9-2015

# The Sample Stream Abstraction for Coordinated Energy-efficient Access to Peripheral Devices

Alexei Colin
*Carnegie Mellon University*, acolin@andrew.cmu.edu

Ragunathan (Raj) Rajkumar
*Carnegie Mellon University*, raj@ece.cmu.edu

# The Sample Stream Abstraction for Coordinated Energy-efficient Access to Peripheral Devices* †

Alexei Colin
acolin@andrew.cmu.edu

Ragunathan (Raj) Rajkumar
raj@ece.cmu.edu

Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA

## ABSTRACT

Battery-powered applications that are coupled to their physical environment through multiple sensors and actuators may not be using their scarce energy in the most efficient way. Acquiring samples from peripherals for fusion by the application software must be coordinated in order to not spend energy on redundant computations. However, coordination, as any sophisticated peripheral control, is a burden to the application developer that detracts from the development of the core logic. We propose the *sample stream abstraction* for accessing peripherals from an application that allows peripherals to be managed and coordinated within the operating system transparently to the application.

Several coordination strategies are developed, and the *sample-ready synchronization* technique is shown to reduce energy and minimize undesirable skew between samples. The feasibility of implementing and using this abstraction as well as its potential for energy savings is evaluated on a custom hardware peripheral emulation testbed and, separately, on a real application that fuses sensor readings from an off-the-shelf accelerometer, gyroscope, compass, and altimeter. Compared to the existing baseline implementation, energy is reduced by 61%, sample skew is negligible, and code is simplified.

## 1. INTRODUCTION

Software applications that closely interact with the physical world do so through a set of sensors and actuators provided by a hardware platform. The sensors that serve one application are usually not independent but related to each other through the top-level application logic. For example, a motion-tracking application might *fuse* samples from an accelerometer, a gyroscope, and a compass by computing the direct cosine matrix (DCM). Such fusion consists of a computation whose inputs are the latest available samples from each sensor and whose output is a state vector. Developer's effort deserves to be focused on implementing this computation and not on the task of getting samples from sensors into program memory. However, carrying out the latter task using abstractions offered by a state-of-the-art embedded operating system, may result in software that uses energy inefficiently and takes more time to write.

An embedded operating system, such as the Real-Time Operating System (TI-RTOS) from Texas Instruments, provides access to each device through a driver whose interface closely resembles the interface exposed by hardware through registers. This approach is more flexible and efficient in terms of memory-space compared to providing generic abstractions common on general-purpose operation systems, such as the file abstraction on Linux. However, the added flexibility comes at the cost of developer time. For the simple goal of obtaining samples at a certain frequency from each device, the developer might end up navigating the maze of hardware state machines specific to each device. Furthermore, differences in device sampling times and data transfer mechanisms, mean that samples become available at different times and may trigger the fusion computation independently. This uncoordinated access to peripherals creates *redundant work*, which consumes extra energy, and leads to fusion calculations on *skewed* sample values acquired at different times, which diminishes application performance.

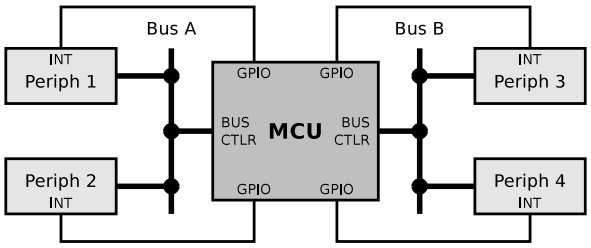



**Figure 1: Platform model: MCU with peripherals.**

We propose an abstraction at the operating-system level that exposes sensor readings as generic streams of data ready for consumption by high-level application code. The runtime system that implements this abstraction coordinates low-level device access by fully controlling communication to all devices. We design a coordination strategy that reduces core energy consumption by eliminating redundant work and minimizes time discrepancies between data samples. Our prototype is evaluated on real hardware using a custom hardware peripheral emulation testbed and in a case study of a motion-tracking application implemented on the Tiva C microcontroller [4] connected to an off-the-shelf accelerometer, gyroscope, compass, and altimeter. As a part of an embedded operating system, this system has the potential to reduce energy consumption and developer effort in sensor-fusion applications.

*This work was supported by Texas Instruments.

†The source code accompanying this report is available at `http://link.alexeicolin.com/coord-peripherals`

## 2. BACKGROUND AND MODEL

Platforms considered in this study consist of a microcontroller (MCU) and a set of peripherals connected to it through buses and interrupt lines, as illustrated in Figure 1. For example, this may be a Tiva C MCU with a magnetometer connected over the I2C bus and a gyroscope connected over the SPI bus. Each peripheral is either a discrete component or a module within an Integrated Circuit (IC) that combines multiple peripherals, such as an Inertial-Motion-Unit (IMU) IC. Integrated peripherals may be synchronized in hardware by virtue of being on the same clock tree and re-using the same signals, which is the case for the accelerometer and the gyroscope in the InvenSense MPU9150. However, this is not always the case. For example, the AsahiKasei AK8975 magnetometer built into the MPU9150 is largely independent from the other two peripherals. The minority of the applications that depend only on a set of peripherals that are all integrated and synchronized in hardware may not benefit from additional coordination support in software. However, when a mixture of discrete sensors is used, coordination is only possible at the software level.

A platform consumes energy on for powering the MCU and the peripherals. The MCU provides at least one *sleep state*, in which power is reduced compared to active operation and wakeup time is short relative to the sampling frequency. For example, the Tiva C provides a sleep state with power consumption equal to 50% of that in active state and a 5 $\mu$s wakeup time, and another with 25% power and 20 ms wakeup time. The energy overhead of transitioning into and out of the sleep states is comparatively small, since the current draw during the transition is close to that during the sleep state. Consequently, the impact of the transition time on transition energy is small regardless of the transition duration. The objective of this study is to reduce the time the MCU spends computing and increasing its time in a deep sleep state.

