.NET Gadgeteer Module Builder's Guide

Version 1.10 – April 19, 2013

Abstract

This *.NET Gadgeteer Module Builder's Guide* is an introduction to manufacturing specifications for Gadgeteer modules. The guide explains mainboard socket types and peripheral buses that Gadgeteer-compatible modules use, assemblies that are written in managed code to provide interfaces to applications, and mechanical design requirements for components.

Microsoft® .NET Gadgeteer extends the Microsoft commitment to .NET developers who are meeting the demand for specialized, embedded hardware components. Modules built on this platform use the .NET Micro Framework to provide direct access to hardware, without an underlying operating system, from a scaled-down version of the .NET Common Language Runtime (CLR).

Beyond embedded applications, Gadgeteer opens a large educational market for the platform. This unique solution to building small devices will be useful to researchers, secondary and tertiary educators, and hobbyists who want to build experimental devices that use Gadgeteer modules.

Open standards for hardware and software interfaces, including open source software for core libraries, enable manufacturers to build compatible mainboards and peripheral modules. The aim is to promote a thriving ecosystem of hardware that is useful to a large and diverse set of users. Early adopters who are building Gadgeteer modules can expect support from Microsoft researchers.

Document History

Date	Change
April 25, 2011	Limited partner release
April 29, 2011	Public Beta – draft 1.0
June 23, 2011	Public Beta – draft 1.3
July 26, 2011	Public Beta – draft 1.4
August 23, 2011	Public Release 1.5
October 3, 2011	Public Release 1.6
October 31, 2011	Public Release 1.7 – updated screenshots
April 5, 2012	Public Beta – draft 1.8
May 15, 2012	Public Release 1.9
April 19, 2013	Public Release 1.10

Contents

Introduction

The *.NET Gadgeteer Module Builder's Guide* introduces procedures for building Microsoft® .NET Gadgeteer-compatible modules, including electronic hardware design, managed driver implementation, and mechanical specifications.

Throughout this document, a *module* refers to a peripheral hardware device that can be connected to a .NET Gadgeteer-compatible mainboard, and is accompanied by a software library that provides an application programming interface (API). The design guidelines for .NET Gadgeteer-compatible mainboards are documented in a separate document, which is available on the .NET Gadgeteer CodePlex page.

The purpose of these guidelines is to help module designers ensure that their modules are compatible with the widest range of .NET Gadgeteer mainboards. We recommend that designers follow these guidelines to ensure compatibility but the guidelines cannot cover every possible situation. Wherever possible, we have tried to explain the rationale behind each guideline to make it clear what issues might arise if it is not followed. We also ask module designers to treat this as an extensible document and contribute feedback to improve and clarify the process for others.

Throughout this document, all dimensions are in millimeters (mm). Unless specified, drawings and images are not to scale.

System Requirements

To use the .NET Gadgeteer platform, you need the .NET Micro Framework Software Development Kit (SDK) and Microsoft .NET Gadgeteer core libraries. The .NET Micro Framework and Gadgeteer core libraries can be installed from Microsoft at the .NET Micro Framework website and the .NET Gadgeteer website, respectively. You do not need a mainboard to build a module, but testing that module requires a mainboard.

The Gadgeteer SDK and the .NET Micro Framework require Microsoft Visual Studio® in either of the following versions:

- Visual Studio 2012, the full-featured Visual Studio application development suite, with support for multiple programming languages.
- Visual Studio Express 2012 for Windows Desktop, a free alternative that provides lightweight, easy-to-learn, and easy-to-use tools for creating Windows® applications.

Connecting Modules to Mainboards

A Gadgeteer module combines device hardware with software that provides a simple API to the functionality of the module from .NET managed code.

When you build a module, you should first consider how to electrically interface between the module hardware and a Gadgeteer mainboard.

We provide a template to help you create the managed driver that is required for new each module. For more information, see **Interfacing with Hardware** later in this document.

Mainboard Socket Types

Gadgeteer mainboards expose their I/O capabilities through *sockets*. Each socket is a 10-way connector, with pins labeled 1 through 10. For detailed information about connector specifications, see **Mechanical Design** later in this document.

Mainboard sockets support one or more different *socket types*. Each socket type is represented by a letter. When a mainboard socket is labeled with a socket type letter, it guarantees a particular set of electrical connections and interfaces on the socket's pins. The following table shows the .NET Gadgeteer Socket Types Definitions. In consultation with mainboard and module manufacturers, new socket types may be added in the future.

Socket Types Table (Version 17)

TYPE	LETTER	PIN 1	PIN 2	PIN 3	PIN 4	PIN 5	PIN 6	PIN 7	PIN 8	PIN 9	PIN 10
3 GPIO	X	+3.3V	+5V	GPIO!	GPIO	GPIO	[UN]	[UN]	[UN]	[UN]	GND
7 GPIO	Y	+3.3V	+5V	GPIO!	GPIO	GPIO	GPIO	GPIO	GPIO	GPIO	GND
Analog In	A	+3.3V	+5V	AIN (G!)	AIN (G)	AIN	GPIO	[UN]	[UN]	[UN]	GND
CAN	C	+3.3V	+5V	GPIO!	TD (G)	RD (G)	GPIO	[UN]	[UN]	[UN]	GND
USB Device	D	+3.3V	+5V	GPIO!	D-	D+	GPIO	GPIO	[UN]	[UN]	GND
Ethernet	E	+3.3V	+5V	[UN]	LED1 (OPT)	LED2 (OPT)	TX D-	TX D+	RX D-	RX D+	GND
SD Card	F	+3.3V	+5V	GPIO!	DAT0	DAT1	CMD	DAT2	DAT3	CLK	GND
USB Host	H	+3.3V	+5V	GPIO!	D-	D+	[UN]	[UN]	[UN]	[UN]	GND
I²C	I	+3.3V	+5V	GPIO!	[UN]	[UN]	GPIO	[UN]	SDA	SCL	GND
UART+ Handshaking	K	+3.3V	+5V	GPIO!	TX (G)	RX (G)	RTS	CTS	[UN]	[UN]	GND
Analog Out	O	+3.3V	+5V	GPIO!	GPIO	AOUT	[UN]	[UN]	[UN]	[UN]	GND
PWM	P	+3.3V	+5V	GPIO!	[UN]	[UN]	GPIO	PWM (G)	PWM (G)	PWM	GND
SPI	S	+3.3V	+5V	GPIO!	GPIO	GPIO	GPIO	MOSI	MISO	SCK	GND
Touch	T	+3.3V	+5V	[UN]	YU	XL	YD	XR	[UN]	[UN]	GND
UART	U	+3.3V	+5V	GPIO!	TX (G)	RX (G)	GPIO	[UN]	[UN]	[UN]	GND
LCD 1	R	+3.3V	+5V	LCD R0	LCD R1	LCD R2	LCD R3	LCD R4	LCD VSYNC	LCD HSYNC	GND
LCD 2	G	+3.3V	+5V	LCD G0	LCD G1	LCD G2	LCD G3	LCD G4	LCD G5	BACK-LIGHT	GND
LCD 3	B	+3.3V	+5V	LCD B0	LCD B1	LCD B2	LCD B3	LCD B4	LCD EN	LCD CLK	GND
Manufacturer Specific	Z	+3.3V	+5V	[MS]	[MS]	[MS]	[MS]	[MS]	[MS]	[MS]	GND
DaisyLink Downstream*	*	+3.3V	+5V	GPIO!	GPIO	GPIO	[MS]	[MS]	[MS]	[MS]	GND

LEGEND:

GPIO A general-purpose digital input/output pin, operating at 3.3 Volts.

(G) In addition to another functionality, a pin that is also usable as a GPIO.

(OPT) A socket type that is optionally supported by a mainboard or a module.

[UN] Modules must not connect to this pin if using this socket type. Mainboards can support multiple socket types on one socket, as long as individual pin functionalities overlap in a compatible manner. A pin from one socket type can overlap with a [UN] pin of another.

[MS] A manufacturer-specific pin. See the documentation from the manufacturer of the board.

! Interrupt-capable and software pull-up capable GPIO (the pull-up is switchable and in the range of 10,000 to 100,000 Ω).

***** Socket type * should not appear on a mainboard, only on DaisyLink modules. The [MS] pins on this socket type can optionally support reflashing the firmware on the module.

NOTES:

- All pins must be at least 3.6V tolerant.
- The low logic input is 0V minimum, 0.4V maximum. Logic input high is 0.7*Vdd min, Vdd+0.2 maximum.
- A module must not assume that the mainboard can pull up or pull down GPIOs using built in pull-ups/pull-downs. If a pull-up or pull-down resistor is required on a GPIO, the module should incorporate it into the PCB.
- A module's screen printing must list both X and Y if it is compatible with both (the mainboard will say only "X" or "Y," but not both). Socket-compatibility labels are case sensitive.
- If pins are shared across multiple sockets, users of shared bus socket types (such as I or S) must not impede use by other devices. For example, type S modules must respect the chip select signal.
- For socket type I, mainboards should include I²C bus pull-ups of 2,200 Ω on both SDA and SCL pins. Modules must *not* include pull-ups on these lines.
- For the DaisyLink protocol, which uses socket type X, pull ups should not be on the mainboard. For details of module pull-up requirements, see the DaisyLink specification.
- For socket types C, K, and U, the rationale for having GPIO-capable function pins is to allow module makers to implement transmit-only or receive-only modules with the flexibility of an additional GPIO pin.
- For socket type A and P, the rationale for having two of the three function pins with (G) and one without (G) is to allow modules to use 1, 2, or 3 Analog Input / PWM pins, and have the flexibility of GPIOs for other pins.
- For DaisyLink modules (using type X), pin 3 is for the DaisyLink neighbor bus, pin 4 is used for I²C SDA, pin 5 is used for I²C SCL. See the DaisyLink spec and Appendix 1, both in this document, for more details.
- Note for UART 'U' and UART+Handshaking 'K' socket types: TX (Pin 3) is data from the Mainboard to the Module, and RX (Pin 4) is data from the Module to the Mainboard. These are idle high (3.3V).
  RTS (Pin 6) is an output from the Mainboard to the Module indicating that the module may send data, and CTS (Pin 7) is an output from the Module to the Mainboard indicating that the Mainboard may send data.
  RTS/CTS lines are "not ready" if high (3.3V) and "ready" if low (0V).

Choosing a Socket Type for a Module

A module typically supports one function type because of the underlying interface that the electronics of that module support, such as universal asynchronous receiver-transmitter—UART (U), SPI (S), or Analog Input (A). If the module simply uses a few digital I/Os, socket type X should be used. For additional information, see the following section, **DaisyLink Protocol, Version 4**. The DaisyLink interface is a custom Gadgeteer interface that also uses socket type X. This protocol enables chaining many devices from a single socket.

In addition to hardware I²C (i.e. I²C with hardware support on mainboard processors) that is provided through Socket type I, .NET Gadgeteer supports Software I²C using GPIOs on socket types X or Y. Note that while Hardware I²C (Type I) modules MUST NOT have pull up resistors, Software I²C modules MUST have pull up resistors (2.2kΩ – 10kΩ), so a module cannot be transparently used with both. A module can be made compatible with both by providing switchable pull-ups.

Another option is to use "DaisyLink", a custom Gadgeteer interface that also uses socket type X and leverages Software I²C to enable chaining many devices from a single socket. An overview of DaisyLink can be found in the following section "DaisyLink Protocol".

Modules may support multiple function types on a single socket, although we do not expect this to be common. Note that a module that is so simple that it supports multiple socket types—such as a button that uses a single Interrupt Input on pin 3—should nonetheless list XY because most mainboard sockets are expected to be compatible with X or Y. Socket-compatibility labels are case sensitive.

Modules may use more than one socket to communicate with the mainboard. In this case, some sockets may be compulsory and some may be optional. Extra functionality is available to users if the optional sockets are connected, but using sockets is not mandatory.

