Design and Testing of a Sensor Middleware for Mobile Phones

by

Clayton James Williams

S.B. CS, M.I.T., 2008

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

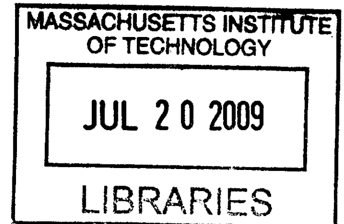
Master of Engineering in Electrical Engineering and Computer Science

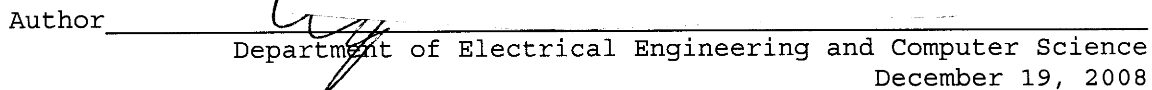
at the Massachusetts Institute of Technology

February, 2009

©2008 Massachusetts Institute of Technology
All rights reserved.

ARCHIVES

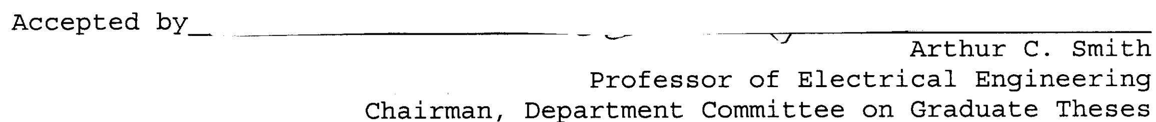




Author
Department of Electrical Engineering and Computer Science
December 19, 2008

Certified by

Dr. Stephen Intille
Research Scientist
MIT Department of Architecture
Thesis Supervisor



Accepted by
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Design and Testing of a Sensor Middleware for Mobile Phones
by
Clayton James Williams

Submitted to the
Department of Electrical Engineering and Computer Science

December 19, 2008

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Sensing of the physical world is a challenging task for mobile applications. There is no standardized method of obtaining data from sensors or for distributing those data to applications that use it. Consequently, many researchers in mobile sensors end up writing custom software to interact with their sensors. Unfortunately, these customized solutions are often tightly integrated with the researcher's specific task and are rarely used for a different sensing application. A sensor framework is presented that offers a standardized, easy-to-use, and efficient application programming interface (API) to sensor software authors. The API lets them write a sensor module focused on producing useful interpretations of the sensor data, without considering how the produced data will be distributed to client applications. As mobile sensors become more ubiquitous, this work will help researchers working with mobile devices to save development time boost the reliability and speed of sensor software that uses sensor input for context detection.

Thesis Supervisor: Dr. Stephen Intille
Title: Technology Director, MIT House_n Research Group

1. Introduction

A central theme of mobile computing research is modeling user context. "Context" refers to the properties of the world surrounding a user that could potentially be used to help a computer make better decisions about how to interact with that person. One important aspect of context is the physical world – properties such as lighting, noise, movement, temperature, etc. If a mobile phone were able to sense these properties it might be able to, for example, automatically silence its ringer when the user is in a movie theater or turn up the brightness of its touch screen when the user is outside. These are relatively simple examples of how contextual information about the user's environment might be used to passively improve human-computer (or human-phone) interaction. The ability to sense physical properties, however, can also add completely new functionality to the mobile device. If the phone were able to sense the user's physical motion, it might be able to monitor the user's activities throughout the day and use its knowledge about those activities to motivate the user to change his behavior, resulting in a more active, healthy lifestyle.

To help develop and study prototypes of such devices, researchers at House_n are designing Wockets[1]. Wockets are small, wireless modules that are designed to be worn on the body and that provide a mobile computer with information from an environmental sensor attached to the module. The first generation of Wockets are being used to sense movement using accelerometers. Applications that use Wockets are custom-built by researchers with a specific purpose in mind. As such, every developer who wants to use a Wocket must first write software that connects to and interprets the bytes coming from the physical device and then write software that derives a useful set of properties from

that incoming data. This is an inefficient use of the developer's time, but it also potentially reduces the amount of useful work a mobile phone can do in a unit of time. For example, if two or more applications that use Wockets are running on the same phone, all of them must perform their calculations independently, even if they need exactly the same information. These redundant operations can be a major performance problem, especially if the properties being derived are processor-intensive.

The WocketSensor framework is a library that allows sensor implementers to create a standardized, easy-to-use, and shareable "sensor module" in software that provides interesting properties of the underlying sensor data to any number of client applications. It also presents a clean, consistent API to those client applications, no matter what type of sensor those applications are interacting with. Finally, it automatically handles sharing of sensors, so that if two applications are interested in the same piece of raw sensor data or in the same derived property of that raw data, the machine need only compute those data once, and forward a copy of it to each of the client applications.

2. Background

The concept of "context" in the mobile computing world is not rigidly defined. Several definitions are given in [2], ranging from the general:

> *Context is the set of environmental states and settings that either determines an application's behavior or in which an application event occurs and is interesting to the user.*

to codifying 4 very specific areas:

- *Computing context*, such as network connectivity, communication costs, and communication bandwidth, and nearby resources such as printers, displays, and workstations.

- *User context,* such as the user's profile, location, people nearby, even the current social situation.
- *Physical context*, such as lighting, noise levels, traffic conditions, and temperature.
- *Time context*, such as time of day, week, month, and season of the year.

Of course, some context detection requires complex inference from sensors. For example, to compute the current social situation, the a computer might need to consider several other pieces of context, drawing at least from the user, physical, and time contexts. As sensor-aware software becomes more prevalent, the number of applications that need to share access to a single physical sensor will only increase, and as algorithms which exploit sensor data become standardized, even intermediate-level derived properties of these sensors will need to be shared across many applications.

2.1 *Multi-sensor awareness and cues derived from raw data*

Gellerson, et al. present a method for determining the physical context of a mobile phone user, the TEA context-awareness module [3]. Their module consists of a prototype board that holds several sensors, connected to a mobile phone via a serial cable. These sensors are each polled by the prototype board at individual frequencies and the results are forwarded to the phone. Several user *contexts* are then computed from the data gathered from the sensors. For example, if the user is in a dark environment, at room temperature, in an "indoor" location, and the time is "night-time", the TEA module might predict that the user is likely to be asleep. Their approach is important to the design of the WocketSensor framework for two reasons: first, they use a module that is aware of the output of many different sensors to make decisions about context and, second, the data that this omniscient module sees is not always the direct value of a physical sensor, but might be pre-processed in some way.

The TEA module uses a layered architecture, as in Figure 1. The physical sensor layer is at the bottom, and each physical sensor generates several "cues" – data streams that are based on the raw sensor data but that are processed or filtered in some way. Gellersen et al. write, "Each cue is a feature extracted from the data stream of a single sensor, and many diverse cues can be derived form the same sensor" [3]. Cues are meant to be relatively simple calculations, such as a mean or standard deviation over a set time period. However, some cues require significant processor resources even if their derivation from the raw data is fairly straightforward – such as calculating the Fast Fourier Transform to find which frequencies carry the most power in an audio signal.

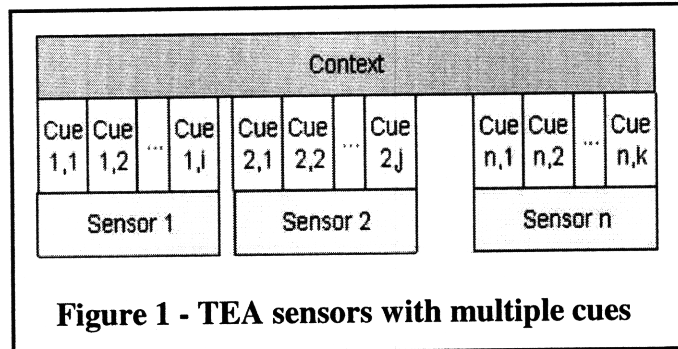


Figure 1 - TEA sensors with multiple cues

2.2 *Logical sensors*

Schmidt et al. expand the TEA module, adding the concept of a *logical* sensor in addition to the normal physical sensors, along with several other enhancements [4]. These logical sensors are based on parts of the user's context that are not physical, such as the current GSM cell or data from an internet source. This is an important generalization of the concept of a "sensor," because with logical sensors it is now possible for an application to get all the information it needs through the same sensor interface, which can be standardized across all sensors.

