CHAPTER 4

New APIs and Hacks

The challenge when Intel Galileo was designed was to create a board that would be compatible with Arduino headers and reference language using only the Quark microprocessor, and to have the same capabilities running Linux and making any needed bridges with microcontrollers.

The competitor's boards that run Linux do in fact use microcontrollers to interface with pin headers and all have decent performance.

Intel Galileo and Intel Galileo Gen 2 do not use a microcontroller, which makes sketch implementations easier because there is no bridge between the microprocessor and microcontroller. On other hand, the implementation of Linux drivers and the Arduino reference API on the Linux userspace context without a microcontroller to guarantee performance in real-time is a huge challenge.

This chapter describes the new APIs as well as some of the design workarounds introduced in the first Intel Galileo version. The chapter also discusses some hacks that you can make if the regular Arduino reference API does not meet the needs of your project.

Some APIs work exclusively with Intel Galileo, some only with Intel Galileo Gen 2, and others with both. Thus, each section on APIs hacks includes a note explaining which version it works with.

Servo API

Servo motors are widely used in robots and devices controlled remotely, such as RC planes. At the same time, they are confusing because the practice is a little different than the theory. Let's take a look at the theory and the practice.

The Theory versus Practice

In theory, the servo rotates between 0 to 180 degrees using PWM pulses that vary between 1 to 2 milliseconds in a period of 20 milliseconds (50Hz) and operating between 4.5 to 6 VDC. Figure 4-1 shows the servo movement according to the pulses to 0, 90, and 180 degrees.

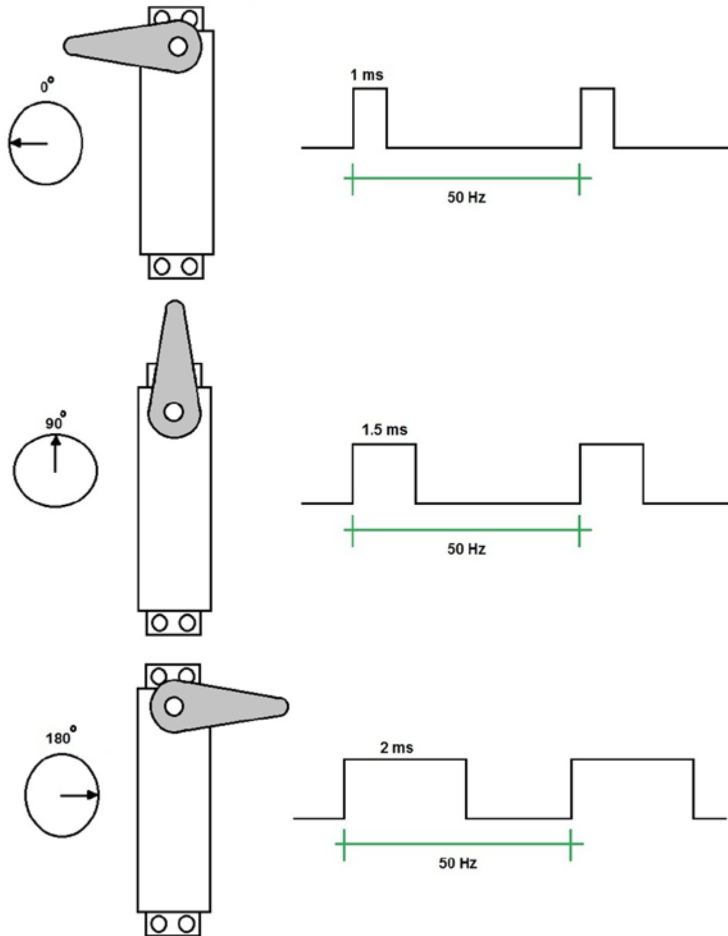

Figure 4-1. *How servos work in theory*

With only three wires, the VCC wire is usually red, the ground is usually black, and the pulse signal wire comes in different colors depending on the manufacterer. It's possible to move the servo to a specific position between 0 to 180 degrees.

To exercise the servos and understand how they really work with the new APIs, connect a servo to an Intel Galileo header. See the materials list in Table 4-1.

Table 4-1. *Materials List for Servo Exercises*

Number	Description
2	Servo Futaba S3003 or equivalent

This servo can be found online for around $7 to $15 US.

Schematic for Servo: One Servo

Connect the VCC to 5V, the ground to a ground, and the pulse to pin 9, as the schematic shows in Figure 4-2.

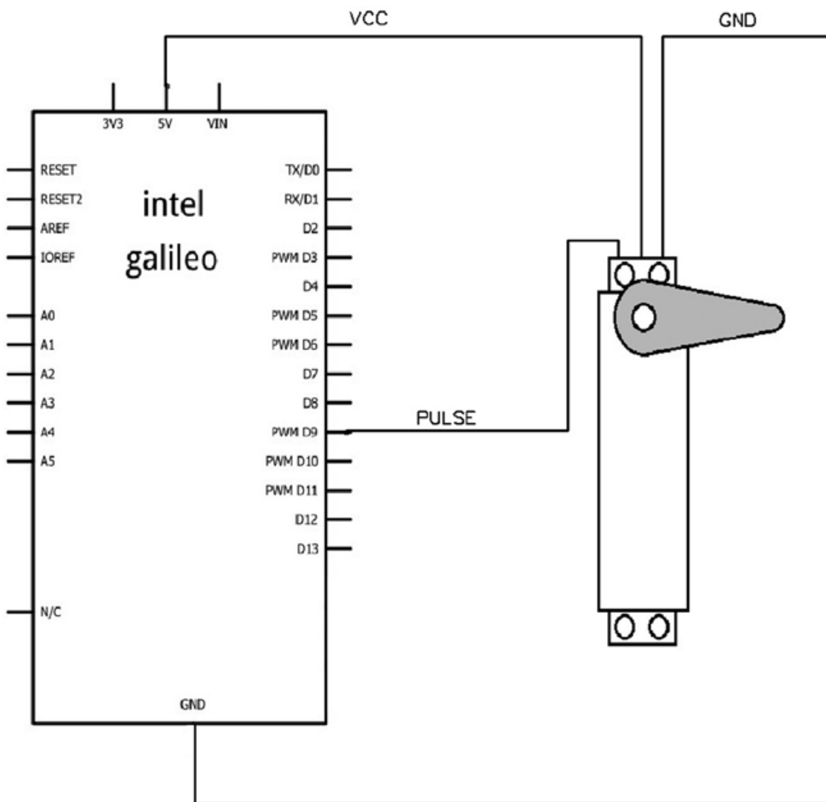

Figure 4-2. *Connecting the servo to Intel Galileo*

With this theory in mind if you decide to buy a servo online, you will see quite a variety of servos with prices ranging between $5 and $15 US.

Choose a servo, such as the servo Futaba S3003 mentioned in the material list Table 4-1.

Then write the code shown in Listing 4-1.

Listing 4-1. servotheory.ino

```
#include <Servo.h>

Servo myservo;

void setup()
{
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop()
{

  // in theory 0 degrees
  myservo.writeMicroseconds(0);
  delay(2000);

  // in theory 180 degrees
  myservo.writeMicroseconds(2000);
  delay(2000);

}
```

The code creates a servo object that's connected to pin 9 using `myservo.attach(9)`. The loop sends a pulse of 0 microseconds that in theory means 0 with `myservo.writeMicroseconds(0)`. After two seconds, it sends a pulse of 2,000 microseconds (2ms) with `myservo.writeMicroseconds(2000)`, which would in theory move the servo to 180 degrees.

If you are using the Futaba S3003 or equivalent, you will realize the servo doesn't move to 180 degrees. It moves to something between 0 and 160 degrees.

Change the code to use the `write()` method, whereby you can exactly define the desired angles. See Listing 4-2.

Listing 4-2. servo_write_angles.ino

```
#include <Servo.h>

Servo myservo;

void setup()
{
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop()
{

  // move to 0 degrees
  myservo.write(0);
  delay(2000);

  // move to 180 degrees
  myservo.write(180);
  delay(2000);

}
```

This servo specifically does not move to 0 degrees with a pulse of 1 millisecond nor to 180 degrees with a pulse of 2,000 milliseconds. What the IDE is doing differently when angles are passed to the `write()` method?

If you run the code in Listing 4-2 you will realize the servo really moves from 0 to 180 degrees without problems.

After you install the IDE, open the file `Servo.h` in the directory `...\arduino-1.5.3\hardware\arduino\x86\libraries\Servo`. Take a look at the definition of this file:

```
#define MIN_PULSE_WIDTH       544     // the shortest pulse sent to a servo
#define MAX_PULSE_WIDTH      2400     // the longest pulse sent to a servo
```

When 0 degrees is specified in the `write()` method, the servo library uses 544 microseconds (0.544ms) and when 180 degrees is specified, it uses 2400 microseconds (2.4ms). The servo used in this experiment actually works with these values, not the theoretical values.

This range between 544 and 2400 microseconds is the most common range that attends the servos you can buy online. You can also change the maximum and minimum if you attach the servos specifying the maximum and minimum according to the specification of the servo. For example, suppose you got a servo where 0 degrees requires a pulse of 800 microseconds and 180 degrees requires a pulse of 2200 microseconds. In this case, it is possible to specify boundaries like so:

```
myservo.attach(9, 800, 2200);  // attaches the servo on pin 9 to the servo
object
```

But this method only accepts maximum and minimum pulses according to the range specified by MIN_PULSE_WIDTH and MAX_PULSE_WIDTH (544 and 2400 microseconds, respectively).

One last observation regarding the write() method. If an angle bigger than 180 degrees is passed to this function, it will be considered microseconds and write() will behave the same way as the writeMicroseconds() method. For example, both methods do the same thing:

```
// move to 180 degrees
myservo.writeMicroseconds(2000);

// the value is > 180, so it means microseconds
myservo.write(2000);
```

The Mistake with Intel Galileo and Servos

If you're using the Intel Galileo Gen 2 board, this section is irrelevant because Intel Galileo Gen 2 uses an expander PCA95555 that works perfectly.

The first Intel Galileo version in the market uses a GPIO expander called CY8C9540A and manufactured by Cypress. It communicates with Intel Galileo through the I2C protocol and unfortunately does not offer high enough precision to have a granularity of 1 degree at a frequency of 50Hz. In other words, the PWM generated by this IC can't provide pulses in one-to-one microsecond increments and you can't move the servo in one-to-one degree increments in a frequency close to 50Hz.

This expander does not work at 50Hz but does work at 48Hz, which is acceptable for working with servos because the tolerance of the servos to the frequencies is around 5 to 10 percent (47.5Hz to 55Hz) and that's acceptable for the domain of 50Hz.

The Cypress CY8C9540A datasheet is in the folder code/datasheets of this chapter in the file named CY8C95x0A.pdf or you can access it at http://www.cypress.com/?rID=3354.

However, some servos might work in higher frequencies and CY8C9540A offers a better angular granularity if the frequency is increased. The duty cycle of this expander is programmed using I2C and only accepts eight bits, which is the source of the problem. Table 4-2 provides a small sample that explains the issues.

Table 4-2. *CY8C9540A Angle Granularity Min Pulse 0.544ms and Max 2.4ms*

Angle	Pulse Needed	Duty Cycle Byte @ 188Hz [0-255]	Duty Cycle Byte @ 48Hz [0-255]
1	0.554311111	26	6
2	0.564622222	27	6
3	0.574933333	27	6
4	0.585244444	28	7
5	0.595555556	28	7
6	0.605866667	29	7
7	0.616177778	29	7
8	0.626488889	30	7
9999	0.636800000	30	7
10	0.647111111	31	7
11	0.657422222	31	7
12	0.667733333	32	8

In the first column of Table 4-2 represents the angle to move, the second column is the ideal pulse for the respective angle considering the minimum pulse of 544 microseconds and maximum pulse of 2400 microseconds (the standard used by the Arduino reference implementation). The third and fourth columns represent the byte necessary to send to the controller to achieve the desired pulse.

As you can see, when the CY8C9540A expander is working at 48Hz the same byte 7 appears to the angles 4 to 11 degrees. If you try to move the servo between 4 and 11 degrees the servo will not move because the byte is the same. When the angle becomes 12 degrees, the byte changes to 8. The servo movements will be choppy and will only move from 4 to 8 degrees drastically, with horrible precision.

However, if you work with a frequency of 188Hz and move 4 to 11 degrees, the servo will move a more smoothly because the granularity is better. For example, when the servo moves from 4 to 6 degrees, it will not move at 5 degrees, because 4 and 5 degrees share the same byte (28). At 6 degrees, the byte is 29, so the servo moves.

So, when you're working with servos at 188Hz, it's still not possible to move in one-to-one degree increments, but the granularity is much better compared to working with the expander at 48Hz.

Table 4-2 contains just a small angle set from 1 to 12 degrees. In the folder called `code` of this chapter, there is a spreadsheet named `frequence_versus_resolution.xls` that contains all angles from 0 to 180 and the resolution offered by CY8C9540A.

What Is New in Servo API?

This section is applicable to Intel Galileo only.

In order to attenuate the problem of servo granularity introduced by the expander CY8C9540A, the software team added an optional argument to the `attach()` method and included two new methods in Intel Galileo—`set48hz()` and `set188hz()`. Keep in mind that the frequency control is not independent per servo and once it's set for one servo, all the other servos must use the same frequency. Otherwise, the servos will not work properly.

void Servo::set48hz()

Forces CY8C9540A to work at 48Hz. As explained, the resolution is very bad and it is useful only if your application does not require precision.

void Servo::set188hz()

Forces CY8C9540A to work at 188Hz and offers a resolution of 2 to 2 degrees. It is not perfect, but for simple robots and RC controllers it is enough. On Intel Galileo, this is the initial frequency used when a servo is created. Thus, if your servo does not support 188Hz, you can force the `attach()` method to set 48Hz in the servo initialization to avoid servo burns.

uint8_t Servo::attach(int16_t pin, bool force48hz = false)

The only difference with this method compared to the traditional Arduino reference servo API is the inclusion of the argument `force48hz`. If this argument is set to `true`, the initial frequency in the servo will be 48Hz; otherwise the default of 188Hz is assumed.

The other argument, called `pin`, only specifies the PWM pin to be used to emit the pulse signals to the servo.

uint8_t Servo::attach(int pin, int min, int max, bool force48hz = false)

Again, the only difference with this method compared to the traditional Arduino reference servo API is the inclusion of argument `force48hz`. If this argument is set to `true`, the initial frequency in the servo will be 48Hz; otherwise, the default of 188Hz is assumed.

The `pin` argument specifies the PWM pin to be used to emit the pulse signals to the servo. The and `min` and `max` arguments specify the minimum and maximum frequency in microseconds, respectively.

Schematic for Servo: Two Servos

In order to create some practical tests with these new methods, let's assemble a circuit using two servos. Connect both VCC to 5V and both grounds to ground. Connect one of the servos' pulse to pin 9 and the another one to pin 3, as the schematic shown in Figure 4-3 shows.