DaisyLink Protocol, Version 4

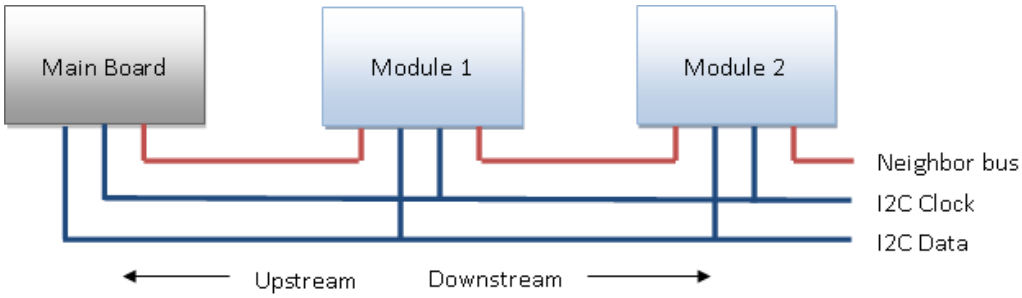
.NET Gadgeteer defines a new interface type: the DaisyLink interface, which is compatible with socket types "XY" (that is, X or Y). Socket-compatibility labels are case sensitive. Although this is just one of many interface options, the DaisyLink interface has the following features that make it preferable for some modules. The DaisyLink interface:

- Allows many modules, which are not required to be of the same type, to share a single socket.
- Allows modules to be connected in a chain fashion instead of a star fashion.
- Uses only three of the seven signal pins on a socket, and these pins are present on many of the defined Gadgeteer socket types.
- Facilitates addressing devices in a location-based fashion, for example, the first light-emitting diode (LED) on socket 5 or the second LED on socket 5.

- Facilitates auto-discovery of the type and number of devices that are chained on a given socket in a way that a shared bus by itself, such as I²C, cannot.

Every DaisyLink module includes its own microcontroller (MCU) that implements the DaisyLink protocol. The MCU also interfaces with any additional devices that may be present on the module, such as a sensor or a LED, and provides a way for the mainboard to read or control their state. In addition the MCU can be used to implement local control logic. For example, the MCU could be used to monitor the value of a sensor and calculate a running average, which can then be read from the mainboard, or maintain the brightness of an LED through pulse-width modulation, at a level specified by the mainboard. In this way, it is possible to offload certain tasks to the MCU that would otherwise consume processing resources on the mainboard.

The following diagram shows a DaisyLink configuration of modules.



DaisyLink includes two buses: an I²C shared data bus for data transfer and a neighbor bus that is used for assigning the I²C ID of each module on the shared bus.

A module is described as "downstream" from its neighbor if it is electrically further away from the mainboard in the chain and "upstream" from its neighbor if it is closer to the mainboard.

Because we expect DaisyLink modules to be widely used, we have chosen to implement the I²C bus in software rather than hardware. This implementation means that several sockets can separately support the DaisyLink interface rather than requiring the single hardware I²C bus to be used for all DaisyLink devices and all standard I²C devices. It also means that the individual pins can be used as GPIOs when they are not used for DaisyLink communication.

For more information about the DaisyLink protocol specification, see Appendix 1.

Module Code Libraries

The following guidelines specify recommended practices in the design of the software libraries that accompany Gadgeteer modules. Note that the easiest way of being compliant with these guidelines is to use the Module Template included in the Gadgeteer Core installer, which is described later under **Module Software Template**.

Assembly Naming

Each module should have its own assembly, which is a DLL that is created as a .NET Micro Framework class library project in Visual Studio. The class should be named *Gadgeteer.Modules.ManufacturerName.ModuleName* and the assembly named GTM.ManufacturerName.ModuleName.dll. This naming convention helps users find the right assemblies in a Visual Studio project and reference required assemblies as needed. Using one module per assembly makes it easy to update each module's code individually. Otherwise, updating the code in one module to get some new functionality might break another module.

To make it easy for the user to find the correct software DLL for a given hardware module, the module and manufacturer names must match the silkscreened names on the hardware module as closely as possible (remove spaces as appropriate).

Version numbers are not required to be included in the software class name, because the same software driver is often expected to work across multiple hardware revisions. Conversely, if the hardware changes significantly enough that the software is incompatible, we recommend that you use a different module name to avoid confusion.

The easiest way to make a module assembly is to use the Gadgeteer Module Template from the Gadgeteer core libraries. Provide the module name and, after the template loads, change the namespace name to *Gadgeteer.Modules.ManufacturerName*.

Summary

- Use the Module Template provided by the Gadgeteer core libraries.
- Create one module class per DLL.
- Name the module class *ModuleName* and use the namespace *Gadgeteer.Modules.ManufacturerName*.

Class Definition, Namespace, and Constructors

To help users find modules quickly, we have varied from the usual .NET naming conventions, which specify that the namespace that a company's libraries use should start with the company name. We ask that manufacturers use a Gadgeteer-specific namespace—*Gadgeteer.Modules.ManufacturerName*—for modules.

The following example illustrates the naming conventions and creates a class for *InterestingSensor* that is exposed in the *Gadgeteer.Modules.ManufacturerName* namespace:

```
using System;
using Microsoft.SPOT;

using GT = Gadgeteer;
using GTM = Gadgeteer.Modules;
using GTI = Gadgeteer.Interfaces;

namespace Gadgeteer.Modules.ManufacturerName
{
```

```
  /// <summary>
  /// Microsoft .NET Gadgeteer module that senses whether something is interesting.
  /// </summary>
  public class InterestingSensor : GTM.Module
  {
    // Note: A constructor summary is auto-generated by the doc builder.
    /// <summary></summary>
    /// <param name="socketNumber">The socket that this module is plugged in to.</param>
    /// <param name="socketNumberTwo">The second socket that this module is plugged in
to.</param>
    public InterestingSensor(int socketNumber, int socketNumberTwo)
    {
      // Implementation details.
    }
  }
}
```

Note that only Gadgeteer modules should have public classes in the Gadgeteer.Modules namespace. Other supporting classes that are public—visible to users—can either be nested classes of the module class or in another namespace (e.g. *ManufacturerName.Gadgeteer.Classname*). To the user, the Gadgeteer.Modules (GTM) namespace should contain physical modules that can be plugged in and nothing else.

Each module must supply a constructor of form shown in the following example. In this case there are two required sockets, but there could be any number of required and optional sockets.

```
    // Note: A constructor summary is auto-generated by the doc builder.
    /// <summary></summary>
    /// <param name="socketNumber">The socket that this module is plugged in to.</param>
    /// <param name="socketNumberTwo">The second socket that this module is plugged in
to.</param>
    public InterestingSensor(int socketNumber, int socketNumberTwo)
    {
    }
```

For optional sockets, the constructor must accept the value Socket.Unused to mean "This optional functionality is not used" (Socket.Unused is a static property defined in the Socket class).

Modules may define additional constructors, but these will NOT be used by the graphic designer. Modules MUST NOT have functionality in constructors that can't also be accessed through methods/properties.
Certain types of modules can usefully derive from some intermediate abstract classes, such as *NetworkModule*, *DaisyLinkModule*, *DisplayModule*, and *SimpleGraphicsInterface.* These are nested classes under Gadgeteer.Module.

Modules that use an intermediate class may need to pass values to that class as part of each constructor that uses the *base* keyword. The following example shows a DaisyLinkModule:

```
/// <summary></summary>
/// <param name="socket">The mainboard socket that has the <see cref="Relay"/> module plugged into it.</param>
public Relay(int socketNumber)
  : base(socketNumber, 0x12, 0, 2, 2, 50, "Relay")
```

```
{
}
```

Summary

- Use the *Gadgeteer.Modules.ManufacturerName* namespace for module implementations.
- Derive the class from the *Gadgeteer.Modules.Module* base class or an intermediate class such as *Gadgeteer.Modules.Module.DaisyLinkModule*.
- Use int *socketNumber* as the parameter name in the constructor.

Interfacing with Hardware

We provide a template that simplifies driver implementation for a new Gadgeteer module. The template installs with the Gadgeteer SDK and is accessible from the **File** menu under **New/Project** in Visual Studio. When you create a **.NET Gadgeteer Module Project**, a read-me text file opens that describes the elements of the module interface and the recommended code patterns.

The example code that the template creates shows how to implement a class constructor, properties, events, and event-handler delegate functions. It shows the triple-slash /// commenting style, which automatically creates an XML file that Intellisense uses to provide context-sensitive help in Visual Studio.

The constructor should use GTI (Gadgeeer.Interfaces) classes where possible to interact with the underlying hardware, such as GTI.Serial. The following example shows a button connected to an interrupt input:

```
public Button(int socketNumber)
{
   GT.Socket socket = Socket.GetSocket(socketNumber, true, this, null);
   this.input = new GTI.InterruptInput(socket, GT.Socket.Pin.Three, GTI.GlitchFilterMode.On,
                                GTI.ResistorMode.PullUp, GTI.InterruptMode.RisingAndFallingEdge, this);

   this.input.Interrupt += new GTI.InterruptInput.InterruptEventHandler(this._input_Interrupt);

}

private GTI.InterruptInput input;

private void _input_Interrupt(GTI.InterruptInput input, bool value)
{
   this.OnButtonEvent(this, value ? ButtonState.Released : ButtonState.Pressed);
}
```

If possible with the interface to the module, the constructor should verify that the module is actually plugged into the correct socket, and, if it isn't plugged in, throw an exception. This is better than the module silently failing. With the DaisyLinkModule subclasses, this is automatic. With a GTI.InterruptInput, as in the previous example, this would be difficult to do and is not done.

The provided GTI (Gadgeteer.Interfaces) classes automatically

- Check the socket type is correct, .e.g GTI.SPI ensures that the socket provided supports type 'S', throwing a helpfully worded exception to the end user if this is not true.
- Reserve pins by using GT.Socket.ReservePin, throwing a helpfully worded exception to the end user if there is a conflict.

If the required functionality cannot be implemented by using a helper class in Gadgeteer.Interfaces, you must manually reserve pins for that interface using Socket.ReservePin(), and you should check that the socket type is correct using Socket.EnsureTypeIsSupported().

Summary

- If you use an intermediate module class, such as GTM.DaisyLinkModule, the class should do all the setup work and provide methods to communicate with the module.
- If no subclass applies, the easiest way is to use GT.Socket.GetSocket() to get an instance of the Socket class for that module, and then GTI.XXX interfaces to communicate with the module.
- If neither of the two preceding points applies, be sure to use the GT.Socket.ReservePin method to reserve the socket pins that the module uses, and the GT.Socket.EnsureTypeIsSupported method to check that the socket is of a valid type for the module.

Debug and Error Printing

Modules should use the Module.DebugPrint and Module.ErrorPrint methods to show debug and error output, respectively. Errors are always printed, and debug messages are printed if the user sets Module.DebugPrintEnabled=true. This gives users per-module control of debug messages that use a standard API.

Events

To provide asynchronous events, use the standard Gadgeteer event pattern as shown in the following basic example:

```
/// <summary>
/// Represents the delegate that is used for the <see cref="MotionDetected"/> event.
/// </summary>
/// <param name="sender">The <see cref="MotionSensor"/> that raised the event.</param>
/// <remarks>
/// The motion sensor takes approximately 10 seconds to calibrate before
/// the <see cref="SensorReady"/> event is raised.
/// During calibration, the <see cref="MotionDetected"/> event is not raised.
/// </remarks>
public delegate void MotionDetectedEventHandler(MotionSensor sender);

/// <summary>
/// Raised when the <see cref="MotionSensor"/> detects motion.
/// </summary>
public event MotionDetectedEventHandler MotionDetected;
```

```
private MotionDetectedEventHandler _OnMotionDetected;

/// <summary>
/// Raises the <see cref="MotionDetected"/> event.
/// </summary>
/// <param name="sender">The <see cref="MotionSensor"/> that raised the event.</param>
protected virtual void OnMotionDetectedEvent(MotionSensor sender)
{
   if (_OnMotionDetected == null)
        _OnMotionDetected =  new MotionDetectedEventHandler(OnMotionDetectedEvent);
    if (Program.CheckAndInvoke(MotionDetected, _OnMotionDetected, sender))
    {
        MotionDetected(sender);
    }
 }
```

The foregoing example consists of:

- A delegate declaration for an event handler.
- An event.
- A private pointer to the event-raiser method.
- The event-raiser method itself.

The event raiser consists of the standard pattern. You must:

- Check that the private pointer to itself is valid and, if not, construct it.
- Use Program.CheckAndInvoke to get onto the Dispatcher thread. This returns *true* if you should trigger the event and *false* otherwise. If it returns *false*, it has scheduled the raiser method to be called on the dispatcher.

The event should be triggered from within the rest of the module's code with the following code:

```
// Raise the motion event.
OnMotionDetectedEvent(this);
```

The first parameter is always the source of the event, which in this case is MotionSensor, so it is normally *this* (C# for "this class instance").

The argument that is provided should be clearly typed, not abstracted to—object, EventArgs—as in the standard pattern for .NET. Generics are not available with the .NET Micro Framework, so using clearly typed parameters avoids confusion. The user is not required to cast the parameters to the correct types.

