

Gadgeteer

Device Driver Authoring Guide

Gadgeteer: Device Driver Authoring Guide

Version 2.0

Publication date \$Date: 2010-06-10 20:45:28 -0500 (Thu, 10 Jun 2010) \$

Copyright © 2003–2010 Iowa State University

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being Appendix B. *GNU Free Documentation License*, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in Appendix B. *GNU Free Documentation License*.

Table of Contents

I. Introduction	1
1. Overview of Gadgeteer	2
Goals of Gadgeteer	2
Goals for Device Driver Authors	2
Portability	3
Maintainability	3
Efficiency	3
Modularity	4
2. Using the VR Juggler Portable Runtime	5
Buffered I/O	5
Serial Ports	5
Sockets	6
Threads	6
Programmer Reference	7
II. Programming	8
3. Drivers and the Input Manager	9
Drivers as Input Manager Plug-Ins	9
Device Types	9
Analog	9
Command	9
Digital	9
Gesture	10
Glove	10
Position	10
Simulator	10
String	10
The Input Mixer	10
4. Device Driver Conventions	12
Separation of Code	12
5. Writing Device Drivers	13
Identifying the Device Type	13
Implementing the Standalone Device Driver	13
Implementing the Gadgeteer Wrapper Class	13
Choose the Base Class(es)	14
Driver Plug-in Entry Points	19
6. Compiling the Driver Plug-in	21
7. Driver Configuration	22
Configuring the Input Manager	22
Driver Configuration Definition File	22
Driver Configuration File	24
Writing Code that Accepts the Configuration	25
getElementType()	25
config()	26
III. Appendices	28
A. Complete Device Driver Code	30
Standalone Driver	30
Gadgeteer Wrapper	30
Makefile Templates	33
B. GNU Free Documentation License	35
PREAMBLE	35
APPLICABILITY AND DEFINITIONS	35

VERBATIM COPYING	36
COPYING IN QUANTITY	36
MODIFICATIONS	37
COMBINING DOCUMENTS	38
COLLECTIONS OF DOCUMENTS	39
AGGREGATION WITH INDEPENDENT WORKS	39
TRANSLATION	39
TERMINATION	39
FUTURE REVISIONS OF THIS LICENSE	40
ADDENDUM: How to use this License for your documents	40
Bibliography	41
Glossary of Terms	42
Index	43

List of Examples

5.1. Spawning a Non-Member Function Thread	15
5.2. Spawning a Static Member Function Thread	16
5.3. Spawning a Member Function Thread	16
5.4. Implementation of <code>sample()</code> Member Function	18
5.5. Implementation of <code>updateData()</code> Member Function	18
5.6. Implementation of <code>getGadgeteerVersion()</code> Entry Point Function	19
5.7. Implementation of <code>initDevice()</code> Entry Point Function	20
6.1. Example Makefile for Device Driver Plug-in	21
7.1. Example Input Manager Configuration	22
7.2. <code>button_device.jdef</code> : Configuration Definition File for Simple Button Device	23
7.3. <code>button_device.jconf</code> : Configuration File for Simple Button Device	24
7.4. Implementation of <code>getElementType()</code> Member Function	26
7.5. Implementation of <code>config()</code> Member Function	27
A.1. <code>buttondevice.h</code>	31
A.2. <code>buttondevice.cpp</code>	32
A.3. <code>Makefile.in</code> for Gadgeteer Build System	33
A.4. Makefile for Use Outside Gadgeteer Source Tree	33
A.5. Visual C++ Project for Use Outside Gadgeteer Source Tree	34

Part I. Introduction

We begin this book with some basic background information about Gadgeteer and the device drivers it uses. This part is written primarily for programmers who are new to Gadgeteer and VR Juggler in general. Rather than including technical content in this part, we instead review concepts and goals to provide new developers with an understanding of our motivations and our long-term goals for Gadgeteer.

Chapter 1. Overview of Gadgeteer

Gadgeteer acts as a hardware device management system. It contains a dynamically extensible Input Manager that treats devices in terms of abstract concepts such as “positional,” “digital,” “gesture,” etc. It also contains a Remote Input Manager that can share device samples between computers. Most importantly, Gadgeteer provides device input for use with VR Juggler applications. As such, Gadgeteer was designed from the beginning to be used with an ever-widening array of virtual reality hardware configurations.

Goals of Gadgeteer

Gadgeteer serves to hide input device hardware from programmers so that immersive software may be written that can take advantage of a wide variety of devices. This goal arises from previous experience with software toolkits that tied immersive applications to specific devices, thereby limiting the portability of the applications between immersive hardware configurations. With Gadgeteer, applications can be written that migrate transparently between different hardware configurations with no required knowledge on the part of the application author relating to vendors, models, drivers, etc.

Gadgeteer categorizes input devices based on abstract input types. The categories are the following:

- Analog
- Command
- Digital
- Glove
- Gesture
- Position
- Simulator
- String

Each of these is described in more detail below in Device Types.

In this categorization, devices from different vendors may return data that maps to the same abstract form. A single piece of hardware may even map to multiple input types, and more device types can be added as new hardware becomes available. Application authors write their code in terms of abstract input types, so as long as a device is available that provides the needed input, the application can function.

Goals for Device Driver Authors

In keeping with the general goals of Gadgeteer, device driver authors should strive to achieve certain goals for each device driver they write. In no particular order, we feel that the most important goals are the following:

- Portability
- Maintainability

- Efficiency
- Modularity

For the most part, these goals are no different than those of any other software project. Nonetheless, we will explain why each is important in the following subsections.

Portability

Gadgeteer is a cross-platform device management system, and as such, the devices it manages should be usable on all platforms supported by Gadgeteer. While this may not always be possible¹, device driver authors should still attempt to make their drivers as portable as possible. The *VR Juggler Portable Runtime* (VPR), introduced later in Chapter 2. *Using the VR Juggler Portable Runtime*, provides many features that simplify the work of writing portable software. This applies to device drivers as much as any other piece of software, and thus, programmers should make use of VPR whenever possible.

Maintainability

Hardware tends to evolve over time, and new versions of a given device may be released. With new hardware, the communication protocol may change, either through extensions or through extensive changes. In order for Gadgeteer device drivers to be used with new hardware, a driver must be written so that it can be maintained by other programmers. That means that a driver should be documented well, and it should not use complex techniques to communicate with the hardware.

Based on our experience, we recommend that the following practices when writing a new driver:

- Do not “brute force” the driver implementation just to get something working. Implement the protocol clearly and completely.
- Do not hard-code maximum values to match a local installation or the current limitations of the hardware. For example, if a positional tracker at the local facility only has two trackers attached to it, do not assume that everyone else has the same configuration.
- Do not do tricks with memory buffers. C and C++ provide very nice features for accessing blocks of memory, so there is usually no need to do pointer math by hand. More often than not, a `struct` or a `union` will do a much better job than an array of bytes.

Efficiency

Input devices used with virtual reality systems tend to sample at a much higher rate than the graphics are rendered (1000 Hz versus 60 Hz). Thus, for a given frame, the driver may make tens or hundreds of samples. Gadgeteer provides some facilities for efficient collection of samples, but ultimately, the driver author must ensure that the driver will not overwhelm the local computer (or the network if the Remote Input Manager is being used). On the other hand, minimizing input latency is very important in achieving good suspension of disbelief on the part of the user. Thus, it is not advisable to discard samples.

