

Gadgeteer

Device Driver Authoring Guide

Gadgeteer: Device Driver Authoring Guide

Published \$Date: 2003/12/05 20:36:50 \$

Table of Contents

I. Introduction	1
1. Overview of Gadgeteer	3
Goals of Gadgeteer	3
Goals for Device Driver Authors	3
Portability	4
Maintainability	4
Efficiency	4
Modularity	5
2. Using the VR Juggler Portable Runtime	6
Buffered I/O	6
Serial Ports	6
Sockets	7
Threads	7
Programmer Reference	7
II. Programming	9
3. Drivers and the Input Manager	11
Drivers as Input Manager Plug-Ins	11
Device Types	11
Position	11
Digital	11
Analog	12
Glove	12
Gesture	12
Simulator	12
The Input Mixer	12
4. Device Driver Conventions	14
Separation of Code	14
5. Writing Device Drivers	15
Identifying the Device Type	15
Implementing the Standalone Device Driver	15
Implementing the Gadgeteer Wrapper Class	15
Choose the Base Class(es)	16
Register the Driver with the Input Manager	19
6. Configuration	21
Configuration Files	21
Writing Code that Accepts the Configuration	23
III. Appendices	24
A. Complete Device Driver Code	26
Standalone Driver	26
Gadgeteer Wrapper	26
Makefile Templates	29
Bibliography	31
Glossary of Terms	32
Index	33

List of Examples

6.1. button_device.jdef: Configuration Definition File for Simple Button Device	21
6.2. button_device.jconf: Configuration File for Simple Button Device	22
A.1. ButtonDevice.h	26
A.2. ButtonDevice.cpp	27
A.3. Makefile.in for Gadgeteer Build System	29
A.4. Makefile for Use Outside Gadgeteer Source Tree	30

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.

Table of Contents

1. Overview of Gadgeteer	3
Goals of Gadgeteer	3
Goals for Device Driver Authors	3
Portability	4
Maintainability	4
Efficiency	4
Modularity	5
2. Using the VR Juggler Portable Runtime	6
Buffered I/O	6
Serial Ports	6
Sockets	7
Threads	7
Programmer Reference	7

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 computers¹. 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:

- Position
- Analog
- Digital
- Glove
- Gesture
- Simulator

Each of these is described in more detail below in the section called “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

¹The Remote Input Manager provides the foundation for Cluster Juggler, the software that allows VR Juggler applications to be run on a cluster of graphics workstations. For more information about Cluster Juggler refer to the *Cluster Juggler Guide* and to the VR Juggler website [<http://www.vrjuggler.org/>].

- 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 possible², 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 read. A balance between low latency, memory efficiency, and possibly network efficiency must be found.

²There 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.

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 re-

view 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
- SPROC: Used for the SPROC thread wrapper class
- 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.

Table of Contents

3. Drivers and the Input Manager	11
Drivers as Input Manager Plug-Ins	11
Device Types	11
Position	11
Digital	11
Analog	12
Glove	12
Gesture	12
Simulator	12
The Input Mixer	12
4. Device Driver Conventions	14
Separation of Code	14
5. Writing Device Drivers	15
Identifying the Device Type	15
Implementing the Standalone Device Driver	15
Implementing the Gadgeteer Wrapper Class	15
Choose the Base Class(es)	16
Register the Driver with the Input Manager	19
6. Configuration	21
Configuration Files	21
Writing Code that Accepts the Configuration	23

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. Position: `gadget::Position`
2. Digital: `gadget::Digital`
3. Analog: `gadget::Analog`
4. Glove: `gadget::Glove`
5. Simulator: `gadget::SimInput`, `gadget::SimPosition`, `gadget::SimDigital`, `gadget::SimAnalog`, `gadget::SimGlove`

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.

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.

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.

Glove

Gesture

Simulator

For each of the above, there is at least one corresponding simulator device type³. 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.

The Input Mixer

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 `gad-`

³Gadgeteer 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.

`get::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.

Note

As of this writing, the Input Mixer is not expected to be a long-term solution. 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 the section called “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 the section called “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 the section called “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:

Choose the Base Class(es)

As discussed earlier in the section called “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::Digital`:

```
class ButtonDevice
    : public gadget::InputMixer<gadget::Input, gadget::Digital>
```

Suppose that a joystick driver supporting buttons and movement is needed. In this case, an additional component, this one for analog input, is needed for 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 C++ multiple inheritance:

```
class JoystickDevice
    : public gadget::InputMixer<gadget::Input,
                               gadget::InputMixer<gadget::Digital,
                                                    gadget::Analog> >
```

Note

To use the joystick in place of a tracker, it should derive instead from `gadget::Position`. This way, you can replace real trackers with your joystick “pseudo tracker”. The main idea is that to be able to replace one device with another, the alternate device class must derive from the same base classes as the original device.

