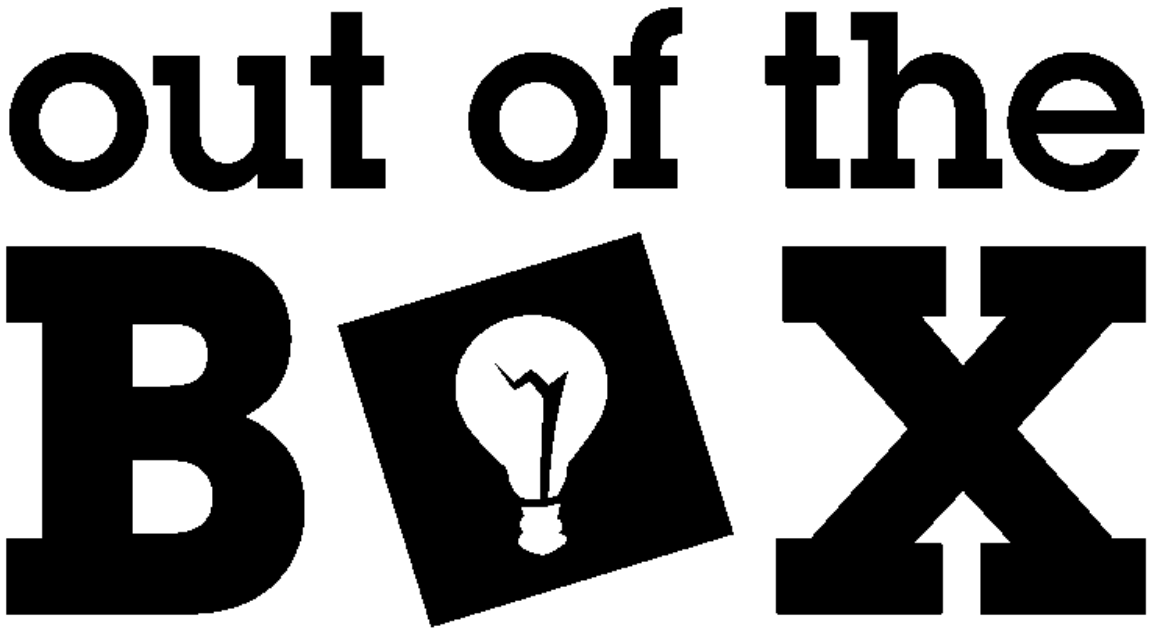


Epiphany Software Reference v2.0

Last Updated August 25, 2012

Table of Contents

Summary

This manual has been created in order to document the code base associated with the Epiphany DIY. This is by no means a comprehensive manual to the Xmega processor. For detailed information on the processor its self please consult the device datasheets. Likewise there is an expectation that the reader has a basic understanding of programming, and more specifically C programming syntax. If this is not the case a reference such as "The C Programming Language" by Brian W. Kernighan, and Dennis M. Ritchie is highly recommended.

Coding With The Xmega An Overview

The Xmega processor is the "New Hottness" of the Atmel 8-bit microcontrollers. These chips are packed to the brim with peripherals, compared to the old Megas, and especially the Tiny series. In addition to this the CPU clock frequency has not only been increased, but it has now become run-time configurable. Another major change is the code implementation itself. Previously all the processors such as the Mega and Tiny utilized non-hierarchical peripheral registers. The same of the descrete register was simply called. Today many manufacturers have started to structure peripherals and their associated registers for greater code readability, portability and organization. The Xmega code base is no exception. Registers are now called in a "parent"."child" syntax.

Example1:

PORTD.DIRSET = 1; //set bit 0 to '1' in the Data Direction register for i/o port D