The key thing to keep in mind when writing a device driver for Gadgeteer is that the driver will be running asynchronously from the graphics. Usually, the sample rate will be limited by how fast the sample can be read from the hardware, be it a memory access, a serial port read, or a network buffer

¹There are various reasons why a given hardware device may not be usable between computers. For example, not all architectures have parallel ports, and thus, a parallel port device could not be expected to be used where no port is available. In general, however, the software device driver should not be the limiting factor in the use of a hardware device.

read. A balance between low latency, memory efficiency, and possibly network efficiency must be found.

Modularity

The current practices used in Gadgeteer encourage modularity of device drivers. Each driver should be able to stand on its own as a single unit within the Input Manager. This philosophy allows individual drivers to be loaded on demand at runtime, and it simplifies compilation of drivers that are not supported on all operating systems.

Chapter 2. Using the VR Juggler Portable Runtime

In this chapter, we will review briefly key components of the VR Juggler Portable Runtime [<http://www.vrjuggler.org/vapor/>] (VPR) that will be used by Gadgeteer device driver authors. This chapter is not meant to be a comprehensive description of VPR but rather a small guide to be used by programmers new to Gadgeteer, VPR, and other modules used by VR Juggler. We assume that the reader has some familiarity with operating system programming, in particular with serial device I/O, socket I/O, and multi-threaded techniques. One or more of these will almost certainly come into play when writing a device driver for use with Gadgeteer.

For those developers new to Gadgeteer and VPR, VPR provides an cross-platform, object-oriented abstraction layer to common operating system features. VPR is the key to the portability of Gadgeteer, Tweek, VR Juggler, and other middleware written at the Virtual Reality Applications Center. It has been in development since January 1997, and it has grown to be a highly portable, robust tool. Software written on top of VPR can be compiled on IRIX, Linux, Windows, FreeBSD, and Solaris, usually without modification.

Internally, VPR wraps platform-specific APIs such as BSD sockets, POSIX threads, and Win32 overlapped I/O. Depending upon how it is compiled, it may also wrap the Netscape Portable Runtime [<http://www.mozilla.org/projects/nspr/index.html>] (*NSPR*), another cross-platform OS abstraction layer written in C. By wrapping NSPR, VPR provides developers with an object-oriented interface and gains even better portability. These details are all hidden behind the classes that make up VPR, and users of VPR do not need to worry about platform-specific details as a result.

Buffered I/O

Before discussing features of VPR useful to device driver authors, we must first understand how I/O is handled in VPR. All I/O classes (file handles, serial ports, and sockets) share the base class `vpr::BlockIO`. Reads and writes are performed using blocks of memory (buffers). This design provides an API that more closely resembles that of the underlying operating system (with methods called `read()` and `write()`), but it is in contrast to stream-oriented I/O that is usually seen in C++. Streams could be written on top of the buffered I/O classes, but thus far, the need has not arisen. With this in mind, the design provides an API that is immediately familiar to programmers used to POSIX-based interfaces, but the API may seem clumsy to C++ programmers who are accustomed to using `std::ostream` and friends.

Serial Ports

Most input devices used for virtual reality systems today make use of a computer's serial port for data communication. For that reason, it is important that device driver authors have at least a basic understanding of the concepts behind the VPR serial port abstraction. In our experience, serial port programming is not much different than other I/O programming. Implementing the communication protocol used by a given device tends to be the hard part, and that will likely be the case regardless of the underlying hardware.

The VPR serial port abstraction is based on the concepts implemented by the standard `termios` serial interface used by most modern UNIX-based operating systems [Ste92]. As such, the API allows enabling and disabling of a subset of the serial device features that can be manipulated using `termios` directly. To provide cross-platform semantics, however, some `termios` features are not included because

there is no corresponding capability with Win32 overlapped I/O. Furthermore, any termios settings that relate specifically to modems are not included in the VPR serial port abstraction.

Sockets

Note

Readers not familiar with socket programming should consult a reference manual ([Ste98] is recommended). We do not attempt to explain the ins and outs of socket programming. Instead, we assume that readers are familiar with socket-level I/O and the ideas involved with various types of network communication.

The socket abstraction follows the concepts set forth by the *BSD sockets* API, which was also the model for the Winsock API used on Windows. In VPR, two types of sockets may be instantiated: stream-oriented (TCP, `vpr::SocketStream`) and datagram (UDP, `vpr::SocketDatagram`). The helper class `vpr::InetAddr` makes use of Internet Protocol (v4) addresses easier. Built on top of `vpr::SocketStream` are two classes that make writing client/server code easier: `vpr::SocketConnector` and `vpr::SocketAcceptor`. The `vpr::System` interface provides cross-platform data conversion functions to deal with endian issues.

The utility of various socket classes will vary depending on the needs of a given driver protocol. It is usually safe to assume that the driver will connect to a server of some sort that will send out device samples. Unpacking information from the samples may or may not be necessary, depending on the protocol. Such concerns are left entirely to the driver authors.

Threads

All device drivers written for Gadgeteer will process samples in a thread separate from the Input Manager. We have chosen this design to avoid the complications that often arise from using non-blocking I/O and to allow the drivers to act more as independent entities. Thus, it will be important to understand how to use the VPR thread interface.

First and foremost, developers must always remember that Gadgeteer uses a shared-memory model for all threads, regardless of the underlying platform-specific thread interface. This follows the lightweight thread model set forth by the POSIX threads (pthreads) standard. With a shared-memory model, all threads have access the same memory, and thus it will almost certainly be necessary to control access to shared variables. In most cases, the class `vpr::Mutex` will provide sufficient control over multi-threaded data access.

Caution

Multi-threaded programming can be tricky, and it is not something that most people can jump into without some background. Those developers who have not done multi-threaded programming before should review a manual or other reference on the topic before beginning work on a new driver. VPR threads are semantically similar to pthreads, and the concepts inherent in multi-threaded programming (e.g., protecting critical sections) will be the same regardless of the specific implementation. To learn more about pthreads specifically, we recommend [Nic96].

Device driver authors will probably not have to do much with shared data access control because the driver will operate almost entirely in the sample loop thread. Any other method invocations (starting the driver, stopping it, configuring it, etc.) will happen in the Input Manager thread, and common memory

accesses have pre-defined helper methods to simplify the work of driver authors. These details will be explained further in later chapters.

Programmer Reference

The various VPR abstraction interfaces are documented extensively, and readers are encouraged to review the VPR Programmer Reference (refer to the VPR website [<http://www.vrjuggler.org/vapor/>] for more information).

The VPR class names follow a standard convention, and understanding this can be helpful in navigating the API documentation. Classes that wrap platform-specific interfaces are named as follows: `vpr::<Type><Platform>`. For example, the NSPR implementation of `vpr::SocketStream` is named `vpr::SocketStreamNSPR`. Here, `<Type>` is “SocketStream”, and `<Platform>` is “NSPR”. The full list of platform names (as spelled in the class names) is as follows:

- Posix: Used for general POSIX-specified interfaces
- BSD: Used for the BSD socket wrapper classes
- Termios: Used for the termios serial port wrapper classes
- NSPR: Used for NSPR wrapper classes
- Win32: Used for Win32-specific wrapper classes

Part II. Programming

In this part of the book, we explain how to write device drivers and add them to Gadgeteer. We begin with a detailed description of device driver conventions in Gadgeteer and how the drivers fit into the Input Manager. We then explain how drivers are configured using JCCL. Throughout the following chapters, example code will be provided.