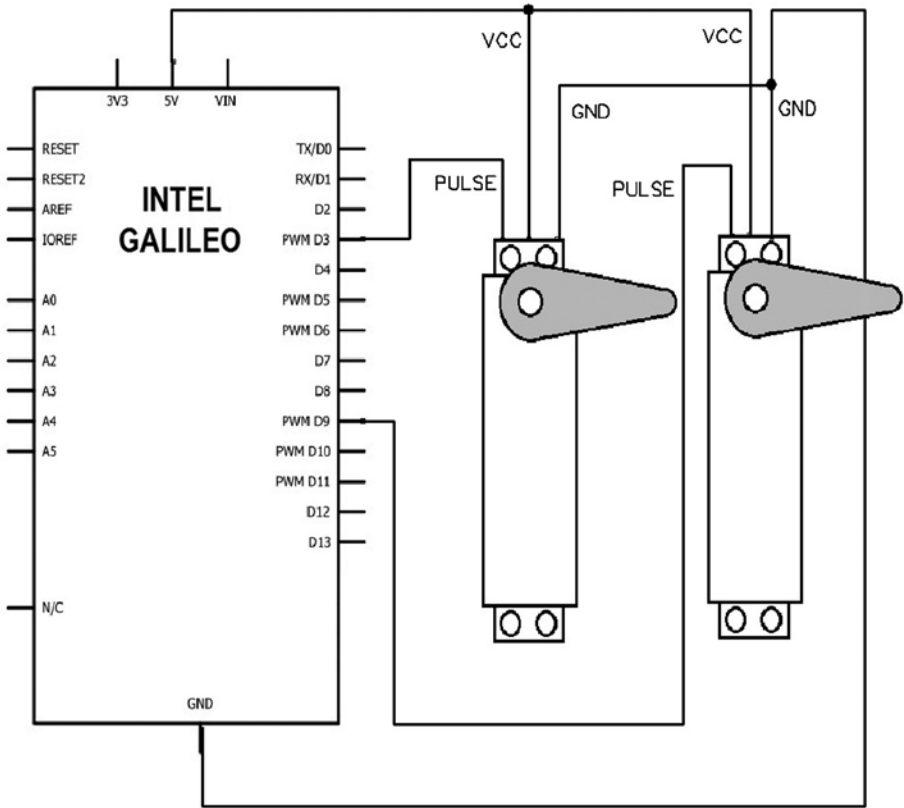

Figure 4-3. *Two servos connected*

Testing the New Servo APIs

Listing 4-3 shows a program to be used with the schematics of Figure 4-3.

Listing 4-3. servo_set_freq.ino

```
#include <Servo.h>

Servo myservo1;
Servo myservo2;

void setup()
{
  myservo1.attach(9);         // attaches the servo to pin 9 at 188 hz
  myservo2.attach(3, true);   // this must work in 48 hz. All servos will
work at 48hz now
  Serial.begin(115200);
}

void loop()
{

    myservo1.write(180);
    myservo2.write(0);

    delay(2000);

    myservo1.write(0);
    myservo2.write(180);

    delay(2000);

    myservo1.set188hz();; // all servos will work at 188hz

}
```

Reviewing servo_set_freq.ino

In the `setup()` function, two servos are added to the system, one in pin 9 at 188Hz (the default frequency). The second servo object is created specifying a servo on pin 3 at 48Hz.

However, as soon the `myservo2.attach(3, true)` method is called, both servos begin operating at 48Hz.

Then in the `loop()` function, the first servo moves 180 degrees and the second moves to 0 degrees. After two seconds, they invert the order to 0 and 180, respectively.

```
myservo1.write(180);
myservo2.write(0);

delay(2000);

myservo1.write(0);
myservo2.write(180);

delay(2000);
```

Thus, in the first interaction of `loop()`, both servos are initially working in 48Hz. After two seconds, all the servos will be working at 188Hz because one of the servos selected this frequency with `myservo1.set188hz()`. The rest of the `loop()` interactions the servos will remain in 188Hz.

Challenges with Servos

The biggest challenge when working with servos is the lack of standardization and the huge variety of servos available in the market, many with different specifications.

For example, there are servo that reach 0 degrees in 1ms and 180 degrees in 2.2ms. This movement is proportional to the pulse length in one single direction.

However, there are servos limited to 120 degrees, so 1.5ms represents 0 degrees, 1ms represents -60 degrees, and 2ms represents +60 degrees. These are totally different than the example in this chapter and the servo API will not have the expected results. Figure 4-4 represents how these kinds of servos work.

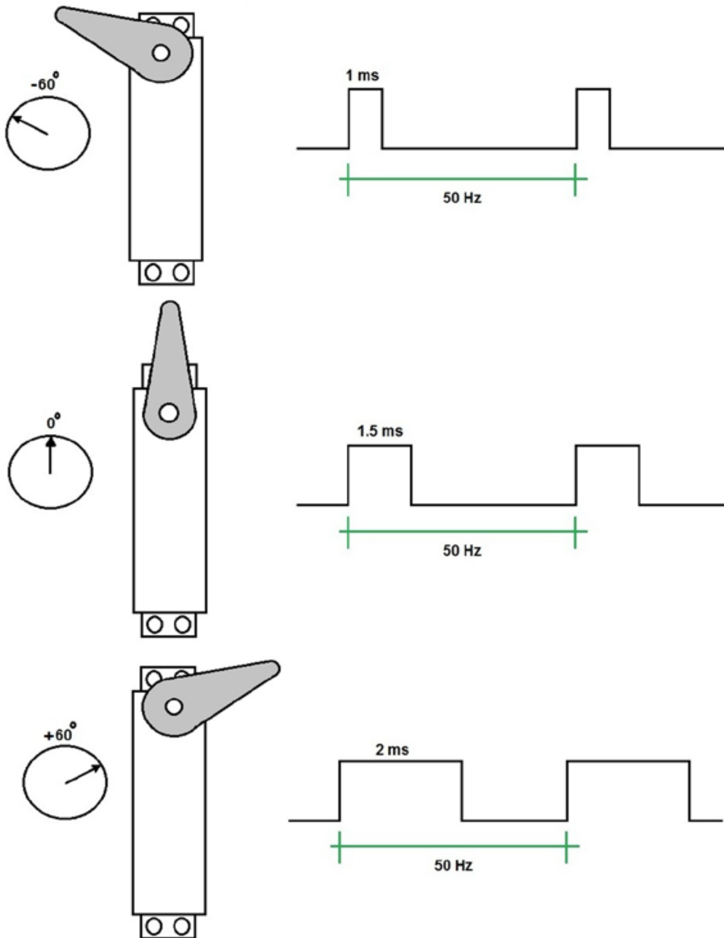

Figure 4-4. *Example of a servo with 0 degrees in 1.5ms*

This does not mean that the servo API is useless in these cases. You simply need to understand your servo specifications and use the API to apply the right pulses at the right time so that the servo moves to the correct angles.

Another common mistake is using servos that do not reach more than 160 degrees. Such servos have an internal mechanical delimiter that prevents them from moving more than 160 degrees.

Some developers remove this mechanical delimiter and make a couple of changes in the servo inputs to make them to have a continuous rotation. In this case, continuous servos do not control the position but the rotation speed.

Another problem is the Arduino reference API. Intel created the servo API according to the reference provided by Arduino, but in practical ways some boards have different behaviors. For example, if you use the Arduino UNO, the initial angle (for most servos) is

90 degrees due to an initial pulse of 1.5ms; however Intel Galileo tries to put your servo initially in 0 degrees. In both implementations, the reference is respected and there is no error in terms of implementation because there is no mandatory requirement that says the initial position must be 0 or 90 degrees.

Serial, Serial1, and Serial2 Objects

This section is applicable to Intel Galileo and Intel Galileo Gen 2.

In Chapter 3, in the section entitled "Debugging with Serial Consoles and Serial Communication," you learned a bit about the usage of Serial to print debug messages from the IDE serial debug console and about the Serial1 object.

In fact Intel Galileo and Intel Galileo Gen 2 count to Serial as any other regular Arduino board, but also include the new Serial1 and Serial2 objects for serial communications.

Serial1 uses pin 0 as its RX port and pin 1 as TX for both Intel Galileo and Intel Galileo Gen 2.

Serial2 uses pin 2 as RX and pin 3 as TX on Intel Galileo Gen 2 and you will use the audio jack on Intel Galileo.

There is a limitation when using Serial2 and a Linux console shell. If Serial2 is used in the sketch, the Linux console is lost because the `mux` are set to redirect the ports used on Linux console to provide the Serial2 functionality.

The baud rates accepted for such objects are 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 230400, and 460800.

The usage is very simple and very similar to the regular Serial object used to debug the sketches. So all methods used to communicate with Serial1 and Serial2 are the same ones explained in Chapter 3 with the Serial object. This section's focus is therefore the few and new details related to Serial1 and Serial2. It includes a simple example for how to use all three objects in the same sketch.

Testing the Serial, Serial1, and Serial2 Objects

This example was created to run on Intel Galileo Gen 2 due to the simplicity of dealing with Serial2 in the I/O headers instead of having to prepare a cable exclusively for the audio jack on Intel Galileo. The idea is to create a sketch that makes a loop between Serial1 and Serial2 using only two wires. Serial1 will transmit a message that will be read by Serial2, and then Serial 2 will read the message and send another message to Serial 1. The process is repeated.

Table 4-3 provides a list of the materials necessary to run this example.

Materials List

Table 4-3. *Materials List for the Serial Object Example*

Number	Description
2	Simple wires for hookup

Schematic for the Serial Example

Figure 4-5 shows how the wires must be hooked up to create a serial loopback.

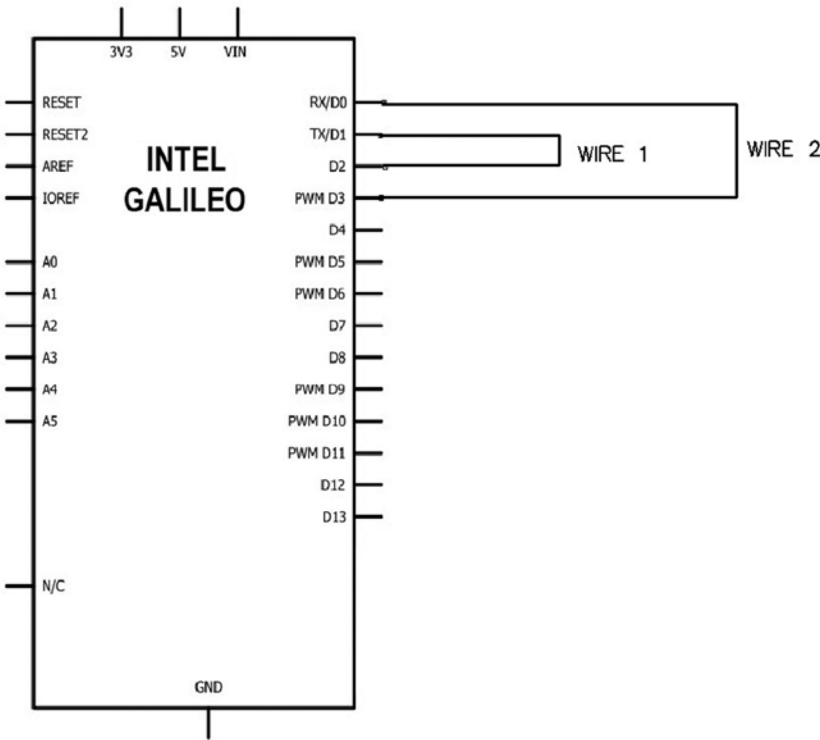

Figure 4-5. *Creating a serial loopback between Serial1 and Serial2*

Serial1 RX (pin 0) is connected to Serial2 TX (pin 3) and Serial1 TX (pin 1) is connected to Serial2 RX (pin 2). This creates a loop between Serial1 and Serial2.

The code for testing is shown in Listing 4-4.

Listing 4-4. all_serials.ino

```
String inputString = "";         // a string to hold incoming data

void setup() {
  // put your setup code here, to run once:
   Serial.begin(115200);    // Serial console debugger
   Serial1.begin(115200);   // PIN 0->RX and 1->TX
   Serial2.begin(115200);   // PIN 2->RX and 3->TX

}

void loop() {

  Serial.println("Transmitting from Serial 1 to Serial 2");
  Serial1.println("Intel"); // this will transmitt using PIN 1
  inputString="";

  // Serial2 will wait for something
  while (Serial2.available()) {
    // get the new byte:
    char inChar = (char)Serial2.read();  // receiving by Serial2 on pin 2

    // add it to the inputString:
    inputString += inChar;

  }

  // Serial 2 receive the word "Intel"
  // let's send the word "Galileo" back to Serial 1

  Serial.print("Message received from Serial 1:");
  Serial.println(inputString);
  inputString = "";

  // transmitting another word to Serial2
  Serial.println("Transmitting from Serial 2 to Serial 1");
  Serial2.println("Galileo");  // transmitting by Serial2 using pin 3

  // Serial1 will wait for something
  while (Serial1.available()) {
    // get the new byte:
    char inChar = (char)Serial1.read();  // receiving by Serial1 using pin 0
```

```
    // add it to the inputString:
    inputString += inChar;

  }

  Serial.print("Message received from Serial 2:");
  Serial.println(inputString);
  inputString = "";

  delay(2000);
}
```

Reviewing all_serials.ino

In the `setup()` function, all three objects are initiated with a 115200 baud rate.

```
Serial.begin(115200);    // Serial console debugger
Serial1.begin(115200);   // PIN 0->RX and 1->TX
Serial2.begin(115200);   // PIN 2->RX and 3->TX
```

In the `loop()` method, the `Serial` object is used only for debugging purposes. Serial1 object starts transmitting the word `"Intel"` to Serial2 using the `println()` method.

```
Serial1.println("Intel"); // this will transmitt using PIN 1
```

Then Serial2 receives the message from Serial1 and reads the whole buffer using the `available()` method. The `available()` method will keep returning the number of bytes available in the buffer and the loop will remain until `available()` returns zero bytes (when there are no more bytes to be read). The `read()` method reads the byte and casts it to a char byte using the `inChar` variable. Once the byte is read the number of bytes reported by the `available()` method is decreased and the byte is accumulated in the string buffer called `inputString`.

```
// Serial2 will wait for something
  while (Serial2.available()) {
    // get the new byte:
    char inChar = (char)Serial2.read();  // receiving by Serial2 on pin 2

    // add it to the inputString:
    inputString += inChar;

  }
```

Once Serial2 reads the "Intel" message from Serial1, the word "Galileo" is sent back to Serial1. Again, the println() method is used for this purpose.

```
  // transmitting another word to Serial 2
...
...
...
  Serial2.println("Galileo");  // transmitting by Serial2 using pin 3
```

After Serial2 sends back the word "Galileo", Serial1 must receive it. The code has the same logic as was used for Serial2, using the available() and read() methods.

```
  // Serial1 will wait for something
  while (Serial1.available()) {
    // get the new byte:
    char inChar = (char)Serial1.read();  // receiving by Serial1 using pin 0

    // add it to the inputString:
    inputString += inChar;

  }
```

■ **Note** To write in the serial port, the println() method is used because it deals with strings more easily. However, if you're transmitting raw bytes or integers, you could use the write() method instead.

As soon as you run the sketch, open the IDE serial console by choosing Tools ➤ Serial Monitor or pressing CTRL+SHIFT+M. You should see something similar to the following messages in the debug serial console:

```
Transmitting from Serial 1 to Serial 2
Message received from Serial 1:Intel

Transmitting from Serial 2 to Serial 1
Message received from Serial 2:Galileo

Transmitting from Serial 1 to Serial 2
Message received from Serial 1:Intel
```

This happens every two seconds due to the delay(2000) command that's used at the end of the loop() function.

Improving the I/O Speed

This section is applicable to Intel Galileo and Intel Galileo Gen 2.

In Chapter 3, you learned how to use basic functions like `digitalWrite()`, `digitalRead()`, and `pinMode()` to manage the I/O header.

