Keywords: MAX3108, UART, RS232, RS485, SPI, I2C, WLP, wafer level package, FIFO

APPLICATION NOTE 5304

Interfacing to the MAX3108 UART

By: Micheal Scherrenburg
Mar 27, 2012

Abstract: This application note, the first in a series exploring the features of the MAX3108 high-performance universal asynchronous receiver/transmitter (UART) in detail, explains the basic interface between the MAX3108 and the controlling microprocessor. The application note briefly covers hardware connectivity via 2, 4, or 6 pins; the implementation of 31 registers, which are accessible through a SPI or an I²C interface; and three reset mechanisms. A more detailed explanation of SPI and I²C interfaces and examples of pseudocode follows.

Introduction

The MAX3108 is a high-performance universal asynchronous receiver-transmitter (UART) in a wafer-level package (WLP), ideal for low-power portable devices. Its advanced features range from individual 128-word transmit and receive FIFOs to extensive hardware-mediated flow control. However, before an application can take advantage of these features, it must reliably communicate with the MAX3108 and its 31 internal registers. This application note covers the basics of MAX3108 communication. Software conventions regarding the pseudocode appear in the Appendix.

Hardware Connectivity

Communication between the MAX3108 and the microprocessor (µP) occurs through 2, 4, or 6 pins. Two dedicated pins, active-low IRQ and active-low RST, typically connect to microprocessor GPIO pins. Active-low IRQ is an input pin to the microprocessor. As a dual-function signal, active-low IRQ both indicates when a MAX3108 reset has been finished and notifies the microprocessor when certain programmable UART events of note have occurred. Active-low RST is an output pin from the microprocessor. It reliably forces a reset of the MAX3108 into a known state. Though it is possible to implement a functional application without using these pins, it is preferable to connect them unless there is a lack of GPIO pins.

The MAX3108 implements 31 registers, each a byte wide and accessible through either a SPI or an I²C interface, which is chosen in the hardware. Because of its much greater bandwidth and simpler implementation, the SPI interface is preferable. Tying the SPI/active-low I²C pin high implements a SPI interface. Tying it low implements an I²C interface.

The MAX3108 SPI interface works best when connected directly to the SPI hardware engines commonly available on microprocessors.

Microprocessor SPI Pin	MAX3108 Pin
MISO	MISO/SDA
MOSI	MOSI/A1
SCLK	SCLK/SCL
SS	Active-low CS/A0

Unlike some other SPI interface implementations, the Slave Select (SS) signal must connect to the MAX3108, even if the MAX3108 is the only device on the SPI bus.

Similarly, the I²C interface works best when connected directly to the I²C hardware engine on the microprocessor.

Microprocessor I²C Pin	MAX3108 Pin
SCL	SCLK/SCL
SDA	MISO/SDA

Unlike with the SPI interface, the I²C interface relies on each peripheral on the bus having a unique I²C address. To help avoid address clashes, hardware pin strapping options on pins MOSI/A1 and active-low CS/A0 fix the MAX3108 I²C peripheral address to one of 16 possibilities. For more details, see the MAX3108 data sheet.

Reset

The MAX3108 has three reset mechanisms, all of which put the MAX3108 in a known state and require a subsequent register set load.

The first reset mechanism is an automatic reset at power-up. For the MAX3108 to come out of this reset, no clock is needed. However, the 1.8V supply must be up and running. Either LDOEN must be tied high, or if LDOEN is tied low, a valid 1.8V supply must be tied to the V18 pin. During reset only, the active-low IRQ pin is not an interrupt indicator, but a reset completion indicator. Once the active-low IRQ pin goes high, reset is complete. If the MAX3108 is absent or is unpowered, program flow could get stuck indefinitely in the `while` loop. If this is a concern, add a timer. If active-low IRQ has not come up after 300µs, you can assume something is amiss. To avoid writing to registers before the MAX3108 is completely ready, the application must follow the procedure below.

```
/*
** Wait for the MAX3108 to come out of reset
*/

   // Wait for the IRQ pin to come up, as this
   // indicates that the MAX3108 reset is complete
   while (POLL_MAX3108_IRQ == 0);
```

