Cyclops: PRU Programming Framework for Precise Timing Applications

Amr Alanwar*, Fatima M. Anwar*, Yi-Fan Zhang*, Justin Pearson†, João Hespanha†, Mani B. Srivastava*
*University of California, Los Angeles
†University of California, Santa Barbara
{alanwar, fatimanwar, yifanz, mbs}@ucla.edu
{jppearson, hespanha}@ece.ucsb.edu

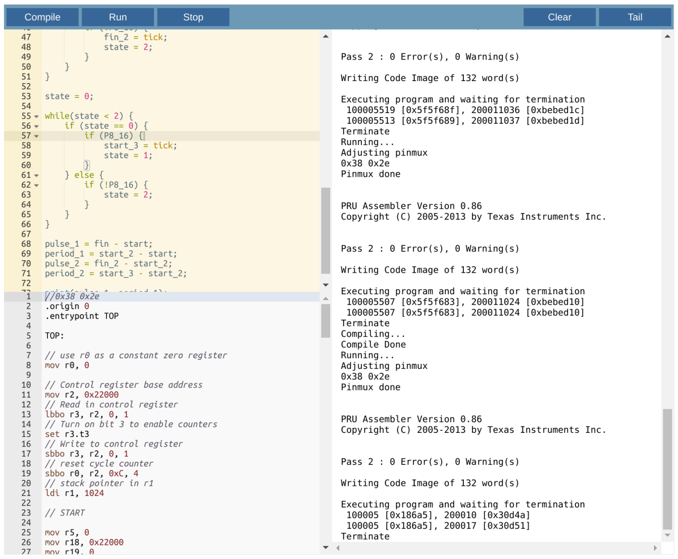
***Abstract*—The Beaglebone Black single-board computer is well-suited for real-time embedded applications because its system-on-a-chip contains two "Programmable Real-time Units" (PRUs): 200-MHz microcontrollers that run concurrently with the main 1-GHz CPU that runs Linux. This paper introduces "Cyclops": a web-browser-based IDE that facilitates the development of embedded applications on the Beaglebone Black's PRU. Users write PRU code in a simple JavaScript-inspired language, which Cyclops converts to PRU assembly code and deploys to the PRU. Cyclops automatically configures the Beaglebone's pinmux controller to use common I/O peripherals like ADC and PWM. Additionally, Cyclops includes a PRU library and Linux kernel module for synchronizing the PRU with the processor clock, enabling the PRU to time-stamp sensor measurements using the Linux processor time within sub-microsecond accuracy.**

I. Introduction

A. Motivation

The Beaglebone Black (BBB) single-board computer (SBC) offers a flexible platform for robotics, sensing, and control system applications [1]. Its 1-GHz processor, 500 MB RAM, and 2 GB of non-volatile flash memory make it more powerful than a "bare metal" microcontroller. Moreover, it ships with the Debian Linux operating system and boots into a common windows-based desktop environment. Its Texas Instruments system-on-a-chip (SoC) includes a wide range of peripherals like an analog-to-digital converter (ADC), general-purpose I/O pins (GPIOs), pulse-width modulation (PWM), and universal asynchronous receiver/transmitter (UART). It includes Python and C software libraries for interfacing with these I/O peripherals. Consequently, building an application that interfaces with physical sensors and actuators takes just a few lines of code.

However, precise timing is often a key requirement in real-time applications; for example, the timing uncertainty suffered by a non-real-time OS like Linux can cause control system instability [2]. A program running on "bare metal" executes somewhat deterministically, due to having essentially sole control of the CPU. In contrast, a program running on top of a multi-threaded OS competes with other programs and OS services, and therefore executes with much less determinacy. To address this, some SoCs include subsidiary co-processors to be used in real-time applications; this architecture is a feature of ARM's "big.LITTLE" heterogeneous computing architecture and also the OMAP family of Texas Instruments SoCs.



```
            fin_2 = tick;
            state = 2;
        }
    }
}

state = 0;

while(state < 2) {
    if (state == 0) {
        if (P8_16) {
            start_3 = tick;
            state = 1;
        }
    } else {
        if (!P8_16) {
            state = 2;
        }
    }
}

pulse_1 = fin - start;
period_1 = start_2 - start;
pulse_2 = fin_2 - start_2;
period_2 = start_3 - start_2;
```

```
//0x38 0x2e
.origin 0
.entrypoint TOP

TOP:

// use r0 as a constant zero register
mov r0, 0

// Control register base address
mov r2, 0x22000
// Read in control register
lbbo r3, r2, 0, 1
// Turn on bit 3 to enable counters
set r3.t3
// Write to control register
sbbo r3, r2, 0, 1
// reset cycle counter
sbbo r0, r2, 0xC, 4
// stack pointer in r1
ldi r1, 1024

// START

mov r5, 0
mov r18, 0x22000
mov r19, 0
```

```
Pass 2 : 0 Error(s), 0 Warning(s)

Writing Code Image of 132 word(s)

Executing program and waiting for termination
 100005519 [0x5f5f68f], 200011036 [0xbebed1c]
 100005513 [0x5f5f689], 200011037 [0xbebed1d]
Terminate
Running...
Adjusting pinmux
0x38 0x2e
Pinmux done


PRU Assembler Version 0.86
Copyright (C) 2005-2013 by Texas Instruments Inc.


Pass 2 : 0 Error(s), 0 Warning(s)

Writing Code Image of 132 word(s)

Executing program and waiting for termination
 100005507 [0x5f5f683], 200011024 [0xbebed10]
 100005507 [0x5f5f683], 200011024 [0xbebed10]
Terminate
Compiling...
Compile Done
Running...
Adjusting pinmux
0x38 0x2e
Pinmux done


PRU Assembler Version 0.86
Copyright (C) 2005-2013 by Texas Instruments Inc.


Pass 2 : 0 Error(s), 0 Warning(s)

Writing Code Image of 132 word(s)

Executing program and waiting for termination
 100005 [0x186a5], 200010 [0x30d4a]
 100005 [0x186a5], 200017 [0x30d51]
Terminate
```