If you tried to use the I/O ports when creating a project that requires high-speed performance, you would probably face some problems due to a latency of the ports limited to 230Hz with Intel Galileo and or a maximum of 470KHz with Intel Galileo Gen 2 (except pin 7, which achieved 1.8KHz). This happens when the regular Arduino functions mentioned in the previous paragraph are used.

If you aren't familiar with this limitation, take a look at Listing 4-5.

Listing 4-5. running_at_230hz_or_max_470KHz.ino

```
/*
 * This program tests the digital port speed
 * using regular Arduino functions
 *
 * Intel Galileo: the max speed is 230Hz
 * Intel Galileo Gen 2: the speed is 470KHz to all pins except
 *                      pin 7, which achieves 1.8KHz
 *
*/
int pin = 2;  // this is pin header 2
void setup() {
  // put your setup code here, to run once:

  pinMode(pin, OUTPUT);
}

void loop() {
    // put your main code here, to run repeatedly:
    int state = LOW;
    while(1){
        digitalWrite(pin, state);
        state =!state;   // if as HIGH it will be LOW and vice-versa
    }
}
```

The `setup()` function sets pin 2 as `OUTPUT`. In the `loop()` function, the variable `"state"` is used in an infinite loop. It switches between LOW (0) and HIGH (1) and the pin state changes according to this variable through `digitalWrite()`.

If you probe an oscilloscope on this pin, the oscilloscope will state a frequency around 230Hz. Try to change the variable `pin` to other pins and see if the same frequency is achieved.

The process of inverting the pins' state in an infinite loop to generate a square wave with a certain frequency will be used in the rest of this chapter. Thus, every time the word "frequency" is used, in all situations this simple algorithm must be considered.

The following factors were responsible for this limitation in the first generation of Intel Galileo:

- When the first Intel Galileo was designed with the GPIO expander CY8C9540A, manufactured by Cypress, some limitations were introduced. All header ports with the exception of pins 2 and 3 were connected to the GPIO expander, which made the management of these pins flow only through I2C commands at 100KHz. This limited the port's speed to 230Hz. Also, the resolution of 8 bits to set the PWM duty cycle on this GPIO expander significantly impacted the performance of the servo motors, as explained in the section entitled "What's New in the Servo API" in this chapter.
- As explained in the beginning of this chapter, Intel Galileo runs Linux OS. The Arduino API is present in the userspace context, where the tasks run in a lower priority compared to the kernel. This makes it more difficult to have precision not only in terms of GPIOs, but also in terms of timer and interruptions controls. Also, Intel Galileo does not create bridges between the processor and the microcontroller. It allows you to run more than one application (or sketch) in Arduino API instead of only one instance. Quark can provide an I/O port speed superior to Arduino Uno if the new API that is discussed later is used.

To improve the I/O performance, some macros were created that allow some Intel Galileo pins to achieve 2.94MHz. They are discussed in the next section.

The New APIs for I/O

Before you learn about the new API, you need to consider a simple concept about how the I/O ports are connected. It's important to understand because there are some limitations.

On Intel Galileo, there are paths that connected the pins 2 and 3 directly to Quark SoC, not with the Cypress GPIO expander, and with these connections the pins can get close to 2.94MHz. With the other pins, there is a software optimization that makes them achieve 470KHz. 470KHz is a good improvement compared to the weak 230Hz, but it's still not enough to make some sensors like the temperature sensor DHT11 work, since this sensor requires the pins to work in higher frequencies.

When Intel Galileo Gen 2 was designed, all pins were connected directly to Quark SoC, except for pins 7 and 8, which are shared with a new GPIO expander PCAL9555 that runs around 357KHz (the kernel stated 400KHz but it is not right) and it is used to reset the mini PCIe card.

In other words, while Intel Galileo has all pins set to "slow" with exception of pins 2 and 3, Intel Galileo Gen 2 has all pins set to "fast" with the exception of pins 7 and 8.

The original Arduino functions called `digitalWrite()` and `digitalRead()` offer 230Hz only and they use `sysfs` to manage the pins. They manage the pins through a Linux character driver that accesses the file system and consequently is very slow.

The idea for improving performance is to create an API that allow you to read and write the pins states and directly access Quark's port-mapped and memory-mapped I/O. Accessing these pins directly from these registers significantly reduces latency.

Access to these registers never conflicts with access done by kernel, thus even the I/O ports being accessed directly through Quark is safe, even in a userspace context.

Memory-Mapped and Port-Mapped I/O

As the developer/maker, you do not need to worry about all the details of the Quark processor in order to use the new API created to improve the I/O performance.

However, some basic concepts regarding the I/O interface and how the pin headers are connected to Quark SoC are important to understand in order to make API calls and obtain the best performance when your software requires more than one pin in high frequencies.

For example, when you're using memory-mapped I/O, one single function call can handle more than one pin; otherwise, if you simply try to make two functions calls and declare the pins as isolated, the performance will be divided into two.

Memory-Mapped I/O

When the I/O interface is memory-mapped, the I/O states, configurations, and reading and writing operations are done in the memory context. Thus, the pin headers connected to the memory-mapped I/O offer the best performance because it is simple memory access. In the case of Quark SoC X1000, such pins are part of the south-cluster and the Linux kernel is configured to support UIO (see `http://www.hep.by/gnu/kernel/uio-howto/` for more info).

Port-Mapped I/O

When the I/O interface is port-mapped, the processor needs to execute some instructions to manage the I/O interface instead of a simple memory access. These I/O offer a slower interface compared to memory-mapped I/O. In the case of Quark SoC X1000, such pins are part of the north-cluster.

If you want to understand how Intel Arduino IDE maps the I/O, you must read sections 19.5.2 and 21.6.5 of the Intel Quark SoC X1000 datasheet at `https://communities.intel.com/docs/DOC-21828` or access a copy of this datasheet from the `code/datasheet` folder.

The I/O Distribution

Intel Galileo and Intel Galileo Gen 2 have the I/O distributed according to Table 4-4.

Table 4-4. *I/O distrubutions*

Pins	Intel Galileo Gen 2	Intel Galileo
0	South-cluster	Expander
1	South-cluster	Expander
2	South-cluster	South-cluster
3	South-cluster	South-cluster
4	North-cluster	Expander
5	North-Cluster	Expander
6	North-Cluster	Expander
7	Expander	Expander
8	Expander	Expander
9	South-cluster	Expander
10	South-cluster	Expander
11	North-cluster	Expander
12	South-cluster	Expander
13	North-cluster	Expander

The rows with "Expander" means that the pin is connected to the GPIO Expander instead. This provide a direct path to Quark SoC; it's not possible to improve the frequency of these pins.

Note that Intel Galileo is the worst case with only pins 2 and 3 available. Intel Galileo Gen 2 uses all pins except 7 and 8, as explained.

At the moment, this table might not seem to be very useful, but it will be later in this chapter.

OUTPUT_FAST and INPUT_FAST

Intel created two modes to be used in conjunction with the `pinMode()` function as the first attempt to improve the read and writing performance of the pins by abolishing the access using the regular `sysfs`. The next sections discusses these two modes and some metrics.

OUTPUT_FAST - 470KHz

It is possible to improve the performance of the `digitalWrite()` and `digitalRead()` functions with a small trick when the pins are configured with `pinMode()`.

In Chapter 3, you learned that `pinMode()` is used to configure the pin direction using the `OUTPUT` and `INPUT` defines.

However, as you can see in Listing 4-5, the frequency reached was 230KHz, but replacing `OUTPUT` with `OUTPUT_FAST` in the `pinMode()` function changes the frequency achieved to 470KHz.

Make this change in Listing 4-6 or simply load `running_at_470hz.ino` using the IDE.

Listing 4-6. running_at_470KHz.ino

```
int pin = 2;  // this is pin header 2
void setup() {
  // put your setup code here, to run once:

  pinMode(pin, OUTPUT_FAST);
}

void loop() {
    // put your main code here, to run repeatedly:
    int state = LOW;
    while(1){
        digitalWrite(pin, state);
        state =!state;   // if as HIGH it will be LOW and vice-versa
    }
}
```

Reviewing the Code

`OUTPUT_FAST`, when used with `digitalWrite()`, works only with pins 2 and 3 on Intel Galileo and all pins on Intel Galileo Gen 2.Recall that pins 7 and 8 reach a maximum of 1.7KHz instead of 470KHz.

INPUT_FAST

In the same manner that `OUTPUT_FAST` replaces the `OUTPUT` when the pin is configured with `pinMode()`, `INPUT_FAST` replaces `INPUT`.

Listing 4-7 compares the performance of `digitalRead()` using `INPUT` and `INPUT_FAST`.

Listing 4-7. INPUT_FAST_example.ino

```
/*
   This program is only a demonstration of INPUT_FAST
*/
#define MAX_COUNTER 200000

void setup() {

  Serial.begin(115200);

  pinMode(2, INPUT_FAST);  // using the pin 2
  pinMode(3, INPUT);       // using the pin 3

  delay(3000); // only to give you time to open the serial debugger terminal

  Serial.print("Number of interactions under test:");
  Serial.println(MAX_COUNTER);

  unsigned long t0,t;
  unsigned int counter = 0;

  t0 = micros(); // number of microseconds since booted
  for (counter = 0; counter < MAX_COUNTER; counter++)
  {
       digitalRead(2);   // this is the fast reading !!!!
  }

  t=micros()-t0; // delta time
  Serial.print("digitalRead() configured with INPUT_FAST took: ");
  Serial.print(t);
  Serial.println(" microseconds");

  t0 = micros(); // resetting to new initial time
  for (counter = 0; counter < MAX_COUNTER; counter++)
  {
     digitalRead(3);   // this is the lazy reading !!!!
  }

  t=micros()-t0; // delta time
  Serial.print("digitalRead() configured with INPUT took:");
  Serial.print(t);
  Serial.println(" microseconds");

}

void loop() {
}
```

Reviewing the Code

The sketch globally defines the maximum number of cycles.

```
#define MAX_COUNTER 200000
```

In the setup() function, pinMode() sets pin 2 as INPUT_FAST and pin 3 as INPUT. This works for Intel Galileo and Intel Galileo Gen 2. A small delay of three seconds was introduced to give you a chance to open the IDE serial console.

The t0 variable is created and receives the number of microseconds passed since the board was booted.

```
t0 = micros(); // number of microseconds since booted
```

Then a for loop calls the digitalRead() macro, which reads the state of pin 2 during MAX_COUNTER for the number of interactions (200000).

```
t0 = micros(); // number microseconds since booted
for (counter = 0; counter < MAX_COUNTER; counter++)
{
     digitalRead(2);   // this is the fast reading !!!!
}
```

When the for loop is finished, the t variable evaluates how many microseconds the digitalRead() took for pin 2:

```
t=micros()-t0; // delta time
```

The same procedure is done using digitalRead() for pin 3, which uses INPUT mode.

Along the program the Serial object is only used to print some debug messages with the time performance.

If you run this code and check the IDE serial console using Tools ➤ Serial Monitor or by pressing CTRL+SHIFT+M, you will see output similar to the following.

Using Intel Galileo Gen 2:

```
Number of interactions under test:200000
digitalRead() configured with INPUT_FAST took: 233937 microseconds
digitalRead() configured with INPUT took:233716 microseconds
```

Using Intel Galileo Gen:

```
Number of interactions under test:200000
digitalRead() configured with INPUT_FAST took: 231954 microseconds
digitalRead() configured with INPUT took:437786889 microseconds
```

As you can see, Intel Galileo took 437786889 microseconds (more than seven minutes) using INPUT and only 231954 microseconds (0.23 seconds) to read 200.000 times pin 3 with INPUT_FAST. Thus, INPUT_FAST is 1,888 times faster than INPUT when you are using Intel Galileo.

On other hand, if your board is Intel Galileo Gen 2, INPUT_FAST and INPUT have the same performance because they share the same implementation. Therefore, if digitalRead() is being used to read a port in Intel Galileo Gen 2, it doesn't matter if INPUT or INPUT_FAST are used. They are both equally fast.

The Fast I/O Macros

Chapter 3 explained how to select the Intel Galileo and Intel Galileo Gen 2 boards using the IDE.

When a board is selected in the IDE, a series of files corresponding to the board are exclusively selected to be part of the sketch compilation. These files bring configurations specific to the hardware (board) selected, such as the mux scheme, the I/O mappings, the SPI settings, and how the pins are connected.

Among these files one of most important is variant.h because it determines the I/O mappings and which pins are connected directly to the Quark SoC.

When you select the Intel Galileo Gen 2 board, the IDE loads the hardware/arduino/x86/variants/galileo_fab_g/variant.h file and contains the following macros:

```
#define GPIO_FAST_IO0   GPIO_FAST_ID_QUARK_SC(0x08)
#define GPIO_FAST_IO1   GPIO_FAST_ID_QUARK_SC(0x10)
#define GPIO_FAST_IO2   GPIO_FAST_ID_QUARK_SC(0x20)
#define GPIO_FAST_IO3   GPIO_FAST_ID_QUARK_SC(0x40)
#define GPIO_FAST_IO4   GPIO_FAST_ID_QUARK_NC_RW(0x10)
#define GPIO_FAST_IO5   GPIO_FAST_ID_QUARK_NC_CW(0x01)
#define GPIO_FAST_IO6   GPIO_FAST_ID_QUARK_NC_CW(0x02)
#define GPIO_FAST_IO9   GPIO_FAST_ID_QUARK_NC_RW(0x04)
#define GPIO_FAST_IO10  GPIO_FAST_ID_QUARK_SC(0x04)
#define GPIO_FAST_IO11  GPIO_FAST_ID_QUARK_NC_RW(0x08)
#define GPIO_FAST_IO12  GPIO_FAST_ID_QUARK_SC(0x80)
#define GPIO_FAST_IO13  GPIO_FAST_ID_QUARK_NC_RW(0x20)
```

When you select the Intel Galileo Gen board, the IDE loads the hardware/arduino/x86/variants/galileo_fab_d/variant.h file and contains the following macros:

```
#define GPIO_FAST_IO2   GPIO_FAST_ID_QUARK_SC(0x40)
#define GPIO_FAST_IO3   GPIO_FAST_ID_QUARK_SC(0x80)
```

When Intel Galileo Gen 2 is selected, the "fast IO" macros exist for all pins except for pins 7 and 8 because they are not connected directly to Quark. When Intel Galileo is selected, only pins 2 and 3 use the respective macro because they are the only pins connected directly to Quark.

If you do not use the right macros, you might see compilation errors. For example, if your board is Intel Galileo and you try to create a sketch to make pin 4 faster, the compilation will fail because there is no GPIO_FAST_IO4 available. The same problem will occur with Intel Galileo Gen 2 if you try to use the GPIO_FAST_IO7 or GPIO_FAST_IO8 macros because they are not defined.

Each macro calls other macros that access the I/O memory mapped through GPIO_FAST_ID_QUARK_SC, where _SC means south-cluster or through port-mapped registers with the macro GPGIO_FAST_ID_QUARK_NC_RW macro, where _NC means north-cluster.

These GPIO_FAST_IOx macros (x refers to the pin number) are used to create the I/O descriptors using 32 bits for each pin. A description is a sequence of bits in memory that describes the ports directions, the state bitmask, and the reading and writing offset of each pin. This explains why each pin contains its own descriptor.

The hexadecimal value in the macro is the bitmask in the I/O register that's used to represent which bit in the register contains each individual the pin state, HIGH or LOW (1 and 0) respectively, and this bitmask is called mask in the IDE code.

These descriptions use other information besides the mask, such as if the type of pin is present in the north-cluster or south cluster and whether the offsets for reading and writing operation are in the register.