The secondary target for energy-reduction pursued in this study is the energy consumed by the peripherals when they are *idle*, i.e. not actively acquiring or transferring samples. Table 1 lists the device parameters present in our model for several off-the-shelf sensors. Each peripheral may implement a *standby state*, in which the power consumption is on the order of micro-Amperes, i.e. comparable to turning the device power supply off by a necessarily-leaky electrical switch. Peripherals that do not provide a standby state are compatible with our system, but their energy share can only be reduced with hardware modifications. A device without a standby state can be converted into one with a standby state by connecting its power rail through an electrically-controlled switch. Energy consumption is minimal when a peripheral is in active mode if and only if it is busy and in standby mode otherwise.

Whether a peripheral device achieves this minimum energy consumption depends on its capabilities in *trigger control* and *power state control*. These capabilities are best explained in the context of the sample acquisition timeline. Before a sample value appears in MCU core memory, the following high-level events take place:

1. **resume**: *(optional)* sensor is woken up from standby
2. **trigger**: a request for acquiring a sample is initiated
3. **acquire**: sensor measures and digitizes quantity
4. **interrupt**: sensor notifies MCU via an interrupt
5. **read**: sample data is transferred to MCU core memory
6. **suspend**: *(optional)* device is put into standby state

Devices differ widely in how each of these actions is initiated and performed. Abstracting these differences away is one of our goals.

The trigger for sample acquisition may be a signal that is generated internally by a clock at a fixed or configurable period, an external electrical signal routed to a pin on the peripheral, or a message that must be sent by the MCU over the bus. The power state of the device may be fixed without possibility of control, controlled automatically by internal hardware logic, controlled through explicit messages over the bus from the MCU, or controlled through external electrical signals. The combination of these capabilities determines the energy profile of the device and vectors for reducing it.

Of particular note are the devices that support both an internal clock trigger and automatic power state control. Such a device consumes the minimum amount of energy by default and without any involvement from the MCU, because it is in standby whenever it is idle. At the opposite extreme are devices that must be explicitly told to acquire each sample and to enter standby mode by the MCU. Energy consumed by the latter class of devices and those in between the extremes can be reduced by managing each device in software. This is the secondary responsibility of the runtime system that implements the proposed sample stream abstraction. This potential for reducing the peripheral share of energy is orthogonal and complementary to the coordination techniques for reduction of the MCU share of energy described next.

## 3. PERIPHERAL COORDINATION

The objective of coordinating access to peripherals is to reduce the share of energy consumed by the MCU. We examine in more detail applications that obtain samples from multiple sensors and fuse them through a calculation. A state vector computed from all sensor inputs is often kept up-to-date by recalculating it when new sample values become available. On one hand, some calculations may be redundant if triggered by updates to inputs that are separate but close in time. On the other hand, the age of samples used in the fusion calculation bears on application performance. For instance, this may affect the timeliness of responding to events in the environment. Furthermore, skew in time between the samples in the input set adversely affects the accuracy of the fusion result. In this section we show how peripheral access can be coordinated to eliminate redundant computation keeping the skew to a minimum. We begin by analyzing the uncoordinated case and introduce a series of techniques that progressively improve in the energy and skew dimensions.

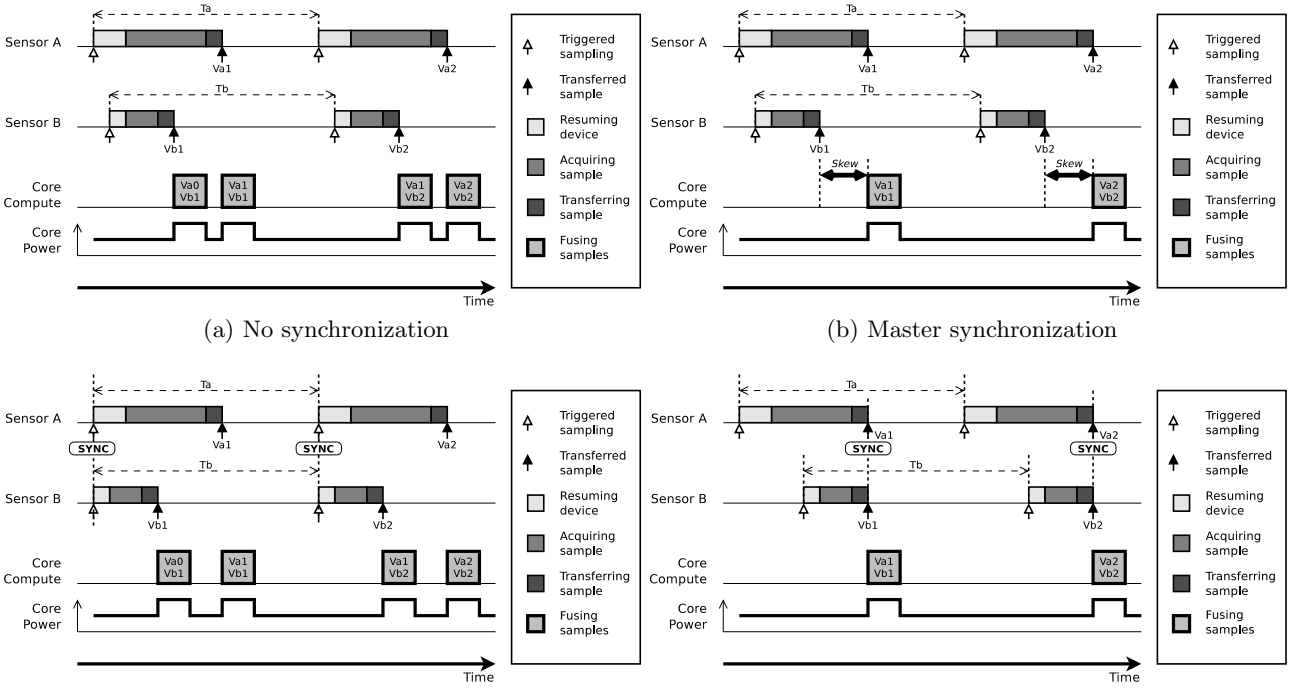
### 3.1 No Synchronization

Uncoordinated acquisition of samples from peripherals creates redundant computation, which costs energy. Figure 2a illustrates a timeline of an example scenario where samples are acquired from two sensors. When a new sample from

