Using the Accelerometer on DE-SoC Boards

For Quartus® Prime 21.1

1 Introduction

This tutorial describes how to use the ADXL345 accelerometer on the DE10-Standard, DE10-Nano, DE1-SoC, and DE0-Nano-SoC boards. For using the ADXL345 accelerometer on the DE0-Nano and VEEK-MT boards, please refer to the document *Accelerometer SPI Mode Core for DE-Series Boards* instead.

The reader is expected to have a basic knowledge of the C programming language, and be familiar with the Monitor Program. Some knowledge of the I2C serial communications protocol is beneficial, but not necessary.

Contents:

- A description of the ADXL345 digital accelerometer
- Communicating with the ADXL345 device on DE-SoC boards
- Using the Cyclone® V HPS's I2C0 controller
- Writing C-language code to operate the ADXL345 device using I2C0

2 The ADXL345 Digital Accelerometer

The ADXL345 device is a 3-axis accelerometer manufactured by Analog Devices Corporation. It provides acceleration measurements in the x, y, and z axes up to a maximum of +/- 16 g (g = 9.81 m/s^2). It is capable of sampling acceleration at regular intervals and storing the measured data for later access by an external device such as a processor. Communication with the ADXL345 device is done using an I2C serial bus. In a typical application, software code running on a processor uses an I2C master to access the ADXL345 device's internal registers. These registers are described in the following section.

2.1 The ADXL345 Internal Registers

An abbreviated list of the ADXL345 internal registers is shown in Table 1. These registers have a width of 8 bits. Only a minimal set of registers required for reading basic acceleration data is described below. For the complete list of registers and detailed descriptions, please refer to the ADXL345 datasheet.

2.1.1 DEVID (0x00)

This register always holds a static value of 0xE5. Reading this register and checking to see that the value 0xE5 is returned can be used as a quick test to see if the I2C connection is working correctly.

Table 1. ADXL345 Internal Registers (Abbreviated)				
Address	**Register name**	**Read/Write**	**Reset Value**	**Purpose**
0x00	DEVID	R	11100101	Device ID (0xE5)
0x2C	BW_RATE	R/W	00001010	Data rate and power mode control
0x2D	POWER_CTL	R/W	00000000	Power state control
0x30	INT_SOURCE	R	00000010	Source of interrupts
0x31	DATA_FORMAT	R/W	00000000	Data format control
0x32	DATAX0	R	00000000	X-Axis Data 0
0x33	DATAX1	R	00000000	X-Axis Data 1
0x34	DATAY0	R	00000000	Y-Axis Data 0
0x35	DATAY1	R	00000000	Y-Axis Data 1
0x36	DATAZ0	R	00000000	Z-Axis Data 0
0x37	DATAZ1	R	00000000	Z-Axis Data 1

2.1.2 BW_RATE (0x2C)

bit$_7$	**bit$_6$**	**bit$_5$**	**bit$_4$**	**bit$_3$**	**bit$_2$**	**bit$_1$**	**bit$_0$**
0	0	0	LOW_POWER	Rate			

This register is used to set the sampling rate of the the accelerometer. The default value is 0x0A, which translates to a sampling rate of 100 Hz. Values between 0x0 and 0xF can be written to the Rate bits, which correspond to sampling rates between 0.098 Hz to 3200 Hz (each increment to Rate doubles the sampling rate). The LOW_POWER bit can be set to 1 to turn on low power mode, but doing so will add noise to the measured data and is not recommended.

2.1.3 POWER_CTL (0x2D)

bit$_7$	**bit$_6$**	**bit$_5$**	**bit$_4$**	**bit$_3$**	**bit$_2$**	**bit$_1$**	**bit$_0$**
0	0	Link	Auto_sleep	Measure	Sleep	Wakeup	

This register is used to configure settings related to the power states of the accelerometer, such as sleep and wakeup. For our purposes, we use only the Measure bit of this register, which turns on measurement of acceleration data when it is set to 1, and turns it off when set to 0.

2.1.4 INT_SOURCE (0x30)

bit$_7$	**bit$_6$**	**bit$_5$**	**bit$_4$**
Data_ready	Single_tap	Double_tap	Activity
bit$_3$	**bit$_2$**	**bit$_1$**	**bit$_0$**
Inactivity	Free_fall	Watermark	Overrun

This register indicates which of eight possible interrupt events has triggered an interrupt signal. Although we do not use interrupts in this tutorial, this register's Data_ready bit can be used to determine whether there is a new acceleration sample that can be read from the DATA registers.