Using basic class declaration for `ButtonDevice` from above, we will proceed with the implementation of the driver class. First, there are six member functions that must be implemented:

`startSampling()`

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

Within this function, a new thread is started. This thread is used to sample the data from the device. The thread creation step may look something like the following:

```
vpr::ThreadMemberFunctor<ButtonDevice>* functor =
    new vpr::ThreadMemberFunctor<ButtonDevice>(this,
                                                &ButtonDevice::sampleFunction,
                                                NULL);
mThread = new vpr::Thread(functor);
```

The above creates a thread that will execute `ButtonDevice::sampleFunction()`, a non-static member function in the class `ButtonDevice`. The implementation of that method would be similar to the following in most cases

```
void ButtonDevice::sampleFunction(void* arg)
{
```

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

The thread can be tested for validity using the method `vpr::BaseThread::valid()`.

stopSampling()

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

The job of this function is to kill the thread created in `startSampling()`.

sample()

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

This method reads data from the device and stores it for later use by `getDigitalData()`. Note that `ButtonDevice::sampleFunction()`, defined above, invokes this method.

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 `gadget::Input` class defines three variables to help programmers keep track of which buffer is in use at any given time: `gadget::Input::current`, `gadget::Input::valid`, and `gadget::Input::progress`. The sampled data would be read into a three-element array of the correct type (this is driver-specific). When writing the freshly sampled data into the array, use `gadget::Input::progress`:

```
mSampledDigitalData[gadget::Input::progress] = sampled_digital_value;
```

updateData()

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

Triple-buffered device drivers use this method to swap the data indices. The member function is usually implemented as follows:

```
void ButtonDevice::updateData()
{
    vpr::Guard<vpr::Mutex> updateGuard(lock);

    // Copy the valid data to the current data so that both are valid
    mSampledDigitalData[current] = mSampledDigitalData[valid];

    // swap the indices for the tri-buffer pointers
    gadget::Input::swapCurrentIndexes();
}
```

Note the use of a `vpr::Guard<>` object to synchronize access to the `mSampledDigitalData` array. This is needed because the sampling and the reading are occurring in separate threads, but both threads need access to `mSampledDigitalData`.

getElementType()

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

In the `getElementType()` function, the *element type* of the device must be returned. Its name must be as it appears in the configuration definition file for the driver. For example, the implementation for the simple button driver would appear as:

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

At this time, it is useful to point out that every Gadgeteer device needs an 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.

getDigitalData()

```
virtual int getDigitalData(int devNum = 0);
```

The Input Manager uses this method to read digital data sampled by the driver. This is when the triple-buffered data scheme becomes especially valuable. To provide the Input Manager with the most up-to-date sample, use `gadget::Input::current` as the index, as shown below:

```
int JoystickDevice::getDigitalData(int devNum)  
{  
    return mSampledDigitalData[current];  
}
```

Note that in this example, the parameter `devNum` is ignored. This is not always the case. Indeed, this button driver would likely have support for more than one button, and in that case, we would use `devNum` as the index into an array or vector containing data sampled from all the buttons.

getAnalogData()

There are other methods that must be implemented depending on the classes from which a given driver class derives. In the joystick example given earlier, the method `getAnalogData()` would have to be implemented in addition to `getDigitalData()`. The prototype for `getAnalogData()` is:

```
virtual float getAnalogData(int devNum = 0)
```

The joystick driver would use this to return values for the X and Y axes. The data here is more complex because it would be for triple-buffered two-dimensional samples. An implementation might look similar to the following:

```
float JoystickDevice::getAnalogData(int axis)  
{  
    vprASSERT(axis >= 0 && axis <= 1 && "only 2 axes (x and y) available");  
    return mSampledAnalogData[current][axis];  
}
```

In this driver, the integer argument to the method is used to represent either the X or the Y axis. The as-

sertion ensures that a valid axis index is passed.

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, for those that can, we implement an “entry point” function named `initDevice()`. Because we are dealing with C++ code, we must indicate to the compiler that this is a C 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 `DriverConfig.h` header.)