| Name | Physical Quantity | Resume Latency (ms) | Conv. Time (ms) | Sampling Rate (kHz) | Sleep Mod | Auto Slp | Slp Current ($\mu$A) | Active Current (mA) | Trig Types |
|---|---|---|---|---|---|---|---|---|---|
| MPU9150 | rotation | 30 | ? | 0.004-8 | 1 | N | 6 | 3.4 | P |
| MPU9150 | accel. | 20 | ? | 0.004-1 | 1 | Y | 6 | 0.550 | P |
| AK8975 | compass | 0.1 | 9 | 0.1 | 1 | Y | 10 | 6 | S |
| MPL3115A2 | altimeter | 60 | 6-512 | 100 | 1 | Y | 2 | 0.265 | P,S |
| FA532002 | rotation | 60 | ? | 12.5-800 | 2 | N | 2.8 | 2.7 | P,H |
| HMC5883L | magnetism | 50 | 6 | 7.5E-4-0.160 | 1 | Y | 2 | 0.1 | P,S |
| TMP006 | temp. | ? | ? | 2.5E-4-0.004 | 1 | Y | 0.5-1 | 0.325 | P |
| BMP180 | pressure | 10 | 25 | 0.128 | 1 | Y | 0.1 | 0.032-1 | S |
| SHT21 | humidity | 150 | ? | 0.002 | 0 | N | N/A | 0.16 | P |
| ISL29023 | light | ? | ? | 0.011Hz-90 | 1 | Y | 0.01-0.3 | 0.085 | P, S |

**Trigger types**: P = periodic internal clock, S = message from MCU software over a bus, H = electrical signal on a pin

**Table 1: Parameters for several off-the-shelf peripheral sensor devices from datasheet specifications.**




(a) No synchronization

(b) Master synchronization

(c) Trigger synchronization

(d) Sample-ready synchronization

**Figure 2: Fusion of samples from two sensors coordinated according to different synchronization strategies.**

either device becomes available (values $V_{a,i}$ or $V_{b,j}$), it triggers the application to re-compute the fused state to keep it as up-to-date as possible. When these computations happen to take place back-to-back, they add little value. Even when the computations are not back-to-back, they are still redundant for a more subtle reason. In this example, both sensors produce samples at the same frequency. If implementation constraints are abstracted away, the application would consume both values at this frequency, which entails one fusion computation per the respective period. Any computations beyond this number stem not from application requirements but from implementation constraints and are avoidable.

## 3.2 Master Synchronization

The problem of redundant computation identified in Section 3.1 can be approached by the following straightforward method. The application can be forced to only re-compute the fused state only when a new sample from the *master* sensor becomes available. Figure 2b illustrates this scheme in action assuming Sensor A is selected as the master. The number of computations matches the minimum required by the application, as desired. However, samples from Sensor B now wait in memory before being incorporated into the fused state. This waiting time is labeled as the skew in Figure 2b. The master-based approach is simple to implement but can save energy only at the cost of introducing skew between sample inputs, which is a sacrifice in application performance.

## 3.3 Trigger Synchronization

Master-based coordination from Section 3.2 passively reacts to the timeline. An alternative approach is to actively manipulate the timeline to achieve coordination. An intuitive strategy that is relatively simple to implement is to start the acquisitions on all sensors at the same time. This is a winning strategy when sensors take the same time to acquire a

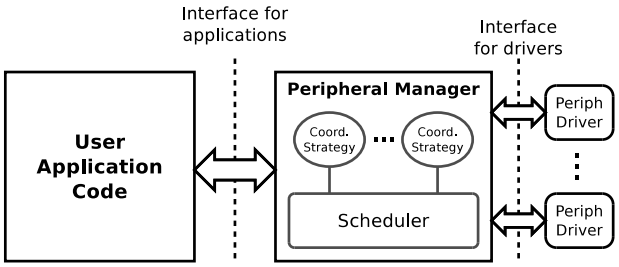



**Figure 3: Peripheral manager system architecture.**

sample, which implies device parameters that are equal or coincidentally add up to the same interval. This situation is uncommon because usually each sensor measures a different physical quantity. The common case for sensors with different characteristics is illustrated in Figure 2c. In this case, synchronized triggers do not decrease the number of fusion computations at all. A combination of this strategy with the master-based would eliminate redundant fusion operation, however it would not improve on the skew over the bare master-based strategy.

## 3.4 Sample-ready Synchronization

The strategy that can eliminate redundant computation without introducing skew must synchronize the instants at which samples become ready from different sensors. The timeline in Figure 2d shows that this strategy produces no redundant computation, and in the common case of equal sampling periods, does not keep any samples waiting in memory before being processed. When periods are different, skew cannot be eliminated, however it might be possible to minimize it by harmonizing the periods by a technique similar to Rate-Harmonized-Scheduling [1].

The sample-ready strategy is advantageous, but it is not trivial to accomplish in practice. No sensor interface explicitly supports a request for delivering a sample at a specific time in the future. The sample-ready synchronization requires an entity which centralizes all peripheral access by channelling all access requests through a generic interface based on a unified abstraction of a peripheral as a sample stream. In the following sections we describe and evaluate the design and implementation of a runtime module for peripheral management capable of coordinating sample acquisitions according to all discussed strategies, including the sample-ready synchronization.