2.1.5 DATA_FORMAT (0x31)

bit$_7$	**bit**$_6$	**bit**$_5$	**bit**$_4$	**bit**$_3$	**bit**$_2$	**bit**$_1$	**bit**$_0$
Self_test	SPI	Int_invert	0	Full_res	Justify	Range	

Bits 0-3 of this register control the format of the data stored in the DATA registers. The Range bits control the g range, and can be set to 0b00 (+/- 2 g), 0b01 (+/- 4 g), 0b10 (+/- 8 g), or 0b11 (+/- 16 g). The Justify selects left-justified mode when set to 1, and right-justified mode when set to 0. Writing 1 to Full_res enables the full resolution mode, which forces the least significant bit (LSB) of the sample to represent 3.9 mg. If full resolution mode is disabled, the data will be limited to 10 bits, and the LSB will represent whatever scale factor is required to cover the range with 10 bits. For example, selecting the +/- 4 g range (total range of 8000 mg) with full resolution mode disabled would result in the LSB representing 7.8 mg ($8000mg/2^{10} = 7.8mg$). The other bits of this register are used for miscellaneous settings that are not important for the purposes of this tutorial.

2.1.6 DATA Registers (0x32-0x37)

bit$_7$	**bit**$_6$	**bit**$_5$	**bit**$_4$	**bit**$_3$	**bit**$_2$	**bit**$_1$	**bit**$_0$
Data							

These registers hold the acceleration data for the three axes. DATA_X0 (0x32) and DATA_X1 (0x33) hold the data for the x-axis, DATA_Y0 (0x34) and DATA_Y1 (0x35) hold the data for the y-axis, and DATA_Z0 and DATA_Z1 hold the data for the z-axis. The latter of each pair of registers holds the most significant bits of the sample for that axis, and together they form a 16-bit 2's complement value. To ensure that these registers are not altered during a read of a sample, an I2C master must perform a single multiple-byte read of these registers rather than multiple single-byte reads.

2.1.7 THRESH_ACT, THRESH_INACT, TIME_INACT, ACT_INACT_CTL (0x24 - 0x27)

These registers allow you to configure interrupts based on activity (changes in acceleration data). The THRESH_ACT and THRESH_INACT registers are used to store eight-bit threshold values (where the values are 62.5 mg/LSB) to detect activity and inactivity, respectively. The TIME_INACT register is used to store an eight-bit value representing the amount of time that acceleration must be less than the value in THRESH_INACT for inactivity to be triggered. The ACT_INACT_CTL register is used to enable or disable activity detection in the X, Y, and Z axes.

3 Communicating with the ADXL345

Communcation with the ADXL345 (the reading and writing of its internal registers) is done through its I2C serial interface. On the DE1-SoC and DE0-Nano-SoC boards, the ADXL345's I2C wires are connected to the Cyclone V HPS, as shown in Figure 1. These wires are then routed through the Pin Multiplexer (Pin Mux) block, shown in Figure 2, which can be configured to route the signals to I2C0 (an I2C controller), GPIO1 (a GPIO controller), or to the FPGA where they can be connected to any user-defined circuit.

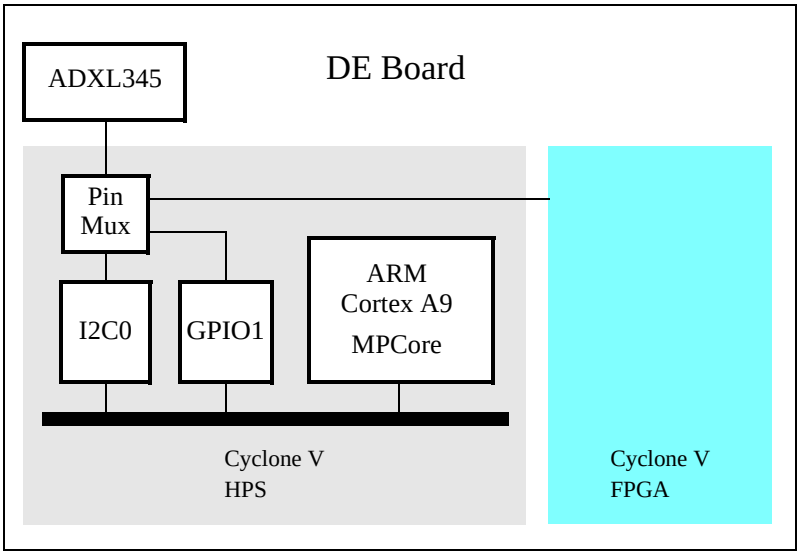

Figure 1. The ADXL345's I2C connection to the Cyclone V SoC chip on DE-Series boards.