2.3 *Application independence and simple interfaces*

One other enhancement of the TEA module is the addition of a fourth layer to the sensor model: the *application* layer [4]. In this new model, the context is still treated as an omniscient observer of all the sensors, but the results of the context layer are available to higher-level applications through a scripting interface. This allows developers who have no knowledge about how the sensors and/or context layer are implemented to use the contextual cues to do interesting things in their programs.

```
Entering a context:
// if the context is: T=h={(v,p)}
// if the situation v is indicated with a certainty of
// p or higher than p the action(i) is performed after
// n milliseconds, v is a situation, p is a number,
if enter(v, p, n) then perform action(i)
```

Figure 2 – TEA scripting interface

Moreover, the interface used to communicate with higher-level programs is small and clearly defined. A set of context states are defined, and the probability of each of these states is continuously calculated by the context layer. Three points are defined for an application to interact with the context layer: When a context state is being *entered*, when a context state is being *left*, and *while* a context state is on-going. The interface description for *entering* states is shown in Figure 2; the other two are similarly simple.

2.4 *Sensor independence*

In the TEA module, the context calculation engine is a monolithic piece of software that knows about all the sensors in the platform and any new sensors that must be custom-added to the context layer. In contrast, MyExperience [5] uses a modular

approach which allows sensors to be added independently of the code of the platform. MyExperience is an electronic experience-sampling tool, designed to collect in-situ usage statistics from populations of mobile phone users by periodically asking them questions in a survey.

Through the use of scripts embedded in surveys, the decision about when to ask a question and which question to ask can be influenced by sensor modules - pieces of software developed independently of the platform that conform to the sensor interface defined by the platform. Through this interface, sensors notify MyExperience when the user's context changes and MyExperience passes that information to a script in the survey that controls the flow of the survey. Because the platform has an interface for outside sensor software, adding a new sensor to MyExperience does not require any changes in or recompilation of the platform code Instead, developers write a sensor module that links to the platform's libraries. This sensor module can then be included in survey scripts and will be loaded at run-time by the platform.

2.5 *Multiplexing sensor data*

All of the sensor platforms in the previous sections are designed to work with a single application consuming sensor data. However, this is not a necessary restriction. The Microsoft GPS Intermediate Driver for Windows Mobile CE-based mobile phones [6] is an example of a middleware that allows an arbitrary number of applications running independently on a phone to each have independent access to the data gathered by the phone's GPS chip. Perhaps because it is one of the few developer-ready sensors to come

built-in with mobile phones, this GPS driver is an example of sensor software that is built to be accessible and re-usable, rather than to fit a specific application's needs.

3. Objectives

Building on the ideas in the previous section, the framework was designed with four goals in mind:

1. Ease of Use
2. Modularity
3. Sharing and Efficiency
4. Robustness

Because the framework is to be used by many independent developers to create software that must work together, the most important design consideration is that sensors be easy to write and easy to use. They must represent the underlying data source in a way that both feels natural to developers and that lets them exploit all of the properties of the physical sensors and external data sources that make up the sensor modules. Furthermore, this representation must be flexible enough to allow for many different types of sensors. For example, the data from an acceleration sensor might be very different than the data from an indoor/outdoor sensor in terms of data type, consistency, and frequency. Accordingly, the framework's API must be both flexible and simple.

One of the problems with sensor software is that it is usually developed with a specific application in mind. This creates a tight bond between the software in charge of using sensor data to produce results and the software in charge of using those results to do something useful. Because the coupling between these two portions of the code is difficult to separate, it is often challenging to re-use the portion of the code that deals with sensors in a different project, even though the two projects might need the same data

from those sensors. The biggest advantage of using the WocketSensor framework to write a piece of sensor software is that once the module is written, it can be used easily by anyone who wants to develop applications that depend on the sensor data it provides. Therefore, sensors in the framework must be self-contained and must not make any assumptions about how the data they provide is going to be used.

Sensors in the framework must also be shareable, meaning that one sensor module that is producing data can be attached to an arbitrary number of client programs that are consuming data. This is a natural model for many types of physical sensors (one sensor can provide data to many applications), but it also increases the efficiency of the sensor modules by reducing the amount of duplicated work. A key idea behind the framework is that a property derived from lower-level data should never be calculated twice. If two (or more) programs need the same piece of data or properties derived from data, they both get it from the same sensor. The sensor calculates the data once, and forwards the result to all applications that want it.

Finally, since the framework is designed to be used as an external library, it must be robust. The library itself should not be a source of unexpected crashes, and it should not have memory leaks. Although it is impossible to completely eliminate all programming errors, the framework should be shown to conform to these principles, under most conditions.

4. Design

At a high level, the WocketSensor framework is a library that manages streams of data between software that is producing data and software that is consuming data.

WocketSensors expose a simple API to developers on each side of the stream that lets them tell the library which streams should be opened and what data should be sent to/received from streams at what times, but those developers are not exposed to - and have no control over - how the library manages memory, threading, inter-process communication, or any other low-level concepts. The word "sensor" in the following sections is used to denote a piece of software written inside the WocketSensor framework that produces some data and sends it to the framework library to be forwarded to applications that want to use those data. Unlike Schmidt et al. [4], the framework draws no distinctions between *physical* and *logical* sensors: the data exported may be based on a physical sensor or on some other form of information available to the sensor, or both.

4.1 *Features*

Every sensor in the framework generates data and makes those data available to other software in the form of one or more *features*. The word "feature" is drawn from the field of pattern recognition, where it denotes an "individual measurable heuristic property of the phenomena being observed" [7]. In this case, the phenomena being observed is some internal state that the sensor is calculating, and the features the sensor exports are some useful properties of that state. The author of a sensor is responsible for defining which features are generated by the sensor.

A feature is similar to the idea of a *cue* [4], but is not required to be a simple derivation of one underlying stream of raw data. For example, a sensor representing a microphone might support features like average volume or highest-energy frequency band, which are both calculated from the audio samples. A sensor representing the user's social context, on the other hand, might use the audio samples from a microphone, in

addition to other sources of information like the user's daily schedule, to produce a feature that indicates whether or not it is a good time to interrupt him.

4.1.1 *Decoupling sensors from applications*

One of the goals of the WocketSensor framework is that the sensors and the software that uses the sensors are separated from each other and can only communicate with each other through the framework, using a strictly defined API. This ensures that sensor modules are re-usable and are not tied to any specific application. Features play an important role in this modularity, because they are used to communicate *about* the data a sensor produces, but they are abstracted from the data itself.

For example, a researcher might create a sensor to support some new application. Since all requests from the application must go through the framework, and features are the only way that sensor and client can communicate about what data should be produced, the sensor cannot tell the difference between the original application requesting those features and a new application that requests the same features.

4.1.2 *Feature identifiers*

One important note about time-dependent features is that the same feature can be calculated at many different sampling periods. For example, consider a sensor that represents an accelerometer. If the accelerometer is being sampled 1/ms in hardware, the sensor could produce a feature called "mean," which represents the average of the signals over some period of time. The time period for "mean" could potentially be anything longer than one millisecond.

For this reason, features are identified by a (*string,duration*) tuple. The string identifies the name of the feature, and the duration identifies which time period the feature is to be calculated over. Each sensor defines which feature names it supports, and for each feature name, the range of durations it supports. Requiring sensor developers to supply this information forces them to think in terms of what the sensor can do abstractly, instead of how to make the sensor produce the exact data needed for a particular application.

All features are identified by this tuple data type, even if having different durations for a particular feature makes no sense. For example, for a feature that only produces data when the user's social activity changes, a sensor might use the feature name "activity" and the period "0," even though the 0 has nothing to do with how often the activity is calculated. Obviously, this sensor would not support the "activity" feature and any other period.