Chapter 3. Drivers and the Input Manager

As its name suggests, the Input Manager is in charge of managing the active input devices and the samples those devices return. Each device driver will hand off a freshly read sample (also known as a sample buffer) to the Input Manager.

Drivers as Input Manager Plug-Ins

The Input Manager itself never cares about the true type of a device. Instead, it looks at each driver as an implementation of the `gadget::Input` interface. This design lends itself well to a plug-in architecture wherein drivers can be loaded at runtime without being compiled into Gadgeteer. Using the Gadgeteer driver plug-in system, users can write their own device drivers without modifying Gadgeteer at all. Indeed, they need not even compile Gadgeteer from its source. All that is needed is a binary installation of Gadgeteer against which the user-written driver can be compiled.

Device Types

As of this writing, there are five key device types handled by the Input Manager:

1. Analog: `gadget::Analog`
2. Command: `gadget::Command`
3. Digital: `gadget::Digital`
4. Glove: `gadget::Glove`
5. Position: `gadget::Position`
6. String: `gadget::String`
7. Simulator: `gadget::SimInput`, `gadget::SimPosition`, `gadget::SimDigital`, `gadget::SimAnalog`, `gadget::SimGlove`

Analog

Analog input represents a continuous range of values. Of course, with digital computers, analog values can only be simulated. In Gadgeteer, this simulation is performed using floating-point values.

At the application level, programmers get values from an analog device in the range 0.0 to 1.0 inclusive. In other words, values returned by an analog device are normalized before they are returned to the application. This allows applications to get analog input from a variety of analog devices without depending on a specific range of values returned by any given device.

Command

Digital

Digital input comes in discrete forms, as its name suggests. However, a digital device in Gadgeteer terms corresponds most closely with a button device that has an “on” state and an “off” state. In that regard, a more appropriate name for a digital device within Gadgeteer would be a Boolean device, except that Gadgeteer provides more than just two values for input from a digital device. Due to its frame-based nature, Gadgeteer can tell users when the state of a digital device has changed since the last frame, thereby allowing for up to four values to be returned from a digital device:

1. On: The device is in the on state.
2. Off: The device is in the off state.
3. Toggle on: The device was in the off state during the last frame and changed to the on state this frame.
4. Toggle off: The device was in the on state during the last frame and changed to the off state this frame.

The management of the toggle states is handled by Gadgeteer; devices simply need to collect the raw on and off values.

Gesture

Glove

Position

Positional input is usually collected from a six-degree-of-freedom (6DOF) tracker such as a Polhemus Fastrak or an Ascension MotionStar. Thus, position devices in Gadgeteer return samples as standard 4×4 transformation matrices representing the position and orientation of a specific tracker. A tracker may not be able to track all six degrees of freedom, and this is allowed with the Gadgeteer position input type.

Simulator

For each of the above, there is at least one corresponding simulator device type¹. Such a device stands in for the corresponding “real” device when one is not available. For example, when using a VR application on the desktop, a 6DOF position tracker is not usually available. Instead, the mouse and keyboard could be used to stand in for the 6DOF tracker. Alternatively, a 3D graphical user interface (GUI) could be written using GLUT to provide a more visually expressive desktop tracker stand-in.

The word “simulator” is a bit of a misnomer. As noted above, these devices act more as stand-ins when another device is not available. To a VR application, the data returned will look exactly the same, but the input mechanism employed by the user will vary.

String

The Input Mixer

¹Gadgeteer is designed so that users may write new simulator devices. In fact, we encourage this so that we can expand on the ways that various input types may be “simulated” for desktop use.

The second version of the Remote Input Manager, introduced in mid-2002, implemented input distribution by sharing devices rather than proxies, as done in the original version [Ols92]. This refactoring has changed the class hierarchy for device drivers. Previously, classes such as `gadget::Digital` and `gadget::Position` derived from `gadget::Input`, and device drivers used multiple inheritance to derive from one or more of `gadget::Analog`, `gadget::Digital`, etc.

With the introduction of `gadget::InputMixer<S, T>`, device drivers now derive from this single template class. More information will be given in Chapter 5. *Writing Device Drivers*, but as an example, consider a driver for a positional device. In VR Juggler 1.0 and in early versions of Gadgeteer, such a driver class would have derived from `gadget::Position`. Now, it would derive from `gadget::InputMixer<gadget::Input, gadget::Position>`. Use of `gadget::InputMixer<S, T>` is required if a device is to be used with the Remote Input Manager. If the old class hierarchy is used (which is still allowed), the device cannot be shared between computers.

When deriving from `gadget::InputMixer<S, T>`, it is highly recommended that one of the predefined instantiations be used to ensure that a known type is used as a base class for the device driver. The order that the classes are listed in the `gadget::InputMixer<S, T>` is critical because the Remote Input Manager only knows about certain instantiations. If an unexpected instantiation is encountered, problems will occur. In the best case, the device data will not be shared in a cluster configuration. In the worst case, the application will crash. The known instantiations are all listed in the file `gadget/Type/InputBaseTypes.h`.

Caution

The Input Mixer may not be a long-term solution, though it has been in place for quite a while. A future version of Gadgeteer may do away with `gadget::InputMixer<S, T>`, and as such, driver authors should be aware of potential API changes in the future.

Chapter 4. Device Driver Conventions

Before we get into the actual coding process, we must first explain the conventions we have used in writing device drivers for Gadgeteer. We strongly recommend that all new drivers follow these conventions as they have proven successful for us for many years.

Separation of Code

The most obvious convention that can be seen upon review of existing device drivers is a separation of the driver code into two pieces: a standalone, “low-level” driver and a Gadgeteer wrapper around the standalone driver.

In this design, the standalone driver implements the complete hardware communication protocol without using any features of Gadgeteer. As such, it stands completely on its own and does not need Gadgeteer to be used. The result is that the driver can be tested and debugged without worrying that some part of Gadgeteer could be causing the driver to malfunction. Driver authors can focus entirely on implementing the hardware communication protocol so as to feel confident that the low-level driver is implemented correctly.

Note

The standalone driver should use VPR to ensure portability. For example, a driver that will communicate with the hardware via the serial port should use the VPR serial port abstraction. For more information, refer to Chapter 2. *Using the VR Juggler Portable Runtime* and to Goals for Device Driver Authors.

Tip

The low-level driver should have an easy-to-use interface that allows effective manipulation of the driver state (starting, stopping, requesting a sample, etc.). To develop a good interface and to test the standalone driver, write an application that creates an instance of the standalone driver, starts the driver running, and collects samples. In writing the test application, the interface can be matured for use by the Gadgeteer wrapper.

Around the low-level driver, a Gadgeteer wrapper is added. This wrapper makes use of the standalone driver interface to activate the driver and read samples. The wrapper class will derive from one or more of the Gadgeteer device types described in Device Types. Instances of the wrapper class will be handled by the Input Manager.

Tip

Do not put a sample loop in the low-level driver. Instead, provide a `sample()` method in the standalone driver API that the wrapper can call repeatedly. This allows the sample thread to be managed by the Gadgeteer wrapper class.

Chapter 5. Writing Device Drivers

At long last, we have covered enough background information to explain how to add device drivers to Gadgeteer. In this chapter, we will examine a very simple device that has an on state and an off state. The general flow of this chapter will model the process that driver programmers would normally follow when writing a new driver from scratch.