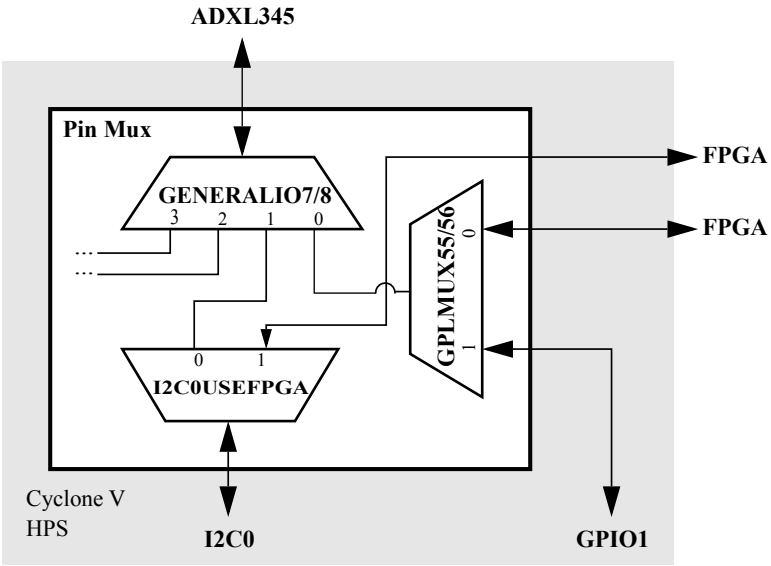

Figure 2. The Pin Mux block in more detail.

3.1 The Pin Multiplexer

In this tutorial we will configure the Pin Mux block to route the ADXL345's I2C signals to I2C0, which is an I2C controller built into the Cyclone V HPS. As we will see in Section 3.2, I2C0 provides a convenient memory-mapped register interface that makes it easy to access the ADXL345 internal registers.

*Table 2. Pin Mux Registers**

Multiplexer	Register name	Address	Reset Value
GENERALIO7/8	GENERALIO7	0xFFD0849C	0x0
	GENERALIO8	0xFFD084A0	0x0
GPLMUX55/56	GPLMUX55	0xFFD086B0	0x0
	GPLMUX56	0xFFD086B4	0x0
I2C0USEFPGA	I2C0USEFPGA	0xFFD08704	0x0
* these registers belong to the HPS's System Manager Module			

We can see from Figure 2 that in order to route the I2C signals to I2C0, multiplexer GENERALIO7/8 should be configured to "1", and I2C0USEFPGA to "0". These multiplexers are controlled by memory-mapped registers, as shown in Table 2. Figure 3 gives C code that writes to these registers to achieve our desired routing. Note that the multiplexers GENERALIO7/8 and GPLMUX55/56 each comprise two multiplexers. While the multiplexers of each pair are controlled by registers at different addresses, they should always be configured identically. In contrast, I2C0USEFPGA is a single multiplexer, and is configured by a single register.

```
1 #define SYSMGR_GENERALIO7   ((volatile unsigned int *) 0xFFD0849C)
2 #define SYSMGR_GENERALIO8   ((volatile unsigned int *) 0xFFD084A0)
3 #define SYSMGR_I2C0USEFPGA  ((volatile unsigned int *) 0xFFD08704)
4
5 void configure_pinmux(){
6     *SYSMGR_I2C0USEFPGA = 0;
7     *SYSMGR_GENERALIO7 = 1;
8     *SYSMGR_GENERALIO8 = 1;
9 }
```

Figure 3. C code that configures Pin Mux to connect the ADXL345 I2C wires to I2C0

Note that there are alternatives to using I2C0. For instance, connecting the ADXL345 to the GPIO controller (GPIO1) would allow the data pins of the GPIO to directly control the I2C wires - this could be done using software code that writes to the GPIO's data register.

3.2 The I2C0 Controller

I2C0 is one of four I2C controllers (I2C0 - I2C3) built into the Cyclone V HPS. These generic I2C controllers are designed to be capable of communicating in the I2C serial protocol with any I2C-compatible device such as the ADXL345. These controllers provide memory-mapped register interfaces that programs can use to communicate with connected I2C devices. A benefit to using the I2C controller is that it provides a high-level interface that abstracts away many of the low-level details involved in the I2C communication protocol.