4.2 *Feature streams*

The channels through which sensors send actual data to client applications are called feature streams. When a client application requests that a sensor start producing a feature, the framework opens a stream for that feature. Feature streams are a form of one-to-many publication: each feature has only one writer (the sensor producing data for it), but can have many readers (every client who is interested in that feature).

4.2.1 *Inter-process communication - memory mapped files*

In order to efficiently share feature streams across process boundaries, the process that contains the sensor and the process that contains the client application must be able to communicate. The WocketSensor framework uses an inter-process communication technique called "memory-mapped files" (MMFs) [9]. MMFs are intended to be an efficient way to load very large files in memory. Instead of copying the entire file into memory to read it, an MMF system call creates a "view" of that file by copying a portion of the file into physical memory and then mapping that memory to an unmapped part of the process' memory space. Subsequently, any changes the process makes to that memory will be written through to the file. If two processes have a view open on the same file, the memory the file is copied into will be mapped into both processes' memory spaces. Thus, if one process writes something to memory in the mapped space, that change is visible to the other process. This scales well to any number of processes because there is only one copy of the data in physical memory.

With most MMFs, each view is associated with a file in the file system. However, there is a significant amount of complexity and performance overhead involved in creating many files and writing changes in memory out to permanent storage. For this reason, the framework uses a special flag in the MMF system call to signify that there should be no physical file backing any of the feature streams; they exist solely in memory.

byte	int	short
sbyte	uint	ushort
double	long	
float	ulong	

Figure 3 supported primitive types

4.2.2 *Feature stream data types*

Feature streams (as represented by MMFs) are simply a contiguous

area of memory, or a series of bytes. Most features, however, will produce data that is more appropriately expressed in some other data type. For example, a feature that produces a likelihood might be expressed as a floating point number between 0 and 1, while a feature that produces the mean of a series of physical sensor readings might be expressed as a 16 or 32-bit integer. To promote ease of use for developers, the framework handles marshalling and unmarshalling to/from any of the primitive types listed in Figure 3. Note that these marshalling functions are included for convenience only. Since feature streams can be read from/written to as `bytes`, it is possible for sensors to do their own marshalling and use a more complex data type, as long as any client applications know how to decode the stream.

However, the sensor and client applications that use the sensor must agree on what types are being produced for each feature. **This aspect of the contract between sensor and client is not enforced by the framework**. For example, there is nothing keeping a sensor from writing `ints` into a stream while a client application reads `doubles` from that stream. Obviously, this would result in nonsensical behavior. Therefore, it falls to the developer of a sensor to document, along with which features are available, what type the output of those features will be.

4.2.3 *Timing guarantees and MMF memory management*

Sensors perform only one operation on feature streams: writing. To the sensor, a stream is indistinguishable from an infinitely large memory block that it writes to contiguously. The sensor produces data and adds it to the end of the stream by passing it to the framework. The framework, in order to minimize the amount of memory needed by

feature streams, implements streams as cyclical buffers on top of the fixed-size blocks allocated by MMFs. When it reaches the end of the block, it wraps around and continues writing from the beginning of the same block. Similarly, when a client application *reads* a feature stream, the framework reads linearly up to the end of the block then wraps around and starts reading from the beginning again.

The reason this approach is efficient for memory usage is that the data provided by sensor normally only needs to exist momentarily – long enough for any client applications to use those data to make a decision or calculate a higher-level result – and can then be discarded. The downside of this approach is that when the sensor writes information too quickly, it might wrap around and overwrite old data values before client applications have had the chance to read those values. To mitigate this disadvantage, the framework allows clients to specify a "minimum temporal requirement" (MTR) for a feature when they open a stream – a minimum amount of time which the framework is required to keep data before overwriting it. The framework keeps track of the longest MTR (over all clients reading a given feature) and - if it is danger of overwriting data that isn't old enough - reallocates a new MMF and copies all the data into it before continuing. To minimize time spent reallocating, the size of the MMF block is doubled with each reallocation. The default starting size of the MMF is small, so this algorithm is also efficient in space, since the amount of memory dedicated to holding the data will never be more than twice the amount required.

4.3 *Framework classes*

In keeping with the theme of simplicity, the framework contains only three classes that developers interact with. The first and most important is the abstract class

`WocketSensor`, from which all sensors in the framework must inherit. Inheriting from the `WocketSensor` class obligates all sensors to implement several abstract functions that form the core of the sensor author's API (discussed in sections 4.4 and 4.5). In addition to handling threading/timing issues, `WocketSensor` also contains the function that a client uses to open a feature stream and the functions that sensor authors use to write data into feature streams.

`Feature` is a simple placeholder data type, which contains the *(name,period)* tuple that identifies a feature, as well as a minimum time requirement field. `Feature` objects are passed by clients to the framework to identify which features should be opened, and they are passed by the framework to sensors to identify which features are open and should be calculated.

`FeatureStream` objects are generated by the framework and held by client applications. They encapsulate a byte-based stream that is ultimately backed by an MMF, and they are responsible for monitoring that stream to see when data are available to be read by client applications. They also have convenience reading functions that read the byte-based MMF stream as longer data types, as described in section 4.2.2.

Finally, the `Controller` class is a singleton class that coordinates every sensor in the machine, across and inside processes. This class is not visible to developers who use the framework; it is only responsible for internal book-keeping. The controller is a singleton, so only one instance can exist in memory in any given process. However, because each instance of `Controller` communicates with all the other processes that use the framework, every instance knows about all the sensors that exist on the machine

and every feature that is open on any of those sensors. `Controller` is the only piece of code that directly deals with inter-process communication and MMFs.

4.4 *The work function and working vs. non-working processes*

To allow the framework to guarantee that a sensor's useful work is being done as efficiently as possible, each sensor is required to implement a "work function." This function must have return type `void`, must accept as an argument a list of `Features` that need to be calculated, and must do these additional three things:

1. Gather data that is relevant to the request features from various sources,
2. Calculate new data points for each requested feature, and
3. Write those data points to the appropriate `FeatureStreams`.

After the sensor has been started and a feature stream has been opened by a client, the framework will launch a thread that periodically calls the work function. When it is called, the work function should gather all the data that it might need from its underlying data sources (physical sensor hardware, internet connection, user's contacts, etc.) and then, for each feature in the list, calculate as many feature values as the underlying data will allow. After it hands those data to the framework so that they can be distributed to the clients that want it, the work function should return immediately. This allows the framework to manage time intelligently by interleaving productive sensor work and necessary framework bookkeeping. Because it eliminates the need for sensor writers to create and manage their own threads, it also helps ensure that no sensor accidentally monopolizes the resources of the machine by inefficiently spinning while waiting for low-level resources to become available. Instead, the developer is presented with an interface that looks like a single shot: the work function should use whatever low-level

data are available at the moment to do useful work, and any further work is deferred to the next call to the work function.

Although the thread that calls the work function can be thought of as a loop that sleeps for some time (usually 100ms) and then calls the work function repeatedly, it also does other book-keeping work that the `WocketSensor` class needs to do periodically. For example, approximately every 2s, the sensor must ask the `Controller` whether any new features have been opened or closed by other processes. If they have, the sensor adjusts the framework's internal list of open features before passing that list to the work function so that the features can be calculated. This highlights one difficult concept involved in creating a sensor: the features being requested by clients can change in between one call to the work function and the next, so the work function must be flexible enough to adjust to those changes.

A sensor in the framework can be used concurrently by many different processes. Since each process has a copy of the sensor (it must have a copy in order to successfully compile and execute), the framework must ensure that only one of those copies is actually doing work. For each sensor, exactly one of the processes that are using that sensor is considered "working" at any given time. Working and non-working processes behave similarly, except that non-working processes do not call the sensor's work function. All other sensor book-keeping is performed normally (updating the open feature list, etc.), and any non-working process may become the working process if the former working process is closed. Normally the working process is whichever process instantiates the sensor first. However, in the case where two processes start at nearly the

same time or a working process is closed, the `Controller` decides which process becomes the working one.