The following, more complicated example has two events, ButtonPressed and ButtonReleased, which share the same event-raiser method. Both events pass a second argument, which is the actual button state. This seems redundant because handlers for the ButtonPressed event could be expected not to need the ButtonState parameter, which will always be *Pressed*. However, this argument enables the user to use the same event handler for two different events. In general, include the parameters because they might be useful. Always pass the module instance itself as the first parameter, as shown in the following example:

```
/// <summary>
/// Represents the state of the <see cref="Button"/> object.
/// </summary>
public enum ButtonState
{
   /// <summary>
   /// The button is released.
   /// </summary>
   Released = 0,
   /// <summary>
   /// The button is pressed.
   /// </summary>
   Pressed = 1
}

/// <summary>
/// Represents the delegate that is used for the <see cref="ButtonPressed"/>
/// and <see cref="ButtonReleased"/> events.
/// </summary>
/// <param name="sender">The <see cref="Button"/> object that raised the event.</param>
/// <param name="state">The state of the button</param>
public delegate void ButtonEventHandler(Button sender, ButtonState state);

/// <summary>
/// Raised when the <see cref="Button"/> is pressed.
/// </summary>
/// <remarks>
/// Handle this event and the <see cref="ButtonReleased"/> event
/// when you want to provide an action associated with button activity.
/// Since the state of the button is passed to the <see cref="ButtonEventHandler"/> delegate,
/// you can use the same event handler for both button states.
/// </remarks>
public event ButtonEventHandler ButtonPressed;

/// <summary>
/// Raised when the <see cref="Button"/> is released.
/// </summary>
/// <remarks>
/// Handle this event and the <see cref="ButtonPressed"/> event
/// when you want to provide an action associated with button activity.
/// Since the state of the button is passed to the <see cref="ButtonEventHandler"/> delegate,
/// you can use the same event handler for both button states.
/// </remarks>
public event ButtonEventHandler ButtonReleased;
private ButtonEventHandler onButton;

/// <summary>
/// Raises the <see cref="ButtonPressed"/> or <see cref="ButtonReleased"/> event.
```

```
/// </summary>
/// <param name="sender">The <see cref="Button"/> that raised the event.</param>
/// <param name="buttonState">The state of the button.</param>
protected virtual void OnButtonEvent(Button sender, ButtonState buttonState)
{
  if (this.onButton == null)
  {
    this.onButton = new ButtonEventHandler(this.OnButtonEvent);
  }

  if (Program.CheckAndInvoke((buttonState == ButtonState.Pressed ? this.ButtonPressed :
this.ButtonReleased), this.onButton, sender, buttonState))
  {
    switch (buttonState)
    {
      case ButtonState.Pressed:
        this.ButtonPressed(sender, buttonState);
        break;
      case ButtonState.Released:
        this.ButtonReleased(sender, buttonState);
        break;
    }
  }
}
```

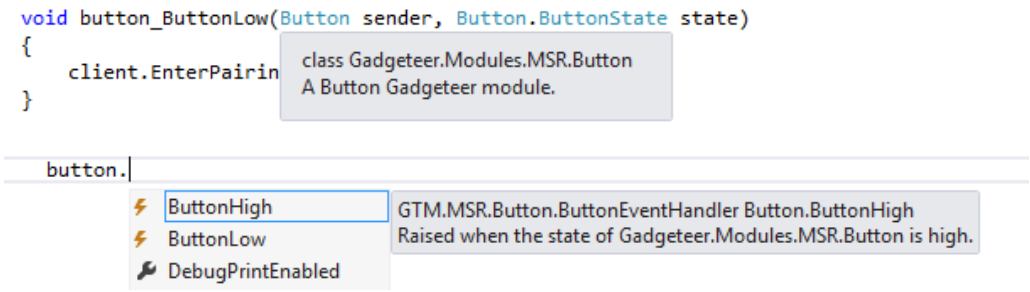
Modules should also provide synchronous access to any information that is available through event handlers. For example, the current button state should be available through a *get* property, such as *IsPressed*. This lets users choose their preferred programming pattern.

Summary

- Provide synchronous properties as well as asynchronous events to allow users' code to be written either way.
- Use the standard event pattern examples to define events and use them from within the rest of your code with OnXXXEvent (this, secondParameter).

Documentation

As shown in the previous examples, be sure to document interfaces by using triple slash comments: ///. This facilitates display in the user's tooltips and IntelliSense, as shown in the following illustrations.



To make Visual Studio generate the documentation as an XML file accompanying the DLL, you must select the **XML documentation File** check box under **Project** -> **Properties** -> **Build** tab. This option is preset by the Module Template.

For more details on this feature, see "How to: Use the XML Documentation Features (C# Programming Guide)" on the Microsoft Developer Network (MSDN®) website.

Summary

- Enable IntelliSense by using triple slash comments: ///.
- Select **XML documentation File** in the project **Properties** check box (or use the Module Template, which does this for you).

Designer F1 Help

The .NET Gadgeteer Designer includes an XSL transform that creates html reference pages for F1 help. The Designer displays these reference pages when the user clicks on a module or mainboard in the Designer and pushes the F1 key.

Both Intellisense and .NET Gadgeteer Designer F1 help originate in triple slash comments /// in C# source files. The .NET Gadgeteer Designer F1 reference pages include remarks and code samples that Intellisense does not display.

To use this feature, include triple slash nodes for <remarks/> and/or <example/> and <code/> in the source code for your module or mainboard. You can use only remarks or both remarks and code examples.

The following example shows triple slash comments that include remarks and example code for a camera module. Intellisense will display only the <summary/> node, but the .NET Gadgeteer Designer contains embedded XSL that transforms the <remarks/>, <example/>, and <code/> nodes into html format for display by F1 help.

```
    /// <summary>
    /// Represents a camera module that you can use to capture images.
    /// </summary>
    /// <remarks>
    /// After you create a <see cref="Camera"/> object, the camera
initializes asynchronously;
    /// it takes approximately 400-600 milliseconds before the <see
cref="CameraReady"/> property returns <b>true</b>.
    /// </remarks>
    /// <example>
    /// <para>The following example initializes a <see cref="Camera"/>
object and the button pressed event delegate in which the camera takes a
picture.
    /// Another delegete is initialized to handle the asynchronous
    /// PictureCaptured event.  In this method the display module uses
    /// the SimpleGraphics class to display
    /// the picture captured by the camera.
    /// <code>
    ///         void ProgramStarted()
    ///         {
    ///             // Initialize event handlers here.
    ///             button.ButtonPressed += new
Button.ButtonEventHandler(button_ButtonPressed);
    ///             camera.PictureCaptured +=
```

```
    ///                             new
Camera.PictureCapturedEventHandler(camera_PictureCaptured);
    ///
    ///             // Do one-time tasks here
    ///             Debug.Print("Program Started");
    ///         }
    ///
    ///         void camera_PictureCaptured(Camera sender, GT.Picture
picture)
    ///         {
    ///             Debug.Print("Picture Captured event.");
    ///             display.SimpleGraphics.DisplayImage(picture, 5, 5);
    ///         }
    ///
    ///         void button_ButtonPressed(Button sender,
Button.ButtonState state)
    ///         {
    ///             camera.TakePicture();
    ///         }
    ///     }
    /// }
    ///
    /// </code>
    /// </example>
```

HelpUrl Attribute of Module Template

If you supply triple slash comments in your module or mainboard source file, F1 help will support a link to addional support information for your module or mainboard. This link is displayed along with F1 help based on the /// comments in source code, so it will not work if F1 help is not in use. To supply the link along with F1 help, assign the HelpUrl attribute of the Module Software Template described in the following section. The HelpUrl attribute is contained by the <ModuleDefinition/> node of the GadgeteerHardware.xml file that the builder templates provide.

The following excerpt from the GadgeteerHardware.xml file shows the HelpUrl attribute.

```
<?xml version="1.0" encoding="utf-8" ?>
<GadgeteerDefinitions
xmlns="http://schemas.microsoft.com/Gadgeteer/2011/Hardware">
  <ModuleDefinitions>
    <ModuleDefinition Name="TestDocModule"
                  UniqueId="871bfa98-5338-4f5c-ab15-6eac97b1b4ee"
                  Manufacturer="Microsoft"
                  Description="A TestDocModule module"
                  InstanceName="TestDocModule"
                  Type="Gadgeteer.Modules.ManufacturerName.TestDocModule"
                  ModuleSuppliesPower="false"
                  HardwareVersion="1.0"
                  Image="Resources\Image.jpg"
                  BoardHeight="22"
                  BoardWidth="44"
                  HelpUrl="http://MicrosoftResearch.com/sample.htm"
                  MinimumGadgeteerCoreVersion="2.41.100"
```

>

Module Software Template

The module template installed with the Gadgeteer core libraries provides an easy way to build software for a hardware module that is compliant with the specifications in this document. The template generates an installer (MSI) that can be distributed to end users. It also generates an installation-merge module (MSM) that can be used to make kit installers that include many modules, mainboards, templates, and other components that might be needed by a development kit.

This version of the Module Builder's Guide is specific to Gadgeteer 2.42.700 and later. The most up-to-date information about building .NET Gadgeteer modules is included in the ReadMe file included with the module template project. Version 2.42.700 of the core libraries support mainboards with built-in modules and modules that only work on particular mainboards. Version 2.42.700 fragments some elements of the core, such as the Networking, Serial, and DisplayModule classes. Refer to the ReadMe.txt for information pertinent to each version.

Both Visual Studio and Visual C# Express support the template. To build the installer MSI automatically, this template requires WiX Toolset 3.5 or newer to be installed. The WiX35 Toolset is available at http://wixtoolset.org/ .

Important: Be sure to install WiX before you create a module template project.

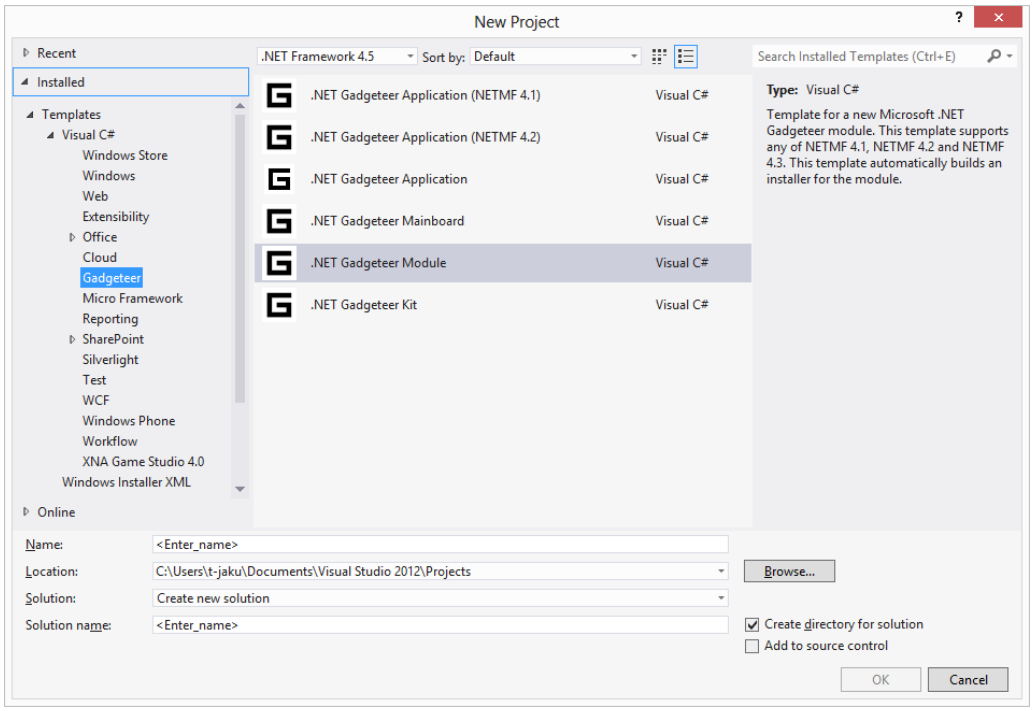
The Readme file installed with the template contains updates of the instructions in this section. The templates currently support modules built on either or both 2.42 and 2.42 assemblies. The 4.2 release of the .NET Micro Framework facilitates this upgrade to the .NET Gadgeteer core libraries, but there are numerous 2.41 modules in production. .NET Gadgeteer developers can implement drivers for either 2.41 or 2.42 assemblies or for both versions of the assemblies.

Visual Studio Module Template Project

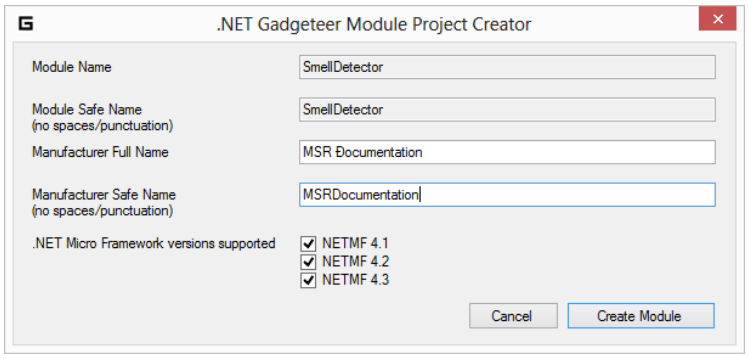
You can create a Visual Studio project that includes the module software template by following these steps.