We will use I2C0's memory-mapped register interface to communicate with the ADXL345. An abbreviated list of I2C0's memory-mapped registers is shown in Table 3, and further descriptions of these registers are provided in the following subsections. These registers have a width of 32 bits.

Table 3. I2C0 Controller Register Address Map (0xFFC04000)

Offset	Register name	Read/Write	Reset Value	Purpose
0x0	`ic_con`	R/W	0x7D	Control Register
0x4	`ic_tar`	R/W	0x1055	Target Address Register
0x10	`ic_data_cmd`	R/W	0x0	Tx Rx Data and Command Register
0x1C	`ic_fs_scl_hcnt`	R/W	0x3C	SCL High Count Register (Fast Speed)
0x20	`ic_fs_scl_lcnt`	R/W	0x82	SCL Low Count Register (Fast Speed)
0x40	`ic_clr_intr`	R	0x0	Clear All Interrupts Register
0x6C	`ic_enable`	R/W	0x0	Enable Register
0x74	`ic_txflr`	R	0x0	Transmit FIFO Level Register
0x78	`ic_rxflr`	R	0x0	Receive FIFO Level Register
0x9C	`ic_enable_status`	R	0x0	Enable Status Register

If you would like a more detailed description of the I2C controller, refer to the document *Cyclone V Device Handbook, Volume 3: Hard Processor System Technical Reference Manual*. For the complete list of I2C0's registers, refer to the document *Cyclone V SoC HPS Address Map and Register Definitions*.

3.2.1 ic_con (0x0)

Bit	Name	Access	Reset
31 - 7	Reserved		
6	`ic_slave_disable`	RW	0x1
5	`ic_restart_en`	RW	0x1
4	`ic_10bitaddr_master`	RW	0x1
3	`ic_10bitaddr_slave`	RW	0x1
2 - 1	`speed`	RW	0x2
0	`master_mode`	RW	0x1

This register is used to set the operating mode of I2C0. The `master_mode` bit controls whether I2C0 acts as an I2C master or slave. A value of 0x1 selects master mode, and 0x0 selects slave mode. The `speed` bits control whether I2C0 operates in fast mode (400 kbit/s) or slow mode (100 kbit/s). The value of 0x2 selects fast mode and 0x1 selects slow mode. The `ic_10bitaddr_slave` and `ic_10bitaddr_master` bits select whether I2C0 uses 7-bit or 10-bit addressing when functioning as a slave or a master, respectively. The value of 0x0 selects 7-bit addressing, and 0x1 selects 10-bit addressing. The `ic_restart_en` bit enables or disables sending the restart condition to the I2C slave, when I2C0 acts as a master. The `ic_slave_disable` bit controls whether I2C0 should function as a slave, and should be set to 0x1 when I2C0 should act as a master, and 0x0 when it should act as a slave.

For our purposes, we write 0x65 to this register. This configures I2C0 to:

- function as an I2C master
- run in fast mode (400 kbit/s)
- use 7-bit addressing mode, which is the addressing mode supported by the ADXL345
- use restart conditions, since this feature is supported by the ADXL345

3.2.2 ic_tar (0x4)

Bit	Name	Access	Reset
31 - 13	Reserved		
12	`ic_10bitaddr_master`	RW	0x1
11	`special`	RW	0x0
10	`gc_or_start`	RW	0x0
9 - 0	`ic_tar`	RW	0x55

This register is used to configure I2C0's communication with a slave. The target slave is selected by writing its slave address to the `ic_tar` bits. The `ic_10bitaddr_master` bit controls whether I2C0 uses 7-bit or 10-bit addressing mode when addressing the slave. Writing 0x1 to this bit selects 10-bit addressing, while writing 0x0 selects 7-bit addressing. For our purposes, we write 0x53 (the slave address of the ADXL345) to `ic_tar` and set `ic_10bitaddr_master` to 0x0, as the ADXL345 supports 7-bit addressing. The `gc_or_start` and `special` bits are not useful for our purposes.