4.5 *Sensor writer's API*

Although many of the individual parts of the API are explained above, this section is a summary of the classes and functions that the implementer of a new sensor must understand and implement to create a sensor module in the framework.

4.5.1 *Functions that must be implemented*

`WocketSensor` is an abstract class, and it has four functions that are unimplemented. When a sensor author makes a sensor class which inherits from `WocketSensor`, that class must implement all of these functions.

1. **`void` OnStart()** – When a client application tells a sensor it should start, the working copy of the sensor will call this function (on itself), before it starts calling its work function. This function should prepare any data sources that the sensor might use to calculate its features. For example, a sensor might want to open a communication channel with a piece of sensor hardware, or start a different WocketSensor whose results it depends on.
2. **`void` OnStop()** – Similarly, when no more clients are using the sensor, the framework will call this function, which should close anything that was opened when the sensor started.
3. **`bool` FeatureSupported(string name, TimeSpan period)** – This function is called by the framework when a client tries to open a feature. If it returns false, an exception is thrown by the framework. If it returns true, the feature can be opened.

4. **`void` CalculateFeatures(List<Feature> features)** – This is the work function, as defined in section 4.4. It must calculate, for each feature in the list, as many data points as possible, and then return immediately.

4.5.2 *Functions That May Be Overridden*

In addition to the four abstract functions above, `WocketSensor` has one virtual function that may be overridden to more carefully control the behavior of the sensor's work thread.

1. **`int` getSleepTimeMillis()** – Specifies the amount of time the sensor's work thread should sleep between calls to the work function, in milliseconds. The default implementation of this function simply returns 100. It may be overridden by sensors that are time-sensitive and must run more quickly, or by sensors whose timing requirements are more complex and may change over the life of the sensor.

4.5.3 *Public Functions Sensor Authors Will Use*

However sensor authors accomplish the calculation of features in their work function, they must all eventually send the calculated data to the framework so that it can be delivered to clients. `WocketSensor` has several overloaded versions of the same function that are used for that purpose:

1. **`void` writeFeatureValues(type[] buffer, int offset, int length, Feature f) –** This function is overloaded once for each type in Figure 3. Sensor writers call this function from their work function, after some data points have been calculated for the feature **f**. This function will add new data to the end of the

appropriate FeatureStream, which can then be read by the sensor's clients.

4.6 *Sensor client's API*

One major goal of the WocketSensor library is to make it as easy as possible to re-use sensors once they have been written. The functions below, along with a basic understanding of what features are and knowledge about what kinds of features a given sensor has, are all the user of a sensor needs to know to write an application that uses that sensor.

4.6.1 *WocketSensor functions*

Since WocketSensor is an abstract class, clients will call these functions on a sensor class that inherits from it, rather than on WocketSensor itself.

1. **void Start()** – Clients call this function to signal to the framework that the sensor should start producing values. Before this function is called and after the Stop() function is called, the sensor will not calculate any values, even if it has open FeatureStreams.
2. **void Stop()** – Clients call this function to signal to the framework that the sensor should stop producing values.
3. **FeatureStream OpenFeature(string name,**

```
TimeSpan minAcceptablePeriod,
TimeSpan maxAcceptablePeriod,
TimeSpan requiredOldnessParam)
```

   Clients use this function to open a new FeatureStream. While Features are normally identified by their (*name,period*) tuple, this function requires a range of possible periods rather than an exact period. This is to maximize possible sharing of open Features. For example, if two clients both want

the "mean" feature, but they each want the mean at a radically different period, the sensor must calculate both features: there is no overlap. On the other hand, if client A can use a mean with any period between 10 milliseconds and 30 milliseconds, and client B must use a mean with a period of 15 milliseconds (i.e. both max and min are 15 milliseconds), there is a potential to save processor time by calculating only one feature (the mean at 15 milliseconds) and sharing that feature with both clients.

When this function is called, the framework will first search for any already-open feature that matches the period requirements. If no match can be found, the framework generates a new `Feature` with a period that is the average of the `min` and `max` arguments and adds it to the list of open features so that it will be calculated the next time the sensor's work function is executed. The last parameter lets the client control how long the framework guarantees those data values will be in memory before they may be overwritten, as in section 4.2.3.

4.6.2 *FeatureStream properties and functions*

Once a client has a FeatureStream, using it is straightforward. FeatureStreams obey the usual stream semantics, with the exception that writing to the stream is not supported (all the writing is done on the sensor side; from the client's point of view these streams are read-only). FeatureStreams have 3 properties and 1 overloaded function:

1. **Available** - This property returns the number of bytes that are in the stream, ready to be read.

2. **Feature** – This is just the (*name,period*) tuple that identifies the Feature for which this stream was opened.
3. **Period** – This is a convenience property that clients can use if they passed a range of possible periods to OpenFeature(), to see which one was ultimately used by the framework.
4. **`int` ReadType (type[] buffer, int offset, int length)** – This function is overloaded once for every type in Figure 3 (although they are not technically overloaded, since none use the name "ReadType." Instead, there is a ReadInts() function, a ReadDoubles() function, etc.). Clients call this function to read data out of the `FeatureStream` and into a local buffer that they can use directly. Note that after data have been read from a stream with this function, that data are considered to be *consumed*, i.e. the `Available` property will be reduced by the appropriate amount and subsequent calls to ReadType() will return only data that was not already read by this call, even if the stream is being read by two different objects. Also note that if two different objects **each** call OpenFeature with the same arguments and each get a `FeatureStream`, one object reading its own stream will **not** consume the data in the other object's stream, even though both streams will contain the same data. In general, every object that needs to independently use the output of a sensor should call OpenFeature to get its own `FeatureStream` object, since the cost of adding another read-only copy of the stream is minimal.

5. Implementation

The framework was implemented and tested on an HTC P3300 mobile phone running Windows Mobile 6 Professional. It is written entirely in C#, and it requires that the .NET Compact Framework 2.0 be installed. The only external dependency (aside from the Memory-Mapped File library below) is a common logging library for C# on mobile phones called NLog [10].

For performing Memory-Mapped File system calls in C#, the framework uses an external library developed by Tomas Restrepo, called *Win32 FileMap Wrapper* [8]. This library is released under the GNU LGPL license, which should present no legal restrictions to developers who use the framework.

Microsoft Visual Studio 2005 [11] was used as the development environment.

5.1 *Obtaining the software*

The WocketSensor code is hosted on Google Code and can be checked out via SVN at https://wockets.googlecode.com/svn/trunk. Instructions for compiling and running an example sensor are available at http://web.mit.edu/mobirnd/wocketsensors.

6. Tests

To measure how well the framework achieved the goals in section 3, several tests were used. Quantitative measures include tests for maximum throughput, scalability, and data loss. All quantitative tests were run on the device described in section 5, and were launched from Visual Studio in Debug mode.[1] A qualitative test of usability was also

[1] Debug code is compiled without optimizations, and so may be slower than release code. However, in a multi-threaded application like the WocketSensor framework, this difference in speed can produce timing problems (race conditions, deadlock, etc.) when the same code is compiled for release. The system still needs to be tested in release mode.

conducted by asking developers who were not familiar with the framework to create a sample sensor.

6.1 *Maximum throughput test*

This test was designed to measure how well the framework works with high-throughput sensors. There is a certain amount of overhead inherent in using a sensor in the framework rather than a custom-coded solution. The goal of this test is to show that the throughput of a sensor in the framework is high enough – and the overhead is low enough – to support sensor modules that generate a large volume of data.

An example sensor was created that produced random integers as quickly as possible. The wait time between calls to the work function was set to 0, and the work function calculated 512,000 random 32-bit integers each time it was called (a total of 2,000 kilobytes). This number was chosen somewhat arbitrarily to be several orders of magnitude higher than the typical amount of data generated by a work function (for comparison, a sensor which represents a 3-axis accelerometer sampled at 350 Hz and that sleeps 100ms between each execution of the work function generates around 300 bytes or less, depending on which features are open). A large number was chosen to ensure that the sensor spent nearly all of its time actually generating data, as opposed to working on framework bookkeeping tasks.