The second reset mechanism is a hardware reset via the MAX3108 active-low RST pin. In this scenario, manually pulse the active-low RST pin low, then high. As in automatic reset at power-up, the active-

low IRQ pin indicates reset completion. As above, to prevent getting indefinitely stuck in the `while` loop during an absent or unpowered MAX3108, add a timer with a minimum 300µs delay. See below for the pseudocode.

```
/*
** Perform a hardware reset
*/
   SET_MAX3108_RESET_PIN_LOW;
   WAIT (1µs);
   SET_MAX3108_RESET_PIN_HIGH;

   // Wait for the IRQ pin to come up, as this
   // indicates that the MAX3108 reset is complete
   while (POLL_MAX3108_IRQ == 0);
```

The third reset mechanism is a software-controlled reset, which relies on SPI or I²C access to the MAX3108. Like other resets, the MAX3108 registers need to be re-loaded after the reset has been completed. Invoke the reset with the following pseudocode.

```
/*
** Software reset of the MAX3108
*/
   MAX3108_Write (MAX3108R_MODE2, 0x01);
   MAX3108_Write (MAX3108R_MODE2, 0x00);
```

Interfacing via SPI

Communication between the MAX3108 and the microprocessor occurs via the active-low RST pin, the active-low IRQ pin (both mentioned above), and an interface between the application microprocessor and the MAX3108 registers. This register interface can be either a SPI or an I²C interface. Details regarding the SPI option appear here, whereas details about the I²C interface appear in the Interfacing via I²C section.

The SPI interface is a commonly available hardware interface on microprocessors. Another advantage of the SPI interface is its speed. The MAX3108 can support SPI data rates up to 26Mbps, faster than most microprocessor SPI hardware available today.

The MAX3108 SPI interface supports not only single register reads and writes, but also burst reads and writes. These burst transactions increase the efficiency of the register interface, further pushing out the point at which the register interface becomes a bottleneck.

Unlike some other SPI implementations, the SS signal from the SPI master hardware on the microprocessor must connect to the MAX3108 active-low CS/A0 pin. Besides its traditional use selecting the SPI bus peripheral, the active-low CS/A0 pin also signals the SPI transaction boundaries to the MAX3108. The falling edge of active-low CS/A0 indicates to the MAX3108 that the next byte on the SPI interface is a MAX3108 register address. The rising edge of active-low CS/A0 signals to the MAX3108 that a SPI transaction (single read, single write, burst read, or burst write) has concluded.

SPI transactions are inherently bidirectional. For each byte written from the microprocessor to the

MAX3108, one byte comes back from the MAX3108 to the microprocessor simultaneously. These can often be ignored; register writes ignore the return bytes from the MAX3108. Similarly, register reads return their values while the microprocessor sends dummy bytes to the MAX3108. These dummy bytes do not slow down the interface because they occur simultaneously with valid register read or write data.

Single SPI Write

The pseudocode for a single write transaction is as follows:

```
/*
** Write one byte to the specified register in the MAX3108
**
** Arguments:
** port: MAX3108 register address to write to (0x00 through 0x1e)
** val: the value to write to that register
**
** return value:  TRUE
**
*/
BOOL MAX3108_SPI_Write (unsigned int port,
                        unsigned char val) {
   unsigned int dummy;

   // Indicate the start of a transaction
   SET_MAX3108_CS_PIN_LOW;

   // Write transaction is indicated by MSbit of the
   // MAX3108 register address byte = 1.  SPI return
   // from the MAX3108 ignored
   dummy = SPI_SEND_BYTE (0x80 | port);

   // Now send the value to write, return value ignored
   dummy = SPI_SEND_BYTE (val);

   // Finally, indicate transaction completion
   SET_MAX3108_CS_PIN_HIGH;

   return TRUE;
}
```

Burst SPI Write

The MAX3108 SPI interface also supports burst writes, which are suitable for quick FIFO fill or for quick register loads. If the register address is zero, then burst writes fill the transmit FIFO. This permits quick FIFO fills without the overhead of toggling the MAX3108 active-low CS/A0 line or sending a register address for each FIFO byte written.