# 4. PERIPHERAL MANAGER MODULE

At the core of our method for coordinating access to peripherals is the sample stream abstraction. Applications targeted in this study are interested in the sample values and not at all in the devices that produce them. Under the proposed abstraction, a peripheral is presented to the application not as a hardware device but as a source of a stream of sample values generated at a specified frequency. This abstraction grants its implementor the flexibility necessary to manipulate the peripherals transparently to the application. Mechanisms for coordination and energy reduction are decoupled from the application and delegated to a dedicated run-time module that implements the abstraction. Our prototype supports all coordination strategies defined in Section 3, including sample-ready synchronization. An overview of the system architecture of the peripheral manager is shown in Figure 3. As the layer between the application and hardware devices, the peripheral manager exposes two separate interfaces. The application developer consumes the *application-facing* interface, and the driver developer implements the *device-facing* interface.

The application-facing programming interface is summarized in Figure 4a. The module is initialized with a coordination mode parameter, which, together with the wait mode described below, selects among the strategies defined in Section 3. An input sample stream is created by supplying a device driver handle and the sample update frequency to the `iomgr_in` method (along with buffer space for internal use). Once all streams are declared and `iomgr_start` method is called, the device manager is actively controlling the devices and retrieving samples at the designated frequency. The samples are retrieved whether or not the application processes them, because the creation of the stream is a declaration of application intent to receive the data.

Once a set of streams is established, samples can then be processed in a coordinated fashion by waiting for an update through `iomgr_wait` and copying the values into application memory with `iomgr_read`. The wait method centralizes all asynchronous communication with all devices into one simple synchronous call. The wait mode specifies whether the wait should return when any one of the specified streams is updated or only when all have been updated. In terms of strategies described in Section 3, the former mode corresponds to no synchronization and the latter to master-based or to sample-ready synchronization depending on manager configuration. The read method does not involve communication with the peripherals but only a copy of an already-transferred sample within memory. If this were not the case, then the application would effectively interact with the device, which would violate the proposed abstraction and its premise. The interface can be extended to support output streams to actuators. This would entail a declaration of a stream with an `iomgr_out` method and an `iomgr_write` method for accepting a value into memory to be transferred to an actuator at a chosen rate.

The device-facing interface is outlined in Figure 4b. The peripheral manager controls each device through a device-specific driver that implements this interface. The interface consists of actions that in most cases require no more than one message to the device, such a write to one device register over an I2C bus. Only actions that are as generic across devices as possible are included. The goal of this interface is to provide only mechanisms for controlling the device without incorporating any policies.

The code listing in Figure 5 outlines the usage of the proposed interface in pseudo-code. After having initialized the device drivers, the application operates exclusively with sample streams and sample values. The run-time module is implemented as a set of libraries for linking into the application, which is the common method for leveraging functionality provided by an embedded operating system. The core library implements the API from Figure 4a and a set of aux-

| | |
|---|---|
| *status* | `iomgr_init`(*coord_mode*) |
| *stream* | `iomgr_in`(*device*, *aux_buffer*, *freq*) |
| *status* | `iomgr_start`() |
| *stream_set* | `iomgr_wait`(*stream_set*, *wait_mode*) |
| *status* | `iomgr_read`(*stream*, *buffer*) |
| *status* | `iomgr_close`(*stream*) |

(a) Interface provided by the peripheral manager for an application.

| | |
|---|---|
| `init`(*instance, params*) | |
| `reset`(*instance*) | |
| `trigger`(*instance*) | |
| `read`(*instance*, *quantity*, *buffer*) | |
| `resume`(*instance*) | |
| `suspend`(*instance*) | |

(b) Interface provided by a device driver for the peripheral manager.

**Figure 4: Programmatic interfaces provided and consumed by the peripheral manager.**

```
main() {
  dev_x_driver_init(&device_x);
  dev_y_driver_init(&device_y);

  stream_x = iomgr_in(&device_x,
    aux_buffer_x, 100 /* Hz */);
  stream_y = iomgr_in(&device_y,
     aux_buffer_y,  50 /* Hz */);

  while (true) {
    iomgr_wait({stream_x, stream_y},
                WAIT_MODE_ALL);
    iomgr_read(stream_x, &sample_x);
    iomgr_read(stream_y, &sample_y);
    process({sample_x, sample_y});
  }
}
```

**Figure 5: Sample application code demonstrating the usage of the peripheral manager API.**

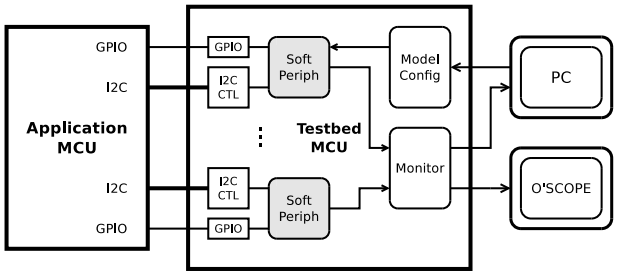



**Figure 6: Hardware peripheral emulation testbed.**

iliary libraries implement the drivers for each device. Our prototype implementation of the libraries used in this study and distributed with this report amounts to approximately 750 lines of C code for the core library and 500 lines of C code per sensor driver.

# 5. EVALUATION

The peripheral manager is evaluated on real hardware to assess the applicability of the sample stream abstraction and the reduction in energy and sample skew it offers. The first set of experiments were conducted on a custom hardware emulation testbed for creating software-defined peripherals. This was complemented by a case study of a motion-tracking application implemented on the Tiva C platform with four off-the-shelf sensors.

## 5.1 Peripheral Emulation Testbed

For testing of the peripheral manager and evaluation of coordination strategies, we developed a hardware *peripheral emulation testbed*. The testbed architecture is shown in Figure 6. It consists of the Tiva C Launchpad board and the firmware that allows creating up to four emulated peripheral devices. Externally, an emulated device presents the same electrical interface to the application MCU as a real device would, i.e. it connects to an I2C bus and provides a sample-ready interrupt line. Internally, an emulated device implements the peripheral model defined in Section 2. The parameters of this model, such as the resume latency and the sampling time, are configurable in software.