The test application that was written to use this sensor opened the test feature's `FeatureStream` and tried to read that stream as quickly possible (a data value was considered "read" when it had been copied into a local buffer the test application had allocated beforehand). The time it took to read 512,000 integers was recorded, and this trial was repeated many times.

On average, a program using the sensor could consume 10,660 integers per second, which equates to a maximum throughput of 41.6 KB/s. To measure the effect of the framework on this number, the same test was conducted with an application that did not use the framework. Instead, it independently generated 512,000 random integers and put them into local storage and measured the time it took. The throughput using this method was 250.4 KB/s on average, around six times higher.

Although the test was expected to show that the framework had a reduced throughput, the difference was expected to be around 2-fold (since each data value must be copied twice before it can be used directly by a client application). The remainder of the difference is likely due to the inefficient method that the framework uses to copy values into `FeatureStreams`, which could potentially be improved. Those improvements are discussed in more depth in section 7.

Even though the framework doesn't compare favorably with a custom-built solution in throughput, the sustained throughput using the framework was still an order of magnitude higher than required for the 3-axis accelerometer sensor mentioned above, and that sensor works in real time. Applications that use the framework will not experience any speed issues until they approach the throughput limit.

6.2 *Scalability test*

One of the major advantages of using the framework is that data generated by sensors can be shared by many applications with very little overhead. This test was designed to measure how quickly client applications could consume sensor data that were being shared, compared to how much data they could consume by calculating it independently (and concurrently).

For this test, a sensor was designed that produced a very small quantity of data. Each execution of the work function on this sensor calculated the sum of 700,000 random doubles and passed that single number to the test application. As before, the test application simply read the data and measured how long it took to calculate each value. However, for this test, the application opened several threads, each of which opened the same feature on the sensor.

Not surprisingly, each of the threads reported the same average time for each data value, regardless of how many threads were open (trials were conducted with up to 10 threads). On average, it took 12 seconds for the sensor to calculate each data value.

For comparison, the sensor code was executed in several concurrent threads in an application that did not use the framework. There was a linear relationship between how many threads were running concurrently and the average amount of time it took to calculate each data value: one thread by itself took 12s, two threads running together took 24s, three threads took 36s, and so on.

This shows that, especially for sensors whose ratio of processing time to unit throughput is high, the framework is a very efficient way to share sensor data.

6.3 *Data loss/timing test*

This test was designed to validate the framework's interface guarantee with regard to the "minimum temporal requirement" that clients can specify when opening a feature. The guarantee is that, although the framework is continuously reusing memory to minimize its overall memory footprint, it will never overwrite data before those data have been in the stream long enough to satisfy the MTR.

An example sensor was designed which generates a series of sequential integers. The sleep time between calls to the work function was 100ms and the work function produced 100 sequential numbers each time it is called. The test application that used this sensor opened a FeatureStream, specifying an MTR of between .5s and 10s. For each MTR value, the test application slept for 100ms less than that value, and then tried to read values from the stream. If a value was read that was not in sequential order, the test application was designed to throw an exception and terminate. In two hours of testing (with the test interrupted periodically to change the MTR), zero exceptions were noted.

6.4 *Robustness test*

The code from the data loss test was reused to show that the framework was sufficiently free of defects to run for long periods. It was ideal for this test because it had moderate throughput and processor requirements that were more realistic models of potential sensors. It also did nothing exotic with memory or anything else in the sensor module itself or the test application, which was necessary to show that any memory leaks or other errors that the test generated were a result of the framework and not of the individual sensor.

For this test, the data loss test was run continuously for two days (the phone was left plugged in, both to save the battery and to keep it from entering sleep mode). At the end of the test, the application was terminated normally, and no exceptions were noted. The phone responded normally to periodic user input throughout.

6.5 *Usability test*

One of the most important goals of the framework is that it should be easy to use, both for developers who are creating sensor modules and for developers who are using

those sensor modules. To test this, two researchers from the House_n group were asked to create a new sensor module. Both had excellent programming backgrounds and a firm grounding in sensor technologies, but no prior experience with the framework. To ensure that they were exposed to both sides of the API, the sensor they created was meant to be a high-level sensor which used an existing, lower-level sensor as its data source. Specifically, the existing sensor represented a Bluetooth accelerometer module. It supported nine features: the raw X, Y, and Z acceleration values plus their means and variances over a given time period between 50ms and 500ms. To simplify the task, the subjects were asked to write a sensor that supported only one feature: The correlation coefficient between the streams of X values and Y values, over the same time period. Correlation was chosen because the calculation of correlation uses the mean and the variance, which would give the subjects a chance to use the client-side API while they were writing their sensor using the sensor-side API.

Each of the subjects had no prior experience with the framework, beyond a general idea of its purpose. Each subject was given the instructions in Appendix A, and they were instructed that they should try to create the sensor using only the information given, but to ask questions if they found themselves confused at any point. These questions were noted, and then an explanation was given. No further interactions were initiated by the tester, who only answered the subjects' questions. At the end of the task, the subjects were asked for their impressions of the difficulty of the task, along with their estimate of how much time they spent on the task.

The first subject reported that it took 1-2 hours to write the correlation sensor. The second spent around the same amount of time, but didn't complete the sensor.

Most of the difficulties the subjects had with the framework were caused by omissions from the instructions, and were easily explained. For example, both subjects were confused by the minimum/maximum periods in OpenFeature, since the instructions only mentioned them in passing, without explaining why two periods were required. Also, both subjects noted confusion about the sleep time, and wondered whether it was related to the period of the features. Their comments were useful in preparing an updated set of instructions, included in Appendix B.

However, user testing also exposed some areas of the framework that were not intuitive. For example, one subject wondered whether it was necessary to start a sensor before opening features, or if the features should be opened before starting. The "state" of the sensor isn't clear from the description of Start(), Stop(), and the constructor, and the subject wasn't sure what the OnStart() and OnStop() functions should do. One subject also noted it was confusing that a `FeatureStream` didn't have a data type associated with it, since the instructions never mentioned that the sensor's author had to publish the type of the data (and in fact doesn't mention that the type of the data it produces is `short`). Both subjects were confused by the fact that reading from a stream consumes the data in that stream. The instruction's don't explain clearly enough that "sharing" the sensor refers to many objects opening their own copy of the feature, not to many objects sharing a reference to one `FeatureStream`.

Finally, the test (especially in asking the subjects to relate what they think of the framework as a whole) also exposed shortcomings in the design of the framework as a whole, which may impact its utility as a sensor platform. These were the most serious concerns, and both of them were related to how the framework handles time.

First, one subject noted that he was unable to correctly calculate the correlation, because if two streams of the raw acceleration data were opened, there was no guarantee that successive values in each stream were actually tied to each other in time. That is, even though the X values and the Y values come from the same underlying data stream, they were split into two different features and since there is no way to "tie" features together in time, users of the sensor can only approximate synchronization of two different `FeatureStreams`.

Second, there is no way for a client application to tell how old the data in a stream are when being read, and data are only cleared from a stream on a read operation. That means that clients must read it continuously. Otherwise, they may encounter several timing problems, including reading "stale" data (data produced at a much earlier time that are no longer relevant), discontinuities in data (allowing a time larger than the MTR to elapse – giving the framework time to overwrite values in the stream – before reading them), and getting "bursts" of data that are difficult to correlate in time (the sensor produces data continuously at a low bandwidth, the client reads data occasionally at a much higher bandwidth).

Many sensor frameworks, like the work of Schmidt, et al. [4], do not have timing considerations like this because the endpoint of all the sensors is known ahead of time – the context layer is free to make assumptions about how the underlying sensor modules behave temporally. MyExperience [5] solves this temporal issue by asynchronously issuing a call into the framework every time the state of the sensor changes, which means that every data point the sensor produces can be correlated exactly in time. However, this is unfeasible in the WocketSensor framework because there is no way to make a function

call in real time across a process boundary. Even if it were possible, doing so for every data point would create an intolerable amount of overhead for a high-bandwidth sensor.