For any other MAX3108 register address, the register address automatically increments for each succeeding byte of the burst, thereby permitting quick register fills.

The pseudocode for a burst write transaction is as follows:

```
/*
** Write a burst of bytes to the MAX3108
**
** Arguments:
** port: MAX3108 register address to write to
** len: number of bytes to send to MAX3108 registers
** ptr: pointer to the bytes to send
**
** return value:  TRUE
**
*/
BOOL MAX3108_SPI_Puts (unsigned int port,
                       unsigned int len,
                       unsigned char *ptr) {

   unsigned int dummy;

   // Indicate the start of a transaction
   SET_MAX3108_CS_PIN_LOW;

   // Write transaction indicated by MSbit of the
   // MAX3108 register address byte = 1.  SPI return
   // from the MAX3108 ignored
   dummy = SPI_SEND_BYTE (0x80 | port);

   // Covers the case where len==0
   while (len--) {
      dummy = SPI_SEND_BYTE (*ptr++);  // return value ignored
   }

   // Indicate transaction completion
   SET_MAX3108_CS_PIN_HIGH;

   return TRUE;
}
```

Note that for burst write transactions, the first byte is the MAX3108 register address and all subsequent bytes are register value writes. The final register write is indicated by the active-low CS/A0 pin going high.

SPI hardware engines (usually DMA-based) also control the MAX3108 active-low CS/A0 pin, usually via a pin labeled SS. SPI transactions end with active-low CS/A0 going high. When active-low CS/A0 goes low, this indicates the start of the next SPI transaction. The minimum time that active-low CS/A0 must remain high is 100ns.This time ensures that the MAX3108 recognizes that the transaction has been completed.

Single SPI Read

Reading a register byte via SPI involves sending the register address, followed by a dummy byte. The SPI feedback from the dummy byte is the value of the MAX3108 register selected.

The pseudocode for a single read is as follows:

```
/*
** Read one byte from the specified register in the MAX3108
**
** Arguments:
** port: MAX3108 register address to read from
**
** return value:  the register value
**
*/
unsigned int MAX3108_SPI_Read (unsigned int port) {
   unsigned int dummy;
   unsigned int val;

   // Indicate the start of a transaction
   SET_MAX3108_CS_PIN_LOW;

   // Read transaction indicated by MSbit of the
   // address byte = 0.  SPI return from the MAX3108
   // ignored
   dummy = SPI_SEND_BYTE (port & 0x7f);

   // Now send a dummy byte to collect the
   // register value
   val = SPI_SEND_BYTE (0x00);

   // Finally, indicate transaction completion
   SET_MAX3108_CS_PIN_HIGH;

   return val;
}
```

Burst SPI Read

The MAX3108 SPI interface also supports burst reads, which are suitable for quick FIFO emptying or for quick register scans. If the register address is zero, then burst reads empty the receive FIFO. This permits quick FIFO dumps without the overhead of toggling the MAX3108 active-low CS/A0 line or sending a register address for each FIFO byte read.

For any other MAX3108 register address, the register address automatically increments for each succeeding byte of the burst, thereby permitting quick register scans.