The composition of 32 bits can be verified in the hardware/arduino/x86/cores/arduino/fast_gpio_common.h file, as shown the code in bold:

```
// Macros to (de-)construct GPIO_FAST_* register descriptors
#define GPIO_FAST_TYPE_NONE                        0x00
#define GPIO_FAST_TYPE_QUARK_SC                    0x01
#define GPIO_FAST_TYPE_QUARK_NC                    0x02
#define GPIO_FAST_ID(type, rd_reg, wr_reg, mask) \
        (0UL | ((type) << 24) | ((rd_reg) << 16) | ((wr_reg) << 8) | (mask))
#define GPIO_FAST_ID_TYPE(id)   (((id) >> 24) & 0xFF)
#define GPIO_FAST_ID_RD_REG(id) (((id) >> 16) & 0xFF)
#define GPIO_FAST_ID_WR_REG(id) (((id) >> 8) & 0xFF)
#define GPIO_FAST_ID_MASK(id)   ((id) & 0xFF)
```

Note the bitmask that defines each pin mask in the descriptor is the last 8 less significant bits (LSB). This is a detail you need to keep in mind because once a pin is configured you usually just want to use the mask to locate the pin state in the register.

fastGpioDigitalWrite(GPIO_FAST_IOx, unsigned int value) - 652KHz to 684KHz

This macro allows you to write to an I/O port and achieve a frequency between 652KHz to 684KHz.

The first parameter, called GPGIO_FAST_IOx, is a fast I/O macro used to identify the pin of interest.

The second parameter, called value, is the value of the pin that might be LOW or HIGH.

If Intel Galileo is used, the pinMode() function must receive the OUTPUT_FAST configuration and only pins 2 and 3 work.

If Intel Galileo Gen 2 is used, the pinMode() function must receive the OUTPUT or OUTPUT_FAST configuration because, unlike Intel Galileo, they provide the same effect. And all pins work except for pins 7 and 8, which achieve 684KHz. Pin 13 reaches 657KHz because it's also connected to a built-in LED and there is a small loss when triggering this LED.

Listing 4-8 shows how to reach 687KHz using pin 2.

Listing 4-8. Running_at_684khz.ino

```
/*
   This program makes the I/O speed to achieve 684KHz.
   If you are using Intel Galileo: Only pins 2 and 3 work
   If you are using Intel Galileo Gen 2: ALL pins work

   Note: if you are using Intel Galileo Gen 2 and change
   this software to support pin 13, the frequency will be
   close to 657KHz and not 687KHz.
*/
void setup() {
  // put your setup code here, to run once:
  unsigned int pin = 2;  // this is pin header 2
  pinMode(pin, OUTPUT_FAST);
}

void loop() {
    // put your main code here, to run repeatedly:
    int state = LOW;
    while(1){
        fastGpioDigitalWrite(GPIO_FAST_IO2, state);
        state =!state;   // if as HIGH it will be LOW and vice versa
    }

}
```

Reviewing 684khz.ino

In the setup() function, the pin variable is created to represent the pin 2 and pinMode() sets this pin to OUTPUT_FAST. This works for Intel Galileo and Intel Galileo Gen 2.

Then in the loop() function, the state variable is initiated with state LOW which creates an infinity loop. In this infinity loop, the fastGpioDigitalWrite() macro identifies the GPGIO_FAST_IO2 macro and the state variable. Finally, the state variable has its stated inverted on each loop interaction, switching between LOW (0) and HIGH (1).

Try to change the variable pin and the corresponding fast I/O macro used in fastGpioDigitalWrite() to explore more other pins.

Frequency Reduction with fastGpioDigitalWrite()

The problem with `fastGpioDigitalWrite()` is the reduction of performance when `fastGpioDigitalWrite()` is called multiple times in a single loop cycle.

It does not matter if `fastGpioDigitalWrite()` is handling different pins; the maximum frequency is divided by the number of `fastGpioDigitalWrite()` methods in one loop interaction.

The code in Listing 4-9 is one example. Considering that `fastGpioDigitalWrite()` is called twice, the end frequency will be 340KHz.

Listing 4-9. fastGpioDigitalWrite_340khz_two_pins.ino

```
/*
   This program makes the I/O speed achieve 340KHz.
   If you are using Intel Galileo: Only pins 2 and 3 work
   If you are using Intel Galileo Gen 2: ALL pins work

*/

void setup() {
  // put your setup code here, to run once:

  pinMode(2, OUTPUT_FAST);
  pinMode(3, OUTPUT_FAST);
}

void loop() {
    // put your main code here, to run repeatedly:
    int state = LOW;
    while(1){
        fastGpioDigitalWrite(GPIO_FAST_IO2, state);
        fastGpioDigitalWrite(GPIO_FAST_IO3, state);

        state =!state;   // if as HIGH it will be LOW and vice versa
    }

}
```

To resolve this problem, a new macro was created called `fastGpioDigitalRegWriteUnsafe()`. It's explained later in this chapter.

int fastGpioDigitalRead(GPIO_FAST_IOx)

This macro allows you to read to an I/O port and achieve much faster performance than the regular `digitalRead()`.

The `GPGIO_FAST_IOx` parameter is a fast I/O macro used to identify the pin of interest.

As a result, the macro might return `0` for `LOW` or any other value representing `HIGH`, because the value returned is the current value in the register mask according to the register offset.

If Intel Galileo is used, the pinMode() function must receive the INPUT_FAST configuration and only pins 2 and 3 work.

If Intel Galileo Gen 2 is used, the pinMode() function must receive the INPUT or INPUT_FAST confuguration because, unlike Intel Galileo, they provide the same effect and all pins work except pins 7 and 8.

Listing 4-10 shows a sketch that compares the performance between digitalRead() and fastGpioDigitalRead() and detects when a state changes in a pin. If you are interested only in checking the performance, it is not necessary to assemble any circuit. In this case, you can just run the sketch. Otherwise, if you want to detect when the pin state is changed, it is necessary to assemble the same circuit and materials mentioned in the section entitled "The Button Example" in Chapter 3.

Listing 4-10. fastGpioDigitalRead_example.ino

```
/*
   This program is only a demonstration of fastGpioDigitalRead()
*/
int pin = 2;

#define MAX_COUNTER 200000

void setup() {

  Serial.begin(115200);

  pinMode(pin, INPUT_FAST);  // using pin 2

  delay(3000); // only to give you time to open the serial debugger terminal

  Serial.print("Number of interactions under test:");
  Serial.println(MAX_COUNTER);

  unsigned long t0,t;
  unsigned int counter = 0;

  t0 = micros(); // number of microseconds since booted
  for (counter = 0; counter < MAX_COUNTER; counter++)
  {
     if (fastGpioDigitalRead(GPIO_FAST_IO2)) // using the fast I/O macro related
                                             // to pin 2
     {
        // the pin is HIGH
         Serial.println("HIGH detected by fastGpioDigitalRead()");
     }

  }
```

```
  t=micros()-t0; // delta time
  Serial.print("fastGpioDigitalRead() took: ");
  Serial.print(t);
  Serial.println(" microseconds");

  t0 = micros(); // reseting to new initial time
  for (counter = 0; counter < MAX_COUNTER; counter++)
  {
     if (digitalRead(pin)) // using the fast I/O macro related
                           // to pin 2
     {
         // the pin is HIGH
          Serial.println("HIGH detected by digitalRead()");
     }

  }

  t=micros()-t0; // delta time
  Serial.print("digitalRead() took: ");
  Serial.print(t);
  Serial.println(" microseconds");

}

void loop() {

}
```

Execute the sketch and open the IDE serial console with Tools ➤ Serial Monitor or by pressing CTRL+SHIFT+M. In the first test, do not press the button even if you assembled the recommended circuit. Simply run it and you will be able to see this output in the serial console:

```
Number of interactions under test:200000
fastGpioDigitalRead() took: 123207 microseconds
digitalRead() took: 239766 microseconds
```

The output indicates that the `fastGpioDigitalRead()` executed 200000 readings in 123302 microseconds, while `digitalRead()` took 239766 microseconds for the same amount of cycles. In other words:

```
239766/123207 = 1.94
```

The `fastGpioDigitalRead()` method was almost twice as fast as the regular `digitalRead()` method using the `INPUT_FAST` mode.

If you assembled the circuit recommended for this test, run the sketch again and press the button to check in the serial console output. Check if the HIGH state is being detected by fastGpioDigitalRead() and digitalRead() in INPUT_FAST mode.

Reviewing fastGpioDigitalRead_example.ino

The sketch globally defines a variable to specify pin 2 and to define to the maximum number of cycles.

```
int pin = 2;
#define MAX_COUNTER 200000
```

In the setup() function, pinMode() sets pin 2 to INPUT_FAST, which works for Intel Galileo and Intel Galileo Gen 2. A delay of three seconds was introduced only to give you a chance to open the IDE serial console.

The variable t0 is created and receives the number of microseconds passed since the board was booted.

```
t0 = micros(); // number microseconds since booted
```

Then a for loop is created by calling the macro fastGpioDigitalRead(), which reads the state of pin 2 and passes the fast I/O macro GPIO_FAST_IO2 as the parameter during **MAX_COUNTER** number of interactions (200000).

```
  t0 = micros(); // number of microseconds since booted
  for (counter = 0; counter < MAX_COUNTER; counter++)
  {
     if (fastGpioDigitalRead(GPIO_FAST_IO2)) // using the fast I/O macro
related
                                             // to pin 2
     {
         // the pin is HIGH
          Serial.println("HIGH detected by fastGpioDigitalRead()");
     }
  }
```

If you assembled the recommended circuit, you still have the option to press the button and check the detection (HIGH state) because the Serial object prints a message reporting the HIGH state.

When the for loop is finished, the t variable evaluates how many microseconds the fastGpioDigitalRead() took and again, using the Serial object, prints the information to the serial console.

```
t=micros()-t0; // delta time
```

The same procedure is done using `digitalRead()` instead of `fastGpioDigitalRead()`, allowing you to compare the performance between both functions when `INPUT_FAST` mode is used.

`fastGpioDigitalRegSnapshot(GPIO_FAST_IOx)`

This macro latches the current register values specified by the macro in the first parameter, called `GPGIO_FAST_IOx`. It's a fast I/O macro used to identify the pin of interest.

An example is shown in Listing 4-11.

Listing 4-11. latch_example.ino

```
/*
   This program is only a demonstration of fastGpioDigitalRegSnapshot()
   and the importance of the bitmask fields.
*/

void setup() {

  unsigned int latchValue;

  Serial.begin(115200);

  delay(3000); // only to give you time to open the serial debugger terminal

  // latches the current value
  latchValue = fastGpioDigitalRegSnapshot(GPIO_FAST_IO3);

  // identifies the bit corresponding to pin 3 in the bitmask
  unsigned int mask = 0x000000ff & GPIO_FAST_IO3;

  if (latchValue & mask)
  {
       // the register indicated the pin is HIGH
       Serial.println("HIGH");
  }
  else
  {
       // the register indicated the pin is LOW
       Serial.println("LOW");
  }

}

void loop() {

}
```

Reviewing latch_example.ino

The setup() method starts with a delay of three seconds to give you a chance to open the IDE serial console with Tools ➤ Serial Monitor or with CTRL+SHIFT+M.

Then fastGpioDigitalRegSnapshot() is called to retrieve the I/O latch regarding pin 3, using the descriptor constructed by the macro GPIO_FAST_IO3.

```
latchValue = fastGpioDigitalRegSnapshot(GPIO_FAST_IO3);
```

As explained in the section called "The Fast I/O" in this chapter, a variable called mask is created by retrieving only the bitmask used an AND operation with 8 fewer significant bits than the description created for pin 3.

```
unsigned int mask = 0x000000ff & GPIO_FAST_IO3;
```

Then the value retrieved by the latchValue variable is compared to the mask variable to determine whether pin 3 is doing another simple AND operation.

```
if (latchValue & mask)
```

This program always prints out LOW because it is the initial state of the pins and the logic in this case is merely illustrative. This same logic is used in the next section with the macro fastGpioDigitalRegWriteUnsafe().

The Serial object is only used to debug messages along the code.

fastGpioDigitalRegWriteUnsafe (GPIO_FAST_IOx, unsigned int value) - 2.94MHz

This macro allows you to write to an I/O port and achieving a frequency up to 2.94MHz.

The first parameter, called GPGIO_FAST_IOx, is a fast I/O macro used to identify the pin of interest.

The second parameter, called value, corresponds to the descriptors and all current pin states.

If Intel Galileo is used, the pinMode() function must receive the OUTPUT_FAST configuration and only pins 2 and 3 work.

If Intel Galileo Gen 2 is used, the pinMode() function must receive the OUTPUT or OUTPUT_FAST configurations because, unlike Intel Galileo, they provide the same effect. All the pins work. Pins 7 and 8 can achieve 2.93MHz whereas pin 13 reaches 1.16MHz, because it's also connected to a built-in LED and there is a small loss when triggering this LED.

This function has the word Unsafe on its name because you must preserve the I/O descriptor and not mess with other I/Os masks when a particular I/O is being used. If you mess with the description and pin states' bitmask, unexpected results will be observed in the pin headers.

The code shown in Listing 4-12 shows how to make pin 2 reach 2.93MHz.

Listing 4-12. running_at_2_93Mh.ino

```
/*
   This program makes the I/O speed achieve 2.93MHz.
   If you are using Intel Galileo: Only pins 2 and 3 work
   If you are using Intel Galileo Gen 2: ALL pins work
   except pins 7 and 8

   Note: if you are using Intel Galileo Gen 2 and change
   this software to support pin 13, the frequency will be
   close to 1.16MHz.
*/

unsigned int latchValue;
unsigned int bmask;

void setup() {
  // put your setup code here, to run once:

  pinMode(2, OUTPUT_FAST);

  // latches the current value
  latchValue = fastGpioDigitalRegSnapshot(GPIO_FAST_IO2);

  // extract the mask that identifies pin 2 in the
  // descriptor
  bmask = 0x000000ff & GPIO_FAST_IO2;

}

void loop() {

  while(1)
  {

     if (latchValue & bmask)
     {
        // state is HIGH
        latchValue = GPIO_FAST_IO2 & !bmask;
     }
     else
     {
        // state is LOW
        latchValue = GPIO_FAST_IO2 | bmask;
     }

     fastGpioDigitalRegWriteUnsafe (GPIO_FAST_IO2, latchValue);

  }

}
```

Reviewing running_at_2_93Mh.ino

The setup() function configures pin 2 as OUTPUT_FAST for Intel Galileo and Intel Galileo Gen 2. Then the variable latchValue reads the latest I/O states according to the GPIO_FAST_IO2 descriptor using fastGpioDigitalRegSnapshot().

```
pinMode(2, OUTPUT_FAST);

// latches the current value
latchValue = fastGpioDigitalRegSnapshot(GPIO_FAST_IO2);
```

At the end of setup(), the variable bmask extracts only the I/O bitmask for pin 2, disregarding the rest of I/O descriptor. Remember this mask occupies only the 8 LSB among the total of 32 bits that compose the I/O descriptor, as explained in the section entitled "The Fast I/O Macros" in this chapter.

```
bmask = 0x000000ff & GPIO_FAST_IO2;
```

In the loop() function, an infinity while is introduced and latchValue is tested to check if the state is HIGH.