Fig. 1: The Cyclops web-based PRU integrated development environment: users write PRU code in JavaScript-like language (upper left), which Cyclops compiles to pasm (lower left), with a status window on the right.

In particular, the Beaglebone Black's Sitara SoC is in the OMAP family; to support real-time applications, it contains two 200-MHz microcontrollers called "Programmable Real-time Units" (PRUs) that run independently from the main 1-GHz CPU. They have their own dedicated data and instruction memory, and have full access to the SoC's peripherals. In principle, a real-time unit paired with an OS running on a processor should provide the best of both worlds: the real-time unit handles time-sensitive aspects of an embedded application, and the OS provides the filesystem, memory management, task management, and networking.

A real-time application that leverages the PRU and the CPU concurrently will naturally require that they share a common sense of time. For example, a control algorithm on Linux runs with respect to the main processor's clock. However, it may need time-stamped sensor measurements from the PRU. The PRU runs independently from the main processor and uses its own internal cycle counter register to track time. Therefore, such a system requires some form of time-synchronization between the PRU and the main processor. We present a tool to address this need.

This paper presents a collection of software for developing real-time applications on the Beaglebone Black. The software is called "Cyclops," for "cycle-level operations". It provides a web-browser-based integrated development environment (IDE) for the PRU, shown in Figure 1. Cyclops streamlines the development of PRU code by offering a JavaScript-inspired language that it compiles into PRU assembly language. Cyclops provides a library of common functions for interfacing the PRU with various peripherals like the ADC, GPIO, and PWM. It expedites the configuration of the PRU and relevant peripherals on the Beaglebone by automatically configuring the Beaglebone's device tree overlays and pinmux. Finally, for time-synchronization, Cyclops provides a C library of PRU functions and a Linux kernel module for disciplining the PRU clock with reference to the main processor clock. We conclude the paper with an evaluation of a PRU/CPU time-synchronization scheme, wherein the PRU's timestamped sensor measurements are expressed in processor time within sub-microsecond accuracy.

B. Related Work

The Beaglebone Black and its Programmable Real-time Unit have been used for several applications. In [3], the authors performed audio processing with sub-millisecond latency using the PRU. The authors of [4] describe how to use the PRU for real-time sensing and control applications. In [5], the PRU's deterministic execution enabled real-time target-detection via a frequency-modulated ultrasonic sensor. Similarly, the BBB was used in [6] to collect and process real-time high-resolution hydro-acoustic data. In [7], the authors use a BBB in the development of a three-phase micro-inverter. Clearly, the Beaglebone SoC has found applications across multiple domains. Our Cyclops tool aims to enable the BBB for further research by facilitating the setup, development, and integration of the PRU with the rest of the BBB.

Using application-specific co-processors beside a general-purpose processor was explored in [8], wherein the authors compared a combined RISC/DSP architecture to a single RISC architecture for 3G multimedia mobile applications.

Existing literature also highlights the importance of synchronizing various peripherals on a single board to a common notion of time. Through a new OS abstraction — timelines — the authors of [9] synchronized a radio peripheral with the host processor's clock. They also implemented a software stack in the Linux kernel built around their timeline abstraction to support multiple simultaneous synchronizations, motivated by the need of a timing stack in [10]. This paper extends the work in [9] by augmenting their time-synchronization framework to the Beaglebone's PRU.

This paper is organized as follows. Section II describes the Beaglebone Black platform, focusing on the PRU, device tree, and Linux clocks. In Section III we describe the Cyclops system. Section IV demonstrates the result of synchronizing the PRU and CPU clocks using Cyclops. We conclude in Section V.