For monitoring and measurements, each emulated peripheral encodes events, e.g. sample trigger, and an emulated power state onto external GPIO pins and into a log streamed to a console on a host workstation. Interference between emulated peripheral instances that run on the same hardware board is minimized by allocating dedicated hardware resources, i.e. timers and I2C controllers, whenever possible. In addition, the MCU is clocked at the maximum frequency (80 MHz), which is often a speedup over the application MCU.

With our peripheral emulation testbed, generic peripheral control mechanisms can be developed without purchasing a large set of hardware sensors. Prototyping time is reduced with no device-specific implementation work. The testbed can also be used to design and optimize interfaces into peripherals.

### 5.1.1 Experimental Setup

An application based on the example in Figure 5 is run on a Tiva C Launchpad board. This application fuses samples from two peripherals at 1 Hz each and does a computation of 250 ms on them. Peripheral access is implemented using the peripheral manager API introduced in Section 4. The peripheral manager is configured to carry out each of the strategies in Section 3 for respective experiments. The two peripherals are emulated using the testbed described above that runs on a separate Tiva C Launchpad board and connects via I2C and GPIO lines to the application board. Each peripheral can be in either active or standby power state.

Throughout each experiment, we measure the energy consumption of the application MCU, the (emulated) power state of each peripheral, and record trigger and sample-ready events. Energy of the MCU is measured using the TIPD135 reference design of a high-side current-sensing circuit based on an PGA281 instrumentation amplifier [2]. For MCU power measurement, the current-sense resistor, $R_{\text{sense}}$, is 6.8 Ω and the gain, $G$, is 22. For peripheral power measurements (in experiments in Section 5.2), $R_{\text{sense}} = 120\Omega$

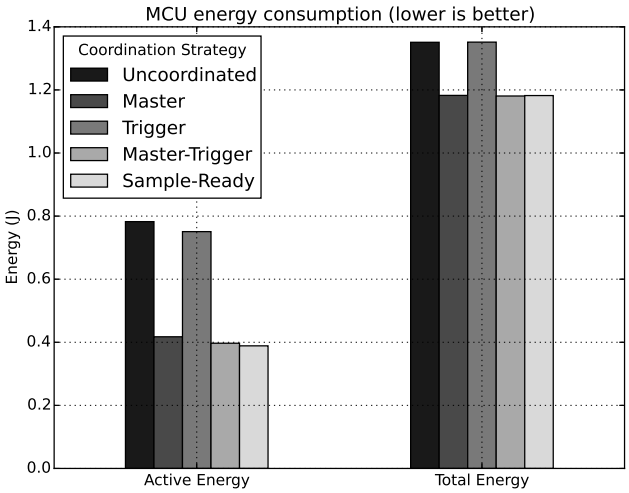



**Figure 7: Energy consumed by the MCU in active state and in all states for different peripheral coordination strategies measured on hardware peripheral emulation testbed.**

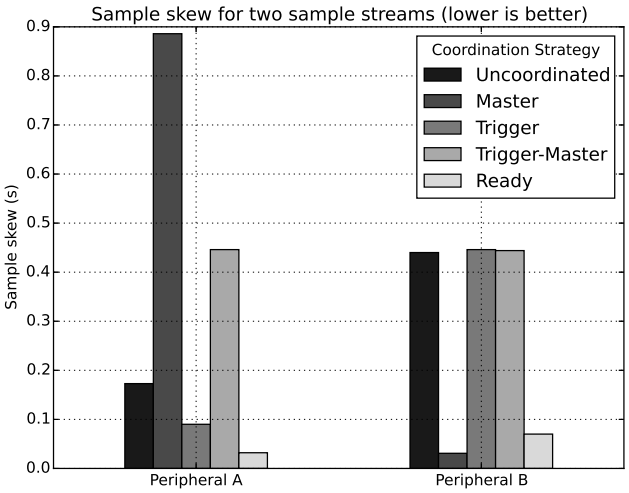



**Figure 8: Skew between the instant when a sample is ready and when it is processed for different peripheral coordination strategies measured on hardware peripheral emulation testbed.**

and $G = 2.75$. The amplified voltage over $R_{\text{sense}}$ is a differential signal that we read out as the difference between two oscilloscope channels. Oscilloscope traces of this voltage are divided by $R_{\text{sense}}$ and integrated to get the total energy consumption of the MCU. The same analog trace is masked by the digital trace of the MCU active/sleep state and integrated in order to get the share of the MCU active energy. The digital event traces are post-processed to calculate the skew between the instant a sample is ready and when it is processed, as discussed in Section 3. Traces for all experiments are 20 seconds long.

### 5.1.2 Results

The coordination strategies proposed in Section 3 were compared according to the energy use and sample skew metrics. Oscilloscope traces for each coordination strategy were post-processed as described in Section 5.1.1, into plots of MCU active and MCU total energy in Figure 7 and sample skew in Figure 8. By active energy use, strategies fall into one of two groups: the 0.4 J group and 0.8 J group. This 50% difference corresponds to whether the strategy successfully eliminates redundant fusion computations. The master-based synchronization, trigger-with-master synchronization, and sample-ready synchronization realize these savings, while unsynchronized and trigger-based synchronization do not.

However, as visible in Figure 8 only the sample-ready strategy reduces the skew to below 0.07 ms for *both* peripherals. The energy-savings viewed as a fraction of total MCU energy are less pronounced (15%) than those in active energy alone, because in our example application, the MCU is idle from 80% to 90% of the time. The longer the fusion computation on the data samples takes, the larger the savings in total energy are. These results from real hardware demonstrate the feasibility of the proposed sample stream abstraction and its energy-saving potential.

## 5.2 Sensor-Fusion Case Study

The proposed approach to coordinating peripheral access is validated in practice in a sensor-fusion application on Tiva C microcontroller and four off-the-shelf sensors.