The pseudocode for a burst read transaction is as follows:

```
/*
** Read a burst of bytes from the MAX3108
**
** Arguments:
** port: MAX3108 register address to read from
** len: number of bytes to get from MAX3108 registers
** ptr: pointer to where to place the bytes
**
** return value:  TRUE
**
*/
BOOL MAX3108_SPI_Gets (unsigned int port,
                       unsigned int len,
                       unsigned char *ptr) {

   unsigned int dummy;

   // Indicate the start of a transaction
   SET_MAX3108_CS_PIN_LOW;

   // Read transaction indicated by MSbit of the
   // address byte = 0.  SPI return from the MAX3108
   // ignored
   dummy = SPI_SEND_BYTE (port & 0x7f);

   // Covers the case where len==0
   while (len--) {
      *ptr++ = SPI_SEND_BYTE (0x00);
   }

   // Indicate transaction completion
   SET_MAX3108_CS_PIN_HIGH;

   return TRUE;
}
```

Note that for burst read transactions, the first byte is the MAX3108 register address and all subsequent bytes are dummies whose returns are the register values. The final register read is indicated by the active-low CS/A0 pin going high.

Interfacing via I²C

Communication between the MAX3108 registers and the microprocessor can also occur via an I²C interface.

The I²C interface is a common hardware-assisted interface on microprocessors. Another advantage of the I²C interface is that it only uses two pins. Instead of individual chip selects for each peripheral (as in the case of a SPI interface), an I²C interface relies on each peripheral having a unique I²C address. To avoid possible clashes with other I²C peripherals on the same I²C bus, the MAX3108 can be pin-

programmed to one of 16 unique I²C peripheral addresses via pin strapping of the MOSI/A1 and the active-low CS/A0 pins. In the pseudocode that follows, the preprocessor variable MAX3108_I2C_PERIPHERAL_ADDRESS is assumed to be defined with an appropriate I²C register write address, which is unique from that of any other I²C peripherals on the same bus.

The MAX3108 supports not only standard mode (100kbps) and fast mode (400kbps) I²C, but also fast mode plus (1Mbps) I²C.

The MAX3108 I²C interface supports not only single register reads and writes, but also burst reads and writes. These burst transactions increase the efficiency of the register interface, further pushing out the point at which the register interface becomes a bottleneck.

Single I²C Write

The pseudocode for a single write transaction is as follows:

```
/*
** Write one byte to the specified register in the MAX3108
**
** Arguments:
** port: MAX3108 register address to write to
** val: the value to write to that register
**
** return value:  TRUE – register successfully written
**                FALSE – I2C protocol error of some kind
**
*/
BOOL MAX3108_I2C_Write (unsigned int port,
                        unsigned char val) {

   // Indicate the start of a transaction and send the
   // MAX3108 write peripheral address (LSbit = 0)
   I2C_SET_START_CONDITION;
   I2C_SEND_BYTE (MAX3108_I2C_PERIPHERAL_ADDRESS);

   // Is anybody out there?
   if (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;  // no – close out the
      return FALSE;            //  transaction
   }

   // The MAX3108 is out there, now send the MAX3108
   // register to write
   I2C_SEND_BYTE (port);

   // Possibly illegal MAX3108 register address?
   If (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;
      return FALSE;
   }
```

```
   // Now send the byte to write
   I2C_SEND_BYTE (val);

   // Did the MAX3108 get confused?
   If (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;
      return FALSE;
   }

   // The MAX3108 is OK with our write, make it so
   I2C_SET_STOP_CONDITION;
   return TRUE;
}
```

Burst I²C Write

The MAX3108 I²C interface also supports burst writes, which are suitable for quick FIFO fill or for quick register loads. If the register address is zero, then burst writes fill the transmit FIFO. This permits quick FIFO fills without the I²C preamble overhead for each FIFO byte written.

For any other MAX3108 register address, the register address automatically increments for each succeeding byte of the burst, thereby permitting quick register fills.