Identifying the Device Type

As discussed in Device Types, there are a set of abstract device types supported by Gadgeteer. Based on its capabilities, a new device will fall into at least one of the device type categories. It is perfectly valid for a single device to provide more than one type of input. For example, an Immersion Tech IBox returns both analog and digital data. Determining the device type for a new piece of hardware should be the easiest part of the driver authoring process.

Implementing the Standalone Device Driver

The standalone device driver makes use of *nothing* in Gadgeteer. It can utilize dependencies of Gadgeteer including VPR and GMTL, however. Reusing code from those projects is encouraged. In particular, writing the driver on top of VPR allows it to be much more portable than it would be if all the cross-platform code were written from scratch. The reason that the standalone driver does not use Gadgeteer is so that it can be tested without needing any of the complexity of the Input Manager, thereby allowing easier, more direct debugging.

In most cases the standalone driver should be an implementation of the hardware communication protocol and nothing more. The standalone driver is written as a single C++ class that provides an interface that the Gadgeteer wrapper class can call. The interface normally has methods such as `open()`, `sample()`, and `close()` for opening the connection to the hardware, collecting a single sample, and closing the connection to the hardware respectively.

The standalone driver class should return data in its most raw form in the majority of cases, but the data should be meaningful. For example, if logic is needed to convert four bytes read from the hardware into a single floating-point value (a float), that should be performed in the standalone driver. That sort of data processing is part of implementing the communication protocol. However, processing such as unit conversion should not be done in the standalone driver in most cases. Instead, such conversions should be handled by the Gadgeteer wrapper class since that is where the unit configuration is done.

Typically, the standalone driver will not be multi-threaded. Instead, a method with a name such as `sample()` should be provided that returns a single sample. Then, test code and the Gadgeteer wrapper class can call the sampling method in a loop which may or may not be run in a thread.

With this design, the standalone driver class can be tested by writing a simple console application that makes an instance of the class and invokes each of the methods. The application can be interactive so that users can configure aspects of the driver and take samples. This makes debugging and data validation easy.

Implementing the Gadgeteer Wrapper Class

The Gadgeteer wrapper class has the job of passing samples read from the standalone driver off to the Input Manager. Depending on the device type, a given sample must be of a certain form. This is where

sample buffers come into play. We will discuss sample buffers later in this section, but the possible sample buffer types are the following:

- `gadget::AnalogData`
- `gadget::CommandData`
- `gadget::DigitalData`
- `gadget::GloveData`
- `gadget::PositionData`
- `gadget::StringData`

Choose the Base Class(es)

As discussed earlier in Device Types, all device drivers in Gadgeteer must derive from one or more classes based on the device type. If a driver is to be used with the Remote Input Manager (i.e., there exists a desire to share a device between two or more computers), then the base class must be `gadget::InputMixer<S, T>` with appropriate device type classes given as the template parameters. If, for whatever reason, the device will not be used with the Remote Input Manager, it may derive from one or more of the device type classes directly using multiple inheritance.

For example, to make a driver that registers button presses, derive from `gadget::input_digital_t` (the `gadget::InputMixer<S, T>` instantiation that includes only `gadget::Digital`):

```
class ButtonDevice
    : public gadget::input_digital_t
```

Suppose that a game controller driver supporting buttons and joystick axes is needed. In this case, an additional component is needed for the for analog input from the X and Y axes. Since the device is both digital and analog, its class must derive from both `gadget::Digital` and `gadget::Analog` using the appropriate `gadget::InputMixer<S, T>` instantiation:

```
class JoystickDevice
    : public gadget::input_digital_analog_t
```

Using the type `ButtonDevice` as declared above, we will proceed with the implementation of the driver class. There are six member functions that must be implemented by every driver class:

1. `startSampling()`: A pure virtual member function declared by `gadget::Input`
2. `stopSampling()`: A pure virtual member function declared by `gadget::Input`
3. `sample()`: A pure virtual member function declared by `gadget::Input`
4. `updateData()`: A pure virtual member function declared by `gadget::Input`
5. `getElementType()`: A static member function
6. `config()`: A virtual member function declared by `gadget::Input`, `gadget::Analog`, `gadget::Digital`, etc., that must be overridden

In this section, we will examine the first four of these. The remaining two will be addressed in Writing Code that Accepts the Configuration.

startSampling()

- Method synopsis:

```
virtual bool startSampling();
```

- Overrides the pure virtual function `gadget::Input::startSampling()` declared in `gadget/Type/Input.h`

Within this function, a new thread is started. This thread is used to sample the data from the device. There are two ways to create threads using `vpr::Thread`. The first uses a non-member function or a static member function. The second uses a non-static member function. To spawn a thread that executes a non-member function or a static member function, the code would be similar to that shown in Example 5.1. Spawning a Non-Member Function Thread and Example 5.2. Spawning a Static Member Function Thread respectively. To spawn a thread that executes a member function, the code would be similar to that shown in Example 5.3. Spawning a Member Function Thread. The thread can be tested for validity using the method `vpr::Thread::valid()`. For a complete description of how to use the cross-platform multi-threading capabilities of the VR Juggler Portable Runtime, refer to the *VPR Programmer's Guide* [<http://www.vrjuggler.org/vapor/docs.php>].

Example 5.1. Spawning a Non-Member Function Thread

```
namespace
{
    void nonMemberSampleFunction(ButtonDevice* devPtr)
    {
        // Keep working until isRunning() returns false.
        while ( devPtr->isRunning() )
        {
            devPtr->sample();
        }
    }
}

bool ButtonDevice::startSampling()
{
    mRunning = true;
    mThread = new vpr::Thread(boost::bind(nonMemberSampleFunction, this));
    return true;
}
```

Example 5.2. Spawning a Static Member Function Thread

```
class ButtonDevice : public ...
{
public:
    ...
    bool startSampling();
    bool sample();

private:
    static void staticMemberSampleFunction(ButtonDevice* devPtr);
    ...
};

bool ButtonDevice::startSampling()
{
    mRunning = true;
    mThread =
        new vpr::Thread(boost::bind(ButtonDevice::staticMemberSampleFunction,
                                    this));
    return true;
}

void ButtonDevice::staticMemberSampleFunction(void* arg)
{
    ButtonDevice* dev_ptr = static_cast<ButtonDevice*>(arg);

    // Keep working until mRunning becomes false.
    while ( dev_ptr->mRunning )
    {
        dev_ptr->sample();
    }
}
```

Example 5.3. Spawning a Member Function Thread

```
bool ButtonDevice::startSampling()
{
    mRunning = true;
    mThread =
        new vpr::Thread(boost::bind(&ButtonDevice::memberSampleFunction, this));
    return true;
}

void ButtonDevice::memberSampleFunction()
{
    // Keep working until mRunning becomes false.
    while ( mRunning )
    {
        this->sample();
    }
}
```

Tip

Using a thread for the sample loop is not always necessary or appropriate. It is most useful for the case of a device protocol implementation that relies on blocking I/O. Such a device could cause the Input Manager to block when querying the latest sample, and this would impact the application frame rate. For drivers that can read data at any time without blocking, simply taking a sample in the implementation of `updateData()` is sufficient. The decision to use a thread or not depends very much on the specific hardware communication protocol, and driver authors must take this into consideration on a per-driver basis.

Caution

In certain cases, it is highly desirable to avoid using a thread. For example, devices that return sample data immediately would collect many, many samples that would have to be shared among cluster nodes every frame. The burden on the network could have a negative impact on the performance of the cluster. The driver we are examining in this section is a good example of a driver that should *not* use a thread because it does not block while waiting for hardware to return a sample. Nevertheless, we illustrate the use of a thread so that readers can see how it works.