### 5.2.1 Motion-Tracking Application

The motion-tracking application acquires samples from multiple sensors and fuses the values into a motion state descriptor. The fusion involves an expensive calculation of a Direct Cosine Matrix (DCM) and displaying the live results via a serial console. The platform consists of a Tiva C Launchpad board with a TM4C microcontroller and break-out boards for an accelerometer, a gyroscope, a AsahiKasei AK8975 compass, all integrated in the InvenSense MPU9150 chip, and a discrete altimeter Freescale MPL3115A2. All peripherals are connected to the same I2C bus. Although it is integrated, the compass can accessed as a standalone device on the bus. In all experiments the MCU is always clocked at 16 MHz, from the main crystal in active mode, and from internal oscillator in deep sleep mode.

Our case study compares two implementations of this motion-tracking application. The baseline version is a modified example from TivaWare library [3], augmented with altimeter sampling and configured to sample all sensors at 8 Hz. The added altimeter code is analogous to the MPU code, except the sampling is triggered in software from a timer ISR, since the hardware trigger supports only rates below 1 Hz. The contrasting version of the application is implemented using the sample stream abstraction introduced in Section 4. The code very closely mimics the pseudo-code from Figure 5. An immediately noticeable advantage of the latter version is its concise code that focuses on the sensor fusion task and hides the data acquisition details. Drivers for each sensor were written to expose the driver-side interface described in Section 4. In our architecture this task is routine, because the drivers implement only the simplest actions and are free of any policies.

The two versions were compared in terms of energy consumption, sample skew defined Section 3, and ease of programmability. Energy consumed by the core and by all peripherals was measured separately and concurrently using two TIPD135 boards (see Section 5.1.1). Digital traces were collected for the data-ready interrupt lines from MPU9150 and MPL3155A2, for MCU state, and for trigger requests and transfer requests encoded onto GPIO pins by the peripheral manager. Sample skew was calculated from these traces as the time difference between the beginning of a fusion operation and the most recent preceding data-ready interrupt for each of the sensors.

### 5.2.2 Execution Trace and Energy Trace

Oscilloscope traces in Figure 10 show both versions of the sensor-fusion application in action. Each trace consists of a sequence of several sample acquisitions and fusion operations. The top two panels of each figure show the instantaneous power usage of the MCU and all peripherals, respectively. The bottom panel contains the signals that expose the events that are happening. The *Rdy* signal pulses low when the device finished acquiring a sample and is ready to transfer it to the MCU whenever the MCU issues this request. The trace for the peripheral manager version in Figure 10b also includes the *Trg* signals, which denote when the manager asked the device to start sample acquisition, after having resumed the device (suspend/resume signals not shown). The *Trg* signals encode the sample acquisition request on rising edge and the transfer request on falling edge. When the main loop fuses the sensor readings, the *Fusing* signal is high.

The data-ready signal from the compass is not exposed at all by MPU9150 and the corresponding two signals shown in Figure 10 are derived from other signals. The *Compass Rdy* signal in the baseline version (Figure 10a) is calculated as the MPU data ready plus the maximum compass acquisition time of 9 ms from the data sheet. The *Compass Rdy* signal in the peripheral manager version (Figure 10b) is the expiration of a timer that counts down the same fixed acquisition time and serves as the data-ready event for the manager. This declared acquisition time was corroborated by tracing the power consumption of MPU9150 with only the compass running and observing spikes equal in length to 9 ms.

The baseline version takes advantage of the hardware synchronization mechanism provided by the MPU9150 to synchronize the accelerometer, gyroscope, and compass. All three of these sensors generate only one data-ready event when acquisition on all completes. The accelerometer and gyroscope are driven from the same clock internal to the peripheral and operate synchronously by default. The MPU hardware automatically transfers a sample and triggers the next acquisition of the compass on each sample clock tick. Effectively, the compass is "polled" by the MPU (not the MCU) at the sampling rate.

### 5.2.3 Energy

Our sample stream abstraction and peripheral manager reduce energy by eliminating redundant fusion computations, placing the core into a high-wakeup-latency deep sleep state between samples, and duty-cycling peripherals. In aggregate, the peripheral manager version of the motion-tracking

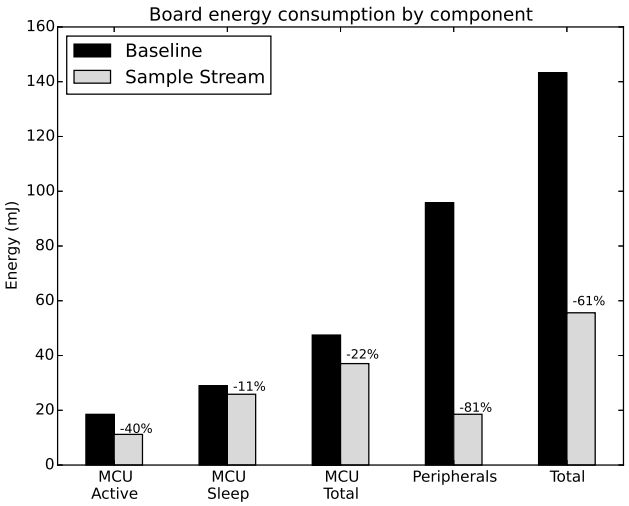
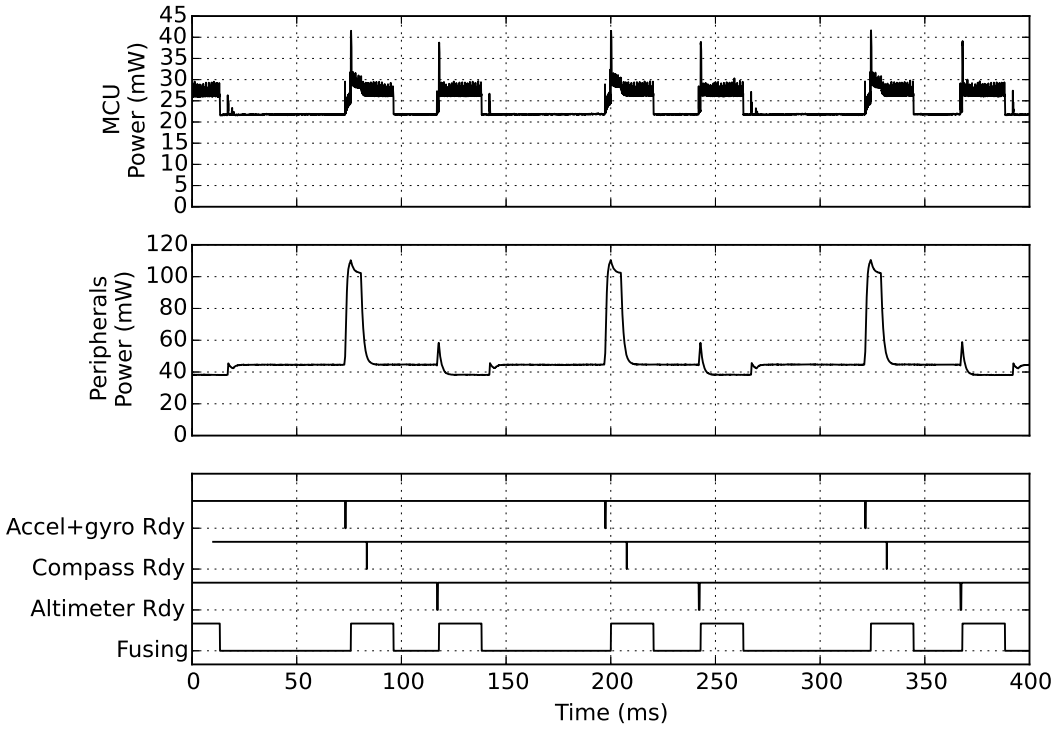



