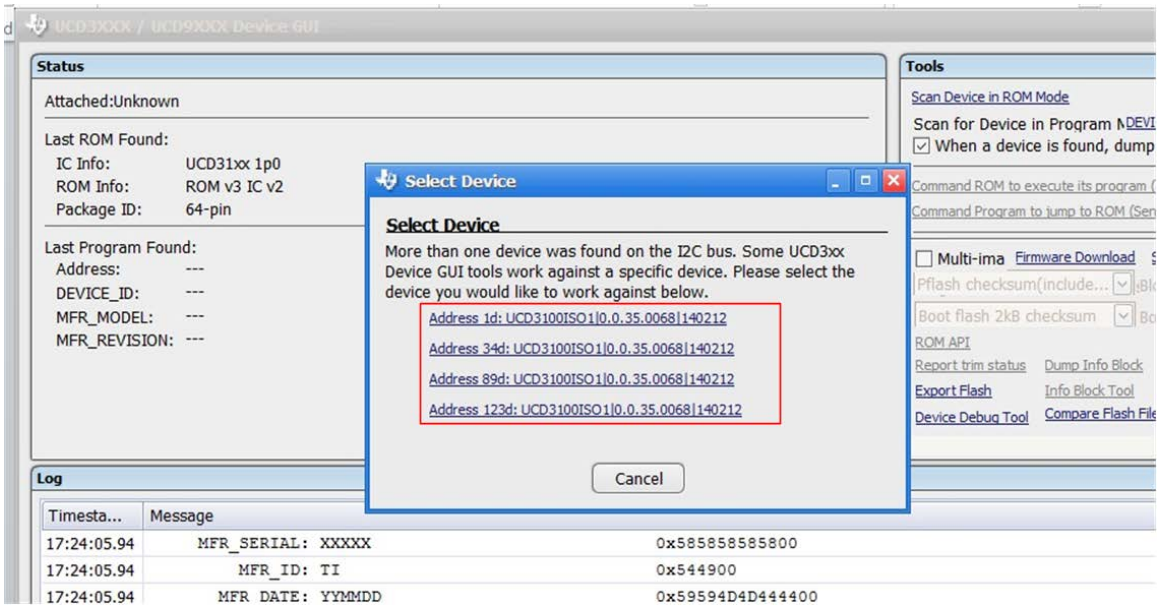

Figure 11. Multiple Slave Addresses

5.2 PMBus Master can get the Different Value in Different Address as There are Multiple Devices in Slave

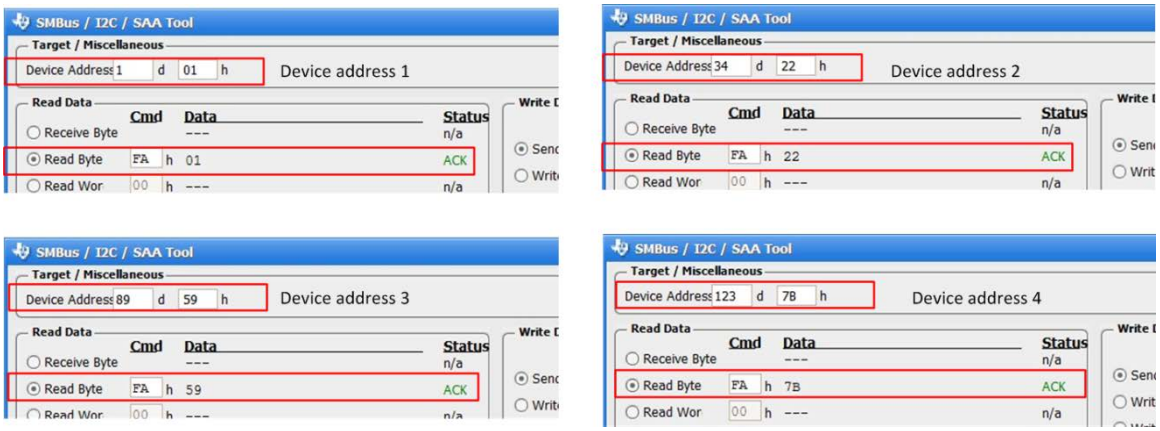

Figure 12. Slaver Sends its own Address to Master

6 Summary

This application note describes a firmware solution to let UCD3138 respond to multiple PMBus addresses. The method involves firmware to ack address and command manually. From the test result, UCD3xxx families can fully support this feature.

7 Reference

1. PMBus – Power System Management Protocol Specification, System Management Interface Forum, Inc. February, 2010
2. UCD3138 Highly Integrated Digital Controller for Isolated Power (SLUSAP2C), Texas Instruments, 2013
3. UCD3138 ARM and Digital System Programmer's Manual (SLUU994), Texas Instruments, July 2012
4. UCD3138 Monitoring and Communications Programmer's Manual (SLUU996), Texas Instruments, April, 2013

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265