If the latched value contains the pin bitmask on its 8 LSB, it means the state is HIGH, because the bit mask will have the bit 1 in the respective pin state; otherwise, the latchValue state is LOW. This operation can be checked with a simple AND:

```
if (latchValue & bmask)
```

If the state is HIGH, it is necessary to change it to LOW. For this, you just need to invert the bitmask bits and create an AND operation again:

```
latchValue = GPIO_FAST_IO2 & !bmask;
```

Otherwise, if the state is LOW, it is necessary to change it to **HIGH**. For this, you must add an OR using the bitmask:

```
latchValue = GPIO_FAST_IO2 | bmask;
```

Finally, fastGpioDigitalRegWriteUnsafe() is called, informing the fast I/O descriptor for pin 2 (GPIO_FAST_IO2) and the current value of latchValue to be changed.

An important point—the bitmask bmask" was retrieved and used with the AND and OR operation, which will preserve the other bits in the I/O descriptor. With this type of logic, you can handle more than one bit safely.

North-Cluster (1.12MHz) versus South-Cluster (2.93MHz)

The previous example in Listing 4-12 uses pin 2. If you take a look in Table 4-4, you will realize this pin belongs to the south-cluster.

Based on this table, you can change the code and replace it with a pin that belongs to the north-cluster, such as pin 4.

With this change, `pinMode()` must be modified and all `GPIO_FAST_IO2` calls must be replaced with `GPIO_FAST_IO4`. You could also just open the code `running_at_1_12MHz.ino`, which contains these changes.

If you run the sketch again, you will see that pin 4 can achieve only 1.12MHz because it is a port-mapped I/O. That means it's slower than the memory-mapped I/O that pin 2 uses.

Keeping the Same Frequency on All Pins

If you create a sketch that works with pins that belong to the same cluster, it's possible to keep the same frequency on all of them using `fastGpioDigitalRegWriteUnsafe()`. This provides a good alternative to `fastGpioDigitalWrite()`, which cannot do this.

Listing 4-13 is an example that runs only on Intel Galileo Gen 2 and shows how to run three pins from the south-cluster, ensuring the speed of 2.93MHz on all of them. You can run this example in the first board Intel Galileo if you remove the code related to pin 12 from this listing.

Listing 4-13. running_at_2_93Mhz_three_pins.ino

```
/*
   This program makes the I/O speed to achieve 2.93MHz.
   If you are using Intel Galileo: Only pins 2 and 3 work
   If you are using Intel Galileo Gen 2: ALL pins work
   except pins 7 and 8

*/

unsigned int latchValue;
unsigned int bmask_pin2;
unsigned int bmask_pin3;
unsigned int bmask_pin12;

void setup() {
  // put your setup code here, to run once:

  pinMode(2, OUTPUT_FAST);
  pinMode(3, OUTPUT_FAST);
  pinMode(12, OUTPUT_FAST);

  // latches the current value
  latchValue = fastGpioDigitalRegSnapshot(GPIO_FAST_IO2);
```

```
  // retrieving bitmasks from descriptors
  bmask_pin2 = 0x000000ff & GPIO_FAST_IO2;      //south-cluster
  bmask_pin3 = 0x000000ff & GPIO_FAST_IO3;      //south-cluster
  bmask_pin12 = 0x000000ff & GPIO_FAST_IO12;    //south-cluster

}

void loop() {

  while(1)
  {

     if (latchValue & bmask_pin12)
     {
        // state is HIGH
        latchValue = GPIO_FAST_IO2 & !bmask_pin2;
        latchValue |= GPIO_FAST_IO3 & !bmask_pin3;
        latchValue |= GPIO_FAST_IO12 & !bmask_pin12;

     }
     else
     {
        // state is LOW
        latchValue = GPIO_FAST_IO2 | bmask_pin2;
        latchValue |= GPIO_FAST_IO3 | bmask_pin3;
        latchValue |= GPIO_FAST_IO12 | bmask_pin12;

     }

     // considering all pins in this example belong to the south-cluster
     // they share the same register in memory-mapped I/O. Only one
     // fastGpioDigitalRegWriteUnsafe() must be called ensuring 2.93MHz
     // to ALL pins
     fastGpioDigitalRegWriteUnsafe (GPIO_FAST_IO2, latchValue);

  }

}
```

Reviewing running_at_2_93Mhz_three_pins.ino

This code has the same logic as Listing 4-12; the difference is that now there are three pins (2, 3, and 12) that belong to the same cluster.

In the `setup()` function, all pins are set to `OUTPUT_FAST` as expected. The bitmask for each descriptor is captured and the `latchValue` latches the current `GPGIO_FAST_IO2` state. The `latchValue` will contain the state of all pins in the south-cluster because all the pins share the same register in the memory (memory-mapped I/O) and it does not matter if the fast macro I/O is used.

In the `loop()` function, the logic is the same. One of the pin states is checked and the `latchValue` changes the state of each pin, inverting the original state.

But the most important point in this code is the fact that only one `fastGpioDigitalRegWriteUnsafe()` is called. The explanation is quite simple—all pins belong to the same cluster and share the same memory registers, so only one call ensures 2.93MHz to all the pins used in the code (pins 2, 3, and 12).

If you change to pins that belong to the north-cluster, the effect will be the same as explained in the previous section. In other words, the maximum frequency will be 1.12MHz.

When Pins from North-Cluster and South-Cluster Are Used in Same Sketch

To achieve best performance, it is recommended that you keep using the pins of same cluster, but in practice this is not always possible. When pins from the north-cluster and south-cluster must be used in a same sketch, you must call `fastGpioDigitalRegWriteUnsafe()` twice in order to configure the I/O pins for both clusters.

When this situation occurs, the maximum frequency for each pin is not respected and you'll get a maximum of 980KHz in this case.

Listing 4-14 shows an example that runs using Intel Galileo Gen 2. It uses two pins from the south-cluster (pins 2 and 3) and one from the north-cluster (pin 4).

Listing 4-14. mixing_north_and_south_clusters.ino

```
/*
   This program is an example of how to mix pins from the
   north-cluster and south-cluster
   If you are using Intel Galileo: Only pins 2 and 3 work
   If you are using Intel Galileo Gen 2: ALL pins work
   except pins 7 and 8

*/

unsigned int latchValue;
unsigned int bmask_pin2;
unsigned int bmask_pin3;
unsigned int bmask_pin4;

void setup() {
  // put your setup code here, to run once:

  pinMode(2, OUTPUT_FAST);
  pinMode(3, OUTPUT_FAST);
  pinMode(4, OUTPUT_FAST);

  // latches the current value
  latchValue = fastGpioDigitalRegSnapshot(GPIO_FAST_IO2);
```

```
  // retrieving bitmasks from descriptors
  bmask_pin2 = 0x000000ff & GPIO_FAST_IO2;     //south-cluster
  bmask_pin3 = 0x000000ff & GPIO_FAST_IO3;     //south-cluster
  bmask_pin4 = 0x000000ff & GPIO_FAST_IO4;     //north-cluster !!!!

}

void loop() {

  while(1)
  {

     if (latchValue & bmask_pin4)
     {
        // state is HIGH
        latchValue = GPIO_FAST_IO2 & !bmask_pin2;
        latchValue |= GPIO_FAST_IO3 & !bmask_pin3;
        latchValue |= GPIO_FAST_IO4 & !bmask_pin4;

     }
     else
     {
        // state is LOW
        latchValue = GPIO_FAST_IO2 | bmask_pin2;
        latchValue |= GPIO_FAST_IO3 | bmask_pin3;
        latchValue |= GPIO_FAST_IO4 | bmask_pin4;

     }

     // pins from cluster different used, so it is necessary
     // to make a couple call using pins from south-cluster and north-cluster
     fastGpioDigitalRegWriteUnsafe (GPIO_FAST_IO2, latchValue);
     fastGpioDigitalRegWriteUnsafe (GPIO_FAST_IO4, latchValue);

  }

}
```

Reviewing mixing_north_and_south_clusters.ino

The code has the same logic as commented in the Listing 4-12 except that there are two pins from the south-cluster (pins 2 and 3) and one from the north-cluster.

As explained, this scenario requires two `fastGpioDigitalRegWriteUnsafe()` calls, one for each cluster.

If you probe each pin, you can verify that the maximum frequency reached is 980KHz.

When Port Speed Is Not Enough - pinMode() Limitations

Until now, you could read methods that help to improve the speed in the digital I/O headers. Sometimes speed is not enough. There is a limitation when `pinMode()` is used to change the pin direction that introduces a three-millisecond delay due to the need to send the I2C command to set the mux in the boards.

Thus, if you are working with devices that require communication in one single pin (when a single pin is used to send commands), you need to consider this delay every time the pin direction is changed.

There are methods for working around this problem, as explained in the section entitled "Practical Project - DHT Sensor Library with Fast I/O APIs" in this chapter.

The Tone API

This section is applicable to Intel Galileo Gen2 only.

The Tone API is part of the Arduino reference and it was created to generate square waves in specific frequencies (tones) with a duty cycle close to 50 percent. The original Arduino reference for the Tone API is discussed at `http://arduino.cc/en/reference/tone`.

Due to the latencies in the I/O ports on Intel Galileo, the Tone API cannot work with pins 2 and 3 when using `OUTPUT_FAST` and `INPUT_FAST` because the frequencies required are not achieved with such modes in other pins. There is no PWM involved.

With Intel Galileo Gen 2, you cannot provide direct connection between Quark SoC I/O with I/O headers.

So, if you are connecting a speaker or a buzzer and want to use this API, make sure that pins 7 and 8 are not being used for this purpose.

What's New in the Tone API?

In the regular Arduino implementation, the Tone API is a blocking call that prevents the use of more than one tone at the same time. In the implementation created for Intel Galileo Gen 2, there are blocking and non-blocking calls, which provides more flexibility to the effect desired when a sketch is created. The next pages discuss these new features.

void tone(unsigned int pin, unsigned int frequency, unsigned long duration = 0)

The first parameter, called `pin`, specifies the pin to be used. Note that if you are using this pin as PWM, the PWM will be disabled automatically.

The second parameter is the frequency in hertz desired.

The third parameter is the duration of the tone in milliseconds to be generated and it is optional. If it's not specified in the function call, the default value is `0`. In this case, the tone will be generated continuously until the `noTone()` function is called. Note when a duration is not specified, the continuous generation of this tone makes the `tone()` function call non-blocking.

With `tone()` called as a non-blocking call, multiple tones can be generated in multiples pins. However, the accuracy goes down as the number of tones increases because each tone called as non-blocking is implemented with POSIX threads called mutually exclusively.

If the `duration` parameter is specified, the function will generate the tone until the duration is attended; the function call in this case is blocking. In these cases, multiples tones cannot be generated; the accuracy is better when compared to non-blocking calls.

void noTone(uint8_t pin)

This function will stop the generation of the tone in the pin that's specified as the function parameter.

A Sample Running Non-Blocking Tone Calls

To run this example, the components listed in Table 4-5 are necessary.

Table 4-5. *Materials List for the Non-Blocking Tone Example*

Number	Description
2	Resistor 100 Ohms
2	Speaker 8 Ohms

The schematics are represented in Figure 4-6.

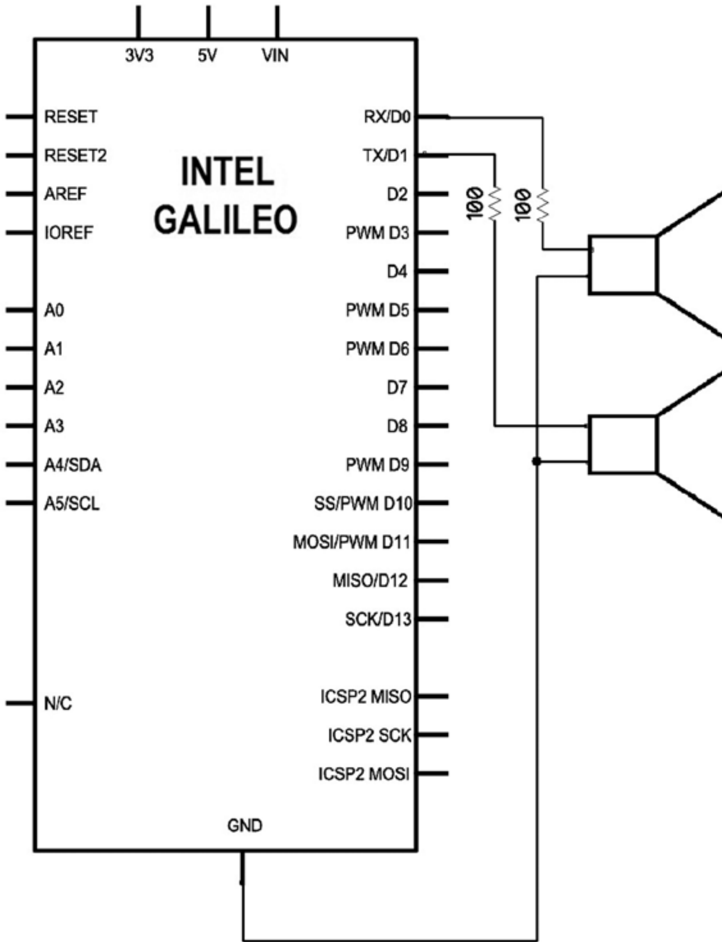

Figure 4-6. *Two speakers connected to Intel Galileo*

Listing 4-15 shows a sketch to be tested.

Listing 4-15. Tone.ino

```
// some tones in hertz
#define NOTE_C4  262
#define NOTE_G3  196
#define NOTE_A3  220
#define NOTE_B3  247

// melody on pin 0:
int melody_pin0[] = {
   NOTE_C4, NOTE_G3,NOTE_G3, NOTE_A3, NOTE_G3,0, NOTE_B3, NOTE_C4};

// melody on pin 1:
int melody_pin1[] = {
   NOTE_G3, NOTE_C4,NOTE_G3, NOTE_G3, NOTE_A3,0, NOTE_C4, NOTE_B3};

void setup() {
  // iterate over the notes of the melody:
  for (int thisNote = 0; thisNote < sizeof(melody_pin0)/sizeof(int);
thisNote++) {

    //the duration is not specified to make a non-blocking call.
    tone(0, melody_pin0[thisNote]);
    tone(1, melody_pin1[thisNote]);

    // small delay
    delay(500);

    // stop the tone playing:
    noTone(0);
    noTone(1);
  }
}

void loop() {
  // no need to repeat the melody.
}
```

When you run the code, you will hear some tones coming out from the speakers connected to pins 0 and 1 in intervals of 500 milliseconds.

Reviewing Tone.ino

The code starts globally defining the tones C4, G3, A3, and B3 in hertz. The `melody_pin0` and `melody_pin1` arrays, which contain the sequence of tones to be played, are created for pins 0 and 1, respectively.

Then in the `setup()` function, a `for` loop scans the tone arrays and plays the arrays on pin 0 and 1 using the `tone()` function. This is done without defining a duration, which means the call is non-blocking.

```
tone(0, melody_pin0[thisNote]);
tone(1, melody_pin1[thisNote]);
```

A small delay of 500ms is added with the `delay()` function. It's used only to allow the perception of tones in the speakers. Finally the tones are stopped by the `noTone()` function call.

```
// stop the tone playing:
noTone(0);
noTone(1);
```

If you are interested in testing the `tone()` function in a blocking scenario, check out http://arduino.cc/en/Tutorial/tone and run the simple sketch.

The pulseIn API

This section is applicable to Intel Galileo and Intel Galileo Gen2.

The `pulseIn()` API is used to measure a pulse length in some specified pin.

unsigned long pulseIn(uint8_t pin, uint8_t state, unsigned long timeout = 1000000)

The first parameter called `pin` specifies the pin to be used. Note if you are using this pin as PWM, the PWM will be disabled automatically.