To create a Visual Studio project that includes the module software template

1. Select **New Project** from the File menu.
2. Select **Gadgeteer** from the Visual C# installed templates.
3. Select **.NET Gadgeteer – Module Template** from the options visible, as shown in the following illustration.
4. Name the project, and click OK.



5. The wizard that opens provides textboxes for **Manufacturer Full Name** and **Manufacturer Safe Name**. It also allows you to specify whether your module targets version 4.1 or 4.2 of the .NET Micro Framework, or both versions.

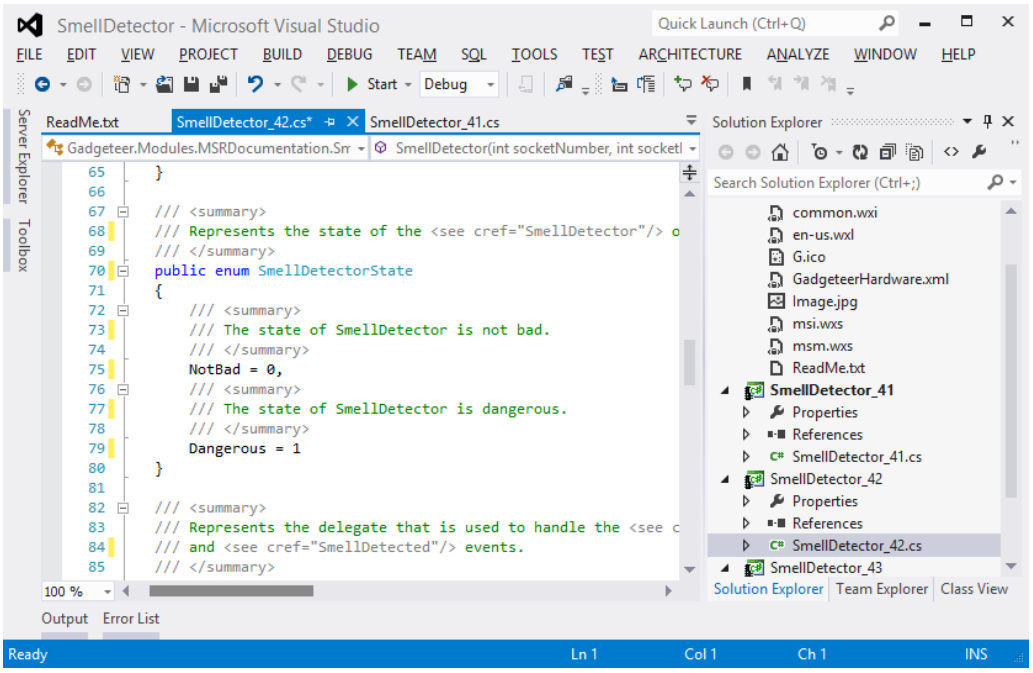


This will replace matches in: [ModuleName].cs, common.wxi, GadgeteerHardware.xml, and AssemblyInfo.cs. This search and replace does not change the **Assembly name** or **Default namespace**, so the previous step is necessary.

Implementing the Module Driver

You do not have to write assembly-language code or do low-level programming to implement the software driver for a Gadgeteer module. The module template creates a file named [ModuleName].cs. This file uses the C# programming language and the features of managed code.

There are comments and examples in the [ModuleName].cs file to assist you with the implementation of the software for your module. The following illustration shows a view of the [ModuleName].cs file in which a hypothetical SmellDetector module is implemented.



The example code that is provided by the SmellDetector.cs files implements a driver in managed code for the module. The module uses a single GTI.InterruptInput to interact with a sensor, which can be in either of two states: low or high. The template example code includes the recommended code pattern for exposing the property (IsHigh). The example also uses the recommended code pattern for exposing two events: **SmellDetectorLow**, **SmellDetectorHigh**.

The triple-slash "///" comments shown in the following code segment are used by the build process to create an XML file named GTM.MSR.SmellDetector.xm that supports IntelliSense and documention for the software interface.

The following code shows the constructor for the hypothetical SmellDetector module.

```
/// <summary>
/// A SmellDetector module for Microsoft .NET Gadgeteer
/// </summary>
public class SmellDetector : GTM.Module
{
    /*  This example implements  a driver in managed code for a simple Gadgeteer module.  The module uses a single GTI.InterruptInput to interact with a sensor that can be in either of two states: low or high.
     The example code shows the recommended code pattern for exposing the property (IsHigh).
    The example also uses the recommended code pattern for exposing two events: SmellDetectorHigh, SmellDetectorLow.
    // The triple-slash "///" comments shown will be used in the build process to create an XML file named
    // GTM.MSRDocumentation.SmellDetector. This file will provide IntelliSense and documentation for the
    // interface and make it easier for developers to use the SmellDetector module.

    // -- CHANGE FOR MICRO FRAMEWORK 4.2 --
    // If you want to use Serial, SPI, or DaisyLink (which includes GTI.SoftwareI2C), you must do a few more steps
```

```
    // since these have been moved to separate assemblies for NETMF 4.2 (to reduce the minimum
memory footprint of Gadgeteer)
    // 1) add a reference to the assembly (named Gadgeteer.[interfacename])
    // 2) in GadgeteerHardware.xml, uncomment the lines under <Assemblies> so that end user apps
using this module also add a reference. */

    // Note: A constructor summary is auto-generated by the doc builder.
    /// <summary></summary>
    /// <param name="socketNumber">The socket that this module is plugged in to.</param>
    /// <param name="socketNumberTwo">The second socket that this module is plugged in
to.</param>
    public SmellDetector(int socketNumber, int socketNumberTwo)
    {
      // This finds the Socket instance from the user-specified socket number.
      // This will generate user-friendly error messages if the socket is invalid.
      // If there is more than one socket on this module, then instead of "null" for the last parameter,
      // put text that identifies the socket to the user (e.g. "S" if there is a socket type S)
      Socket socket = Socket.GetSocket(socketNumber, true, this, null);

      // This creates an GTI.InterruptInput interface. The interfaces under the GTI namespace to provide
easy ways to build common modules.
      // This also generates user-friendly error messages automatically, e.g. if the user chooses a socket
incompatible with an interrupt input.
      this.input = new GTI.InterruptInput(socket, GT.Socket.Pin.Three, GTI.GlitchFilterMode.On,
GTI.ResistorMode.PullUp, GTI.InterruptMode.RisingAndFallingEdge, this);

      // This registers a handler for the interrupt event of the interrupt input, which is below.
      this.input.Interrupt += new GTI. InterruptInput.InterruptEventHandlerr(this._input_Interrupt);
    }
```

The following code segment implements the input interrupt and a property that indicates the state of the module.

```
    private void _input_Interrupt(GTI.InterruptInput input, bool value)
    {
      this.OnSmellDetectorEvent(this, value ? SmellDetectorState.Low : SmellDetectorState.High);
    }

    private GTI.InterruptInput input;

    /// <summary>
    /// Gets a value that indicates whether the state of this SmellDetector is high.
    /// </summary>
    public bool IsHigh
    {
      get
      {
        return this.input.Read();
      }
    }

    /// <summary>
    /// Represents the state of the <see cref="SmellDetector"/> object.
    /// </summary>
    public enum SmellDetectorState
    {
      /// <summary>
      /// The state of SmellDetector is low.
```

```
        /// </summary>
        Low = 0,
        /// <summary>
        /// The state of SmellDetector is high.
        /// </summary>
        High = 1
    }
```

The remainder of the template code implements a delegate that is used to handle the events that are raised when the state of the module changes.

```
    /// <summary>
    /// Represents the delegate that is used to handle the <see cref="SmellDetectorHigh"/>
    /// and <see cref="SmellDetectorLow"/> events.
    /// </summary>
    /// <param name="sender">The <see cref="SmellDetector"/> object that raised the event.</param>
    /// <param name="state">The state of the SmellDetector</param>
    public delegate void SmellDetectorEventHandler(SmellDetector sender, SmellDetectorState state);

    /// <summary>
    /// Raised when the state of <see cref="SmellDetector"/> is high.
    /// </summary>
    /// <remarks>
    /// Implement this event handler and the <see cref="SmellDetectorLow"/> event handler
    /// when you want to provide an action associated with SmellDetector activity.
    /// The state of the SmellDetector is passed to the <see cref="SmellDetectorEventHandler"/>
delegate,
    /// so you can use the same event handler for both SmellDetector states.
    /// </remarks>
    public event SmellDetectorEventHandler SmellDetectorHigh;

    /// <summary>
    /// Raised when the state of <see cref="SmellDetector"/> is low.
    /// </summary>
    /// <remarks>
    /// Implement this event handler and the <see cref="SmellDetectorHigh"/> event handler
    /// when you want to provide an action associated with SmellDetector activity.
    /// Since the state of the SmellDetector is passed to the <see cref="SmellDetectorEventHandler"/>
delegate,
    /// you can use the same event handler for both SmellDetector states.
    /// </remarks>
    public event SmellDetectorEventHandler SmellDetectorLow;

    private SmellDetectorEventHandler onSmellDetector;

    /// <summary>
    /// Raises the <see cref="SmellDetectorHigh"/> or <see cref="SmellDetectorLow"/> event.
    /// </summary>
    /// <param name="sender">The <see cref="SmellDetector"/> that raised the event.</param>
    /// <param name="SmellDetectorState">The state of the SmellDetector.</param>
    protected virtual void OnSmellDetectorEvent(SmellDetector sender, SmellDetectorState
SmellDetectorState)
    {
        if (this.onSmellDetector == null)
        {
            this.onSmellDetector = new SmellDetectorEventHandler(this.OnSmellDetectorEvent);
        }
```

```
        if (SmellDetectorState == SmellDetector.SmellDetectorState.High)
        {
          // Program.CheckAndInvoke helps event callers get onto the Dispatcher thread.
          // If the event is null then it returns false.
          // If it is called while not on the Dispatcher thread, it returns false but also
          // re-invokes this method on the Dispatcher.
          // If on the thread, it returns true so that the caller can execute the event.
          if (Program.CheckAndInvoke(SmellDetectorHigh, this.onSmellDetector, sender,
SmellDetectorState))
          {
            this.SmellDetectorHigh(sender, SmellDetectorState);
          }
        }
        else
        {
          if (Program.CheckAndInvoke(SmellDetectorLow, this.onSmellDetector, sender,
SmellDetectorState))
          {
            this.SmellDetectorLow(sender, SmellDetectorState);
          }
        }
      }
```

Building the project in the **Debug** configuration creates the module class library (dll). The **Debug** configuration does not build the installer .msi or .msm files. To build the installers, change the configuration to **Release** mode.

The build process takes longer in **Release** mode, so using the **Debug** configuration until the implementation is complete and tested will save time. If you see the error "The system cannot find the file..." try **Rebuild** rather than **Build**.

Testing the Module Software

Modules cannot be run directly because they are class libraries (.dll) not executables (.exe). Testing is most easily accomplished by adding a new Gadgeteer project to the same Visual Studio or Visual C# Express solution in which the module driver software is implemented.

To add a new Gadgeteer project and reference the module library

1. Right click on the module Solution in **Solution Explorer**.
2. Click Add->New Project.
3. Select Gadgeteer – Blank Template.
4. If you have a mainboard, it will appear on the designer surface, but the new module has not been installed yet, so it will not be available.
5. In the Solution Explorer, right click the test project project.
6. Click **Add Reference**.
7. Browse to the DLL for your module and add a reference to it.
8. Add a using directive for the library in the Program.cs file.

9. In the test project Program.cs file, instantiate the module as shown in the following example.

```
// Define and initialize GTM.Modules here, specifying their socket numbers.
GTM.MSR.SmellDetector smellDetector = new GTM.MSR.SmellDetector(4);
```

10. Add a handler for one of the events that your module implements.

The following code segment shows initialization of a delegate that handles the event that is raised when the state of the SmellDetector module changes to high.