3.2.3 ic_data_cmd (0x10)

Bit	Name	Access	Reset
31 - 11	Reserved		
10	`restart`	W	0x0
9	`stop`	W	0x0
8	`cmd`	W	0x0
7 - 0	`dat`	RW	0x0

This register is used to send or receive a byte to or from the ADXL345. When sending, the byte is written to `dat`, along with a 0x0 to `cmd`. When receiving, a 0x1 is first written to `cmd`, which sends a read request to the ADXL345. Once the ADXL345 replies with a byte, it can be read from `dat`. The `stop` bit is set to 0x1 when a stop signal should be sent after the byte. The `restart` bit is set to 0x1 when a restart signal should be sent before the byte. Note that `stop`, `restart`, `cmd` and `dat` are written together; a stop or restart signal cannot be sent independently of a byte.

3.2.4 ic_fs_scl_hcnt (0x1C)

Bit	Name	Access	Reset
31 - 16	Reserved		
15 - 0	`ic_fs_scl_hcnt`	RW	0x3C

This register is used to configure the serial clock signal (SCL) that is used for synchronizing data bits sent/received by I2C0. The `ic_fs_scl_hcnt` register sets the SCL clock's high period when running in the 400 kbit/s (fast) mode. The value written to this register corresponds to the number of cycles of the input clock to the I2C0 for which the SCL should remain high in each SCL clock period. The input clock to the ADXL345 is controlled the HPS, and is by default 100 MHz. We set the content of this register to 90, as described in Section 4.2.

3.2.5 ic_fs_scl_lcnt (0x20)

Bit	Name	Access	Reset
31 - 16	Reserved		
15 - 0	`ic_fs_scl_lcnt`	RW	0x82

This register is used to configure the serial clock signal (SCL) that is used for synchronizing data bits sent/received by I2C0. The `ic_fs_scl_lcnt` register sets the SCL clock's low period when running in the 400 kbit/s (fast) mode. The value written to this register corresponds to the number of cycles of the input clock to the I2C0 for which the SCL should remain low in each SCL clock period. The input clock to the ADXL345 is controlled the HPS, and is by default 100 MHz. We set the content of this register to 160, as described in Section 4.2.

3.2.6 ic_clr_intr (0x40)

Bit	Name	Access	Reset
31 - 1	Reserved		
0	`clr_intr`	R	0x82

This register is used to clear software-clearable interrupts. A read of this register triggers the clear. For our purposes, we read this register when first initializing the I2C0 to clear any currently-pending interrupts.

3.2.7 ic_enable (0x6C)

Bit	Name	Access	Reset
31 - 2	Reserved		
1	`txabort`	RW	0x0
0	`enable`	RW	0x0

This register is used to enable or disable I2C communication. A 0x1, or 0x0, is written to `enable` to enable, or disable, I2C communication, respectively. A 0x1 is written to `txabort` to abort any pending transmissions.

3.2.8 ic_txflr (0x74)

Bit	Name	Access	Reset
31 - 7	Reserved		
6 - 0	`txflr`	R	0x0

This register reports the number of valid data entries currently in I2C0's transmit FIFO buffer. When `txflr` is greater than 0, it means that there are bytes waiting to be sent to the ADXL345.

3.2.9 ic_rxflr (0x78)

Bit	Name	Access	Reset
31 - 7	Reserved		
6 - 0	`rxflr`	R	0x0

This register reports the number of valid data entries currently in I2C0's receive FIFO buffer. When `rxflr` is greater than 0, it means that there are bytes that I2C0 has received from the ADXL345 that we have not yet read.

3.2.10 ic_enable_status (0x9C)

Bit	Name	Access	Reset
31 - 3	Reserved		
2	`slv_rx_data_lost`	R	0x0
1	`slv_disabled_while_busy`	R	0x0
0	`ic_en`	R	0x0

This register is used to check whether I2C0 has reached the enabled state. After writing a 0x1 to the `enable` bit in the `ic_enable` register, we poll this register until the `ic_en` bit becomes 0x1, signalling that I2C0 is ready for operation. Similarly, after writing a 0x0 to the `enable` bit, we poll until `ic_en` becomes 0x0, signalling that I2C0 has been disabled.

4 Using the Accelerometer in C-Language Code

The following sections provide C-language code for configuring and operating the ADXL345, the I2C0 controller, and the Pin Multiplexer block. This code can also be found in the files `ADXL345.c` and `ADXL345.h` which accompany this tutorial. The code should be compiled, loaded, and executed using the Monitor Program. If you are not familiar with the Monitor Program you can refer to the document *Monitor Program Tutorial*.