The pseudocode for a burst write transaction is as follows:

```
/*
** Write a burst of bytes to the MAX3108
**
** Arguments:
** port: MAX3108 register address to write to
** len: number of bytes to send to MAX3108 registers
** ptr: pointer to the bytes to send
**
** return value:  TRUE – register successfully written
**                FALSE – I2C protocol error of some kind

**
*/
BOOL MAX3108_I2C_Puts (unsigned int port,
                       unsigned int len,
                       unsigned char *ptr) {

   // Indicate the start of a transaction and send the
   // MAX3108 write peripheral address (LSbit = 0)
   I2C_SET_START_CONDITION;
   I2C_SEND_BYTE (MAX3108_I2C_PERIPHERAL_ADDRESS);

   // Is anybody out there?
```

```
   if (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;  // no – close out the
      return FALSE;            //  transaction
   }

   // The MAX3108 is out there, now send the MAX3108
   // register address
   I2C_SEND_BYTE (port);

   // Possibly illegal MAX3108 register address?
   If (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;
      return FALSE;
   }

   while (len--) {
      // Now send the byte to write
      I2C_SEND_BYTE (*ptr++);

      // Did the MAX3108 get confused?
      if (!I2C_TEST_ACK) {
         I2C_SET_STOP_CONDITION;
         return FALSE;
      }
   }

   // The MAX3108 is OK with our write, make it so
   I2C_SET_STOP_CONDITION;
   return TRUE;
}
```

Note that for burst write transactions, the first byte is the MAX3108 I²C peripheral address, the second byte is the MAX3108 register address, and all subsequent bytes are register values to write. The final register write is indicated by the STOP condition.

Single I²C Read

Reading a register byte via I²C involves sending the MAX3108 read register address, followed by an I²C RESTART condition to turn the I²C bus around from a write (to the MAX3108 to specify the MAX3108 register) to a read (to get the register value).

The pseudocode for a single read is as follows:

```
/*
** Read one byte from the specified register in the MAX3108
**
** Arguments:
** port: MAX3108 register address to read from
**
** return value:  the register value (0x00XX)
```

```
**                0xff00 is there was some error
**
*/
unsigned int MAX3108_I2C_Read (unsigned int port) {

   unsigned int val;

   // Indicate the start of a transaction and send the
   // MAX3108 write peripheral address (LSbit = 0)
   I2C_SET_START_CONDITION;
   I2C_SEND_BYTE (MAX3108_I2C_PERIPHERAL_ADDRESS);

   // Is anybody out there?
   if (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;  // no – close out the
      return 0xff00;           //  transaction
   }

   // The MAX3108 is out there, now send the MAX3108
   // register address
   I2C_SEND_BYTE (port);

   // Possibly illegal MAX3108 register address?
   If (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;
      return 0xff00;
   }

   // Now turn the I2C bus around by sending the MAX3108
   // I2C peripheral read address (LSbit = 1)
   I2C_SET_RESTART_CONDITION;
   I2C_SEND_BYTE (MAX3108_I2C_PERIPHERAL_ADDRESS | 0x01);

   if (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;
      return 0xff00;
   }

   // Now get the register value
   val = I2C_RECEIVE_BYTE;
   I2C_SEND_NACK;
   I2C_SET_STOP_CONDITION;

   return (val);
}
```

Burst I²C Read

The MAX3108 I²C interface also supports burst reads, which are suitable for quick FIFO emptying or for

quick register scans. If the register address is zero, then burst reads empty the receive FIFO. This permits quick FIFO dumps without the overhead associated with the I²C preamble for each FIFO byte read.

For any other MAX3108 register address, the register address automatically increments for each succeeding byte of the burst, thereby permitting quick register scans.