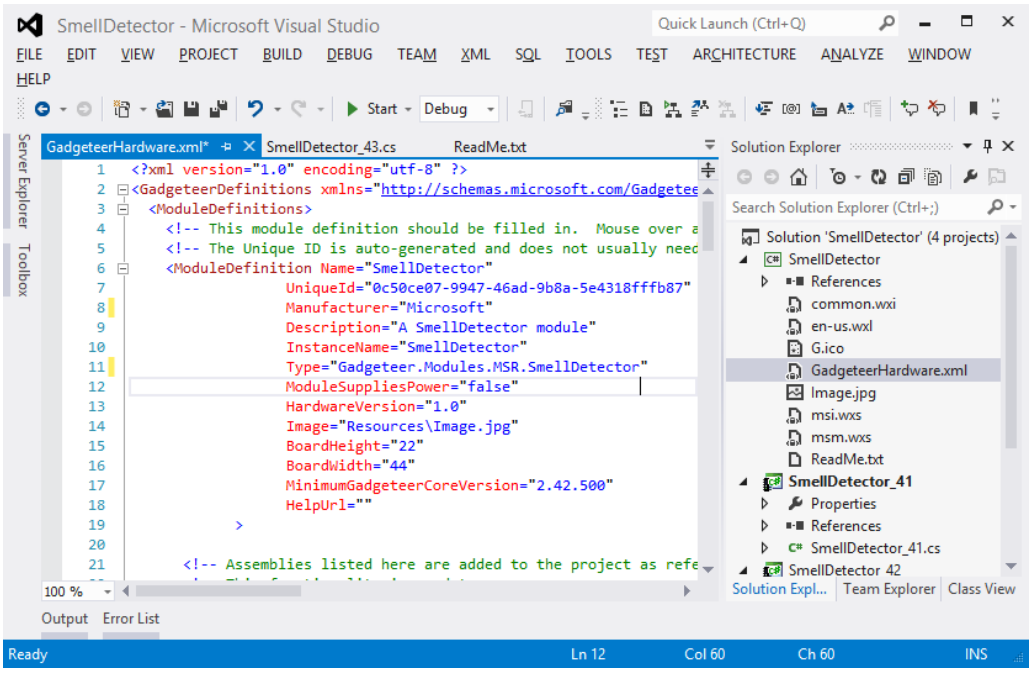
```
void ProgramStarted()
{
  // Initialize event handlers here.
  // e.g. button.ButtonPressed += new Button.ButtonEventHandler(button_ButtonPressed);

  smellDetector.SmellDetectorHigh +=
   new GTM.MSR.SmellDetector.SmellDetectorEventHandler(smellDetector_SmellDetectorHigh);

  // Do one-time tasks here
  Debug.Print("Program Started");
}

void smellDetector_SmellDetectorHigh(GTM.MSR.SmellDetector sender,
                                     GTM.MSR.SmellDetector.SmellDetectorState state)
{
  Debug.Print("Smell detector alarm state.");
}
```

Building the MSI and MSM Installers

When the module implementation is complete, you are ready to build the installers for the module software. The template supports .msi and .msm installers. The output of the template includes both an .msi installer for the single module and and .msm installer that can be used for kit installers that incorporate one or more modules and mainboards.

The GadgeteerHardware.xml file specifies information about your module's assembly name and pin mappings. You should edit this file as needed beyond the changes to the manufacturer, description, and fully qualified assembly name. IntelliSense information is available as you mouse over each element of the xml document. For more information about pin and socket types, see the heading **Connecting Modules to Mainboards** heading in this document.

The following illustration shows the GadgeteerHardware.xml file in Visual Studio.



In version 2.42 of the .NET Gadgeteer core libraries, some components are in separate assemblies to make it possible to exclude them when they are not required. This saves RAM and flash space when deployed. The designer can automatically include such core "fragments" when including a modulethat relies on them or is commonly used with them. The fragment libraries should be listed in the Assemblies section of the module alongside that module's driver, so that the designer will automatically include a reference to them when that module is included in a project.

The fragments are:

- Gadgeteer.SPI
- Gadgeteer.Serial
- Gadgeteer.DaisyLink (inc. SoftwareI2C)
- Gadgeteer.WebClient and Gadgeteer.Webserver

If you use any of these in your module, then it will NOT work in 4.2 unless you add an Assemblies entry in GadgeteerHardware.xml.

The **Resources** folder that the .Net Gadgeteer Module Template includes contains a blank Image.jpg file. Change the Resources\Image.jpg file to a **good** quality top-down image of the module with the socket side facing up. The image should be cropped tight (no margin), in the same orientation (not rotated) as the width and height specified in GadgeteerHardware.xml.

Finally, edit the Setup\common.wxi file to specify parameters for the installer. The parameters are described in the .wxi file. You don't need to edit any other file in the **Setup** directory for the first build.

Change the build configuration to **Release**, and build the module project. When the build completes, you should find the .msi and .msm files in the \bin\Release\Installer directory. The example used for illustrations in this example builds files named SmellDetector.msi and SmellDetector.msm.

Making Changes to Module Software

If you want to release a new version of your module, make sure to change the version number in Setup\`common.wxi`. Otherwise, the auto-generated installer will not be able to upgrade the older version correctly, and an error message will result. It is also necessary to change the versions in Properties\`AssemblyInfo.cs`, else the existence of the newer assembly will not be picked up by Visual Studio.

If you want to change the name of your module, be sure to search all the files for instances of the name. Per the Module Builder's Guide, the software module name should match the name printed on the module itself. The ManufacturerName should match the manufacturer name printed on the module (remove any spaces/punctuation).

Comments in `AssemblyInfo.cs` describe the version numbers as shown below.

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: AssemblyTitle("GTM.MSRDocumentation.SmellDetector")]
[assembly: AssemblyDescription("Driver for SmellDetector module made by MSR
for Microsoft .NET Gadgeteer")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Microsoft")]
[assembly: AssemblyProduct("SmellDetector")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// This is the assembly version.  It is often useful to keep this constant
between major releases where the API changes
// since a dll relying on this dll will need to be recompiled if this
changes.
// Suggestion: use a version number X.Y.0.0 where X and Y indicate major
version numbers.
[assembly: AssemblyVersion("1.0.0.0")]

// These numbers must be changed whenever a new version of this dll is
released, to allow upgrades to proceed correctly.
// Suggestion: Use a version number X.Y.Z.0 where X.Y.Z is the same as the
installer version found in Setup\common.wxi
[assembly: AssemblyFileVersion("1.1.0.0")]
[assembly: AssemblyInformationalVersion("1.1.0.0")]

// Alternatively, if this module is included as part of a kit installer and
you want to synchronize version numbers to the kit's, follow the instructions
in the kit's readme to include the kit's AssemblyInfoGlobal.cs version
numbers, and comment out the two version numbers above (AssemblyFileVersion
and AssemblyInformationalVersion BUT NOT AssemblyVersion)
```

Similar comments in the `common.wxi` describe how to change the version.