4.1 Configuring the Pin Multiplexer

As described in Section 3.1, the first step in using the ADXL345 is to configure the Pin Mux block in the Cyclone V HPS to connect the ADXL345's I2C wires to I2C0. Figure 4 shows the function `Pinmux_Config()` that accomplishes this. The function writes to the memory-mapped registers GENERALIO7, GENERALIO8, and I2C0USEFPGA which control multiplexers inside the Pin Mux block (shown in Figure 2).

```
1 void Pinmux_Config(){
2     *SYSMGR_I2C0USEFPGA = 0;
3     *SYSMGR_GENERALIO7 = 1;
4     *SYSMGR_GENERALIO8 = 1;
5 }
```

Figure 4. C code that configures Pin Mux to connect the ADXL345's I2C wires to I2C0

4.2 Configuring I2C0

Once the ADXL345's I2C wires are connected to I2C0, you must configure I2C0 for communication with the ADXL345. Figure 5 shows the function `I2C0_Init()` that configures I2C0 with the appropriate settings. Important lines of the code are described below:

- Line 4 aborts any pending transmissions and disables I2C0. This is required before modifying any of I2C0's settings.
- Line 7 polls the enable status register until the I2C0 is fully disabled.
- Line 11 writes 0x65 to I2C0_CON, which configures I2C0 to:
  - function as an I2C master. It will control the ADXL345, which functions as the slave.
  - run in fast mode (400 kbit/s), as supported by the ADXL345.
  - use 7-bit addressing mode, as required by the ADXL345.
  - use restart conditions, as supported by the ADXL345.
- Line 14 writes 0x53 to I2C0_TAR, which configures I2C0 to target the ADXL345.
- Lines 19 and 20 configure the SCL clock that will drive the ADXL345. The ADXL345 requires the SCL clock period to be at least 2.5 μs, the SCL high time to be at least 0.6 μs, and the SCL low time to be at least 1.3 μs. All three conditions are met by setting the high period to be 0.9 μs (90 cycles of the 100 MHz input clock to I2C0), and setting the low period to be 1.6 μs (160 cycles of the 100 MHz input clock to I2C0).

- Line 23 re-enables I2C0 now that all settings have been configured, and Line 26 waits until I2C0 reaches its operational state.

```
1  void I2C0_Init(){
2
3      // Abort any ongoing transmits and disable I2C0.
4      *I2C0_ENABLE = 2;
5
6      // Wait until I2C0 is disabled
7      while(((*I2C0_ENABLE_STATUS)&0x1) == 1){}
8
9      // Configure the config reg with the desired setting (act as
10     // a master, use 7bit addressing, fast mode (400kb/s)).
11     *I2C0_CON = 0x65;
12
13     // Set target address (disable special commands, use 7bit addressing)
14     *I2C0_TAR = 0x53;
15
16     // Set SCL high/low counts (Assuming default 100MHZ clock input to
          I2C0 Controller).
17     // The minimum SCL high period is 0.6us, and the minimum SCL low
          period is 1.3us,
18     // However, the combined period must be 2.5us or greater, so add 0.3us
          to each.
19     *I2C0_FS_SCL_HCNT = 60 + 30; // 0.6us + 0.3us
20     *I2C0_FS_SCL_LCNT = 130 + 30; // 1.3us + 0.3us
21
22     // Enable the controller
23     *I2C0_ENABLE = 1;
24
25     // Wait until controller is powered on
26     while(((*I2C0_ENABLE_STATUS)&0x1) == 0){}
27 }
```

Figure 5. A function that configures I2C0.

4.3 Reading and Writing the ADXL345 Internal Registers

Once the ADXL345's I2C wires are routed to I2C0 and I2C0 has been configured, you can use I2C0's memory-mapped registers to read and write the ADXL345 internal registers.

The `ADXL345_REG_READ(uint8_t address, uint8_t *value)` function shown in Figure 6 performs a read of a single internal register at internal address `address` and writes the value read into `value`. It does this in three steps. First, it sends the address of the target register, along with a START signal (line 5). Second, it sends the read request (line 8). Finally in lines 11 and 12, the function waits until I2C0 has received a response from ADXL345, then writes the value to `value`.

```
1  // Read value from internal register at address
2  void ADXL345_REG_READ(uint8_t address, uint8_t *value){
3
4      // Send reg address (+0x400 to send START signal)
5      *I2C0_DATA_CMD = address + 0x400;
6
7      // Send read signal
8      *I2C0_DATA_CMD = 0x100;
9
10     // Read the response (first wait until RX buffer contains data)
11     while (*I2C0_RXFLR == 0){}
12     *value = *I2C0_DATA_CMD;
13 }
```

Figure 6. A function that reads the ADXL345 internal registers.