How To Determine the Syntax/hierarchy

Method1

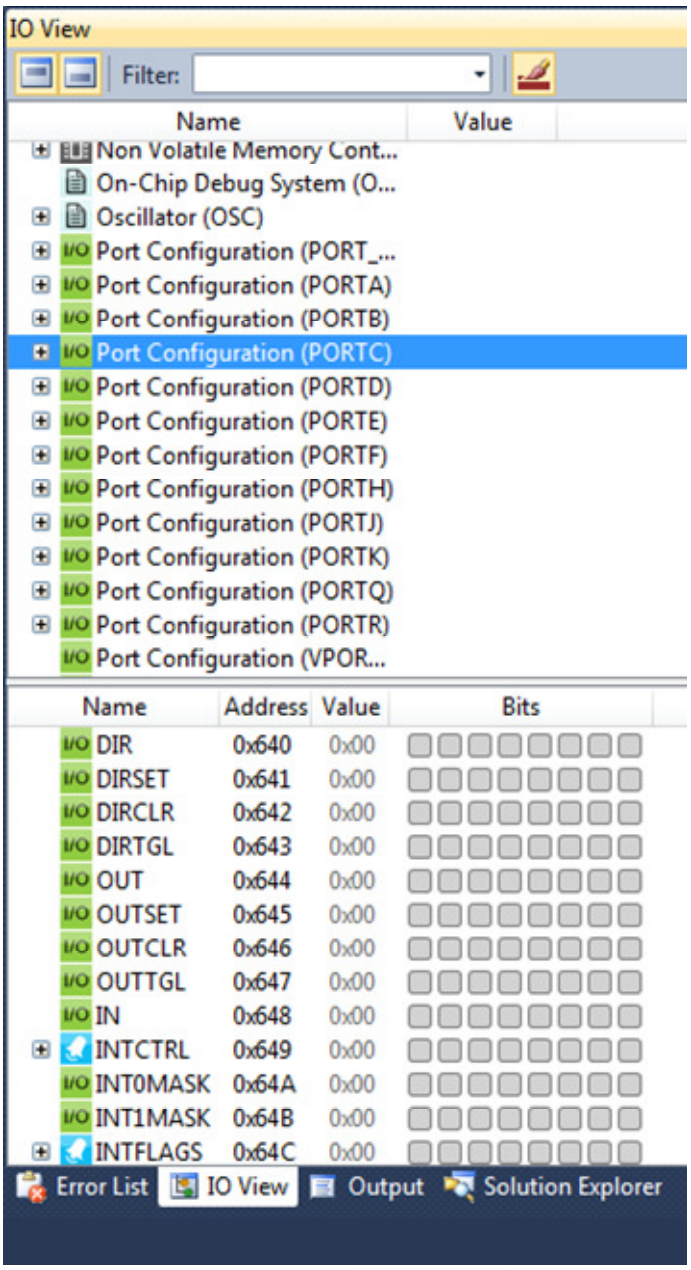
For the savvy C programmer.

When programming for any Atmel processor one of the required dependencies is <io.h>. This header file points to the device specific header containing definitions of all the registers i.e. assignments to their respective memory map locations. To access the device specific io header file in AVR/ATMEL Studio first locate the "Solution Explorer" window. Once there expand the "Dependencies" folder to find a file like "iox64a1.h". Note the exact name will vary based upon the specific chip you Epiphany has. The '64' is the memory size, and could be 128 in some cases. The 'a1' portion specifies the series. So this could also be 'a1u'.

Once the file is located determining the hierarchy of peripherals and their registers involves just looking though the declaration of structures.

Method2

In this author's opinion the easiest way of determining hierarchy is to use "IO View" inside of AVR/ATMEL Studio.



Shown above is an IO View window for a Xmega series processor. The IO View utility allows you to select the peripheral in question, allowing you to see in detail the associated registers.

Timing

CPU Clock

The CPU frequency of the Xmega processors can be changed during run time. By default the clock frequency is set to 2MHz, and the maximum speed is 32MHz. Many frequencies can be set by playing with the PLL settings, and by switching between various clock sources. For more information consult the device doumentation

clockInit(void)

This function sets the clock frequency of the Xmega to 32MHz. This function is typically called upon startup.

Real Time Clock

The real time clock is a means of using timing with the Epiphany. The functions associated with this peripheral are designed to setup delays that do not interfere with CPU operation. Put simply the clock is set with a value of time to count down. In code a flag is read to determine whether the delay has elapsed. This allows the CPU to continue its business with the rest of the program until the specified time has elapsed.

Functions

RTC_DelayInit(void)

This function is declared in order to initialize the RTC peripheral. This must be declared before the RTC_Delay_ms() can be called.

RTC_Delay_ms(uint16_t delayTime)

This function sets up the RTC to count down **delayTime** milliseconds. To determine that the specified duration has elapsed a Boolean i.e. true/false statement should be implemented utilizing one of the flags below.

Flags

delayNotOver

This is a flag that evaluates as true while the RTC peripheral is still counting down. So as the flag reads, the delay is not yet over.

delayOver

This is the same flag as **delayNotOver**, except the logic has been flipped.

Input and Output

SET CLR, and TGL

When manipulating bits at in a register the conventional method is to use and/or instructions via bit-masking. The Xmega accelerates this process by having hardware set clear and toggle functions.