`stopSampling()`

- Method synopsis:

```
virtual bool stopSampling();
```

- Overrides the pure virtual function `gadget::Input::stopSampling()` declared in `gadget/Type/Input.h`

The job of this function is to kill the thread created in `startSampling()` and release resources such as open file handles. If no thread was created, then this method only needs to release any open resources. Remember that `startSampling()` could be called again later, so drivers must be written to handle the case where they are shut down and restarted.

`sample()`

- Method synopsis:

```
virtual bool sample();
```

- Overrides the pure virtual function `gadget::Input::sample()` declared in `gadget/Type/Input.h`

This method reads data from the device and stores it for later use by `gadget::Digital::getDigitalData()`. Note that `ButtonDevice::sampleFunction()`, defined above, invokes this method. It is the responsibility of the driver itself to call `sample()`, and therefore, its implementation may be empty depending on how the driver is designed.

Gadgeteer devices typically use triple-buffered data management. This is done to ensure that data is not being written into a buffer when the Input Manager is trying to read the most recent value. The various base classes for device types provide helper methods for handling triple-buffered data. This is a big help over VR Juggler 1.0 and early versions of VR Juggler 1.1 where the triple-buffering had to be managed manually in every driver.

We are not actually implementing a hardware protocol, so we simply use dummy values as the sample buffers. For this, we use the protected base class method `gadget::Digital::addDigitalSample()`. Note that this method expects a `std::vector<T>` containing sample buffers as its parameter. In our case, we will have one sample buffer in the vector, as shown below:

Example 5.4. Implementation of `sample()` Member Function

```
bool ButtonDevice::sample()
{
    bool status(false);

    if ( mRunning )
    {
        // Here you would add your code to sample the hardware for a
        // button press:
        std::vector<gadget::DigitalData> digital_samples(1);
        digital_samples[0] = 1;
        addDigitalSample(digital_samples);

        // Successful sample.
        status = true;
    }

    return status;
}
```

For a device driver that supports multiple data types, there are additional inherited helper methods such as `gadget::Position::addPositionSample()`, `gadget::Analog::addAnalogSample()`, etc. Refer to the Gadgeteer Programmer Reference for complete details about the class declarations.

`updateData()`

- Method synopsis:

```
virtual void updateData();
```

- Overrides the pure virtual function `gadget::Input::updateData()` declared in `gadget/Type/Input.h`

This method is invoked by the Input Manager once per frame for each active device to prepare the latest input data for use by higher level code such as a VR Juggler application object. Again, we can use a helper function inherited from `gadget::Digital` to perform this operation. As usual, there are similar helper functions that would be inherited from the other device types if our driver supported multiple types of input data.

Example 5.5. Implementation of `updateData()` Member Function

```
void ButtonDevice::updateData()
{
    if ( mRunning )
    {
        swapDigitalBuffers();
    }
}
```

Note

As mentioned above, some drivers may perform their entire sample operation in their override of `gadget::Input::updateData()`. This driver would be a good candidate for that use of the `updateData()` method. Instead of building up a large number of sample buffers in a separate thread, our sample could be taken here. This would be accomplished by combining the implementations of `ButtonDevice::sample()` and `ButtonDevice::updateData()` into this method, thereby leaving `ButtonDevice::sample()` empty.

Driver Plug-in Entry Points

All the device drivers that can be loaded at run time by the Input Manager have what are known as *entry point functions*. These are free functions with C-style signatures that may be looked up at run time using operating system features. In Gadgeteer 1.0, every driver plug-in must include two exported functions: `getGadgeteerVersion()` and `initDevice()`. The first is used for simple version checking, and the second registers the driver with the Input Manager.

For Windows programmers, these are the only two symbols that must be exported from the driver DLL. The device driver class or classes do not have to be exported from the DLL.

Driver Version Checking

As of the release of Gadgeteer 1.0 Beta 1, all driver plug-ins must have an entry point function that allows the Input Manager to perform version checking. This is done to ensure that the driver is compatible with the Input Manager. The implementation of this function will be the same for all drivers, and it is shown in Example 5.6. Implementation of `getGadgeteerVersion()` Entry Point Function. Because we are dealing with C++ code, we must indicate to the compiler that this is a C-style function, so no name mangling should occur when its symbol table entry is created. We do this by wrapping the function body in an `extern "C"` block. For cross-platform plug-in capabilities, we use the `GADGET_DRIVER_EXPORT()` macro. On Win32 systems, this will add the appropriate type modifiers to declare `initDevice()` as a function exported by the DLL that will be compiled. For other platforms, the macro simply evaluates to the void type. (These details are handled within the `gadget/Devices/DriverConfig.h` header.)

Example 5.6. Implementation of `getGadgeteerVersion()` Entry Point Function

```
#include <gadget/Devices/DriverConfig.h>
#include <vpr/vpr.h>
#include <gadget/gadgetParam.h>

extern "C"
{
    GADGET_DRIVER_EXPORT(vpr::UInt32) getGadgeteerVersion()
    {
        return __GADGET_version;
    }
}
```

The result of this implementation is that the Gadgeteer version number is compiled into the driver plug-in. At run time, the Input Manager compares this value with the version of the Gadgeteer library. If the two match exactly, the driver loading process continues. If not, the driver plug-in is ignored.

This form of version compatibility testing is rudimentary, but because the Juggler libraries do not guarantee binary compatibility between releases, it is necessary. In the future, a more sophisticated version comparison system may be devised for driver plug-ins. Presently, it is already possible to get the individual version numbers of the Gadgeteer library (major, minor, and patch), and these could be used as the foundation for a more advanced compatibility management system.

Register the Driver with the Input Manager

Device driver registration is done through a template type called `gadget::DeviceConstructor<T>`. When this type is used with a special “factory function” called `initDevice()`, the driver can be used as a plug-in to the Input Manager. While there are some drivers that cannot currently be loaded dynamically, those that can include an entry point function named `initDevice()`.

With all of that, we can now write the body for `initDevice()`. No declaration in a header file is needed because this function will be looked up dynamically at run time. The implementation of `initDevice()` will appear in `ButtonDevice.cpp` as follows:

Example 5.7. Implementation of `initDevice()` Entry Point Function

```
#include <gadget/Devices/DriverConfig.h>
#include <gadget/Type/DeviceConstructor.h>
#include "ButtonDevice.h"

extern "C"
{

GADGET_DRIVER_EXPORT(void) initDevice(gadget::InputManager* inputMgr)
{
    new gadget::DeviceConstructor<ButtonDevice>(inputMgr);
}

}
```

Chapter 6. Compiling the Driver Plug-in

At this point, our device driver is not yet complete because it cannot be configured. Nevertheless, there will be nothing for the Input Manager to load—and hence no driver to configure—until we are able to compile the driver plug-in. In this chapter, we explain how to compile the driver as a dynamically loadable component. We present this from the perspective of a driver that exists outside the Gadgeteer driver source tree. For drivers that are integrated into the Gadgeteer driver source tree, refer to Makefile Templates.

The device driver must be compiled into a standalone dynamically loadable component. The usual file extension names for plug-ins are `.so`, `.dll`, or `.dylib` depending on the host operating system. This component will act as the Input Manager plug-in. In this way, there is no need to modify the Gadgeteer source code to add a new driver. Thus, the driver code is collected into a cohesive unit that can be distributed as a plug-in for Gadgeteer.