The second parameter is the `state`. If `HIGH` is specified, `pulseIn()` will start timing when the pin changes from `LOW` to `HIGH`, and stop timing when it moves to the `LOW` state again. If the state parameter is `LOW`, the inverse order will be used to measure the pulse from `HIGH` to `LOW`, and `LOW` to `HIGH` again.

The third parameter called `timeout` is in microseconds and is optional. It imposes the maximum timeout that `pulseIn()` needs to wait for the pin state transition. If it is not specified, the default value is 1.000.000 microseconds (1 second).

If `pulseIn()` measures the pulse length, the length in microseconds is returned. Otherwise, if some timeout occurred, the value returned is 0.

For example, if you use the following code in your sketch:

```
duration = pulseIn(3, HIGH);
```

The `pulseIn()` method will wait for pin 3 to reach the `HIGH` state and then start timing until the state becomes `LOW`. The `pulseIn()` function returns the pulse length in microseconds (duration).

The pulseIn API created for Intel Galileo boards follows the same proposal as the Arduino reference; see `http://arduino.cc/en/Reference/pulseIn`. However, it is crucial to understand its limitations if your sketch intends to use this API.

There is a limitation of the Tone API related to pins that work on Intel Galileo. This API works only if pins 2 and 3 are used; Intel Galileo Gen 2 works in all pins except 7 and 8.

What's New with pulseIn()

The implementation created for Intel Galileo boards has as timeout parameter that's an unsigned long that occupies 32 bits. It has certain advantages compared to a board with 16 bits because the timeout can be set to measure pulses from 3 microseconds to 2^32 microseconds (71.58 minutes).

The accuracy is around 2 microseconds.

A Sample Running pulseIn()

The idea of this sample is to have a button connection to one of the digital I/O headers and, using this button, you can generate a pulse. The `pulseIn()` function will then read how long the pulse was present.

The pin chosen is pin 2 because it works with Intel Galileo and Intel Galileo Gen 2.

The material and schematics used in this example are the same as explained in the section entitled "The Button Example" in Chapter 3. This circuit allows you to inject 5V into pin 2 when the button is pressed; `pulseIn()` will measure how long the pulse is generated by the button and will remain in the HIGH state during this time.

Listing 4-16 shows an example of how to use the pulseIn API.

Listing 4-16. pulseIn.ino

```
int pin = 2;
unsigned long duration;

void setup()
{
  Serial.begin(115200);
  pinMode(pin, INPUT);
}

void loop()
{
  // measuring the pulse lengh during 20 seconds
  duration = pulseIn(pin, HIGH,20000000);
  Serial.println(duration);
}
```

Run the sketch and open the IDE serial console by choosing Tools ➤ Serial Monitor or pressing CTRL+SHIFT+M. Then, try to press the button during intervals of 20 seconds. The IDE serial console will display 0 if the pulse could not be read; otherwise, the reading is displayed in microseconds.

Reviewing pulseIn.ino

The code starts globally defining the variable pin, which is used to indicate that pin 2 is used to receive the pulse event. Another variable duration is used to receive the metric returned by the pulseIn() function.

In the setup() function, pinMode() is used only to set the pin to the INPUT mode. The Serial object is initiated for debugging purposes.

In the loop() function, pulseIn() is set to read pin 2 and measure the transition from LOW to HIGH using 20 seconds as the timeout.

When read with success, the duration variable receives the measurement during a HIGH state; otherwise, it receives **0**. The Serial object prints the results in microseconds.

Hacks

Sometimes there are requirements that the regular Arduino reference API cannot address. For example, when you need a different PWM frequency and duty cycle than provided by the regular analogWrite(), or need a servo motor that works with pules out of regular Arduino reference range, or want to create a single sketch that works in both Intel Galileo and Intel Galileo Gen 2.

The next sections provide a few hacks that you might find useful.

Hacking the Servo Library

This hack is applicable to Intel Galileo and Intel Galileo Gen 2.

Suppose you acquired a new servo motor and the specification of this servo is a pulse of 500ms to 0 degrees and 2600ms to 180 degrees. This servo will not work correctly if you use the Arduino IDE with any changes in the library. This is not specific to Intel Galileo; the reference implementation of Arduino IDE really uses 544 (MIN_PULSE_WIDTH) and 2400 (MAX_PULSE_WIDTH) microseconds.

To resolve such a problem, you must change these defines. In the case of the servo mentioned, the hack would be to change the ...\arduino-1.5.3\hardware\arduino\x86\libraries\Servo.h file:

```
#define MIN_PULSE_WIDTH        500     // the shortest pulse sent to a servo
#define MAX_PULSE_WIDTH       2600     // the longest pulse sent to a servo
```

Then the servo can receive the pulses that attend its specification.

Hacking the GPIO Expander for New PWM Frequencies

This hack is applicable to Intel Galileo only.

The first generation of Intel Galileo uses the expander CY8C9540A from Cypress. It is a PWM generator that can be commanded using I2C commands and the base PWM frequency can be changed by changing the clock sources and the divider in the IC.

Before reading this section, it's best to have the datasheet on hand. The Cypress CY8C9540A datasheet is in the `code/datasheets` folder of this chapter in the file named `CY8C95x0A.pdf`. You can also access it at http://www.cypress.com/?rID=3354.

In case of Intel Galileo, the CY8C9540A can be reached by I2C commands through the address 0x20.

Four simple I2C commands must be sent to generate your custom PWM, discussed in the following sections.

Changing the Frequency

It is necessary to use the divider register 0x2C.

The formula to specify the frequency is:

```
Freq = Clock Source/Divider
```

The divider is 8 bits—something from 1 to 255 (0 is not a valid divider).

Selecting the Clock Source

The clock source must be selected using the register 0x29. Check Table 11 in the datasheet, named "PWM Clock Sources," for all the available PWM clock sources.

Selecting the Duty Cycle

The duty cycle is set using the command 0x2B and must be represented for 8 bits. It follows the regular theory, in other words:

```
Duty cyle = Pulse Width/Period
```

Setting the Period Register

Sets the period of thePWM counter; only two values are accepted—0 for falling pulse edge and 255 (0xff) for rising pulse edge.

Changing the PWM

Suppose you want to generate a PWM with 123Hz, with a minimum duty cycle possible with a rising edge period.

Looking at the datasheet, the clock source closest to this frequency is 367.6Hz (see Table 11 of the datasheet). So the divisor that must be used to get close to 123Hz is 3 because 367.6/3 = 122.533Hz.

Thus the clock source register 0x29 is 4 and the divisor 0x2C is 3.

The duty cycle register 0x2b can be set 1 to 255. Dividing the maximum period of 122.533Hz by 255 gives you the minimum granularity of a duty cycle. In this case you have:

```
122.533 hz = 8.16ms
Minimum granularity is: 8.16 / 255 = 32 microseconds
```

So the minimum duty cycle is 32 microseconds and must be represented by `1` in the 0x2b register. Figure 4-7 shows this logic graphically.

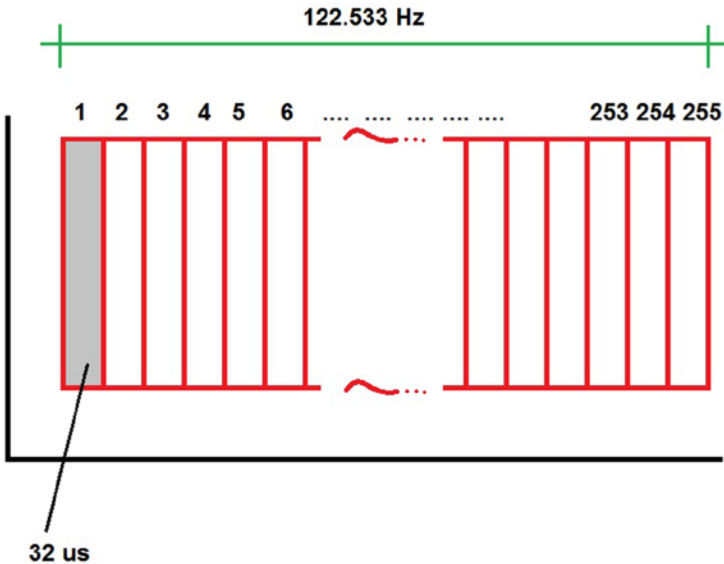

***Figure 4-7.** Minimum granularity for duty cycle in 255 possibilites*

Thus, if a duty cycle of 64 microseconds is desired, the 0x2b register will receive the value of `2`; if the duty cycle desired is 94 microseconds, the value received is `3`, and so on.

Knowing all registers that must be used and the respective values that each one must receive, it is time to create the sketch. The Arduino reference has a library called Wire API that is created to send I2C commands.

Basically to send an I2C command using Wire API, you use three methods—`beginTransmission()`, `write()`, and `endTransmisstion()`.

The basic sequence to use these commands is as follows:

```
Wire.beginTransmission(DEVICE_ADRESS_HERE);
Wire.write(BYTE1);
Wire.write(BYTE2);
Wire.write(BYTE3);
...
...

Wire.endTransmission();
```

It's simple to create the sketch in order to send such commands using I2C and pin 9. This process is demonstrated in Listing 4-17.

Listing 4-17. custom_pwm.ino

```
#include "Wire.h"

int PIN = 9;

void setup()
{

   // Set divider to get 122.533Hz freq.
   Wire.beginTransmission(0x20);
   Wire.write(0x2C);
   Wire.write(0x03);
   Wire.endTransmission();

   // Select PWM source clock
   Wire.beginTransmission(0x20);
   Wire.write(0x29);
   Wire.write(0x04);
   Wire.endTransmission();

   // Set period register
   Wire.beginTransmission(0x20);
   Wire.write(0x2a);
   Wire.write(0xff);
   Wire.endTransmission();

   // Duty cycle of 32us @ 122.533hz
   Wire.beginTransmission(0x20);
   Wire.write(0x2b);
   Wire.write(0x01);  // 1 is the minimum granularity
   Wire.endTransmission();

}

void loop()
{
}
```

Run the code using the IDE. If you have an oscilloscope, you will be able to see the PWM on pin 9.

Single Code for Intel Galileo and Intel Galileo Gen 2

This chapter covers some particularities that are supported by Intel Galileo Gen 2 but are not supported or only partially supported by Intel Galileo.

How do you guarantee compatibility when it is necessary to create a sketch that must run in these different boards? Or how do you port libraries and ensure they have code that deals with these particularities?

One way is to use preprocessing by adding a directive that differentiates one board from other. This directive is present only in Intel Galileo core libraries and is defined as `PLATFORM_ID` representing the value 0x06.

Thus, during compilation time it is possible to implement the following checking logic:

```
#if PLATFORM_ID == 0x06
// this is Intel Galileo
#else
// this is Intel Galileo Gen 2
#endif
```

It is also possible to check the board used during run-time using the `PLATFORM_NAME` directive, which is represented by the string `"GalileoGen2"` for Intel Galileo Gen 2 and `"Galileo"` for Intel Galileo. For example, the following code snippet shows how to check the board during run-time:

```
if(PLATFORM_NAME == "GalileoGen2")
{
    // this is Intel Galileo Gen 2
...
...
...

} else if (PLATFORM_NAME == "Galileo")
{
    // this is Intel Galileo
...
...
...
}
```

Project: DHT Sensor Library with Fast I/O APIs

The idea is to create a project that provides environmental temperature and humidity control using sensors called DHT. This project runs on Intel Galileo and Intel Galileo Gen 2.

You'll do the following with this project:

1. Introduce a simple project that explores fast I/O APIs.
2. Provide a workaround to solve the impact caused by `pinMode()` when the pin direction changes.
3. Create a library especially for Intel Galileo to communicate with DHT sensors.

Materials List

This project requires the materials listed in Table 4-6.

Table 4-6. *Materials List for the DHT Sensor Library Project*

Number	Description
1	DHT11 sensor
1	1/4W 5K Ohm resistor
1	Low enable tri-state buffer NTE74HC125 or equivalent

The cost of this project is estimated at about $6.00 US, not including the Intel Galileo board.

The DHT Sensor

The DHT sensor is a humidity and temperature sensor that communicates using one single wire connected to one of the digital ports on the microcontrollers. This connection is called a single-wire two-way.

The basic connection of the DHT is shown in Figure 4-8.

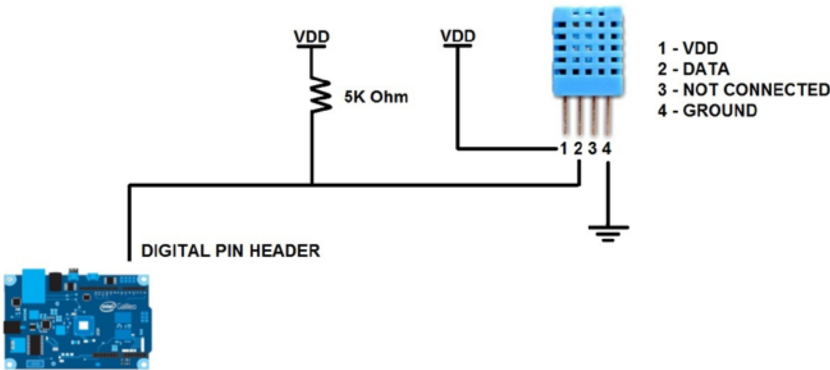

Figure 4-8. *Connecting DHT11 to Intel Galileo*

To communicate with this sensor, you need a simple yet fast protocol, so the fast I/O API discussed in this chapter will be used. Basically, the protocol sends a command to the sensor and waits for a response that contains the local temperature and humidity.

Figure 4-9 shows how the protocol works when a command is sent and data is received.

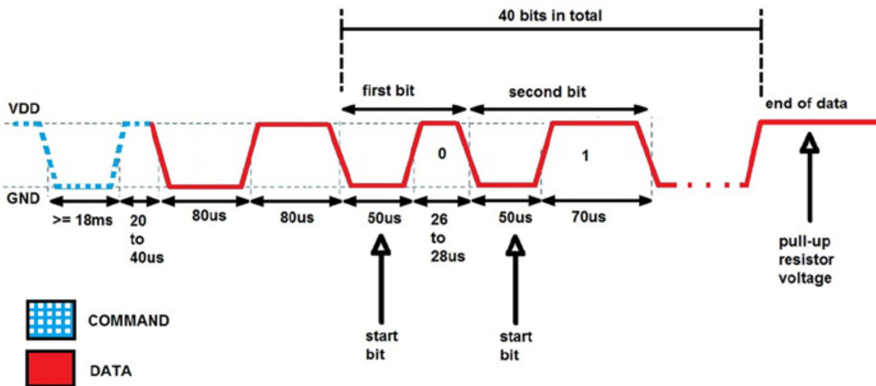

Figure 4-9. *DHT11 protocol*

Initially Intel Galileo must pull down the digital port at at least 18ms. The sensor's interpreter will intepret that as a command and start to transmit the data response after 20 to 40 microseconds.

The sensor pulls down and pulls up the voltage level during 80 microseconds on each state.

Then the transmission of bits that will report the temperature and humidity starts. Each bit is composed of a start bit and the bit value in general; in other words, a signal will tell if the bit value is `0` or `1`. The start bit is always identified by a pull down voltage of 50 microseconds. When the bit is a pull up voltage that remains between 26 to 28 microseconds, the bit value is `0`. If the voltage remains at 70 microseconds, the bit value is `1`. It is expected to receive 40 bits that will be used to fill five bytes (5 bytes = 8 bits * 5 = 40 bits).

These five bits will be parsed as follows:

- **Byte 0** -> Contains the humidity
- **Byte 1** -> Always 0
- **Byte 2** -> Contains the temperature
- **Byte 3** -> Always 0
- **Byte 4** -> The checksum that must match the sum of other four bytes

If you understand how the protocol works, it is time to learn what must be done that make it work using Intel Galileo's digital I/O headers.