The `ADXL345_REG_WRITE(uint8_t address, uint8_t value)` function shown in Figure 7 writes the value `value` to the internal register at address `address`. It does this in two steps. First, it sends the address of the target register, along with a START signal. Then it sends the value `value`.

```
1  // Write value to internal register at address
2  void ADXL345_REG_WRITE(uint8_t address, uint8_t value){
3
4      // Send reg address (+0x400 to send START signal)
5      *I2C0_DATA_CMD = address + 0x400;
6
7      // Send value
8      *I2C0_DATA_CMD = value;
9  }
```

Figure 7. A function that writes to ADXL345 internal registers.

The `ADXL345_REG_MULTI_READ(uint8_t address, uint8_t values[], uint8_t len)` function shown in Figure 8 performs a read of `len` consecutive internal registers, starting at internal address `address`. It stores the values read in the array `values`. This function is used when reading the six DATA registers inside the ADXL345, which are at consecutive addresses 0x32 - 0x37. Because this function performs a single read of multiple consecutive registers, it ensures that none of the DATA registers are modified while the read is being performed. This operation is preferable to performing multiple single reads, where the DATA registers could be modified if the ADXL345 samples its acceleration values in between the reads.

```
1  // Read multiple consecutive internal registers
2  void ADXL345_REG_MULTI_READ(uint8_t address, uint8_t values[], uint8_t
       len){
3
4      // Send reg address (+0x400 to send START signal)
5      *I2C0_DATA_CMD = address + 0x400;
6
7      // Send read signal len times
8      int i;
9      for (i=0;i<len;i++)
10         *I2C0_DATA_CMD = 0x100;
11
12     // Read the bytes
13     int nth_byte=0;
14     while (len){
15         if ((*I2C0_RXFLR) > 0){
16             values[nth_byte] = *I2C0_DATA_CMD;
17             nth_byte++;
18             len--;
19         }
20     }
21 }
```

Figure 8. A function that reads `len` consecutive registers.

4.4 Configuring the ADXL345

The ADXL345 has various settings that can be configured to alter its operation. These settings are configured by writing to the ADXL345 control registers, which you can do using the `ADXL345_REG_WRITE` function. Figure 9 shows the function `ADXL345_Init()` which configures the ADXL345 to run in +/- 16 g mode, and sample acceleration at a rate of 100 Hz. It also configures the ADXL345 to run in full resolution mode, which forces a resolution of 3.9 mg (the least significant bit of the DATA values represents 3.9 mg).

```
1  // Initialize the ADXL345 chip
2  void ADXL345_Init(){
3
4      // +- 16g range, full resolution
5      ADXL345_REG_WRITE(ADXL345_REG_DATA_FORMAT, XL345_RANGE_16G |
           XL345_FULL_RESOLUTION);
6
7      // Output Data Rate: 200Hz
8      ADXL345_REG_WRITE(ADXL345_REG_BW_RATE, XL345_RATE_200);
9
10     // The DATA_READY bit is not reliable. It is updated at a much higher
           rate than the Data Rate
11     // Use the Activity and Inactivity interrupts as indicators for new
           data.
12     ADXL345_REG_WRITE(ADXL345_REG_THRESH_ACT, 0x04); //activity threshold
13     ADXL345_REG_WRITE(ADXL345_REG_THRESH_INACT, 0x02); //inactivity
           threshold
14     ADXL345_REG_WRITE(ADXL345_REG_TIME_INACT, 0x02); //time for inactivity
15     ADXL345_REG_WRITE(ADXL345_REG_ACT_INACT_CTL, 0xFF); //Enables AC
           coupling for thresholds
16     ADXL345_REG_WRITE(ADXL345_REG_INT_ENABLE, XL345_ACTIVITY |
           XL345_INACTIVITY );  //enable interrupts
17
18     // stop measure
19     ADXL345_REG_WRITE(ADXL345_REG_POWER_CTL, XL345_STANDBY);
20
21     // start measure
22     ADXL345_REG_WRITE(ADXL345_REG_POWER_CTL, XL345_MEASURE);
23 }
```

Figure 9. A function that configures the ADXL345 mode of operation.