```
  <!-- Change this whenever releasing a new kit installer.  The fourth number
is ignored, so change one of the top three.
  Otherwise, users will not be able to upgrade properly; Windows Installer
will exit with an error instead of upgrading. -->
  <?define AppVersion = "1.1.0.0" ?>
```

IF there is only one other assembly referring to a given DLL, for example an end user's Program.cs file, then the **AssemblyVersion** can change, because there is no possibility of conflicting references. So for **Gadgeteer** modules, if the modules are only directly used in end user code, then the modules can change **AssemblyVersion**, and you can keep these synchronized with the **AssemblyFileVersion**.

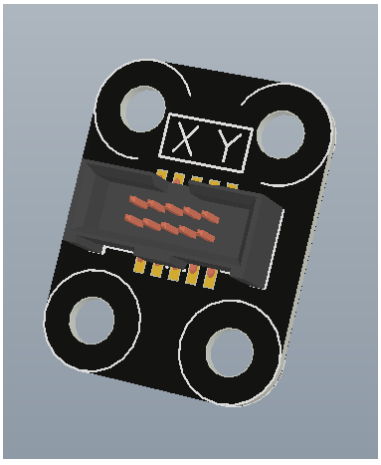
IF, however, you expect an assembly might be used to make a library class that is then passed on to other users in BINARY form, which is what we face with Gadgeteer.dll, then it may be better to keep the **AssemblyVersion** the same, as long as you don't remove elements from the API.

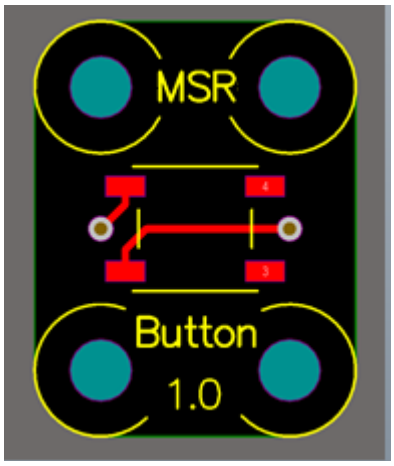
Mechanical Design

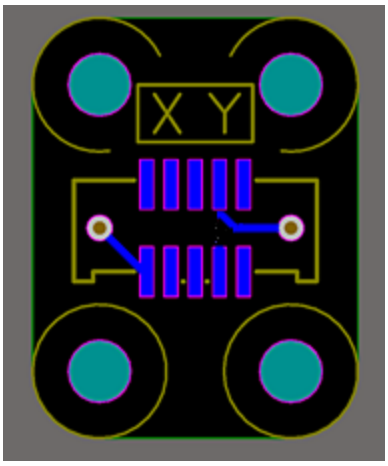
The following guidelines specify best practices in mechanical design of Gadgeteer Modules.

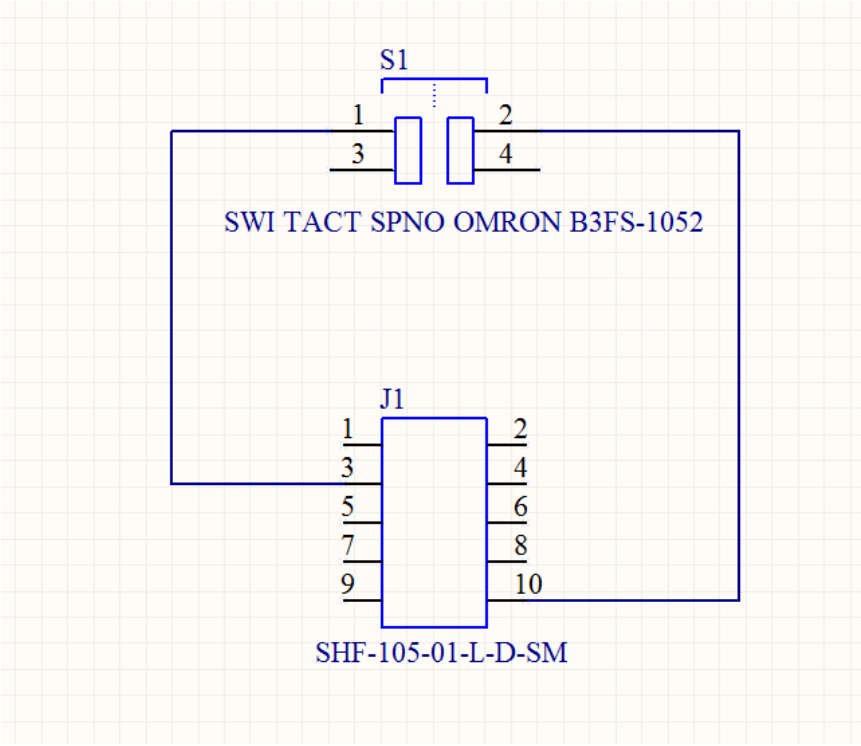
PCB Layout and Silkscreen Guidelines

The following PCB design is a sample Gadgeteer module that illustrates the guidelines in this document. The example and schematic show a simple push-button module that is named Button, which is designed by a company with the initials MSRC and is version 1.0 of the design. The module is compatible with Socket Types X and Y. Version numbers are unnecessary in the software class name because the same software driver might work across multiple hardware revisions. Conversely, if the hardware changes so significantly that the software is incompatible, we recommend that you use a different module name, such as ButtonLarge.





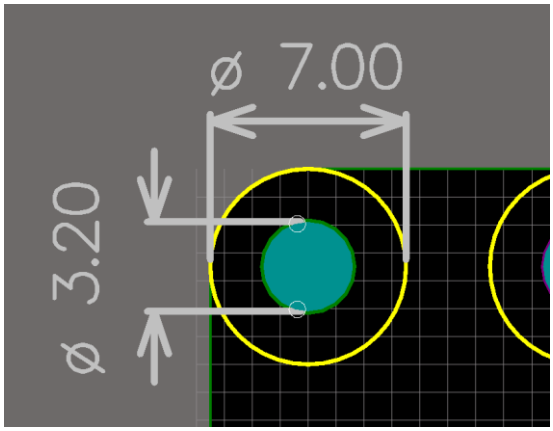




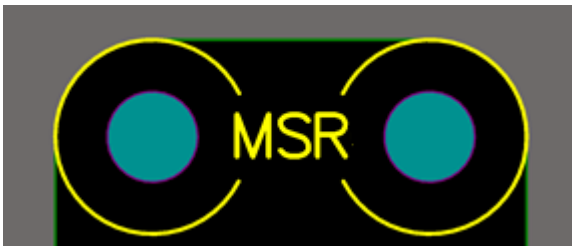
Mounting Holes

Mounting holes should be placed at the corners of the board. A design should have at least two mounting holes, but we recommend more holes if the module is subject to mechanical stress during use.

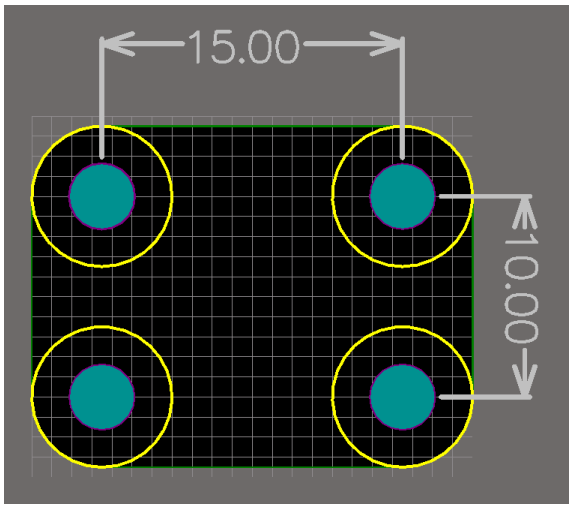
The mounting holes should have a 3.2-mm diameter, with a 7-mm-diameter component keep-out area around them, as shown in the following illustration.



The keep-out area should be clearly delimited in the silkscreen on both sides of the PCB, as shown in the following illustration. For small modules, where space is tight, it is possible to interrupt the keep-out delimiter silkscreen to make space for other labeling or silkscreen elements. Under no circumstances should you place components inside the keep-out area.

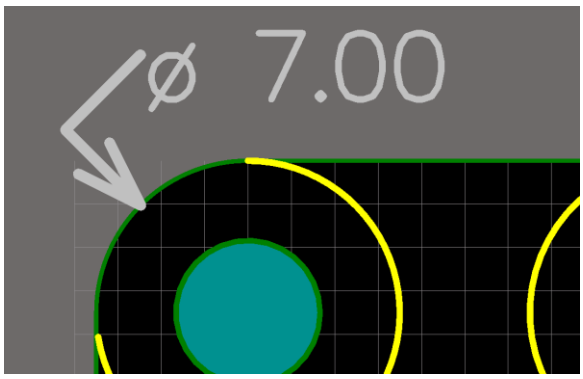


All mounting holes should be placed on a 5-mm grid, that is, the distance between adjacent holes should be a multiple of 5 mm, as shown in the following illustration.



Corners

Corners should be rounded, with a 7-mm-diameter curve that is concentric with a mounting hole's keep out area, as shown in the following illustration.



If a corner does not include a mounting hole, the corner does not need to be rounded. However, we recommend that you maintain the same 7-mm rounding diameter for consistency.

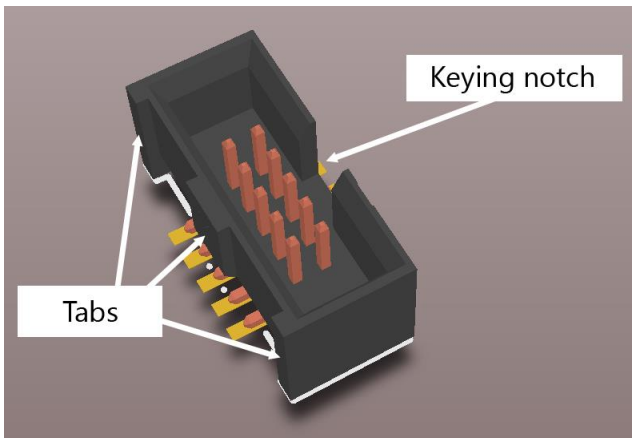
Connectors

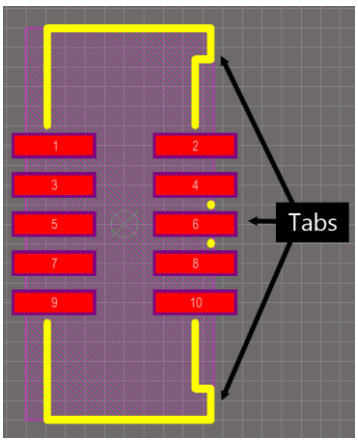
All Gadgeteer modules are equipped with at least one of standard 10-pin, 1.27-mm pitch keyed headers, which this document refers to simply as the connector. The connector allows the module to connect to a Gadgeteer mainboard.

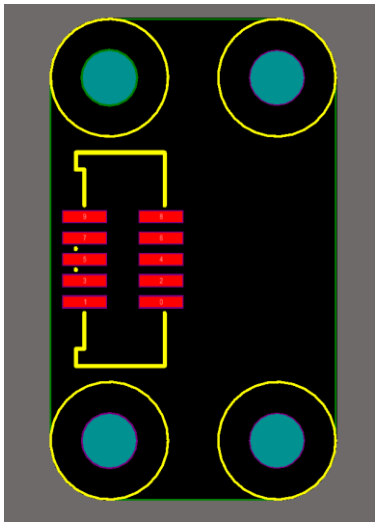
Connectors must be compatible with the Samtec part SHF-105-01-L-D-SM.

On mainboards, connectors must be oriented with their keying notch facing outward, toward the nearest outer edge of the board. On modules, connectors may be placed anywhere on the board, but when placed near an outer edge of the PCB, they should be placed with their notch facing inward. Mainboards and modules are different so that a cable can easily go from a mainboard to a module, with a socket near the edge of the mainboard.

Connectors should also comply with the silkscreen guidelines in "PCB Layout and Silkscreen Guidelines" earlier in this document. Connectors may have tabs as well as notches, as shown in the following illustration.







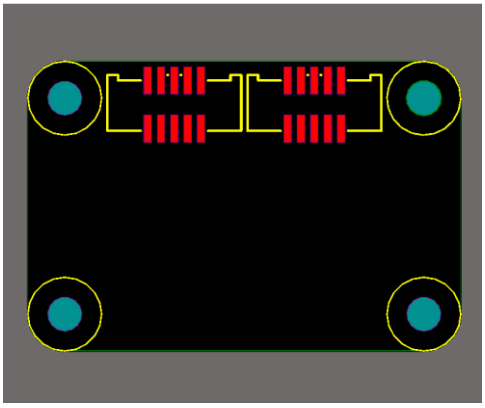
Multiple Connectors

A module might include more than one connector in the following two instances:

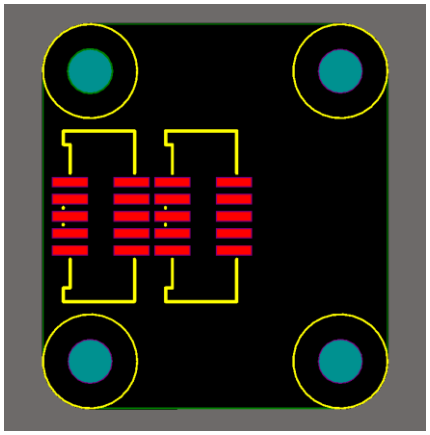
- The module needs to provide connection to more than one mainboard socket at a time.
- The module is compatible with the .NET Gadgeteer DaisyLink Protocol and can be daisy-chained with other modules.

Modules with Multiple Socket Connectors

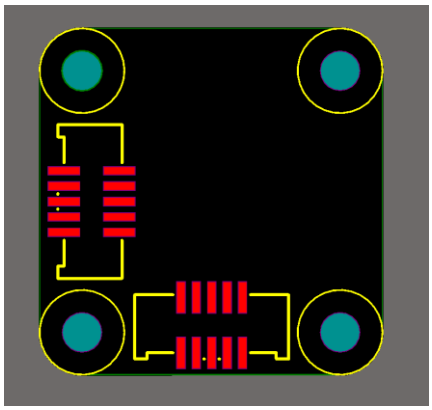
In this case, connectors can be placed anywhere on the board, following the guidelines under "Connectors" earlier in this document. If possible, connectors should be placed side by side near an edge of the board, as shown in the following illustration.



If the dimensions of the board prohibit a side-by-side arrangement of connectors, the next best option is to place a second connector immediately behind the first, in the same orientation, as shown in the following illustration.



The third least desirable option involves placing the two connectors on adjacent board edges. Again, it is important to observe the connector orientation guidelines. Tabs should face outward of the PCB, and keyed notches should face into the PCB, as shown in the following illustration.



These same guidelines extend to modules using more than two connectors.

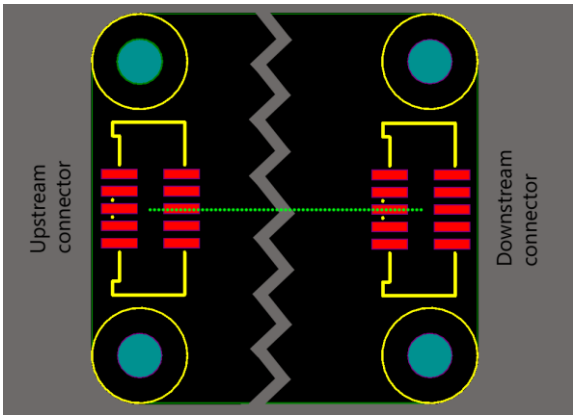
DaisyLink-Compatible Modules

A DaisyLink-compatible module needs two connectors: one that connects to the mainboard or to other modules upstream from it (the upstream connector) and a second that connects to modules downstream from it (the downstream connector).

The upstream connector should be placed like a regular connector and should follow the normal orientation guidelines: placed near an edge, tabs facing outward and keyed notch facing inward.

The downstream connector should be placed directly opposite the upstream connector next to the opposite edge of the PCB, center-aligned with the upstream connector and in the same orientation. Note that this orientation results in a reversal of the normal guideline: the tabs of the downstream connector facing into the PCB and its keyed notch facing outward toward the nearest edge. This orientation is shown in the following illustration.

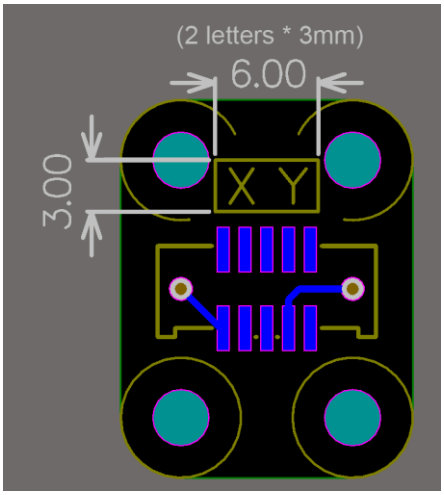
(For more information about the DaisyLink protocol specification see the heading DaisyLink Protocol, Version 4 earlier in this document and Appendix 1.)



Socket-Type Compatibility Labels

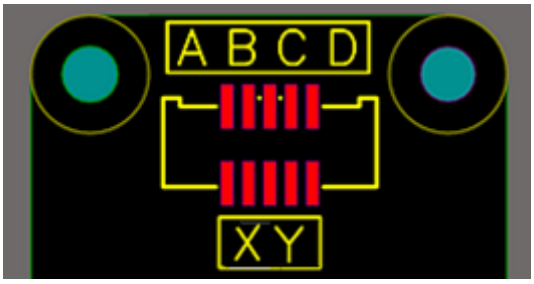
Connectors should be clearly labeled on the PCB's silkscreen with a series of letters that indicate that connector's socket-type compatibility. The label should be placed on the same side of the PCB as the connector and as close to the connector itself as possible.

The label should be surrounded by a bounding box, which is 3 mm high and (3 mm times the number of letters) wide. The bounding box for the label in the Button example, which consists of the two letters X and Y, is 3x6 mm. The recommended stroke thickness of the bounding box lines is 0.2 mm, as shown in the following illustration.



The bounding box provides a series of 3x3-mm spaces for each of the letters in the label. Each letter should be centered on this 3-mm grid. The recommended type is no serif, 2 mm high, and a 0.2-mm stroke width. Letters should be listed in alphabetical order.

If space constraints make it impossible to list all the necessary letters within a single bounding box, it is possible to provide two separate labels. As much as possible, letters should still read in alphabetical order between the two labels and both labels should be placed as close as possible to the connector, as shown in the following illustration.

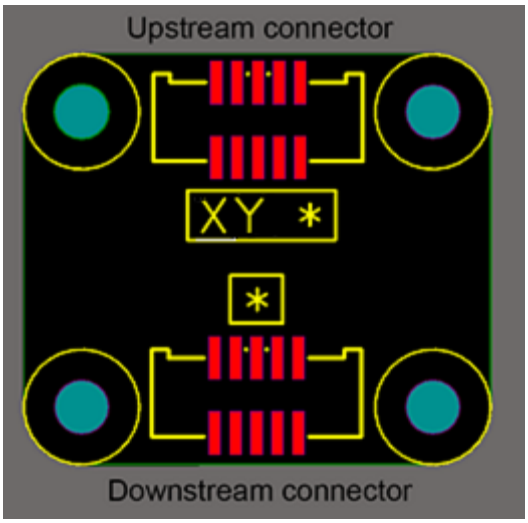


Additional Socket-Type Labeling Guidelines

Modules that are compatible with socket Types X or Y may also be implicitly compatible with a large number of additional socket types. To minimize the number of letters on a label, modules that are compatible with socket Type X should list X and Y on their label and no other socket types. Modules that are compatible with socket Type Y should list only Y on their label. Socket-compatibility labels are case sensitive.

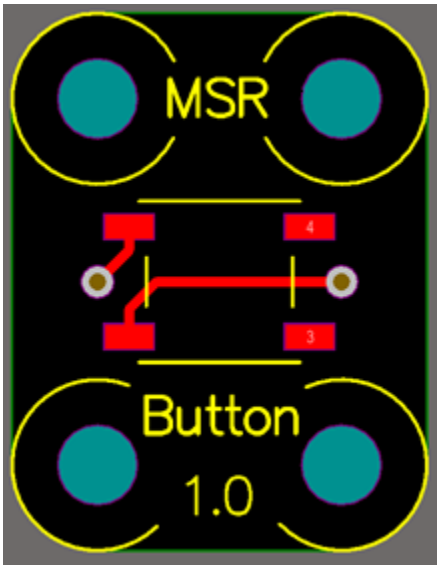
Socket-Type Compatibility Labels for DaisyLink Modules

The upstream connector in DaisyLink modules should include a label with the letters X Y and the asterisk (*) character. Socket-compatibility labels are case sensitive. The downstream connector should be labeled only with the asterisk (*) character, as shown in the following illustration.



Manufacturer, Module Name, and Version Number Labels

Modules should be clearly labeled with the manufacturer's or designer's name , an abbreviation of that name, or a URL clearly containing that name, e.g. www.ManufacturerName.com, as well as the module name. The minimum recommended type is no serif, 1.5 mm high, and a 0.2-mm stroke width. These labels should not be surrounded by a bounding box of any kind, as shown in the following illustration.



Solder-Resist and Silkscreen Colors

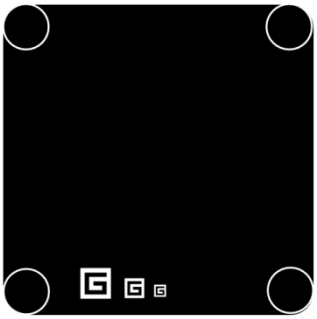
For most modules, the solder-resist color should ideally be black but can be a standard green. Vias should be tented to improve silkscreen readability. Silkscreen color should be white.

If the module acts as a power supply, for example, if it provides power to the mainboard when it connected to the board, the solder-resist color must be red.

Gadgeteer Graphic

The following illustration shows the suggested Gadgeteer identifier. We recommend that it be placed in a visible location further from any socket than the socket type letter so no confusion exists that the module is for a hypothetical socket G.

The identifier is shown in three sizes, but only one should be used on a mainboard or module in production.



We specify the following guidelines for use of the graphic:

- You may use the graphic for any purpose that is consistent with its Creative Commons license. We recommend that you print it on the PCBs of .NET Gadgeteer-compatible hardware. To view a copy of the license, visit the Creative Commons website.
- You may use terminology such as Gadgeteer-compatible "For .NET Gadgeteer" to describe mainboards or modules where appropriate.
- You may not use ".NET" or "Gadgeteer" in your product name.

Additional Recommendations

The following items will make components more usable and appealing to users but are not required:

- All zeros on silkscreens should have diagonal lines through them because "0" is similar to the letter "O" and could be misread.
- Boards should be small as possible while they follow the other recommendations.

JPEGs and CAD Models

When possible, Gadgeteer module designers should provide a high-resolution image of the module front and back, and also an accurate 3D CAD model of their module. Ideally, the CAD model should be generated directly from the PCB layout data. CAD diagrams can also be provided as STEP, IDF, or IGES files.

DesignSpark

DesignSpark is a free PCB design tool that includes the ability to view and export PCBs as 3D CAD models. It includes the ability to import Eagle CAD designs. To export a CAD model, go to **PCB View** and select the **Output** -> **IDF** check box.

Altium

Altium provides built-in support for viewing and exporting your PCB as a 3D model. Exported models are in STEP format.

To provide a basic but accurate model, components should have their **Height** property field populated. Altium can then generate a basic bounding cube based on the components footprint and height, which represents the volume that the component occupies.

To create more realistic models, it is also possible to associate STEP models of individual components with their PCB library entry. Many component and connector manufacturers provide 3D CAD models of their parts through their websites. Another good resource for user-generated CAD models of parts is the 3D ContentCentral website.

To be sure that the PCB thickness is accurate, specify the appropriate layer thickness in the **Design** -> **Layer Stack Manager** menu.

To export the 3D CAD model, on the **File** menu, click **Save As**, and then select **Export STEP** or **Export IDF** as the file type.

If export as STEP, be sure that the following options are selected in the **Export Options** menu:

- **Components with 3D Bodies: Export All**
- **3D Bodies Export Options: Prefer STEP Models**
- **Pad Holes: Export All**

Releasing Modules to Users

The current installation of Gadgeteer core libraries includes a Module Template that automatically packages modules by using the Visual Studio .msi extension file format. This tool provides information about the module—such as name, manufacturer name, socket type compatibility, or pins used—and the DLLs that form the software. For the details of building module installation packages, see the heading *Module Software Template* under *Module Code Libraries* in this document.

The .NET Gadgeteer website hosts a list of .NET Gadgeteer-compatible modules including specifications and links so that end users can purchase them. This provides Gadgeteer users with a single point of reference to find Gadgeteer modules that interest them.

Open Source Community Participation

The .NET Gadgeteer team is looking for ways to establish communication in the broader community of vendors and users that will help to ensure that the value proposition of the Gadgeteer ecosystem continues to grow and is not diminished through fragmentation. We plan to set up a "counsel" of vendors and users as a focal point for input on the direction of the technology. We would like your input about how best to engage the community and to make the most of this approach to building devices. If you have ideas on this, send an email to netgt@microsoft.com.

Resources

Additional information about the .NET Gadgeteer and the .NET Micro Framework can be found at the following sites:

.NET Gadgeteer
http://netmf.com/gadgeteer/

.NET Gadgeteer SDK
http://gadgeteer.codeplex.com/

.NET Micro Framework
http://netmf.com/

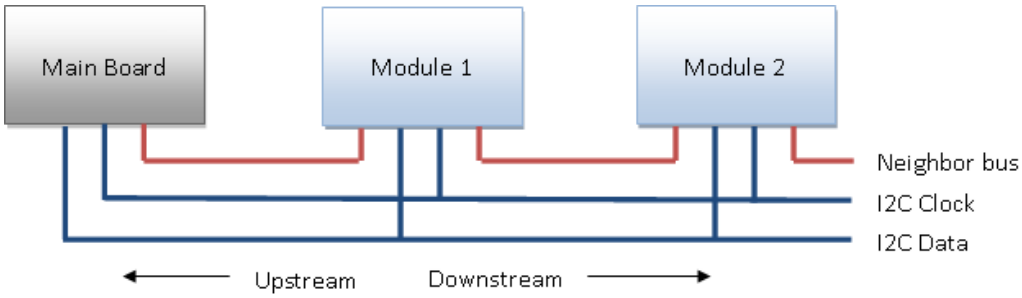
.NET Micro Framework SDK
http://netmf.codeplex.com/

How to: Use the XML Documentation Features (C# Programming Guide)
http://msdn.microsoft.com/en-us/library/z04awywx.aspx8

Appendix 1: DaisyLink Protocol Specification – Version 4

DaisyLink includes two buses: an I²C shared data bus for data transfer and a neighbor bus that is used for assigning the I²C ID of each module on the shared bus.

The following diagram shows a DaisyLink configuration of modules.



DaisyLink modules can be connected to any type X or Y sockets. The neighbor bus is connected to pin 3, the I²C data is connected to pin 4, and the I²C clock is connected to pin 5.

The following table shows the pin mapping specification and pull-up resistor requirements.

Socket pin	Function	Specification	Pull-ups
3	Neighbor Bus	This document	Every module pulls up both upstream and downstream buses with fixed 10,000 Ω resistors.
4	Shared Bus	I²C SDA	Every module has switchable 10,000 Ω pull-up resistors that are applied during Setup mode and released or enabled by command from the mainboard.
5	Shared Bus	I²C SCL	Every module has switchable 10,000 Ω pull-up resistors that are applied during Setup mode and released or enabled by command from the mainboard.

Every module must have switchable pull-ups for the I²C shared bus pins. The pull-ups must have the capability of being switched on or off during module initialization. When initialization is complete, regardless of how many modules are connected to the I²C bus, only one set of pull-ups is active and those pull-ups are on the last module in the chain.

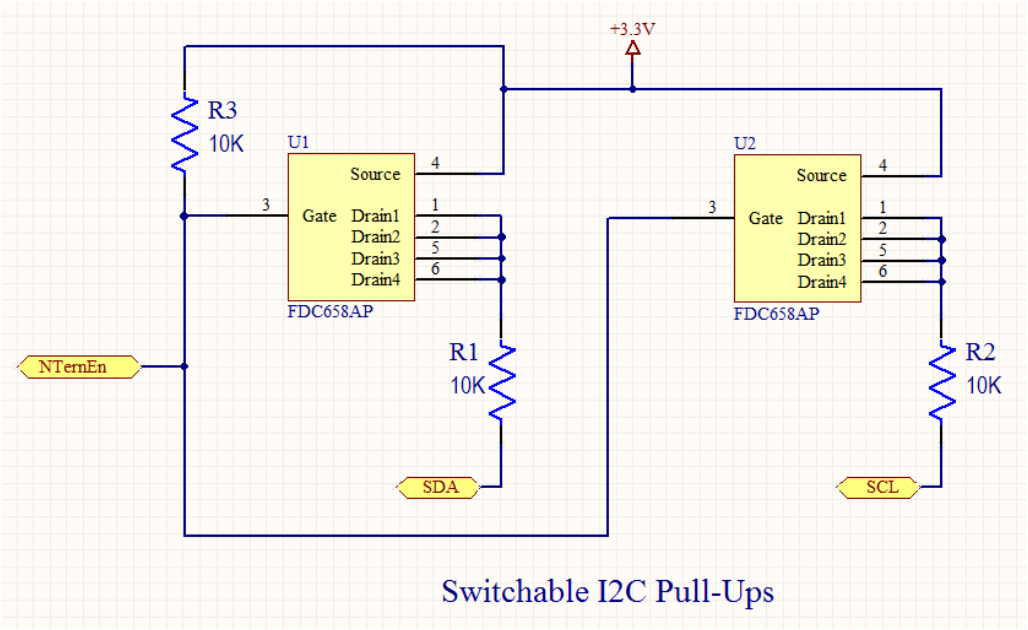
The module connects these pull-ups either by using discrete field-effect transistors (FETs) to switch them to the 3.3V supply or by tying them to a tristate-able processor pin and driving that pin high, assuming that the pin is capable of sourcing the required 0.5 milliamp.

The logic high state for the I²C and neighbor bus is 3.3V, and the logic low state is ground: 0V.

The following schematic shows the switchable I²C pull-up resistors: R1 on the data bus (SDA) and R2 on the clock bus (SCL). The schematic shows one way of driving the

pull-ups on the I²C bus, via FETs. "NTermEn" is the signal from the module's MCU that enables the pull-ups. In this case, pulling the NTermEn line low enables the pull-ups.

For information on the neighbor bus, see the following sections: "Neighbor Bus" and "DaisyLink Module Initialization."

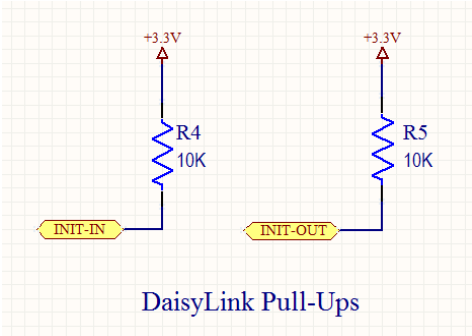


Switchable I2C Pull-Ups

Neighbor Bus

The neighbor bus is used during the initialization of the DaisyLink modules and whenever a module must interrupt—to get the attention of—the mainboard. The neighbor bus is composed of an upstream signal and a downstream signal. Both signals must be similar to the I²C pins in that they drive low only—open collector or open drain—and allow pull-up resistors to provide the logic high. The upstream neighbor bus signal is connected to the module's upstream neighbor or to the mainboard if the module is first in the chain. The downstream neighbor bus signal is connected to the module's downstream neighbor or unconnected if the module is the last in the chain.

The following schematic shows the fixed pull-up resistors on the neighbor bus lines. The INIT-IN signal is connected to the upstream side of the neighbor bus and the INIT-OUT signal is connected to the downstream side.



DaisyLink Pull-Ups

During normal operation, the neighbor bus is pulled to 3.3V by the pull-up resistors and goes to logic low during module initialization or when a module must communicate an interrupt condition to the mainboard.

During module initialization, the mainboard pulls a module's upstream neighbor bus signal low. Each module then passes this active low signal from its upstream bus to its downstream bus. During an interrupt condition, a module's downstream neighbor bus may be pulled low. In that case, each module passes this active low signal from its downstream bus to its upstream bus. When this behavior is implemented, the mainboard may assign each module its own unique I²C ID during initialization and any module in the chain may request the attention of the mainboard by pulling its upstream neighbor bus line low.

DaisyLink Module Initialization

During module initialization, the daisy-chain nature of the neighbor bus is used to assign a unique I²C ID to each module in the chain. In addition, during initialization, the mainboard tells each module whether to activate its pull-ups on the I²C bus lines.

Initialization is accomplished by the mainboard taking each module, in order, through four distinct states: Reset, Setup, Standby, and Active.

The following is an abbreviated outline of what happens during initialization:

1. The mainboard pulls a module's upstream neighbor bus low, which places the module in the Reset state. The module pulls its downstream bus low also so that all the downstream modules also enter a Reset state. In the Reset state, the module ignores signals on the I²C bus.
2. The module's upstream neighbor bus goes high, which places the module in the Setup mode. In Setup mode, the module continues to drive its downstream neighbor bus low so that downstream modules remain in Reset mode. During Setup mode, the module responds to a default I²C ID (127). The mainboard uses this default I²C ID to determine if the module exists and then sets the final unique I²C ID of the module.
3. When the mainboard sets the module's I²C ID, the module enters the Standby mode. In Standby mode, the module responds to its new unique I²C ID. It releases its upstream neighbor bus so that it goes high. This places the module's downstream neighbor (if one exists) in Setup mode.
4. The mainboard determines if the module's downstream neighbor exists and then either activates or deactivates the module's I²C pull-up resistors. This places the module in Active mode. Initialization is complete.