**Figure 9: Energy consumed by two alternative implementations of the sensor-fusion application measured on Tiva C MCU with four off-the-shelf sensors.**

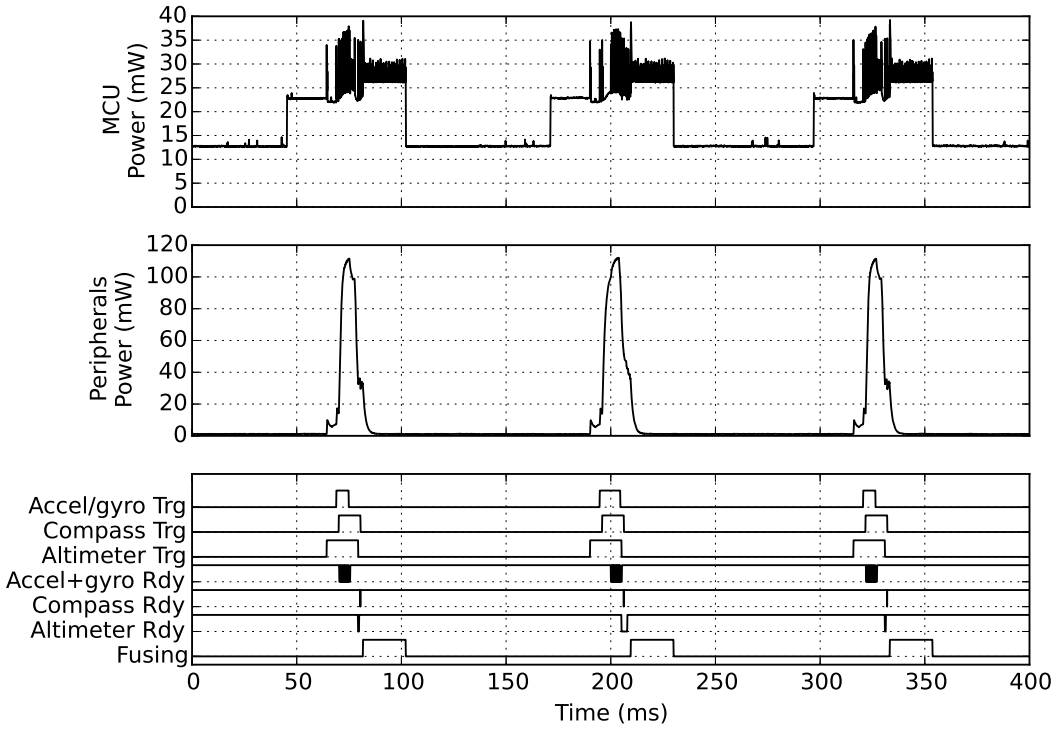
application uses 61% less energy than the baseline version. Figure 9 breaks down these savings into components. In the motion-tracking application, the number of fusion computations is reduced in half, as is seen in Figure 10, because the altimeter is coordinated with the other sensors. This yields a reduction of 40% in MCU active energy as plotted in Figure 9. Alternatively, the redundant computation could be eliminated by synchronizing the altimeter using the previously-mentioned secondary bus mechanism provided by this particular MPU hardware. However, this method is limited to trigger-based synchronization, which inevitably introduces skew, analogously to the compass case to be discussed in Section 5.2.4.

A second upshot from centralizing coordination in the peripheral manager is an opportunity to keep the MCU in deep sleep between samples. In this case study, we observed a reduction of 11% in sleep energy share, plotted in Figure 9. This is the net reduction after the increase in sleep energy due to the increased time the MCU spends in sleep mode. The manager is able to compensate for the wakeu-up latency through scheduling by arranging exactly one wakeup per sampling period. This compensation is required for sample-ready synchronization strategy, because otherwise a skew up to the wakeup latency (20 ms on Tiva C) would be introduced. The alternative approach of transitioning the MCU into deep sleep whenever it has no pending work was attempted in the baseline version of the application and determined to not be viable. The modified code produced an effective sampling rate that was six times smaller than the explicitly configured one. This means that under this simplistic alternative, the implicit assumptions about timing made in the code no longer hold, producing incorrect behavior. For example, the wakeup latency is now incurred at arbitrary interrupt service routines, including the frequent one for I2C bus transactions.