4.5 Reading the Acceleration Data

Now that the ADXL345 is configured for operation, you can read acceleration data from its DATA registers. Figure 10 shows the function `ADXL345_XYZ_READ(int16_t szData16[3])` which reads the acceleration for the X, Y and Z axes and stores the data in `szData16[0]`, `szData16[1]`, and `szData16[2]`, respectively. Line 5 calls `ADXL345_REG_MULTI_READ` to read the ADXL345's six DATA registers. This reads two registers for each axis, one representing the bottom 8 bits and the other representing the top 8 bits of the acceleration sample. Lines 7 to 9 combine these two halves for each axis and write the 16-bit values into `szData16`. The least-significant bit of these values represents 3.9 mg, and the values range from -4096 to 4095 (a resolution of 13 bits).

```
1  // Read acceleration data of all three axes
2  void ADXL345_XYZ_Read(int16_t szData16[3]){
3
4      uint8_t szData8[6];
5      ADXL345_REG_MULTI_READ(0x32, (uint8_t *)&szData8, sizeof(szData8));
6
7      szData16[0] = (szData8[1] << 8) | szData8[0];
8      szData16[1] = (szData8[3] << 8) | szData8[2];
9      szData16[2] = (szData8[5] << 8) | szData8[4];
10 }
```

Figure 10. A function that reads the acceleration data for the x, y, and z axes.

The ADXL345 is set to sample acceleration 100 times a second, which is very slow relative to a modern processor. This makes it possible for a program to read the ADXL345 far more often than 100 times each second, leading to duplicate reads of the same samples. To prevent this, we can check the `Data_ready` bit of the ADXL345's `INT_SOURCE` register and only call `ADXL345_XYZ_Read` when there is new data available. Figure 11 shows the function `ADXL345_IsDataReady()` which checks the `Data_ready` bit and returns true if there is new data, and false otherwise.

```
1  // Return true if there is new data
2  bool ADXL345_IsDataReady(){
3      bool bReady = false;
4      uint8_t data8;
5
6      ADXL345_REG_READ(ADXL345_REG_INT_SOURCE,&data8);
7      if (data8 & XL345_ACTIVITY)
8          bReady = true;
9
10     return bReady;
11 }
```

Figure 11. A function that checks if there is new acceleration data.

4.6 Putting it All Together: An Example C Program

Figure 12 shows a simple program that calls the functions described in the previous sections to initialize the required components and then loops forever to read and print out acceleration data. Important lines of the code are explained below:

- Line 1 includes the header file `ADXL345.h`, which contains all of the functions for working with the ADXL345 and the I2C0 controller.
- Line 11 calls the function `Pinmux_Config()` which configures the Pin Mux block to connect the ADXL345's I2C wires to I2C0.
- Line 14 calls the function `I2C0_Init()` which configures the I2C0 controller to communicate with the ADXL345 chip.

```
1  #include "ADXL345.h"
2  #include <stdio.h>
3
4  int main(void){
5
6      uint8_t devid;
7      int16_t mg_per_lsb = 4;
8      int16_t XYZ[3];
9
10     // Configure Pin Muxing
11     Pinmux_Config();
12
13     // Initialize I2C0 Controller
14     I2C0_Init();
15
16     // 0xE5 is read from DEVID(0x00) if I2C is functioning correctly
17     ADXL345_REG_READ(0x00, &devid);
18
19     // Correct Device ID
20     if (devid == 0xE5){
21         // Initialize accelerometer chip
22         ADXL345_Init();
23
24         while(1){
25             if (ADXL345_IsDataReady()){
26                 ADXL345_XYZ_Read(XYZ);
27                 printf("X=%d mg, Y=%d mg, Z=%d mg\n", XYZ[0]*mg_per_lsb,
                       XYZ[1]*mg_per_lsb, XYZ[2]*mg_per_lsb);
28             }
29         }
30     } else {
31         printf("Incorrect device ID\n");
32     }
33
34     return 0;
35 }
```

Figure 12. C code that reads acceleration data from the ADXL345

- Line 17 reads the Device ID internal register of the ADXL345 chip. If the I2C communication is working correctly, the device ID 0xE5 is read.
- Line 22 calls the function `ADXL345_Init()` which configures the ADXL345 chip to start measuring acceleration data.
- Line 26 calls the function `ADXL345_XYZ_Read(XYZ)` which reads the acceleration data for all three axes, and stores the data in the XYZ array. The least significant bit of the acceleration values read from the ADXL345 represent 3.9 mg increments (approximately 0.038 m/s^2, since g = 9.81 m/s^2). To account for this when printing out the data in line 23, the values are multiplied by `mg_per_lsb` to output numbers in mg units.