The preceding outline omits a few details, such as whether the I²C pull-up resistors are active or inactive during each state. All the modules cannot leave their I²C pull-up resistors off during initialization or the I²C bus would fail to work correctly. Neither can all modules have their resistors active or it would be impossible to drive the I²C bus signals to logic low.

The following table shows the module behavior during each of the states in the preceding outline.

State name	Upstream neighbor bus	Downstream neighbor bus	I²C pull-ups	I²C behavior
Reset	Driven low by mainboard	Driven low by module	Disabled	Ignores I²C bus.
Setup	Mainboard releases and is pulled high by module's pull-ups.	Driven low by module	Active	Lets all DaisyLink registers be read and the I²C ID register to be written to by using the default I²C ID (127).
Standby	Pulled high by module's resistors	Module releases and is pulled high by module's pull-up resistors	Active	Lets all DaisyLink registers be read and the Config register be written by using the module's new unique I²C ID.
Active	Pulled high by module's resistors	Pulled high by module's resistors	As set by the mainboard	Lets all DaisyLink registers be read and all device-specific functions be available by using the module's new unique I²C ID.

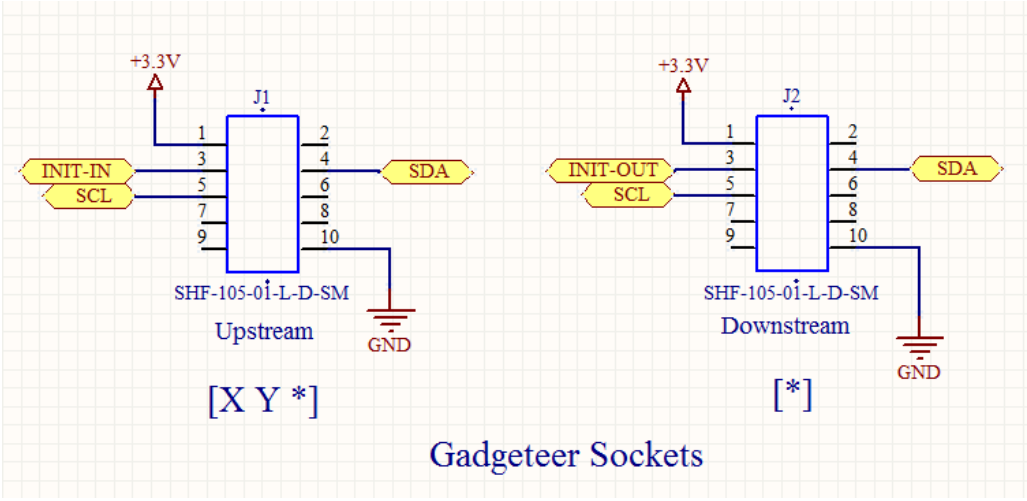
To provide a bit more detail, from the mainboard's perspective the initialization sequence is similar to the following outline:

1. The mainboard drives its neighbor bus line low to start initialization. It maintains this line low for at least 2 ms to confirm that the signal has propagated to all modules in the chain. After the 2 ms elapses, it releases this line and lets it float high with the pull-up resistors on the first module in the chain.
2. In response to this line going high, the first module in the chain turns on its I²C pull-ups and waits for its I²C ID to be set. At this point, the mainboard tries to read the DaisyLink version number from the first module—as described in "Memory Map Interface" earlier in this document—by using the default I²C ID of 127 (hexadecimal 7F). If the mainboard does not receive the correct version number (4), it assumes that no modules are connected to the socket and initialization is considered to be complete.
3. If the mainboard detects a connected module, it writes to the I²C ID register by using the default I²C ID of 127 (0x7F). This sets the unique I²C ID of the module. The module responds to this ID until power down or until module initialization recurs. When the module's I²C ID is set, the module lets its downstream neighbor bus be pulled high, which starts initialization of the next module in the chain.
4. At this point, if a second module exists in the chain, its I²C pull-up resistors are enabled in addition to the I²C pull-up resistors on the first module. The mainboard then tries to determine if the second module exists as it did with the first one, by trying to read the DaisyLink version number by using the default I²C ID. If the mainboard cannot read the version number, it performs a write to the DaisyLink configuration byte of the first module, which tells it to turn on its pull-ups. This enables the unique functionality of the first module.
5. If the mainboard finds a second module, it performs a write to the DaisyLink configuration byte of the first module, which tells it to turn off its I²C pull-ups.

This enables the device-specific functionality of the first module, and the only I²C pull-ups that are enabled are those on the second module in the chain. Initialization continues with the second module in the chain as it did with the first module in step 3. This pattern continues until all modules in the chain have been discovered and the only I²C pull-ups that have been enabled are on the last module in the chain.

At any time during or after initialization, the mainboard can decide to abort the process and begin again by bringing the module's upstream neighbor bus line low. If this happens, the module must return to the Reset state.

The following schematic shows the sockets on a module: one for the upstream connection and another for the downstream connection. The neighbor bus, which carries the initialization logic, connects to pin 3 in both cases (INIT-IN) and (INIT-OUT).



DaisyLink Module Interrupts

DaisyLink communication on the I²C bus is a master/subordinate configuration in which the mainboard is always the master and the DaisyLink modules are subordinates. This situation prevents modules from spontaneously sending messages to the mainboard. Some modules, however, may have to inform the mainboard when an operation is complete or a new measurement arrives.

DaisyLink modules are able to generate interrupts via the neighbor bus to signal their need for attention from the mainboard. At any time after module initialization is complete, a module may get the attention of the mainboard by driving its upstream neighbor bus low for a maximum of 1 mm. Back at the mainboard, this causes an event to be raised for the interrupting module.

Note that modules that are upstream of the interrupting module must return this active low signal to the mainboard. Because of this, all modules must implement this interrupt behavior even if they never interrupt the mainboard themselves.

So that interrupts for all modules work correctly, each module must watch its downstream neighbor bus line while it is in its Active state (as described earlier in this section). If the module's downstream neighbor bus line is driven low by its downstream neighbor, the module must also drive its upstream neighbor bus low.

When the downstream neighbor bus goes to logic high again, the module must release its upstream neighbor bus line also—unless it is itself driving its own 1-ms interrupt signal.

An interrupting module uses both the neighbor bus and the DaisyLink Config register (as described in "Memory Map Interface" later in this document). When a module wants to interrupt the mainboard, it first sets the most significant bit of its DaisyLink Config register high and then drives its upstream neighbor bus line low for no more than 1 ms.

This causes the mainboard to start reading the DaisyLink Config register of each module in the chain until it finds the interrupting module. Then the mainboard writes a zero to the module's DaisyLink Config register. This does not affect the module's pull-up state, but does clear the interrupt condition of the device. If it is still within the 1-ms time period, the module releases the upstream neighbor bus—unless another downstream module is still trying to interrupt the mainboard.

Note that although a module may not drive its upstream neighbor bus line low for more than 1 ms for its own interrupt, it may need to drive the upstream bus low for more than 1 ms to propagate the interrupt conditions of modules downstream.

Shared Data (I^2C) Bus

The shared data bus should comply with the I^2C Standard mode by using a 100-kilohertz (kHz) maximum clock rate and a single master, which is the mainboard.

Each DaisyLink module should be able to switch its pull-up resistors on or off between the two shared bus signals (I^2C clock and I^2C data) and the 3.3V rail. The resistors must be 10,000 ohms and the pull-ups must be able to sink 0.5 milliamp (mA). The device turns on its pull-up resistors only during initialization or as directed by the mainboard. If the state of a device is reset, the device must disable the resistors.

Memory Map Interface

A memory map interface is implemented by using the shared data (I^2C) bus with the MCU implementing a standard I^2C-based EEPROM interface. A write transaction comprises the following bytes:

- A start condition.
- The I^2C ID with the *write* bit set (ID 1–127, LSB 0=write).
- The memory address to write (0–255).
- One or more bytes of data to write.
- A stop condition.

A read transaction includes the following bytes:

- A start condition.
- The I^2C ID with the write bit set (ID 1–127, lsb 0=write).
- The memory address to read (0–255).
- A repeated start condition (or stop condition and then start condition).

- The I²C ID with the read bit set (ID 1–127, LSB 1=read).
- One or more bytes of data that are written by the subordinate device clocked in by the master according to the I²C specification.
- A stop condition.

The address space is split into two regions: DaisyLink and device-specific. The first 8 bytes (addresses 0–7) are reserved for DaisyLink protocol operation, and the rest of the address space is for device-specific functionality.

The DaisyLink portion of the module address map is shown in the following table.

Address	Register Description
0	I²C ID of the module (the ID can be written only in Setup mode).
1	Configuration byte (Bit 0: low=pull-ups off, high=pull-ups on. Bit 1: low=module in Active mode, high=module is not initialized. These two bits can be written to only during Standby mode. Bit 7: low=normal operation, high=interrupt state. This bit is set by the module and cleared by the mainboard, which writes a zero to this register. This is the only bit of the DaisyLink map that can be affected in Active mode.
2	DaisyLink version (always 4 for devices that are compatible with this specification).
3	Module type (specific to the manufacturer)
4	Module version (assigned by the manufacturer).
5	Manufacturer code (the code is assigned to the manufacturer and maintained in a list at gadgeteer.codeplex.com).
6	Reserved (for future use; return 0 if read).
7	Reserved (for future use; return 0 if read).
8 to 255	Can be used for any purpose by the module designer.

These bytes are read-only except during module initialization as described in "DaisyLink Module Initialization" earlier in this document. At all other times, write operations must be ignored. The manufacturer code, module type, and module version are hardcoded by the manufacturer in the firmware of the device. The DaisyLink version must be 4.

If a multi-byte read operation occurs on the DaisyLink registers, the module should return data at successive addresses, beginning from the provided address. Thus, one read operation can read all eight registers. For module-specific registers, the manufacturer decides whether this same behavior is used or whether a read returns many bytes from the same address. The module could use an alternative model such as implementing streams of data at each address or 32-bit words of data at each address.