The pseudo code for a burst read transaction is as follows:

```
/*
** Read a burst of bytes from the MAX3108
**
** Arguments:
** port: MAX3108 register address to read from
** len: number of bytes to get from MAX3108 registers
** ptr: pointer to where to place the bytes
**
** return value:  TRUE – if all read
**                FALSE – if there was any error
**
*/
BOOL MAX3108_I2C_Gets (unsigned int port,
                       unsigned int len,
                       unsigned char *ptr) {



   // Indicate the start of a transaction and send the
   // MAX3108 write peripheral address (LSbit = 0)
   I2C_SET_START_CONDITION;
   I2C_SEND_BYTE (MAX3108_I2C_PERIPHERAL_ADDRESS);

   // Is anybody out there?
   if (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;  // no – close out the
      return FALSE;           //  transaction
   }

   // The MAX3108 is out there, now send the MAX3108
   // register address
   I2C_SEND_BYTE (port);

   // Possibly illegal MAX3108 register address?
   If (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;
      return FALSE;
   }
```

```
   // Now turn the I2C bus around by sending the MAX3108
   // I2C peripheral read address (LSbit = 1)
   I2C_SET_RESTART_CONDITION;
   I2C_SEND_BYTE (MAX3108_I2C_PERIPHERAL_ADDRESS | 0x01);

   if (!I2C_TEST_ACK) {
      I2C_SET_STOP_CONDITION;
      return FALSE;
   }

   // Now get the register values
   while (len--) {
      *ptr++ = I2C_RECEIVE_BYTE;
      if (len)
         I2C_SEND_ACK;     // all but last read
      else
         I2C_SEND_NACK;    // last read only
   }

   I2C_SET_STOP_CONDITION;
   return TRUE;
}
```

Note that for burst read transactions, the first byte is the MAX3108 I²C peripheral write address, the second byte is the MAX3108 register address, the third byte is the MAX3108 I²C peripheral read address, and all subsequent bytes are register values read. The microprocessor responds to each byte read with an ACK condition as more registers are to be read. A NACK condition indicates the final register read from the MAX3108.

Conclusion

Following the coding guidelines provided in this application note, one can quickly get an interface between the microprocessor and the MAX3108 up and running.

Much of the translation of the pseudocode routines in this application note will be microprocessor specific. Except for interrupt handlers, other code-specific application notes in this series, which explore the features of the MAX3108 in detail, will need little translation to a specific target microprocessor. They will rely on the read, write, gets, and puts primitives described in this application note to encapsulate microprocessor-specific issues.

Appendix

This appendix contains the definitions necessary to understand the pseudocode in this application note as well as others in this series, which explore the features of the MAX3108 in detail.

Definitions Relating to Pin I/O

POLL_MAX3108_IRQ: This pseudocode returns the digital state of the MAX3108 active-low IRQ pin. This pin requires an external pullup to work properly.

SET_MAX3108_CS_PIN_HIGH: This pseudocode drives the MAX3108 active-low CS/A0 pin high.

SET_MAX3108_CS_PIN_LOW: This pseudocode drives the MAX3108 active-low CS/A0 pin low.

SET_MAX3108_RESET_PIN_HIGH: This pseudocode drives the MAX3108 active-low RST pin high.

SET_MAX3108_RESET_PIN_LOW: This pseudocode drives the MAX3108 active-low RST pin low.

WAIT: This pseudocode delays execution of the following instruction, at a minimum, by the specified time.

Definitions Relating to the I²C Interface

I2C_SEND_BYTE: This pseudocode sends one byte to the MAX3108 via the I²C bus.

I2C_SET_RESTART_CONDITION: This pseudocode puts a RESTART condition on the I²C bus.

I2C_SET_START_CONDITION: This pseudocode puts a START condition on the I²C bus.

I2C_SET_STOP_CONDITION: This pseudocode puts a STOP condition on the I²C bus.

I2C_TEST_ACK: This pseudocode returns TRUE if the MAX3108 responded to a byte written with an ACK and returns FALS if the MAX3108 responded to a byte written with a NACK.

SPI_SEND_BYTE: One byte is sent to the MAX3108 via the SPI interface, and the byte simultaneously returned by the MAX3108 is captured.

Definitions Related to the Register Interface

MAX3108_Gets: This routine is either MAX3108_I2C_Gets or MAX3108_SPI_Gets, depending on which interface is implemented in the application.

MAX3108_Puts: This routine is either MAX3108_I2C_Puts or MAX3108_SPI_Puts, depending on which interface is implemented in the application.