In a single wire, commands and responses move between the sensor and the Intel Galileo based on the transition of pin states (high and low) during a specific short period of time. This protocol is based on the voltage transition in a short period of time. This is possible because Intel Galileo is supposed to change the port direction all the time. In other words, when a command must be sent to the sensor, the port I/O is configured as OUTPUT. The pin state changes during short periods and the sensor detects it, interpreting as a command.

The next step is to be ready to receive the response. For this, Intel Galileo must configure to port direction to INPUT. With the command received by the sensor, a response is sent back to Intel Galileo. It reads the state transition during a short period of time, thereby converting it to a sequence of bits that will be used to "read" the sensor's values.

In case of the DHT sensor, this transition between OUTPUT and INPUT that sends the commands and receives the responses is done in the order of microseconds (20 to 40 microseconds). This is impossible to do using `pinMode()` with Intel Galileo because it takes around 3 milliseconds (see the section entitled "When Port Speed Is Not Enough - pinMode() Limitations" in this chapter).

Figure 4-10 shows the catastrophic result with the addition of the DC level while the `pinMode()` is still processing the change of pin direction. Pay attention to the numbered arrows. The number 1 arrow points to the oscilloscope channel that's set to 5ms per time division. The number 2 arrow shows how `pinMode()` takes almost 3ms (less than one division of 5ms) and applies a DC level in the port.

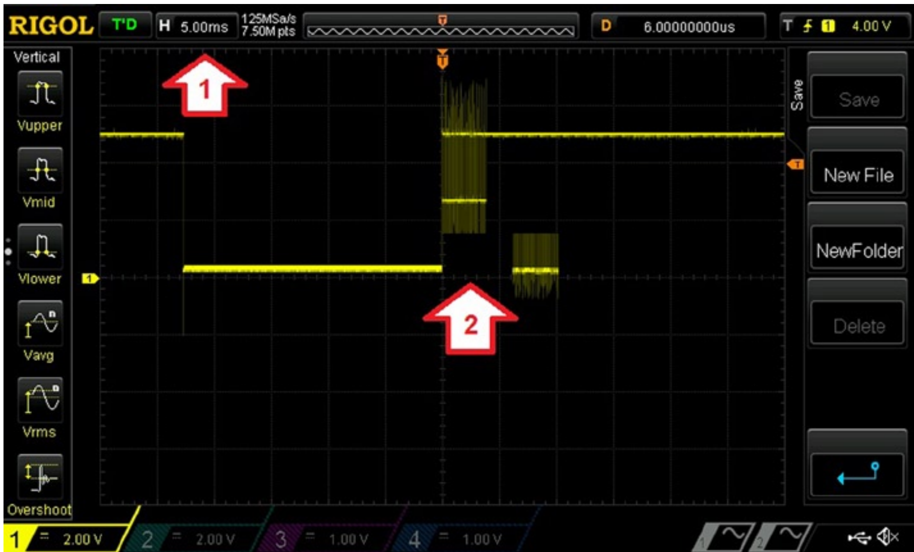

Figure 4-10. *When `pinMode()` takes 3ms to change the port direction*

A Workaround Using Tri-State Buffers

If `pinMode()` does not have enough performance for this sensor; however, it is possible to create a workaround and separate commands from responses using two different pins even if the sensor uses a single wire to communicate.

One of the techniques used to make such an isolation is to use a low-enabled tri-state buffer, as shown in Figure 4-11.

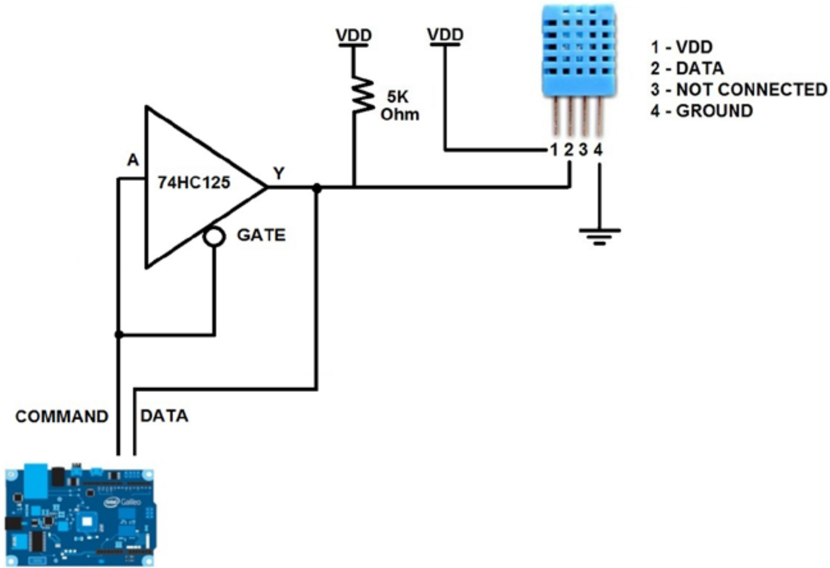

Figure 4-11. *Transforming a one-wire to a two-wires connection*

The NTE74HC125 is a low-enabled tri-state buffer that internally contains four gates. Figure 4-12 shows the gates disposition internally in the NTE74HC125. The letter `G` is used to represent the gate, `Y` is the output, and `A` is the input. The numbers in front of each letter represent the gate number. You need only one gate for this project.

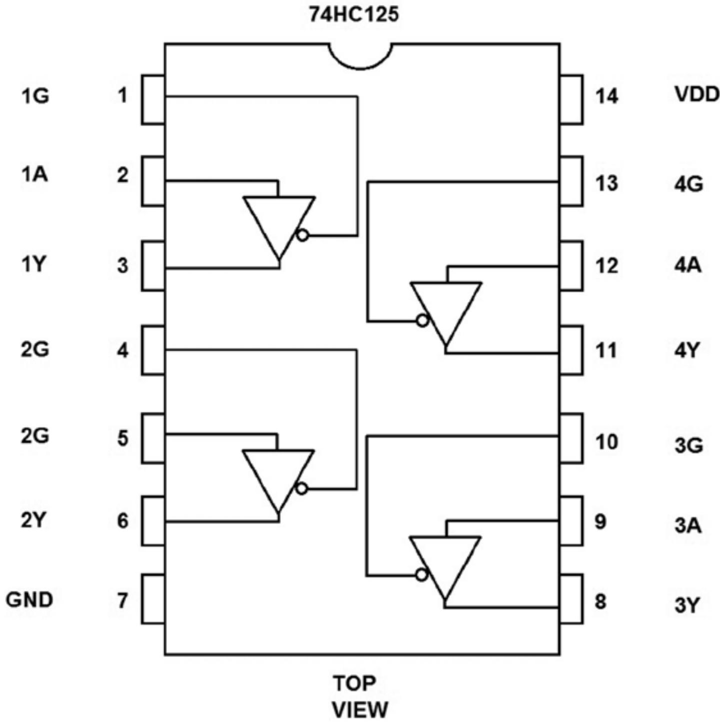

Figure 4-12. *Top view of the 74HC125 and pin-out disposition*

To understand how this is possible, it is necessary to understand some basic concepts related to tri-state buffers.

The tri-state provides three logic states: LOW (0), HIGH (1), and Z (high impedance), as shown in Table 4-7.

Table 4-7. *Logic Table of a Low-Enabled Tri-State Buffer*

INPUT	GATE	OUTPUT
0	0	0
1	0	1
0	1	Z (High impedance)
1	1	Z (High impedance)

In a low-enabled tri-state buffer, the gate enables the input to the output only when the gate has a low state (0). When the gate is high (1), the output is high impedance, in other words, it's floating.

Floating means that the current never will pass over there due to the high impedance, so the circuit is totally isolated. Figure 4-13 shows a representation of the low-enabled gate operation in a tri-state buffer.

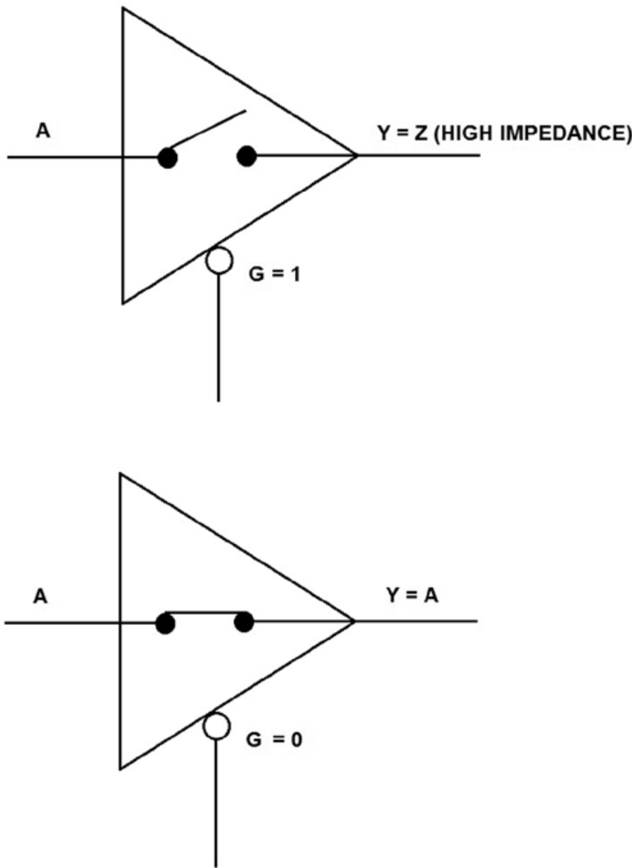

Figure 4-13. *Low-enabled tri-state gate operation*

The idea for resolving the `pinMode()` issue is very simple. As soon as the command is sent, the gate will be opened, which will isolate the input pin. The current will flow exclusively to the output pin, which will receives the response.

In that way, it is possible to isolate commands from responses even when the sensor uses a single wire to communicate and there are not concerns regarding voltage level.

Creating a New Library for DHT11 Sensor

This section explains how to create a functional DHT sensor library for Intel Galileo boards.

The Problem with Public DHT Libraries

The DHT11 sensors have different free libraries available on the Internet that you can download to the `libraries` directory of the IDE and run a sketch example.

Unfortunately, these libraries were mostly created for microcontrollers with a fixed clock and loop counters used as timing purposes to detect different components in the DHT11 protocol. For example, at http://playground.arduino.cc/main/DHT11Lib, there is a library created for DHT11, and in the `DHT.cpp` file, it is possible to see the following:

```
// ACKNOWLEDGE or TIMEOUT
        unsigned int loopCnt = 10000;
        while(digitalRead(pin) == LOW)
                if (loopCnt-- == 0) return DHTLIB_ERROR_TIMEOUT;

        loopCnt = 10000;
        while(digitalRead(pin) == HIGH)
                if (loopCnt-- == 0) return DHTLIB_ERROR_TIMEOUT;
```

This technique never will be precise enough for Intel Galileo. Once the Arduino libraries run in the Linux userspace context, it will receive a different priority. Fixed clocks and loop counting are not applicable to Galileo.

The second problem is the lack of performance of `pinMode()` to switch the pin direction, as explained.

Designing a New DHT Library

Based on these two problems, a new library was created, as shown in Listings 4-18 and 4-19.

The code of this library is located in the folder `code/DHT11_Galileo`, which must be installed in your IDE. The process is simple:

1. Close all IDEs running.
2. Move the folder called `DHT11_Galileo` within the directory called `arduino-1.5.3/libraries` of your IDE installation.
3. Open the IDE again and select Sketch➤Import Library from the menu. Check if `DHT11_Galileo` is listed. If it is, open the sketch called `DHT_4_Galileo.ino` in the `examples` folder of the library. You can also select Files➤Examples➤ DHT11_Galileo➤ DHT11_4_Galileo using the menu. If the library is not present, check if the library is properly installed as recommended. You can also learn more about the libraries installed at http://arduino.cc/en/Guide/Libraries.

Listing 4-18. DHT_4_Galileo.h

```
#ifndef DHT_4_GALILEO
#define DHT_4_GALILEO
#include <Arduino.h>
#include <stdint.h>

class DHT_4_Galileo
{

public:

   // to report ACK status
   static const int DHT_ACK_OK = 0;
   static const int DHT_ACK_ERROR = -1;

   // to report reading status
   static const int DHT_READ_SUCCESS = 0;
   static const int DHT_READ_TIMEOUT_ERROR = -1;
   static const int DHT_READ_CHECKSUM_ERROR = -2;

   int humidity;
   int temperature;

    /*
     * The class constructor must receive the pins
     * used as gate/command and the pin to read
     * the data from the sensor
     */
    DHT_4_Galileo(uint8_t p_gate, uint8_t p_read);

    /*
     *  sendCommand():
     *  sends the commands and wait for the acknowledgement from the sensor
     *
     */
    int sendCommand();  // end of send_command()

    /*
     *  read():
     *  Reads the 40 bits of data, parse the data in
     *  temperature and humidity
     */
     int read ();
```

```
private:
   int pin_gate;
   int pin_read;

   /*
    * getFastIOMacro(int pin)
    * only an utility function to return
    * the right macro according to the pin
    */
   static int getFastIOMacro(int pin);

}; // end of class
#endif
```

The idea is to create a very simple class thay sends a command to the sensor and read its response. That's exactly what the class DHT_4_Galileo does with the sendCommand() and read() methods.

Note that the class constructor asks for two pins—p_gate is the pin that is connected to the gate of one of the tri-state gates (G) of 74HC125 and p_read is for the pin that will receive the data (Y).

Some constants were created to check if acknowledgement signals were received or if an error happened during the data readings:

```
// to report ACK status
static const int DHT_ACK_OK = 0;
static const int DHT_ACK_ERROR = -1;

// to report reading status
static const int DHT_READ_SUCCESS = 0;
static const int DHT_READ_TIMEOUT_ERROR = -1;
static const int DHT_READ_CHECKSUM_ERROR = -2;
```

Listing 4-19 shows the DHT_4_Galileo implementation.