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:

```
#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);
}

}
```

The new device driver can be compiled into a standalone library (`.so`, `.dll`, or `.dylib` are the usual suffix choices for plug-ins). This library 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 (in other words, a component) for Gadgeteer.

Runtime driver registration depends on the Input Manager configuration. Assuming a UNIX-like environment, the Input Manager could be configured to load our driver plug-in using the following configuration file:

```
<?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>ButtonDevice_drv</driver>
      </input_manager>
    </elements>
  </configuration>
```

Here, the driver plug-in is named `ButtonDevice_drv.so` (or some other platform-specific name), and it is found in the user's home directory.

Chapter 6. Configuration

To configure a device, two things are needed:

1. Configuration files
2. Driver code that accepts the configuration

Configuration Files

Before configuring a device, a new configuration definition must be created. We recommend that this be done using VRJConfig. For the button device, the definition file will be the following:

Example 6.1. `button_device.jdef`: Configuration Definition File for Simple Button Device

```
<?xml version="1.0" encoding="UTF-8"?>
<?org-vrjuggler-jccl-settings definition.version="3.0"?>
<definition xmlns="http://www.vrjuggler.org/jccl/xsd/3.0/definition"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.vrjuggler.org/jccl/xsd/3.0/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>
    <category>/Devices/Digital</category>
    <property valuetype="string" variable="false" name="port">
❷         <help>Serial port to which this device is connected.</help>
         <value label="Port" defaultvalue="/dev/ttyd1"/>
    </property>
    <property valuetype="integer" variable="false" name="baud">
❸         <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 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** 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.
- ❸** 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 *Cluster Juggler Guide* for more information about this property.

For a more complex device, a more complex configuration definition may be needed. Again, the VRJ-Config configuration definition editor simplifies the creation of this definition.

Once the configuration definition is in place, a new configuration element can be created. Once again, VRJConfig makes the step easier. The following is an example configuration file that configures the one-button device we have been using thus far:

Example 6.2. `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">
    <driver_path>${HOME}</driver_path>
    <driver>ButtonDevice_drv</driver>
  </input_manager>
  <button_device name="Button Device" version="1">
    <port>/dev/ttyd4</port>
    <baud>9600</baud>
    <device_host />
  </button_device>
</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}/ButtonDevice_drv.so`, that should be loaded at runtime.
- ❷ Next, we have an instance of the configuration definition shown in Example 6.1, “`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 *Cluster Juggler Guide* for more information about this.

Writing Code that Accepts the Configuration

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

1.

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

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 accept. The type value is returned by this method, and the implementation for our simple driver is the following:

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

Note that the string returned matches the token we defined in Example 6.1, “button_device.jdef: Configuration Definition File for Simple Button Device”.

2.

```
virtual bool gadget::Input::config(jccl::ConfigElementPtr e);
```

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. For more information about how to use instances of `jccl::ConfigElementPtr`, refer to the *JCCL Programmer's Reference*. The following is a simple example for the basic button device we have used thus far:

```
bool ButtonDevice::config(jccl::ConfigElementPtr e)
{
    if ( ! 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	26
Standalone Driver	26
Gadgeteer Wrapper	26
Makefile Templates	29

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.

Example A.1. ButtonDevice.h

```
1 #ifndef _MY_BUTTON_DEVICE_H_
  #define _MY_BUTTON_DEVICE_H_

  #include <gadget/Devices/DriverConfig.h>
5
  #include <stdlib.h>

  #include <vpr/vpr.h>
  #include <vpr/Thread/Thread.h>
10 #include <gadget/Type/Input.h>
  #include <gadget/Type/Digital.h>
  #include <gadget/Type/InputMixer.h>

15 class ButtonDevice
    : public gadget::InputMixer<gadget::Input, gadget::Digital>
  {
  public:
    ButtonDevice()
20      : mSampleThread(NULL)
      , mRunning(false)
    {
      /* Do nothing. */ ;
    }
25
    virtual ~ButtonDevice()
    {
      if ( mRunning )
30      {
        this->stopSampling();
      }
    }

    virtual void updateData();
35    virtual bool startSampling();
    virtual bool sample();
    virtual bool stopSampling();

40    static std::string getElementType();

    /**
```

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

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

60 private:
    static void    sampleFunction(void* classPointer);
    int           mDigitalData;
    vpr::Thread*  mSampleThread;

65     bool        mRunning;

    // configuration data set by config()
    std::string   mPortName;
    int           mBaudRate;
70 };