Compiling device driver plug-ins for use with Gadgeteer is a relatively simple process that can use either GNU Make or a Visual C++ project file. To compile a driver plug-in on Windows, a Visual C++ project file must be used; using GNU Make on Windows is not supported by Gadgeteer 1.0.

In either case, the name of the compiled driver plug-in must follow certain conventions. A debug version of the driver must be named as `<device>_drv_d.<ext>`. Here, `<device>` provides a meaningful, unique name for the driver plug-in, and `<ext>` is the platform-specific file extension for plug-ins. The `_d` part indicates that the driver is debug-enabled and on Windows that it is linked against the Visual C++ debug runtime. Examples are `IS900_drv_d.so` and `MotionStar_drv_d.dll`. On Mac OS X, the extension `.dylib` is used rather than `.bundle`. An optimized (“release” in Visual C++ terminology) version of the driver uses the same convention but drops the `_d` part.

When using GNU Make, a very short makefile is all that is required. An example for our button driver is shown in Example 6.1. Example Makefile for Device Driver Plug-in. The makefile allows either a debug or an optimized version of the driver to be compiled depending on the value assigned to the `$(BUILD_TYPE)` variable. The `$(SRCS)` variable must contain the complete list of source files for the driver plug-in. The file extensions `.c`, `.C`, `.cc`, `.CC`, `.cxx`, `.cpp`, and `.c++` are recognized. The environment variable `GADGET_BASE_DIR` must be set for use by this makefile. It would usually be set to the same value as `VJ_BASE_DIR`.

Example 6.1. Example Makefile for Device Driver Plug-in

```
srcdir=          .
BUILD_TYPE=     dbg
#BUILD_TYPE=    opt

DRIVER_NAME=    button
SRCS=          buttondevice.cpp

include $(GADGET_BASE_DIR)/share/gadgeteer/gadget.driver.mk
```

To use a Visual C++ project file, the example provided in `$(GADGET_BASE_DIR)/share/gadgeteer/samples/tutorials/device.driver` can be used as a starting point. It sets all the compiler and linker options. Users must change the list of source and header files and the names of the output files. It is also recommended that users edit the `.vcproj` file with a text editor to remove the `ProjectGUID` setting on line 6 before loading the project file into Visual Studio. This will ensure that the new Visual C++ project file has a unique identifier.

Chapter 7. Driver Configuration

In Gadgeteer, device drivers should be parameterized. This means that aspects of the driver that can vary should be configurable rather than hard-coded values. This is in line with the general Juggler philosophy of making software components flexible to provide a high degree of adaptability. In this chapter, we explain how to configure the Input Manager to load a device driver plug-in and how to make a device driver configurable. For a more detailed explanation of the Juggler configuration system, refer to the VR Juggler *Configuration Guide*.

Configuring the Input Manager

Run-time driver registration depends on the Input Manager configuration. For our button device driver, the Input Manager would be configured to load our driver plug-in using the configuration file shown in Example 7.1. Example Input Manager Configuration. Here, the driver plug-in is named `button_drv.so` (or some other platform-specific name), and it is found in the user's home directory. Never include the `_d` part of the driver plug-in name or the platform-specific file extension. The Input Manager determines these details automatically, thus making the configuration files highly portable across different operating systems. For more details about how to configure the Input Manager, refer to the VR Juggler *Configuration Guide*.

Example 7.1. Example Input Manager Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<?org-vrjuggler-jccl-settings configuration.version="3.0"?>
<configuration
  xmlns="http://www.vrjuggler.org/jccl/xsd/3.0/configuration"
  name="Configuration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.vrjuggler.org/jccl/xsd/3.0/configuration http://w
  <elements>
    <input_manager name="Button Device Input Manager" version="2">
      <driver_path>${HOME}</driver_path>
      <driver>button_drv</driver>
    </input_manager>
  </elements>
</configuration>
```

Driver Configuration Definition File

Every Gadgeteer device needs a unique *element type* associated with it. An element type is similar to a struct in C or C++. The data structure is defined in an configuration definition file (which usually has the extension `.jdef`). Once defined, the type for a new driver can be used in JCCL configuration files.

Therefore, a new driver-specific configuration definition must be created before a driver can be configured. We recommend that this be done using the VRJConfig Config Definition Editor. For the button device, the definition file is shown in Example 7.2. `button_device.jdef`: Configuration Definition File for Simple Button Device.

Example 7.2. `button_device.jdef`: Configuration Definition File for Simple Button Device

```
<?xml version="1.0" encoding="UTF-8"?>
<?org-vrjuggler-jccl-settings definition.version="3.1"?>
<definition xmlns="http://www.vrjuggler.org/jccl/xsd/3.1/definition"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.vrjuggler.org/jccl/xsd/3.1/definition h
    name="button_device"> ❶
  <definition_version version="1" label="My Button Device">
    <help>Configuration for simple one-button device.</help>
    <parent>digital_device</parent> ❷
    <category>/Devices/Digital</category>
    <property valuetype="string" variable="false"
      name="port"> ❸
      <help>Serial port the device is connected to.</help>
      <value label="Port" defaultvalue="/dev/ttyd1"/>
    </property>
    <property valuetype="integer" variable="false"
      name="baud"> ❹
      <help>Serial port speed.</help>
      <value label="Baud" defaultvalue="38400"/>
    </property>
    <upgrade_transform/>
  </definition_version>
</definition>
```

- ❶ This begins the definition for our device type. The name attribute must be named as a valid XML tag (a CNAME in XML terminology) because it will be used as such in a configuration file. A free-form, human-friendly string may be specified in the `label` attribute of the `definition_version` element. This string will be presented to the user of VRJConfig, and as such, it should be a meaningful identifier.
- ❷ The parent (or base type) for this config definition. Zero or more of these are allowed. In this case, we must indicate that `digital_device` is a parent type so that VRJConfig will know that digital proxies can be pointed at config elements for our driver. We also inherit property definitions from `digital_device` including “`device_host`”, which is needed for cluster configurations.
- ❸ This declares the “`port`” property that will provide the name of the serial port to which the hardware is connected. The serial port name will be interpreted as a string, and it has the default value of “`/dev/ttyd1`”. In the case of our simple button driver, there is no serial port, but we include this property definition to demonstrate how the whole configuration definition works.
- ❹ This declares the “`baud`” property that will provide the baud setting for the serial port to which the hardware is connected. The baud value will be interpreted as an integer, and it has the default value of 38400 (kilobits per second). In the case of our simple button driver, there is no serial port, but we include this property definition to demonstrate how the whole configuration definition works.

Note

In the above configuration definition, we do not declare a “`device_host`” property, which is used in conjunction with the Remote Input Manager. This is not necessary because we have declared our parent type to be “`digital_device`”, and we inherit its property definitions. All drivers that may be used with the Remote Input Manager must have the “`device_host`” property, and configuration definition inheritance ensures that this will be the case. Refer to the *VR Juggler Configuration Guide* for more information about this property.

For a more complex device, a more complex configuration definition may be needed. Again, the VRJConfig Configuration Definition Editor simplifies the creation of this definition.

Driver Configuration File

Once the configuration definition is in place, a new configuration element can be created. Once again, VRJConfig makes the step easier. In Example 7.3. `button_device.jconf`: Configuration File for Simple Button Device, we see a configuration file that configures the one-button device we have been using thus far.

Important

In order for VRJConfig and the JCCL Configuration Manager to find the driver-specific `.jdef` file at run time, it may be necessary to extend the `.jdef` search path using the environment variable `JCCL_DEFINITION_PATH`. This environment variable is set in the same manner as the `PATH` environment variable, meaning that it uses platform-specific conventions. On UNIX-based platforms, the directories to search are separated with the colon (`:`) character; on Windows, the path separator character is the semi-colon (`;`). If the `.jdef` file is installed in the default search path (`$VJ_BASE_DIR/share/vrjuggler/data/definitions`), there is no need to set `JCCL_DEFINITION_PATH`.

Example 7.3. `button_device.jconf`: Configuration File for Simple Button Device