Both of these issues could be solved by "marking" the data coming from a sensor with timestamps. It is possible for sensors whose output must be correlated precisely in time to write a timestamp at whatever point makes the most sense for those data (every sample, every five samples, etc.). Of course, this is inelegant and cumbersome for sensor writers, so the framework could also be extended to support a "time stamped" version of `FeatureStream`, which simply inserts a timestamp every time the sensor writes to the stream and reports that time whenever a client reads from the stream – in effect, tying writes and reads together in a 1-to-1 way where the write time is always recorded.

The only inherent limitation on temporal information that is part of the framework's design is the MTR. Client applications must read the stream at least as often as they promise to when they specify the MTR, or risk losing old data. However, there is no limitation of the length of the MTR, so if the sensor is specifically designed to be very low-bandwidth a client application can specify a long MTR and check the stream whenever it wants to.

Despite the limitations expressed during the user testing, both subjects were able to grasp the concepts used to build the framework after explanation, and thought that it was feasible to use the framework to develop reusable sensors. However, they both thought the instructions needed improvements before they would be useful to new developers. One subject used the code of the supplied sensor as a template. The other subject didn't, but said that having an example sensor would be the easiest way to understand some of the framework concepts.

7. Conclusions and future work

The WocketSensor framework builds on the work of several disparate research efforts in mobile sensors, combining many powerful ideas into a single software library. The resulting work is a platform that supports re-usable sensor modules that can be shared efficiently between many different processes. The efficiency properties were demonstrated by tests built within the framework. Usability was tested by asking developers unfamiliar with the framework to create a sensor. The usability test demonstrated several problems with the design of the framework, and especially with how the framework was explained to new developers.

There are three specific ways that the framework can be improved, based on the findings in section 6. First, a version of `FeatureStream` that is time stamped, as discussed above, would help developers deal with time-sensitive sensors and with multiple features that should be synchronized in time. This would require modification of the code that writes to streams as well as the code that reads from streams. Essentially, each write operation would be written into memory as three components: a timestamp, an amount of data, and a payload of the data itself. Conversely, each read operation would read exactly one payload's worth of data, along with the timestamp metadata.

Second, the way that data are copied to a `FeatureStream` when a sensor calls the writeFeatureValues() function is inefficient. This is because a function call is generated for every single element of the array that the sensor is trying to copy into the stream. This is enforced by the stream abstraction that the Win32 FileMap Wrapper library uses, which doesn't allow direct access to the destination buffer. This efficiency improvement would require significant code revision, again to the code in

writeFeatureValues() that actually copies data from the sensor's working memory into the memory backing the MMF that is the basis for `FeatureStreams`. Specifically, instead of using a `Stream` wrapper around the memory block, as the Win32 FileMap Wrapper library does, the pointer to that memory block which is returned by the MMF system call [9] would be written to directly. By eliminating the cost of a function call for each data point, this modification could reduce the 6x cut in maximum throughput to around the expected 2x.

Finally, it was noted in section 4.2.2 that the framework does not enforce a data type on any given `FeatureStream`; data from any stream can be read as any data type. This was a design decision made early on, to ensure that the type of data generated by sensors was not limited to the set of built-in types supported by C#. However, the usability value of opening a feature and getting a stream that explicitly reads the correct data type is substantial. Adding type metadata to the FeatureStreams, as long as a special "other" type is supported for complex data structures, is a feature that can be added to the framework with relative ease and few usability side-effects.

Acknowledgements

The author would like to thank the members of the House_n Research group and Jon Froelich, the founder of the MyExperience open source project, which was used in this work. Portions of this research were funded by NIH Grant #1R21LM008553, NIH Grant #1U01HL091737, and the MIT House_n Research Consortium.

References

[1] n.a. (n.d.). Wockets: Open Source Accelerometers for Phones. Retrieved January 27, 2009. http://web.mit.edu/wockets/

[2] Chen, G. and Kotz, D., A Survey of Context-Aware Mobile Computing Research, tech. report TR2000-381, Dept. of Computer Science, Dartmouth College, Hanover, N.H., 2000.

[3] Gellersen, H., Schmidt, A., Beigl, M. Multi-sensor Context-Awareness in Mobile Devices and Smart Artifacts. *Mobile networks and applications, vol. 7,(pp. 341-351).* 2002

[4] Schmidt, A., Aidoo, K. A., Takaluoma, A., Tuomela, U., Van Laerhoven, K., and Van de Velde, W. Advanced interaction in context. In H. Gellersen (Ed.) *Proceedings of First International Symposium on Handheld and Ubiquitous Computing, HUC '99*, (pp. 89-101). Springer.

[5] Froehlich, J., Chen, M., Consolvo, S., Harrison, B., Landay, J.: MyExperience: A system for in situ tracing and capturing of user feedback on mobile phones. In: *Proc. Int. Conf. Mobile Systems, Applications, and Services (MobiSys)* (June 2007)

[6] n.a. (August 8, 2008). GPS Intermediate Driver. *Windows Embedded Developer Center.* Retrieved January 27, 2009. http://msdn2.microsoft.com/en-us/library/bb202086.aspx

[7] Yoon, S., Intel Corporation (n.d.). Technologies and Analysis Methods for Detecting Gene Expression by DNA Microarrays. *IEEE Explore.* Retrieved January 21, 2009. http://ieeexplore.ieee.org/techsurvey/ts2_section3.jsp

[8] Restrepo, T. (n.d.). DotNET: Filemap. *Winterdom.* Retrieved on January 27, 2009. http://winterdom.com/dev/dotnet/index.html

[9] Kath, R. (February 9, 1993). Managing Memory-Mapped Files in Win32. *Microsoft Developer Network.* Retrieved January 27, 2009. http://msdn.microsoft.com/en-us/library/ms810613.aspx

[10] Kowalski, J. (n.d.). Introduction to NLog. *NLog: A .NET Logging Library.* Retrieved January 29, 2009. http://www.nlog-project.org/

[11] n.a., Microsoft Corp. (n.d.). Visual Studio 2005. *Visual Studio 2005 Developer Center.* Retrieved January 29, 2009. http://msdn.microsoft.com/en-us/library/ms950416.aspx

Appendix A: Usability test – subject instructions

Thank you for agreeing to test out the Wocket Sensor Library.

In the real world, the word "sensor" generally refers to the physical electrical object that is capable of sensing some properties of the world. In this software framework, we will be using the word "sensor" to mean a software construct that provides a stream of data to other software. This stream of data is usually at least partly based on the readings of an actual physical sensor, but that is not a requirement.

Specifically, a sensor is a class which extends the class WocketSensor. Each sensor consumes data from various sources (physical sensor hardware, other software "sensors", internet, etc.) and produces streams of "features" of those data. These are meant to be features in the machine learning sense, i.e. a descriptive property of some piece of data that is expressed in a short and easily comparable form. For example, useful features of the data sampled from a microphone might be the Fourier transform, the overall volume of the signal, or even something specific like the probability that the audio signal is a sample of someone speaking the words "yes" or "no".

In the WocketSensor framework, a Feature is defined to be a (name,period) pair, where the name is a string representing the name of a feature and the period is the inverse of the frequency of the feature. For example, ("volume", .002 seconds) would mean a feature where the volume is calculated once every .002 seconds, or 500 Hz. We use period rather than frequency because it tends to be easier to represent and use in software. Note that although two features may share the same name, they are distinct if their periods are different (i.e. volume reported every .002 seconds is distinct from volume reported every .2 seconds).

Each sensor that is written for the framework has a set of pre-defined feature names and period ranges that it can support. The author must define which features are allowed, and must ensure that the sensor is capable of calculating any number of allowed features.

Conceptually, the framework operates by managing "streams" between the sensor and users of the sensor.

-As a user of the sensor, you open a stream for some feature and read it.