#endif
```

Example A.2. ButtonDevice.cpp

```
1 #include <gadget/Devices/DriverConfig.h>

    #include <vpr/vpr.h>
    #include <vpr/System.h>
5 #include <gadget/InputManager.h>
    #include <gadget/Type/DeviceConstructor.h>

    #include "ButtonDevice.h"

10 using namespace gadget;

    extern "C"
    {
15 GADGET_DRIVER_EXPORT(void) initDevice(InputManager* inputMgr)
    {
        new DeviceConstructor<ButtonDevice>(inputMgr);
    }
20 }

    /** Returns a string that matches this device's configuration element type. */
    std::string ButtonDevice::getElementType()
25 {
    return std::string("MyButtonDevice");
}
```

```
    }

    //: When the system detects a configuration change for your driver, it will
30 // pass the new jccl::ConfigElement into this function. See the documentatio
    // on config elements, for information on how to access them.
    bool ButtonDevice::config(jccl::ConfigElementPtr e)
    {
        if ( ! Digital::config(e) )
35     {
            return false;
        }

        mPort = e->getProperty<std::string>("port");
40     mBaud = e->getProperty<int>("baud");

        return true;
    }

45 void ButtonDevice::updateData()
    {
        if ( mRunning )
        {
            swapDigitalBuffers();
50     }
    }

    /**
    * Spawns the sample thread, which calls MyButtonDevice::sample() repeatedly.
55 */
    bool ButtonDevice::startSampling()
    {
        mRunning = true;
        mSampleThread = new vpr::Thread(threadedSampleFunction, (void*) this);
60     if ( ! mSampleThread->valid() )
        {
            mRunning = false;
            return false; // thread creation failed
65     }
        else
        {
            return true; // thread creation success
70     }
    }

    /**
    * Records (or samples) the current data. This is called repeatedly by the
    * sample thread created by startSampling().
75 */
    bool ButtonDevice::sample()
    {
        bool status(false);

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

            // Successful sample.
            status = true;
90     }
    }
```

```
        return status;
    }

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

100     if (mSampleThread != NULL)
        {
            mSampleThread->kill(); // Not guaranteed to work on all platforms
            mSampleThread->join();
            delete mSampleThread;
105     mSampleThread = NULL;
        }
        return true;
    }

110 /**
    * Our sampling function that is executed by the spawned sample thread.
    * This function is declared as a static member of MyButtonDevice. It simply
    * calls MyButtonDevice::sample() over and over.
    */
115 void ButtonDevice::threadedSampleFunction(void* classPointer)
    {
        ButtonDevice* this_ptr = static_cast<ButtonDevice*>( classPointer );

        // spin until someone kills "mSampleThread"
120     while ( this_ptr->mRunning )
        {
            this_ptr->sample();
            vpr::System::sleep(1); //specify some time here, so you don't waste CPU c
125 }
    }
```

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

```
1 default: all

    # Include common definitions.
    include @topdir@/make.defs.mk
5
    DRIVER_NAME=      ButtonDevice

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

15  SRCS=             ButtonDevice.cpp \
                    DriverStandalone.cpp
```

```
include $(MKPATH)/dpp.obj.mk
include @topdir@/driver.defs.mk
20 # -----
# Include dependencies generated automatically.
# -----
ifndef DO_CLEANDEPEND
25 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.

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

```
BUILD_TYPE= dbg

DRIVER_NAME= button
SRCS=        buttondevice.cpp

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

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/>

thread abstraction, 7
using, 6

Index

C

classes

- gadget::Analog, 12
- gadget::DeviceConstructor<T>, 19
- gadget::Digital, 11, 12, 12
- gadget::Glove, 11
- gadget::Input, 11, 12
- gadget::InputMixer<S, T>, 12, 12, 13, 13
- gadget::Position, 11, 11, 12, 12, 16
- gadget::SimAnalog, 11
- gadget::SimDigital, 11
- gadget::SimGlove, 11
- gadget::SimInput, 11
- gadget::SimPosition, 11

D

device drivers

- configuring, 21
- example, 26
- implementing
 - Gadgeteer wrapper class, 15
 - identifying device type, 15
 - standalone driver, 15
- plug-ins, 11
- registering, 19
- writing, 15

device types, 11

- analog, 12
- digital, 11
- gesture, 12
- glove, 12
- position, 11
- simulator, 12

G

Gadgeteer

- goals, 3
- overview, 3

I

Input Manager, 3

input mixer, 12

R

Remote Input Manager, 3, 12

V

VPR

- overview, 6
- programmer reference, 7
- serial port abstraction, 6
- socket abstraction, 7