Listing 4-19. DHT_4_Galileo.cpp

```
#include "DHT_4_Galileo.h"

#define DEBUG 1  // change to 0 if you do not want debug messages

    /*
     * The class constructor must receive the pins
     * used as gate/command and the pin to read
     * the data from the sensor
     */
```

```
DHT_4_Galileo::DHT_4_Galileo(uint8_t p_gate, uint8_t p_read)
{
     pin_gate = p_gate;
     pin_read = p_read;

     pinMode(pin_gate, OUTPUT_FAST);
     pinMode(pin_read, INPUT_FAST);
};

/*
 *  sendCommand():
 *  sends the commands and wait for the acknowledgement from the sensor
 *
 */
int DHT_4_Galileo::sendCommand()
{

   // pull down during 18 microseconds.. this is our command!
   fastGpioDigitalWrite(getFastIOMacro(pin_gate), LOW);
   delay(18);
   fastGpioDigitalWrite(getFastIOMacro(pin_gate), HIGH);  // High
impedance
   delayMicroseconds(40);

   // now let's check if some ACK was received
   unsigned long t0,ti;

   int state = fastGpioDigitalRead(getFastIOMacro(pin_read));
   int new_state = state;

   boolean ack = false;
   t0 = micros(); // number microseconds since booted
   while (micros()-t0<80)
   {
        new_state = fastGpioDigitalRead(getFastIOMacro(pin_read));
       if (new_state==0)
       {
           // cool!!! first ACK received during 80 microseconds
           ack = true;
           break;
       }
   }
```

```
       if (!ack)
       {
#if (DEBUG)
          Serial.println("problem in FIRST PART OF ACK");
#endif
          return DHT_ACK_ERROR;
       }

       ack=false;
       t0 = micros(); // number microseconds since booted
       while (micros()-t0 < 80)
       {
           // waiting for HIGH
           new_state = fastGpioDigitalRead(getFastIOMacro(pin_read));
           if (new_state!=0)
           {
              // the second ACK received!!! let's wait for the data!!!
              ack = true;
              break;
           }
       }

       if (!ack)
       {
#if (DEBUG)
           Serial.println("problem in SECOND PART ACK");
#endif
           return DHT_ACK_ERROR;
       }

       return DHT_ACK_OK;
    }  // end of send_command()

    /*
     *  read():
     *  Reads the 40 bits of data, parse the data in
     *  temperature and humidity
     */
     int DHT_4_Galileo::read () {

        unsigned long t0;

            // BUFFER TO RECEIVE
        uint8_t bits[5];
        uint8_t cnt = 7;
        uint8_t idx = 0;
```

```
int start_bit[40];
int reading_bit[40];

// cleaning arrays
for (int i=0; i<40; i++)
{
  start_bit[i] = 0;
  reading_bit[i] = 0;
}

for (int i=0; i<5; i++)
{
  bits[i] = 0;
}

 // READ OUTPUT - 40 BITS => 5 BYTES or TIMEOUT
for (int i=0; i<40; i++)
{

        //start bit
        // will stay low for 50us

        t0 = micros(); // number microseconds since booted
        while(fastGpioDigitalRead(getFastIOMacro(pin_read)) == 0)
        {
            // using 70 instead 50 us due to the remaining
            // state of last ack
            if ((micros() - t0) > 70) {
                return DHT_READ_TIMEOUT_ERROR;
            }
        }

        start_bit[i] = micros() -t0;

        t0 = micros(); // number microseconds since booted

        //reading bit
        // 26 to 28us  -> 0
        // up tp 70 us -> 1
        //int c = 0;
        while( fastGpioDigitalRead(getFastIOMacro(pin_read)) != 0) {
                if ((micros() - t0) > 77) {
                     return DHT_READ_TIMEOUT_ERROR;
            }
        };
```

```
                unsigned long delta_time =  micros() - t0;
                reading_bit[i] = delta_time;

                if (delta_time > 50) bits[idx] |= (1 << cnt);
                if (cnt == 0)       // next byte?
                {
                        cnt = 7;    // restart at MSB
                        idx++;      // next byte!
                }
                else cnt--;
        }

        // dump
#if (DEBUG)
        Serial.println();
        for (int i=0; i<40; i++)
        {
          Serial.print(i);
          Serial.print("   ");
          Serial.print(start_bit[i]);
          Serial.print("   ");
          Serial.print(reading_bit[i]);
          Serial.print("   ");
          if (reading_bit[i] > 40)
            Serial.println("1");
          else
            Serial.println("0");
        }

        Serial.println();
        Serial.println("BYTES PARSED:");
        Serial.println("------------");

        for (int i=0; i<5; i++)
        {
          Serial.print(i);
          Serial.print(": ");
          Serial.println(bits[i]);
        }
                 Serial.println();
#endif
        // parsing the bits
        humidity    = bits[0];
        temperature = bits[2];
        uint8_t sum = (bits[0] +  bits[1] + bits[2] +  bits[3]) & 0xff;
```

```
        if (bits[4] != sum)
        {
          return DHT_READ_CHECKSUM_ERROR;
        } else {
                   return DHT_READ_SUCCESS;
        }

}

   /*
    * getFastIOMacro(int pin)
    * only an utility function to return
    * the right macro according to the pin
    */
   int DHT_4_Galileo::getFastIOMacro(int pin)
   {
        int macro;
        switch (pin)
        {
#if PLATFORM_ID != 0x06
             // this is Galileo Gen 2
            case 0: macro = GPIO_FAST_IO0;
              break;
            case 1: macro = GPIO_FAST_IO1;
               break;
#endif
            case 2: macro = GPIO_FAST_IO2;
               break;
            case 3: macro = GPIO_FAST_IO3;
               break;
#if PLATFORM_ID != 0x06
            // this is Galileo Gen 2 - no fast I/O for pins 7 and 8
            case 4: macro = GPIO_FAST_IO4;
               break;
            case 5: macro = GPIO_FAST_IO5;
               break;
            case 6: macro = GPIO_FAST_IO6;
               break;
            case 9: macro = GPIO_FAST_IO9;
               break;
            case 10: macro = GPIO_FAST_IO10;
               break;
            case 11: macro = GPIO_FAST_IO11;
               break;
```

```
            case 12: macro = GPIO_FAST_IO12;
               break;
            case 13: macro = GPIO_FAST_IO13;
               break;
#endif
            default:
                     macro = 0;
               break;

        }

        return macro;
   } // end of getFastIOMacro()
```

Reviewing the DHT_4_Galileo.cpp Library

The constructor of this class receives the pin connected to the gate and the pin connected to the data, as explained. In the constructor, the pins are set as INPUT_FAST and OUTPUT_FAST through the pinMode() function. It defines the possibility for using the new fast I/O functions.

The second method implemented is sendCommand(). It pulls the pin connected to the tri-state gate up and down during the 18 milliseconds, which immediately keeps the acknowledgement waiting. The sensors pull the voltage levels down and up during the 80 microseconds on each state. If the acknowledgement is not received, sendCommand() reports this by returning DHT_ACK_ERROR; otherwise, DHT_ACK_OK is returned. Note that this method uses fastGpioDigitalWrite() and fastGpioDigitalRead() fast I/Os. The static function getFastIOMacro() is used only to convert the regular pin to the specific fast I/O macros.

Finally, the read() method tries to read the 40 bits transmitted by the sensors containing the data to be parsed. The fast I/O fastGpioDigitalRead() function reads the start and data bits. In the first code sequence, the code waits for the start bit:

```
t0 = micros(); // number microseconds since booted
while(fastGpioDigitalRead(getFastIOMacro(pin_read)) == 0)
 {
           // using 70 instead 50 us due to the remaining
           // state of last ack
           if ((micros() - t0) > 70) {
                 return DHT_READ_TIMEOUT_ERROR;
           }
 }

 start_bit[i] = micros() -t0;

 t0 = micros(); // number microseconds since booted
```

```
 //reading bit
 // 26 to 28us  -> 0
 // up tp 70 us -> 1
  while( fastGpioDigitalRead(getFastIOMacro(pin_read)) != 0) {
          if ((micros() - t0) > 77) {
             return DHT_READ_TIMEOUT_ERROR;
     }
 };
unsigned long delta_time =  micros() - t0;
reading_bit[i] = delta_time;
```

Note that the `reading_bit[]` array saves how long each data bit took in microseconds. Thus to decide if the bit received was a 0 or a 1, you use this simple code:

```
if (delta_time > 50) bits[idx] |= (1 << cnt);
   if (cnt == 0)    // next byte?
   {
           cnt = 7; // restart at MSB
           idx++;   // next byte!
   }
else cnt--;
```

The `bits[]` array is responsible for saving the five bytes expected from the sensor after parsing the 40 bits received. The code checks if the time that the bit was received is bigger than 50us. If it is, it's considered bit 1; otherwise, it is 0 and each bit is shifted properly until eight bits are received. When the total of eight bits are received, the index in the array is incremented and the logic continues by saving the next eight bits in the next element of `bits[]` array.

With the `bits[]` array properly filled, the `humidity` and `temperature` classes members are loaded and the checksum is checked to make sure the data received is correct:

```
// parsing the bits
humidity    = bits[0];
temperature = bits[2];
uint8_t sum = (bits[0] +  bits[1] + bits[2] +  bits[3]) & 0xff;

if (bits[4] != sum)
{
    return DHT_READ_CHECKSUM_ERROR;
} else {
     return DHT_READ_SUCCESS;
}
```

Creating the Sketch for DHT Sensor

Listing 4-20 shows the sketch itself. Note that the sketch is still simple but now two pins are defined in the contructor of the `DHT_4_Galileo` class.

Listing 4-20. DHT11_4_Galileo.ino

```
#include <DHT_4_Galileo.h>

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  Serial.println("start......");
  delay(3000);
}

void loop() {
  // put your main code here, to run repeatedly:

  Serial.println("send command..");
  Serial.println();

  DHT_4_Galileo dht(3, 2);

  dht.sendCommand();

  int response = dht.read();
  if (response == DHT_4_Galileo::DHT_READ_SUCCESS)
  {
     Serial.println("RESULTS:");
     Serial.println("----------------");
     Serial.print("Humidity:");
     Serial.print(dht.humidity,1);
     Serial.print(",\t");
     Serial.print("Temperature (C):");
     Serial.println(dht.temperature,1);
  }
  else
  {
     Serial.print("there is an error:");
     Serial.println(response);
  }
  delay(5000);

}
```

With the class, the sketch is very simple. In `loop()`, a `dht` object is created using pin 3 as the gate and pin 2 as the data. The methods `sendCommand()` and `read()` are used.

Running the Code

As soon you run the sketch, open the IDE serial console by chooseing Tools ➤ Serial Monitor or pressing the keys CTRL+SHIFT+M.

With the debug enabled, you will be able to see the index of the bit received, the respective start time, the bit time duration, and whether the bit is considered a 0 or a 1 in the last column for all 40 bits. You will also see these 40 bits parsed in five bytes and, if the checksum matches, the humidity and temperature.

```
0   2   19  0
1   53  23  0
2   54  70  1
3   53  25  0
4   53  22  0
5   54  69  1
6   54  23  0
7   53  69  1
8   54  22  0
9   54  23  0
10  53  24  0
11  53  23  0
12  54  23  0
13  54  23  0
14  53  24  0
15  53  25  0
16  53  23  0
17  54  23  0
18  54  23  0
19  53  71  1
20  53  70  1
21  54  23  0
22  54  23  0
23  53  24  0
24  53  23  0
25  54  23  0
26  54  23  0
27  53  24  0
28  53  23  0
29  54  22  0
30  54  22  0
31  53  26  0
32  53  24  0
33  53  23  0
34  54  70  1
35  53  71  1
36  53  70  1
37  54  70  1
38  54  23  0
39  53  67  1
```

```
BYTES PARSED:
------------
0: 37
1: 0
2: 24
3: 0
4: 61

RESULTS:
-----------------
Humidity:37,    Temperature (C):24
```

If you probe pins 2 and 3 in the oscilloscope, you will see signals similar to the ones in Figure 4-14.

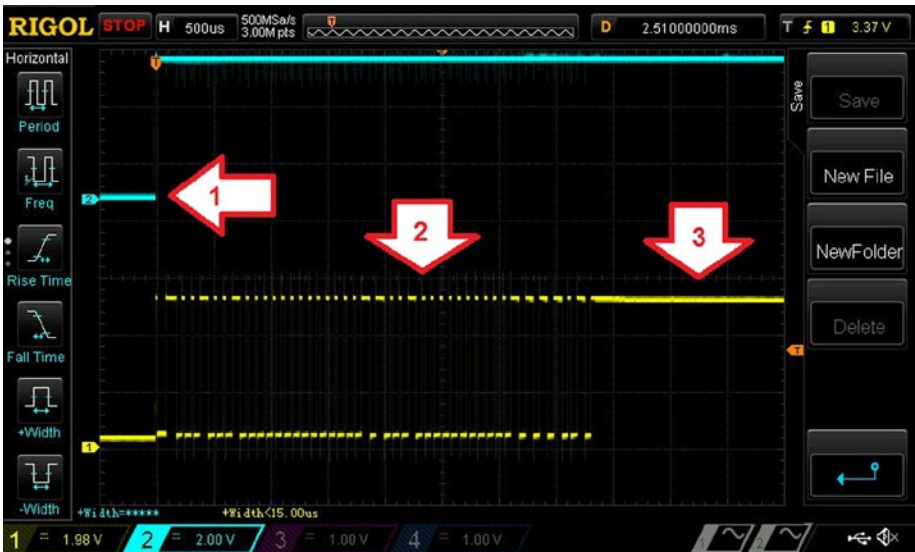

Figure 4-14. The correct signals when operating a DHT11 sensor

As you can see in Figure 4-14, the arrow 1 pointing to the signal is the command, the arrow 2 is pointing to the acknowledgements and the sequence of 40 bits received from the sensor, and the arrow 3 is the end of transmission with the tension in the pull-up 5K Ohms resistor.

Replacing the Tri-State Buffer with a Diode

Suppose you are very excited to have your DHT sensor working with Intel Galileo but you don't have a tri-state buffer. You can't wait for the electronic store near you to open and did not yet order a buffer online.

A simple, but not perfect, solution is to replace the tri-state buffer with a diode, as shown in Figure 4-15.

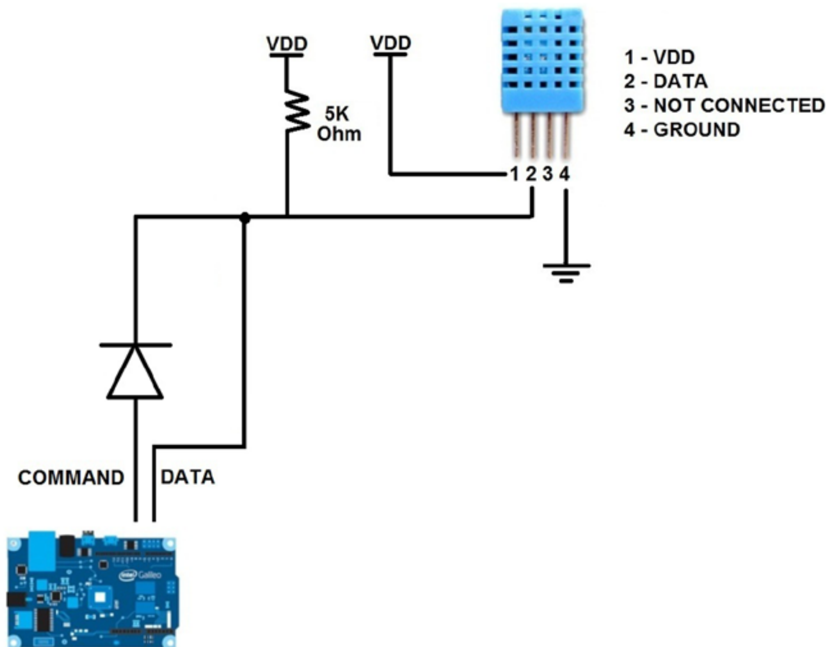

Figure 4-15. *Replacing the tri-state buffer with a diode*

It's best to use a 1N4148, 1N4448, 1N916A, or 1N916B diode, as they are the fastest.

The diode offers a voltage level reduction in the digital port input on Galileo but does not compromise the detection of the level state (HIGH or LOW).

This is not a perfect solution, but if you have a diode in hand, you can make the test.

Summary

This chapter discussed the new APIs created specially for Intel Galileo boards, including the differences when the APIs are used on Intel Galileo and Intel Galileo Gen 2, the best way to optimize the pins' performance, choosing the right clusters, and what's new with tone and pulsein APIs.

You also learned about techniques for hacking the servo API, the GPIO expander needed to generate different frequencies, and how to create a single sketch that can run in different Intel Galileo boards.

At the end of this chapter, a real project using temperature and humidity DHT11 sensor was demonstrated.

The information provided in this chapter is essential for making Intel Galileo communicate with devices that need high-speed performance in the port I/O and also helps to port libraries created exclusively for Atmel microcontrollers that require performance not achieved by Intel Galileo when the regular Arduino functions are used.