To reduce the share of energy consumed by peripherals, the proposed peripheral manager can duty-cycle a peripheral




(a) Baseline imlementation from TivaWare example




(b) Implementation using sample-stream peripheral manager

**Figure 10: Oscilloscope traces showing peripheral sampling events and power consumption for two alternative implementations of a sensor-fusion application on Tiva C MCU with four real off-the-shelf sensors.**

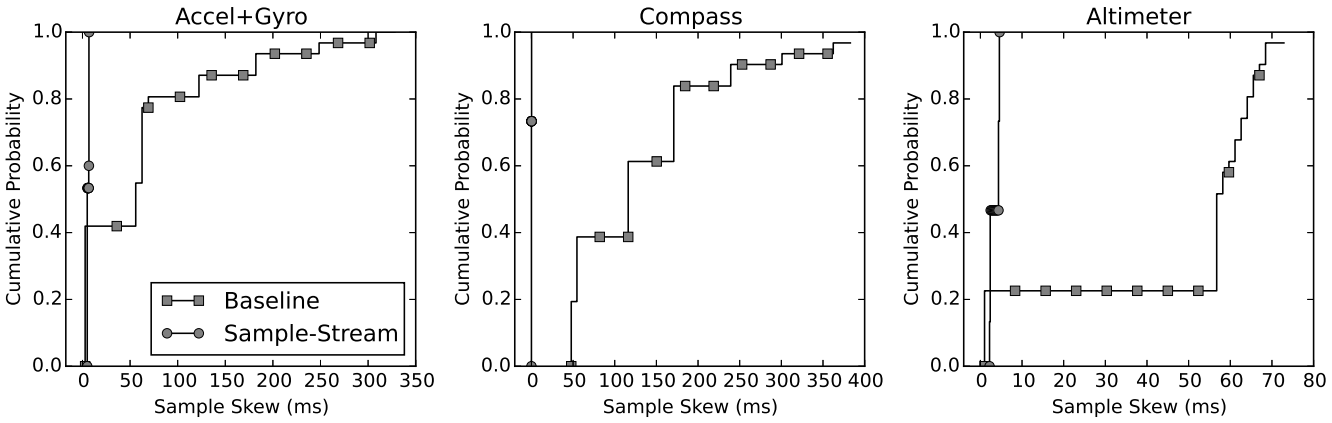



**Figure 11: Measured sample skew for each sensor for two alternative implementations of the sensor-fusion application running on Tiva C MCU with four real off-the-shelf sensors.**

whenever such functionality is not built into the peripheral hardware. As discussed in Section 2, some peripherals automatically enter standby mode after having acquired a sample. For a given sampling rate, the energy share consumed by such a peripheral is already at its minimum by default, since the device is active if and only if it is performing useful work. The compass and the altimeter in our case study are both in this category. However, the accelerometer supports automatic duty-cycling only if the gyroscope is disabled. This limitation disqualifies this hardware functionality from our motion-tracking application, which relies on all sensors. The gyroscope does not support any automatic duty-cycling at all. By default, the power consumption of the gyroscope is constant.

The peripheral manager offers a zero-effort solution to overcoming this lack of hardware functionality. The manager automatically suspends and resumes devices for each sample. The wakeup time is automatically measured during an initialization phase and factored into the scheduling for the sample-ready coordination. In the sample-stream version of our application, both the gyroscope and the accelerometer are duty cycled by the peripheral manager behind the scenes. The savings in peripheral energy consumption are 81%, most of which is attributed to the reduction in the gyroscope share.

### 5.2.4 Sample Skew

The hardware synchronization mechanism in the MPU discussed in Section 5.2.2 effectively implements the trigger-master synchronization scheme defined in Section 3.3. The main implication of this scheme is the sample skew in the compass readings. In the absence of other peripherals (i.e. without the altimeter), each compass reading would be stale by at least one sampling period (125 ms) when fused. With other peripherals present (i.e. with the altimeter), data-ready interrupts may happen earlier reducing this skew for some of the samples. This effect is observed as the bimodal nature of the CDF of the skew in Figure 11 (middle), where roughly half of the samples experience the skew above one sampling period (125 ms). The skew in the accelerometer-gyroscope and altimeter samples is also bimodal. Fusion operations that are triggered by Sensor A use fresh data from Sensor A (low skew) but stale data from Sensor B (high skew) and *vice versa*. Furthermore, the skew varies on the longer time scale of tens of seconds due to drift between the MPU clock and the MCU clock. In contrast, the vertically sloping CDFs near zero skew, show that skew is practically absent for the data-ready synchronization strategy from Section 3.4 implemented in the proposed peripheral manager.

## 6. CONCLUSION

This study identified sources of energy loss common in applications that are coupled with physical processes. Uncoordinated access to a set of peripherals may lead to redundant computation if sensor samples are fused with each update. The microcontroller and the peripherals may waste energy if not transitioned into a standby state when idle. We proposed the sample stream abstraction for accessing peripherals that decouples the application logic from mechanisms for coordinating peripheral devices. A sequence of coordination mechanisms was described, including a sample-ready synchronization strategy that saves energy by eliminating redundant computation while keeping sample skew to a minimum. The proposed coordination strategies were implemented in a run-time peripheral manager module. The proposed system was first evaluated on a custom hardware peripheral emulation testbed for creating software-defined peripherals. A standalone case study with four off-the-shelf sensors confirmed the general applicability of the proposed abstraction and its potential for reducing energy, eliminating sample skew, and simplifying application code.

## 7. REFERENCES

[1] A. Rowe, K. Lakshmanan, H. Zhu, and R. Rajkumar. Rate-harmonized scheduling and its applicability to energy management. *Industrial Informatics, IEEE Transactions on*, 6(3):265–275, Aug 2010.

[2] Texas Instruments. High-Side Current Sensing Solution Reference Design. `http://www.ti.com/tool/TIPD135`.

[3] Texas Instruments. Tiva C Series LaunchPad Evaluation Kit. `http://www.ti.com/tool/sw-tm4c`.

[4] Texas Instruments. TivaWare for C Series. `http://www.ti.com/tool/ek-tm4c123gxl`.