-As the author of a sensor, you are given a list of features that users of your sensor have opened, along with a stream for each one, and are expected to calculate values for those features and write those values to the feature's stream.

Your goal for this test is to create a sensor that calculates the correlation between the three axes of an accelerometer. To accomplish this task, you will be writing a new sensor which supports just one feature name: "correlationXY" (the other two correlations will be omitted because they are obvious after you've finished the first). Your sensor must be able to calculate this feature at any period between .05 and .5 seconds (2-20Hz), using a discrete (non-sliding) window. Although your sensor only supports one feature name,

keep in mind that it must be able to calculate that feature name at an arbitrary number of periods (frequencies).

Below you'll find the API for the sensor framework and the underlying hardware/sensors you'll be using to implement this sensor

Sensor User API

If you are writing software that uses a sensor in the framework, here is the API you will use to get data from it.

WocketSensor methods:
Constructor:
Call the sensor's constructor to get an instance of that sensor
Start():
Call Start() when you are ready for the sensor to start producing data

OpenFeature(string name, TimeSpan minPeriod, TimeSpan maxPeriod, TimeSpan oldness):
Call OpenFeature() to get a **FeatureStream**. A FeatureStream represents a read-only view of a stream of data. Each discrete data point in this stream represents exactly 1 value of the feature (for example, every point in the "meanX" stream of the WiTiltSensor is the mean of the underlying acceleration values over the feature's period length).

To open a feature, you must specify its name as well as a range of periods that you are willing to accept. The period of the feature that is actually opened can be read via the **Period** property on the FeatureStream.

Finally, you must specify an "oldness" requirement for the feature. This is a guaranteed length of time that the sensor will keep old sensor values in memory before discarding them and using that memory for new values. This is generally related to how long your software usually goes between calls to FeatureStream.Read() – i.e. if you are only reading the feature every 500 milliseconds, make the oldness requirement 750 milliseconds or 1 second. Keep in mind that raising this value will cause the sensor to use linearly more memory.

Stop():
Call Stop() when you are done with the sensor. After this call, FeatureStreams will stop producing data

FeatureStream methods/properties:
Available:
Use this property to determine how many data points are available in the stream
Period:
Use this property to determine which period (in the range of allowable periods) that this stream's feature actually opened
ReadInts(), ReadShorts(), ReadDoubles(), etc…:

Call these to read data from the stream. Note that the particular method you call is determined by the output type of the feature. The WiTiltSensor uses the **short** data type for all its features.

Sensor Author API:
As the author of a new sensor which inherits from WocketSensor, here is the API you will be expected to implement and use:

Constructor:

Your constructor should instantiate other objects you need, but your sensor should refrain from opening any communication channels/starting other sensors. It should still be in a "non-started" state (a state where the sensor might or might not be able to calculate any of the features it supports).

OnStart():

You must implement this abstract method. This method is called by the framework when a user of your sensor calls Start(). It should open any communication streams, start any other sensors it is using, and generally do whatever is necessary to put the sensor into a "started" state (one where it is capable of calculating any features that it supports)

OnStop():

You must implement this abstract method. Similar to OnStart(), this method should close/stop any resources that the sensor is using

bool FeatureSupported(string name, TimeSpan period):

You must implement this abstract method. It should return true for (name,period) pairs that are supported and false for pairs that are not

CalculateFeatures(List<Feature> features):

You must implement this abstract method. This is where most of the work of the sensor is accomplished. This method will be called periodically by the framework. It is expected that by the end of this method, the sensor will have written out up-to-date data for all the features in the list.

As part of the operation of the sensor, this method should first read any communication channels/other sensors that your sensor is using. The list of features that is passed to this method by the framework represents all the features that have been opened via **OpenFeature** by users of your sensor. When it has up-to-date data from all of its data sources, it should, for each feature in the list, calculate as many data points as possible and pass those data points back to the framework by calling **writeFeatureValues()**

writeFeatureValues(array[], Feature f):

Call this method when you have finished calculating values and are ready to pass them to the users of your sensor. There is a version of this method available for each of the C# integral types (int[], short[], double[], etc…)

Along with the data values you've calculated you must also pass the feature you are calculating so that the framework knows which stream to route the values to

getSleepTimeMillis():

You may override this method. It specifies how long the framework will sleep (in milliseconds) between calls to **CalculateFeatures**(). If you don't override it, the default value is 100

The Sensor

To get physical acceleration data, we will be using the Sparkfun WiTilt sensor over a Bluetooth link. There is a pre-existing software sensor class (written using the WocketSensor framework) called WiTiltSensor which represents this hardware.

The WiTiltSensor supports the following features:

1. rawX
2. rawY
3. meanX
4. meanY
5. varianceX
6. varianceY

The raw signal features are only supported at the actual reporting period of the hardware: 1/350 Hz ~= 2.9 milliseconds (this value is represented as the static variable WiTiltSensor.RAW_SIGNAL_PERIOD). The other features are supported at periods between 10 milliseconds and 1 second, and are based on discrete (non-sliding) windows.

The Task

Your sensor must calculate the correlation between the three axes of the accelerometer. The formula for correlation of two discrete signals is:

$$\frac{1}{N}\sum_{i=1}^{N}\left(\frac{X_i-\mu_x}{\sigma_x}\right)\left(\frac{Y_i-\mu_y}{\sigma_y}\right)$$

Where N is the number of samples in the window, X and Y are the samples, mu_x and mu_y are the means of those samples, and sigma_x and sigma_y are the standard deviations of those samples. You should end up using the raw, mean, and variance features for the X and
Y axes on the WiTiltSensor to calculate the correlation (remember standard deviation is the square root of variance).

Getting the code

To check the code out from the wockets repository on Google Code, use SVN with the following parameters:

Repository Location: https://wockets.googlecode.com/svn/trunk

Appendix B: Updated instructions

Thank you for agreeing to test out the Wocket Sensor Framework.

In the real world, the word "sensor" generally refers to the physical electrical object that is capable of sensing some properties of the world. In this software framework, we will be using the word "sensor" to mean a software construct that provides a stream of data to other software. This stream of data is usually at least partly based on the readings of an actual physical sensor, but that is not a requirement.

Specifically, a sensor is a class which extends the class WocketSensor. Each sensor consumes data from various sources (physical sensor hardware, other software "sensors", internet, etc.) and produces streams of "features" of those data. Features are defined as "individual, measurable heuristic properties of the phenomena being observed". For example, useful features of the data sampled from a microphone might be the Fourier transform, the overall volume of the signal, or even something specific like the probability that the audio signal is a sample of someone speaking the words "yes" or "no".

In the WocketSensor framework, a Feature is defined to be a (name,period) pair, where "name" is a string representing the name of a feature and "period" is a period of time over which the feature should be calculated. For example, ("volume", .002 seconds) would mean a feature where the volume is calculated once every .002 seconds, or 500 Hz. We use period rather than frequency because it tends to be easier to represent and use in software. Note that although two features may share the same name, they are distinct if their periods are different (i.e. volume reported every .002 seconds is distinct from volume reported every .2 seconds).

Your goal for this test is to create a sensor that calculates the correlation between the three axes of an accelerometer. To accomplish this task, you will be writing a new sensor which supports just one feature name: "correlationXY". This is defined as the statistical correlation between the raw acceleration values on the X axis and those on the Y axis (the other two correlations, YZ and ZX, do not need to be implemented). Your sensor must be able to calculate this feature at any period between .05 and .5 seconds (2-20Hz), using a discrete (non-sliding) window. Although your sensor only supports one feature name, keep in mind that it must be able to calculate that feature name at an arbitrary number of periods.

Each sensor that is written for the framework has a set of pre-defined feature names and period ranges that it can support. As the author of a feature, you must define which Features are allowed to be opened, meaning both which feature names your sensor supports and what range of periods is valid for each feature. You must also ensure that your code is capable of producing any number of those features at the same time, because there is no way to know ahead of time what features the users of your sensor are going to open. Specifically, your code must be able to calculate the same feature at two different periods simultaneously, as in the volume example above.