```
<?xml version="1.0" encoding="UTF-8"?>
<?org-vrjuggler-jccl-settings configuration.version="3.0"?>
<configuration xmlns="http://www.vrjuggler.org/jccl/xsd/3.0/configuration"
    name="Configuration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.vrjuggler.org/jccl/xsd/3.0/configura
<elements>
  <input_manager name="Button Device Input Manager"
    version="2"> 1
    <driver_path>${HOME}</driver_path>
    <driver>button_drv</driver>
  </input_manager>
  <button_device name="Button Device" version="1"> 2
    <port>/dev/ttyd4</port> 3
    <baud>9600</baud> 4
    <device_host /> 5
  </button_device>
  <digital_proxy name="Button Proxy 0" version="1"> 6
    <device>Button Device</device>
    <unit>0</unit>
  </digital_proxy>
  <alias name="VJButton0" version="1">
    <proxy>Button Proxy 0</proxy>
  </alias>
</elements>
</configuration>
```

- ❶ The `input_manager` element configures the Gadgeteer Input Manager. In this case, we are telling the Input Manager about a driver plug-in, found at `${HOME}/button_drv.so`, that should be loaded at run time.

Tip

Earlier, we introduced the Input Manager configuration in a separate configuration file. In general, we recommend configuring a given device entirely within one `.jconf` file—including the `input_manager` config element that loads the driver plug-in. That is precisely what we are showing in this example.

- ❷ Next, we have an instance of the configuration definition shown in Example 7.2. `button_device.jdef`: Configuration Definition File for Simple Button Device. As described above, `<button_device>` is named based on the `name` attribute of the `definition` element in our configuration definition file. The `name` attribute here gives this *instance* a unique identifier.
- ❸ Now, we set the value for the serial port name. As noted above, our simple button device does not actually use the serial port, but this demonstrates how the property value is used in a configuration file. If no value were given here, the default value set in `button_device.jdef` would be used.
- ❹ This provides a value for the serial port baud setting. Again, this will not actually be used by our simple device, but we show it here to give a complete example.
- ❺ For our example, we will not fill in a value for `device_host` because we are not dealing with the Remote Input Manager. Refer to the *VR Juggler Configuration Guide* for more information about this.
- ❻ Finally, we point a digital proxy at our device. Since the device has only one input source, we have to use 0 for the unit value.

Writing Code that Accepts the Configuration

In the driver, recall that there are two methods that must be implemented in order to handle config elements:

1. `getElementType()`
2. `config()`

We now present these two methods, thereby completing the interface of the driver as far as the Input Manager is concerned.

`getElementType()`

- Method synopsis:

```
static std::string getElementType();
```

- Overrides the pure virtual function `gadget::Input::updateData()` declared in `gadget/Type/Input.h`

When the configuration changes, the JCCL Configuration Manager asks every registered configuration handler for their respective configuration element types. If the type matches the type of the newly received configuration element, then the handler's `config()` method is invoked. All device drivers are

configuration handlers and thus need to indicate the configuration element type they accept¹. The type value is returned by the member function `getElementType()`.

In this function, the element type of the device must be returned. The string returned by `getElementType()` must be *exactly* the same as the name attribute for the driver configuration definition that we saw above in Driver Configuration Definition File. For example, the implementation for the simple button driver would appear as shown in Example 7.4. Implementation of `getElementType()` Member Function.

Example 7.4. Implementation of `getElementType()` Member Function

```
std::string ButtonDevice::getElementType()
{
    return std::string("button_device");
}
```

Important

Returning the correct string from `getElementType()` is very important. Failing to do so will result in the config element for the device driver ending up in the JCCL Configuration Manager's pending list at run time. For device driver authors, this can be a serious point of confusion. The driver plug-in will load correctly, but if the string returned by `getElementType()` is incorrect, the driver will never be configured or started.

config()

- Method synopsis:

```
virtual bool updateData(jccl::ConfigElementPtr element);
```

- Overrides the virtual function `gadget::Input::config()` declared in `gadget/Type/Input.h` as well as the virtual functions `gadget::Analog::config()`, `gadget::Position::config()`, etc., depending on the type parameters used in the base class declaration

When the Configuration Manager detects a configuration change for a given driver, it will pass the new `jccl::ConfigElementPtr` object as the parameter to this method. This is when the driver must handle its configuration and store the information for use when the Input Manager tells the device to start up. For more information about how to use instances of `jccl::ConfigElementPtr`, refer to the *JCCL C++ Programmer's Reference*. The implementation of `config()` for our simple button device driver is shown in Example 7.5. Implementation of `config()` Member Function.

Important

The override of `gadget::Input::config()` must invoke the implementations of `config()` that it overrides from its base classes. All the base classes defined by Gadgeteer for different input device types have a `config()` method, though some of them have an empty method body. Nevertheless, *all* inherited `config()` methods must be invoked by the driver in order to ensure that the driver is fully configured. In general, this should be done first before reading any of the device-specific properties from the config element. This is demonstrated in the example below.

¹A comprehensive explanation of the full functionality of the Juggler run-time (re-)configuration system is beyond the scope of this document. Here, we present only the basic information about config definitions and config elements that is needed for writing a complete device driver. For more details, see the VR Juggler *Configuration Guide*.

Example 7.5. Implementation of `config()` Member Function

```
bool ButtonDevice::config(jccl::ConfigElementPtr e)
{
    // Configure all our base classes first.  If any of those fail,
    // we cannot finish configuring ourselves.
    if ( ! gadget::Input::config(e) && ! gadget::Digital::config(e) )
    {
        return false;
    }

    mPortName = e->getProperty<std::string>("port");
    mBaudRate = e->getProperty<int>("baud");

    return true;
}
```

Part III. Appendices

Table of Contents

A. Complete Device Driver Code	30
Standalone Driver	30
Gadgeteer Wrapper	30
Makefile Templates	33
B. GNU Free Documentation License	35
PREAMBLE	35
APPLICABILITY AND DEFINITIONS	35
VERBATIM COPYING	36
COPYING IN QUANTITY	36
MODIFICATIONS	37
COMBINING DOCUMENTS	38
COLLECTIONS OF DOCUMENTS	39
AGGREGATION WITH INDEPENDENT WORKS	39
TRANSLATION	39
TERMINATION	39
FUTURE REVISIONS OF THIS LICENSE	40
ADDENDUM: How to use this License for your documents	40

Appendix A. Complete Device Driver Code

Standalone Driver

Gadgeteer Wrapper

Now that we have explained the concepts involved in adding a device driver to Gadgeteer, we can show some code. The following example is for a fictitious piece of hardware that has only one button.

Important

This implementation is trivial by design. A more appropriate implementation for this case would not use a thread for sampling and would instead call `sample()` from `updateData()` so that only one sample is recorded per frame. The reason for doing this is because this “driver” is not I/O-bound and instead just adds a value to the digital sample buffer every time `sample()` is invoked.