MAX3108_Read: This routine is either MAX3108_I2C_Read or MAX3108_SPI_Read, depending on which interface is implemented in the application.

MAX3108_Write: This routine is either MAX3108_I2C_Write or MAX3108_SPI_Write, depending on which interface is implemented in the application.

MAX3108_I2C_Gets: This routine, described in this application note, burst reads from the MAX3108 receive FIFO or other MAX3108 registers via the I²C interface.

MAX3108_I2C_Puts: This routine, described in this application note, burst writes to the MAX3108 transmit FIFO or other MAX3108 registers via the I²C interface.

MAX3108_I2C_Read: This routine, described in this application note, reads the value of one of the MAX3108 registers via the I²C interface.

MAX3108_I2C_Write: This routine, described in this application note, writes a value to one of the MAX3108 registers via the I²C interface.

MAX3108_SPI_Gets: This routine, described in this application note, burst reads from the MAX3108 receive FIFO or other MAX3108 registers via the SPI interface.

MAX3108_SPI_Puts: This routine, described in this application note, burst writes to the MAX3108 transmit FIFO or other MAX3108 registers via the SPI interface.

MAX3108_SPI_Read: This routine, described in this application note, reads the value of one of the MAX3108 registers via the SPI interface.

MAX3108_SPI_Write: This routine, described in this application note, writes a value to one of the MAX3108 registers via the SPI interface.

MAX3108 Register Defines

```
//
// MAX3108 Register map defines
//
#define MAX3108R_RHR          (0x00)
#define MAX3108R_THR          (0x00)
#define MAX3108R_IRQEN        (0x01)
#define MAX3108R_ISR          (0x02)
#define MAX3108R_LSRINTEN     (0x03)
#define MAX3108R_LSR          (0x04)
#define MAX3108R_SPCLCHRINTEN (0x05)
#define MAX3108R_SPCLCHARINT  (0x06)
#define MAX3108R_STSINTEN     (0x07)
#define MAX3108R_STSINT       (0x08)
#define MAX3108R_MODE1        (0x09)
#define MAX3108R_MODE2        (0x0a)
#define MAX3108R_LCR          (0x0b)
#define MAX3108R_RXTIMEOUT    (0x0c)
#define MAX3108R_HDPLXDELAY   (0x0d)
#define MAX3108R_IRDA         (0x0e)
#define MAX3108R_FLOWLVL      (0x0f)
#define MAX3108R_FIFOTRGLVL   (0x10)
#define MAX3108R_TXFIFOLVL    (0x11)
#define MAX3108R_RXFIFOLVL    (0x12)
#define MAX3108R_FLOWCTRL     (0x13)
#define MAX3108R_XON1         (0x14)
#define MAX3108R_XON2         (0x15)
#define MAX3108R_XOFF1        (0x16)
#define MAX3108R_XOFF2        (0x17)
#define MAX3108R_GPIOCONFG    (0x18)
#define MAX3108R_GPIODATA     (0x19)
#define MAX3108R_PLLCONFIG    (0x1a)
#define MAX3108R_BRGCONFIG    (0x1b)
```

```
#define MAX3108R_DIVLSB       (0x1c)
#define MAX3108R_DIVMSB       (0x1d)
#define MAX3108R_CLKSOURCE    (0x1e)
```

Related Parts		
MAX14830	Quad Serial UART with 128-Word FIFOs	
MAX3107	SPI/I²C UART with 128-Word FIFOs	Free Samples
MAX3108	SPI/I²C UART with 128-Word FIFOs in WLP	
MAX3109	Dual Serial UART with 128-Word FIFOs	Free Samples

More Information
For Technical Support: http://www.maximintegrated.com/support
For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact

Application Note 5304: http://www.maximintegrated.com/an5304
APPLICATION NOTE 5304, AN5304, AN 5304, APP5304, Appnote5304, Appnote 5304