- A ‘1’ in a SET register bit will set the same bit in the host register
- A ‘1’ in a CLR register bit will clear the same bit in the host register
- A ‘1’ in a TGL register bit will toggle the same bit in the host register

Example2:

Traditional bitmask

PORTD.OUT |= 1<<5; //sets bit 5. Requires a read, an or and a store instruction

Hardware bitmask

PORTD.OUTSET = 1<<5; //sets bit 5. Requires just a store instruction

Direction Registers

When using an i/o pin the first priority is choosing whether that pin is an input or an output. This is done via the direction register associated with the respective port. The convention for AVR processors is '1' is for output and '0' is for input. It's also worth knowing that by default every pin is set as an input. This prevents bus conflicts due to the chip being unprogrammed.

Example3:

PORTC.DIR = 1<<3;//port C pin 3 is now an output, while the rest of the port is an input.

Output Register

The Output register is used for controlling the voltage level of an output pin. The relationship is high true.

Example4:

PORTC.DIRSET = 0xFF;//set the port to all outputs

PORTC.OUTCLR = 0x55;//send out hex '55' to port C

Input Register

The input register is used to read the data in from a pin/port.

Example5:

PORTE.DIRCLR = 0x00;//set all of pins of port E to inputs

If(PORTE.IN & 1<<2)someFunctionCall();//This is an example of not only reading but masking a bit

Configuring Ports

Each pin of the Xmega has pull-up and pull-down resistors available, as well as other useful features. Manipulating the PIN#CTRL registers these features can be harnessed. For details on all these features consult the device datasheets.

Example6:

PORTF.PIN5CTRL = 0x10;//enables a pull-down resistor on pin 5 of port C

PORTF.PIN6CTRL = 0x18;//enables a pull-up resistor on pin 6 of port C

Analog input

The Epiphany has up to 16 analog inputs 8 of which are scaled for 5V. The precision is 12-bit. Note the precision will be truncated if the ADC voltage selection is 3.3V on the PCB

The ADC modules on the Xmega have several “channels” and result registers. These are used to request up to 4 conversions as once, that will then be sequentially calculated. This is faster and simpler than reusing a single result register.

adcInit(ADC_t *adcModule)

This function is used to initialize the ADC. **adcModule** is either ADCA or ADCB.

Example7:

adcInit(&ADCA);//this will initialize ADCA note & is used since the address is what is being passed

adcChannelMux(ADC_t *adcModule, uint8_t channelNum, uint8_t inputPin)

adcChannel Mux is used to set the input source(**inputPin** 0-7) for one of the 4 ADC channels (**channelNum** 0-3) to the respective **adcModule** ADCA or ADCB.

Example8:

adcChannelMux(&ADCB,0,4);//channel 0 of ADCB now will convert from the source on pin 4

analogRead(ADC_t *adcModule, uint8_t channelNum)

analogRead is used to get the actual data from the analog result register of the respective channel (**channelNum** 0-3). Prior to using this function the adcChannelMux function should be used in order to select the conversion source.

Analog output

Coming soon

Actuation

Motors

Functions

motorInit()

This function should be called to initialize the motor controllers.

setMotorEffort(uint8_t motorNum, uint16_t effort, motorDirection_t motorDir)

This function is used to control low power motors with an **effort** value 0-1024. **motorNum** specifies the motor 1-4. **motorDir** is used to set the direction of the motor. The choices are: **MOTOR_DIR_NEUTRAL, MOTOR_DIR_FORWARD, MOTOR_DIR_BACKWARD**

Example9:

setMotorEffort(1,800, MOTOR_DIR_BACKWARD)// moves motor 1 backward at (100*800/1024)%

setMotorEffort_HP(uint8_t hpMotorNum, uint16_t effort, motorDirectionHP_t motorDir)
This function is used to control high power motors with an **effort** value 0-1024. **motorNum** specifies the motor 1-4. **motorDir** is used to set the direction of the motor. The choices are:
MOTOR_DIR_NEUTRAL_HP, MOTOR_DIR_FORWARD_HP, MOTOR_DIR_BACKWARD_HP

Servos

Functions

ATtinyServoInit(void)
This function should be called in order to initialize the Atiny Servo controller.