```

45     /**
        * Spawns the sample thread, which calls ButtonDevice::sample()
        * repeatedly. Complete Device Driver Code
        */


---


50     virtual bool startSampling();
Example A.1. buttondevice.h
    /**
        * Records (or samples) the current data. This is called
        * repeatedly by the sample thread created by startSampling().
55     */
    virtual bool sample();

    /** Kills the sample thread. */
    virtual bool stopSampling();
60
    virtual void updateData();

    /**
        * Returns a string that matches this device's configuration
65     * element type.
        */
    static std::string getElementType();

    /**
70     * Invokes the global scope delete operator. This is required
        * for proper releasing of memory in DLLs on Win32.
        */
    void operator delete(void* p)
    {
75     ::operator delete(p);
    }

protected:
    /**
80     * Deletes this object. This is an implementation of the pure
        * virtual gadget::Input::destroy() method.
        */
    virtual void destroy()
    {
85     delete this;
    }

private:
    /**
90     * Our sampling function that is executed by the spawned
        * sample thread. This function simply calls
        * ButtonDevice::sample() over and over.
        */
    void threadedSampleFunction();
95
    vpr::Thread* mSampleThread;
    bool mRunning;

    // configuration data set by config()
100    std::string mPort;
    int mBaud;
};

#endif /* _EXAMPLE_BUTTON_DEVICE_H */

```

```

        << std::endl;
    }
75     return mRunning;
}
Complete Device Driver Code

```

Example A.2: ButtonDevice.cpp

```

// ButtonDevice.cpp
// Reads (or samples) current data. This is called
80 // repeatedly by the sample thread created by startSampling().
bool ButtonDevice::sample()
{
    bool status(false);

85     if ( mRunning )
    {
        // Here you would add your code to sample the hardware for
        // a button press:
        std::vector<DigitalData> samples(1);
90         samples[0] = 1;
        addDigitalSample(samples);

        // Successful sample.
        status = true;
95     }

    return status;
}

100 // Kills the sample thread.
bool ButtonDevice::stopSampling()
{
    mRunning = false;

105     if (mSampleThread != NULL)
    {
        mSampleThread->kill();
        mSampleThread->join();
        delete mSampleThread;
110         mSampleThread = NULL;
    }
    return true;
}

115 void ButtonDevice::updateData()
{
    if ( mRunning )
    {
        swapDigitalBuffers();
120     }
}

// Our sampling function that is executed by the spawned sample
// thread.
125 void ButtonDevice::threadedSampleFunction()
{
    // spin until someone kills "mSampleThread"
    while ( mRunning )
    {
130         sample();
    }
}

```

Makefile Templates

The following is an example `Makefile.in` that could be added to the Gadgeteer build system.

Example A.3. Makefile.in for Gadgeteer Build System

```
default: all

# Include common definitions.
include @topdir@/make.defs.mk

DRIVER_NAME=    ButtonDevice

srcdir=         @srcdir@
top_srcdir=     @top_srcdir@
INSTALL=        @INSTALL@
INSTALL_FILES=
SUBOBJDIR=      $(DRIVER_NAME)
C_AFTERBUILD=   driver-dso

SRCS=           ButtonDevice.cpp \
                DriverStandalone.cpp

include $(MKPATH)/dpp.obj.mk
include @topdir@/driver.defs.mk

# Include dependencies generated automatically.
ifndef DO_CLEANDEPEND
ifndef DO_BEFOREBUILD
    -include $(DEPEND_FILES)
endif
endif
```

The following is a makefile for a driver that is built outside of the Gadgeteer source tree. This cannot be used on Windows with Gadgeteer 1.0.

Example A.4. Makefile for Use Outside Gadgeteer Source Tree

```
srcdir=         .
BUILD_TYPE=     dbg
#BUILD_TYPE=    opt

DRIVER_NAME=    button
SRCS=           buttondevice.cpp

GADGET_BASE_DIR=$(shell flagpoll gadgeteer --get-prefix)
GADGET_VERSION= $(shell flagpoll gadgeteer --modversion)

include $(GADGET_BASE_DIR)/share/gadgeteer-$(GADGET_VERSION)/gadget.driver.mk
```

The following is a Visual C++ project file for a driver that is built outside of the Gadgeteer source tree. A Visual C++ project file must be used for compiling driver plug-ins on Windows.

Appendix B. GNU Free Documentation License

Version 1.2, November 2002

FSF Copyright note

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant.

The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose

the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

GNU FDL Modification Conditions

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.

- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same

name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void,

and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Sample Invariant Sections list

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

Sample Invariant Sections list

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

- [Nic96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. A POSIX Standard for Better Multiprocessing. O'Reilly & Associates. 1996.
- [Ols92] Eric Olson. *Cluster Juggler: PC cluster virtual reality*. Iowa State University. Dept. of Electrical and Computer Engineering. 2002.
- [Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley. 1992.
- [Ste98] W. Richard Stevens. *UNIX Network Programming*. Volume 1. Network APIs: Sockets and XTI. Second Edition. Prentice-Hall PTR. 1998.

Glossary of Terms

B

BSD sockets The socket programming interface introduced with the Berkeley Software Distribution version of the UNIX operating system. It is made up of a collection of system calls that allow highly flexible socket programming. Most UNIX variants in use today use the BSD sockets API. Moreover, the Winsock API used on Windows is based on this API.

N

Netscape Portable Runtime More information can be found at <http://www.mozilla.org/projects/nspr/index.html>

V

VR Juggler Portable Runtime More information can be found at <http://www.vrjuggler.org/vapor/>

Index

C

classes

- gadget::Analog, 11
- gadget::AnalogData, 14
- gadget::Command, 9
- gadget::CommandData, 14
- gadget::DeviceConstructor<T>, 20
- gadget::Digital, 9, 11, 18
- gadget::DigitalData, 14
- gadget::Glove, 9
- gadget::GloveData, 14
- gadget::Input, 9, 11
- gadget::InputMixer<S,T>, 11
- gadget::Position, 9, 9, 11, 11
- gadget::PositionData, 14
- gadget::SimAnalog, 9
- gadget::SimDigital, 9
- gadget::SimGlove, 9
- gadget::SimInput, 9
- gadget::SimPosition, 9
- gadget::String, 9
- gadget::StringData, 14
- vpr::BlockIO, 5
- vpr::InetAddr, 6
- vpr::Mutex, 6
- vpr::SocketAcceptor, 6
- vpr::SocketConnector, 6
- vpr::SocketDatagram, 6
- vpr::SocketStream, 6
- vpr::System, 6
- vpr::Thread, 15

D

device drivers

- configuring, 22
- example, 30
- implementing
 - Gadgeteer wrapper class, 13
 - identifying device type, 13
 - standalone driver, 13

- plug-ins, 9
- registering, 20
- writing, 13

device types, 9

- analog, 9
- command, 9
- digital, 9
- gesture, 10
- glove, 10
- position, 10

- simulator, 10
- string, 10

G

Gadgeteer

- goals, 2
- overview, 2

I

- Input Manager, 2
- input mixer, 10

R

- Remote Input Manager, 2, 11

V

VPR

- overview, 5
- programmer reference, 7
- serial port abstraction, 5
- socket abstraction, 6
- thread abstraction, 6
- using, 5