Each sensor exists in one of two states. In the "running" state, the sensor is gathering data from its constituent parts and calculating Feature values from that underlying data. In the "stopped" state, the sensor does no work. Client applications that use the sensor can control this state by calling Start() and Stop() on the sensor. When a sensor is initially instantiated, it is in the "stopped" state until some application calls Start().

Each sensor has a "work function", which processes all the sensor's open features. As a feature author, you must implement this work function (called CalculateFeatures()). The work function returns no value and takes in a list of open Feature objects. Whenever the sensor is in the "running" state, it has an internal thread open. This internal thread both performs the internal bookkeeping work of the framework and calls the work function at fixed intervals. The interval the thread uses can be changed by overriding the getSleepTimeMillis() function. This function defines how long the thread sleeps in between each call to your work function. If not overridden, the default value is 100ms.

When implementing the work function, your code should do three things:

1. Gather data from underlying data sources,
2. Calculate, for each Feature in the openFeatures list, as many data points as the underlying data will allow, and
3. Write the calculated data points for each Feature back into the framework so that they can be distributed to client applications that have opened that Feature.

Because this test is designed to test the usability of the framework as a whole, you will also be using the client API, which lets user applications open features on a sensor and read feature values from it. The sensor you will be using – the WiTiltSensor – was written in the framework, and represents the output of a wireless WiTilt accelerometer from SparkFun. The sensor exports features that will are useful in calculating the correlationXY feature.

The framework is designed such that many independent client applications can use the feature values produced by a single sensor. Conceptually, the framework works by managing "streams" between sensor that are producing data and clients that are consuming data. In the framework, these are called FeatureStreams. Each time a client application opens a Feature on some sensor, it gets back a FeatureStream that it can read to get the values of that feature. A FeatureStream can be thought of as a one-to-many publishing model, where the sensor dumps data on one end of the stream and several clients read data from the other end of the stream. However, because each open Feature can correspond to many open FeatureStreams (one for each client application that opened that Feature), it's more like the sensor is writing simultaneously to every FeatureStream open for that Feature, and the client who opened those FeatureStream each read values from them at their own independent rates. Each FeatureStream should correspond to exactly one reader, since values are consumed after they are read from the streams. However, the cost of opening many readers is minimal if they are all opening the same Feature.

Also, you should note that FeatureStreams are represented internally as a sequence of bytes, and can be read by clients as any of the fundamental C# data types. You must make sure that you are reading values from FeatureStreams as the same type that the sensor is writing them. For example, the WiTiltSensor uses only `short` data types, since it has a 10-bit A/D converter.

Below you'll find the API for the sensor framework and the underlying hardware/sensors you'll be using to implement this sensor

Sensor Client API

If you are writing software that uses a sensor in the framework, here is the API you will use to get data from it.

WocketSensor methods:

Start():

Call Start() when you are ready for the sensor to start producing data. It will enter the "running" state and start calling its work function.

Stop():

Call Stop() when you no longer need values from the sensor. It will enter the "stopped" state and stop calling its work function.

OpenFeature(string name, TimeSpan minPeriod, TimeSpan maxPeriod, TimeSpan oldness):

Call OpenFeature() to get a **FeatureStream**. To a client, a FeatureStream represents a read-only view of a stream of data. Each discrete data point in this stream represents exactly 1 value of the feature (for example, every point in the "meanX" stream of the WiTiltSensor is the mean of the underlying acceleration values over the feature's period length).

To facilitate many clients sharing any open streams, you are required to specify a range of possible periods that you find acceptable for this feature. If the sensor happens to already be producing a Feature that has the same name and whose period is in the range you specify, no new Feature will be opened, but a new FeatureStream containing the data of the already open Feature. Otherwise, a new Feature will be opened which uses the average of the min and max periods you supplied.

Finally, you must specify an "oldness" requirement for the feature. This is a guaranteed length of time that the sensor will keep old sensor values in memory before discarding them and using that memory for new values. This is generally related to how long your software usually goes between calls to FeatureStream.Read() – i.e. if you are only reading the feature every 500 milliseconds, make the oldness requirement 750 milliseconds or 1 second. Keep in mind that raising this value will cause the sensor to use linearly more memory.

FeatureStream methods/properties:

Available:

Use this property to determine how many bytes are available in the stream. Keep in mind that this does not correspond exactly to the number of available data points, because there may be many bytes in a single data point which is some other fundamental type (for example, `ints` are four bytes, while `shorts` are two.

Period:

Use this property to determine which period (in the range of allowable periods you specified) that this stream's feature actually opened.

ReadInts(), ReadShorts(), ReadDoubles(), etc…:

Call these to read data from the stream. Note that the particular method you call is determined by the output type of the feature. The WiTiltSensor uses the **short** data type for all its features.

Sensor Author API:

As the author of a new sensor which inherits from WocketSensor, here is the API you will be expected to implement and use:

Constructor:

Your constructor should instantiate other objects you need, but your sensor should refrain from opening any communication channels/starting other sensors. It should still be in a "stopped" state.

OnStart():

You must implement this abstract method. This method is called by the framework when a user of your sensor calls Start(). It should open any communication streams, start any other sensors it is using, and generally do whatever is necessary to put the sensor into the "running" state (one where it is capable of calculating any features that it supports)

OnStop():

You must implement this abstract method. Similar to OnStart(), this method should close/stop any resources that the sensor is using, returning the sensor to the "stopped" state.

bool FeatureSupported(string name, TimeSpan period):

You must implement this abstract method. It should return true for (name,period) pairs that are supported and false for pairs that are not.

CalculateFeatures(List<Feature> features):

You must implement this abstract method. This is the sensors "work function", as defined above. This method will be called periodically by the framework. It is expected that by the end of this method, the sensor will have written out up-to-date data for all the features in the list.

As part of the operation of the sensor, this method should first read any communication channels/other sensors that your sensor is using. The list of

features that is passed to this method by the framework represents all the features that have been opened via **OpenFeature** by users of your sensor. When it has up-to-date data from all of its data sources, it should, for each feature in the list, calculate as many data points as possible and pass those data points back to the framework by calling **writeFeatureValues()**

writeFeatureValues(array[], Feature f):

Call this method when you have finished calculating values and are ready to pass them to the users of your sensor. There is a version of this method available for each of the C# fundamental types (int[], short[], double[], etc…)

Along with the data values you've calculated you must also pass the feature you are calculating so that the framework knows which stream to route the values to

getSleepTimeMillis():

You may override this method. It specifies how long the framework will sleep (in milliseconds) between calls to **CalculateFeatures**(). If you don't override it, the default value is 100

The Sensor

To get physical acceleration data, we will be using the Sparkfun WiTilt sensor over a Bluetooth link. You will be supplied with a software sensor class (written using the WocketSensor framework) called WiTiltSensor which represents this hardware.

The WiTiltSensor supports the following features (each with the `short` type):

7. rawX
8. rawY
9. meanX
10. meanY
11. varianceX
12. varianceY

The raw signal features are only supported at the actual reporting period of the hardware: 1/350 Hz ~= 2.9 milliseconds (this value is represented as the static variable WiTiltSensor.RAW_SIGNAL_PERIOD). The other features are supported at periods between 10 milliseconds and 1 second.

The Task

Your sensor must calculate the correlation between the three axes of the accelerometer. The formula for correlation of two discrete signals is:

$$\frac{1}{N}\sum_{i=1}^{N}\left(\frac{X_i-\mu_x}{\sigma_x}\right)\left(\frac{Y_i-\mu_y}{\sigma_y}\right)$$

Where N is the number of samples in the window, X and Y are the samples, mu_x and mu_y are the means of those samples, and sigma_x and sigma_y are the standard deviations of those samples. You should end up using the raw, mean, and variance features for the X and Y axes on the WiTiltSensor to calculate the correlation (remember standard deviation is the square root of variance).

Getting the code

To check the code out from the wockets repository on Google Code, use SVN with the following parameters:

Repository Location: https://wockets.googlecode.com/svn/trunk