setServoAngle(uint8_t servoNumber,uint8_t angle)
setServoAngle is used to move the servo **servoNumber** to a set **angle**. Be aware that setting new angles too fast will make it impossible for the servo to keep up. Servo signals are updated every 20ms so writing faster than that will cause the servo to become erratic. Also be aware of the angular speed of the servo. Servos are typically rated by their speed to rotate 60 degrees. It's not recommended to command the servos to run faster than this rating because they will not keep up

Example10:
setServoAngle(5,45);//moves the servo # 5 to an angle of 45 degrees

setServoPosition(uint8_t servoNumber, uint16_t servoPosition)
setServoPosition is similar to setServoAngle, but instead of using angles the means of controlling the servo is a number **servoPosition.** In order to understand this number an explanation of servo signaling is required. Servo signals use a techique called Pulse Code Modulation. The protocol works as follows, a positive digital pulse is sent once every 20ms. The width of this pulse generally varies from 600 to 2100us, and consequently moves the servo from 0 to 180 degrees. Getting back to **servoPosition** this number corresponds to the pulse width in increments of 1/3us.

Example11:
setServoPosition(4,2250);//This command sends a 1500us pulse to servo 4 which corresponds to a 90 degree center position.

getServoAngle(uint8_t servoNumber)
This function returns the angle for a given servo **servoNumber**

getServoPosition(uint8_t servoNumber)
This function returns the servoPosition in 1/3us for a given servo **servoNumber**

Communication

<stdio.h> Crash Course

Special Character Reference

specifier	Output	Example
c	Character	a
d or i	Signed decimal integer	392
e	Scientific notation (mantissa/exponent) using e character	3.9265e+2
E	Scientific notation (mantissa/exponent) using E character	3.9265E+2
o	Unsigned octal	610
s	String of characters	sample
u	Unsigned decimal integer	7235
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (capital letters)	7FA
p	Pointer address	B800:0000
n	Nothing printed. The argument must be a pointer to a signed int, where the number of characters written so far is stored.	
%	A % followed by another % character will write % to stdout.	%

Table taken from http://www.cplusplus.com

Special Character Escape Sequence

alert (beep)	\a
backslash	\\
backspace	\b
carriage return	\r
double quote	\"
formfeed	\f
horizontal tab	\t
newline	\n
null character	\0
single quote	\'
vertical tab	\v
question mark	\?

Taken from http://web.mit.edu

Functions

fprintf

fprintf is the basis of output communications via the USARTs, and the LCD. By using this simple function formatted output can be obtained.

Example12:

fprintf(&USB_str,"Hello world\r\n");

That prints:

Hello world followed by a carriage return and a new line

This would be sent out the USB to serial port.

fprintf(&Xbee_str,"I am %d years old\r\n",24);

That prints:

I am 24 years old followed by a carriage return and a new line

This would be sent out the Xbee to serial port.

For further details on fprintf and the whole printf family consult a C reference

fscanf

fscanf is a means of formatted input. This is how data is read in through the USART peripherals

Example13:

Fscanf(&usartC1_str,"%c",&someVariable);

This will read in a character from the USARTC1 rx buffer and store it to the variable "someVariable".

For further details on fscanf and the whole scanf family consult a C reference

USART

Streams

Every USART has a stream associated with it. The convention is **usartC0_str** where 'C0' is the particular usart. C is the port and 0 is the designation for which usart on that port is used.

For simplicity and readability aliases were created for the USB to serial, and Xbee ports. These aliases are **USB_str**, and **Xbee_str**

Functions

usartInit(USART_t *uart, long baud)
use this function to initialize the usart.

Example14:
usartInit(USARTC0,115200);//this initializes the usart C0 (USB usart) at a baud of 115200

dataInBuf
dataInBuf is a series of functions used to tell if unread data is in the rx buffer.

Example15:
dataInBufD1();//this function returns a booloean value of true '1' if unread data is present. D1 refers to the port and usart number.

LCD

Streams
LCD_str is the stream of choice for the LCD module

Functions

LCDInit(void)
Call this function **ONLY** if an LCD is connected This function does not sense for the existence of the LCD, and as a result it can cause the program to hang. When using an LCD this function must be called before anything can be displayed.

General
The LCD module uses fprintf similarly to the USART modules, only the escape charaters '\r', and '\n' have special meaning.

'\r' will both clear and home the lcd

'\n' will move the cursor to base position on line 2

Example16:
fprintf(&LCD_str,"\rhello world");

this will clear and home the screen. Then hello world will be printed out

SPI
Coming Soon

TWI (I2C)
Coming Soon

Appendix

Coming Soon

Data Types

Coming Soon

Functions and Library Dependencies

Coming Soon