II. Background

A. Programmable Real-time Unit (PRU)

The Beaglebone Black includes two 200-MHz microcontrollers called "Programmable Real-time Units" (PRUs) that operate independently from the 1-GHz main processor that runs Linux. Each PRU contains a 32-bit RISC processor core with dedicated instruction and data RAM (8 kB of each). Another 12 kB RAM is shared between the PRUs for fast communication between them. The PRUs also have full access to the Beaglebone's main 500 MB RAM and all the SoC's peripherals. Moreover, the PRUs have single-cycle access to several GPIO pins on the BBB's header, making them suitable for fast nanosecond-level I/O. The PRU is programmed in PRU assembly language ("pasm") or C; Texas Instruments provides closed-source compilers to compile either language into PRU machine language.

For time-keeping, the PRU has access to several counters. The simplest counter is the 32-bit "PRU cycle counter," which merely counts the number of cycles since the PRU started executing. This counter is not suitable to drive higher-level clocks because it does not wrap automatically upon overflow; when it reaches $(2^{32}-1)$, which takes about 20 seconds, it simply stops counting and must be reset explicitly. Instead, for our work, the PRU keeps time via a counter within the Industrial Ethernet Peripheral (IEP). The IEP contains a 32-bit timer with capture and compare events, and can be configured to wrap automatically on overflow. The PRU accesses the IEP timer very deterministically, allowing us to account for that latency in our PRU timekeeping logic described in Section III-E.

B. The Linux Device Tree

When a system boots up, the kernel needs to learn what memory address ranges correspond to peripheral hardware registers. An embedded system typically has a fixed hardware configuration for a particular use-case, so its device configuration is typically hard-coded into the Linux kernel. This practice resulted in a situation where the mainline Linux kernel was being polluted by board- and architecture-specific configuration files.

The solution adopted by Linus Torvalds was to separate the hardware configuration from the Linux kernel into a data structure called a *device tree*. The device's bootloader passes the device tree file to the kernel at boot-time as a kernel parameter. The kernel then reads it to learn the hardware configuration it is running on and load the appropriate drivers. This enables a single kernel to be run on multiple hardware platforms.

A drawback of the device tree is that the Linux kernel reads it only once at boot-time. Hardware platforms like the Beaglebone and the Raspberry Pi can be customized with auxiliary daughter-boards that contain radios, accelerometers, and other sensors and actuators. Users often want to reconfigure their boards without having to reboot. To accommodate this, the device tree concept was expanded in Linux kernel version

3.17 to accommodate dynamic reconfiguration during runtime. The user loads a binary file called a *device tree overlay* that contains the hardware configuration of an auxiliary daughterboard. The kernel then integrates the overlay's device drivers into the existing device tree.

However, the implementation of device tree overlays is relatively immature and changed significantly between versions 3.18 and 4.1 of the Linux kernel. Consequently, it is a significant challenge for new Beaglebone users to write, compile, and load the appropriate device tree overlays. In Section III-C we describe how Cyclops circumvents the device tree overlay system by implementing a Linux kernel module to automatically configure the Beaglebone's pinmux controller.

C. Clocks in Linux

Operating systems play a key role in how time is managed and delivered to applications. In the Beaglebone, Linux monitors a single timer peripheral — a simple 32-bit counter — and maintains clock abstractions with names like *CLOCK_REALTIME*, *CLOCK_MONOTONIC*, and *CLOCK_MONOTONIC_RAW*. Despite being derived from the same timer, these virtual clocks have different properties; for example, *CLOCK_MONOTONIC* counts up strictly monotonically, even if the user tries to rewind the system clock. *CLOCK_REALTIME* is the Linux system clock that can be disciplined through synchronization algorithms.

Clock synchronization algorithms like NTP [11] adjust a clock by modifying how it converts the underlying timer peripheral's counter into a clock value. However, the accuracy of NTP is limited by the timing accuracy of Linux's software timestamps. Instead, the Precision Time Protocol (PTP) can achieve nanosecond-level synchronization using clocks based on timers capable of hardware-based timestamps, see [12], [13]. We employed this idea in [9], in which we made the BBB's processor clock PTP-compliant, then used this clock in a PTP-based clock-synchronization algorithm for networks of Beaglebones. In the present paper, we extend the scope of this effort by synchronizing the BBB's processor clock with its PRU's clock, which is derived from a timer capable of hardware timestamps. This allows the PRU to share a sense of time with PRUs in other BBBs across a network, useful for distributed systems.

III. Cyclops Architecture

In this section, we present the details of the Cyclops system architecture. The block diagram in Figure 2 shows the main pieces, which are described in detail in the following subsections and summarized as follows: The user writes PRU code in a web browser that runs a simple IDE from a web server. The IDE is shown in Figure 1. Code can be written in PRU assembly language (pasm), or a custom JavaScript-like language we developed for Cyclops. The IDE converts this code into pasm, compiles it into an executable binary, then copies it to the PRU's instruction memory using a program called the "PRU Loader". The PRU Loader then uses a Linux kernel module (LKM) we wrote called "pin-pirate" to

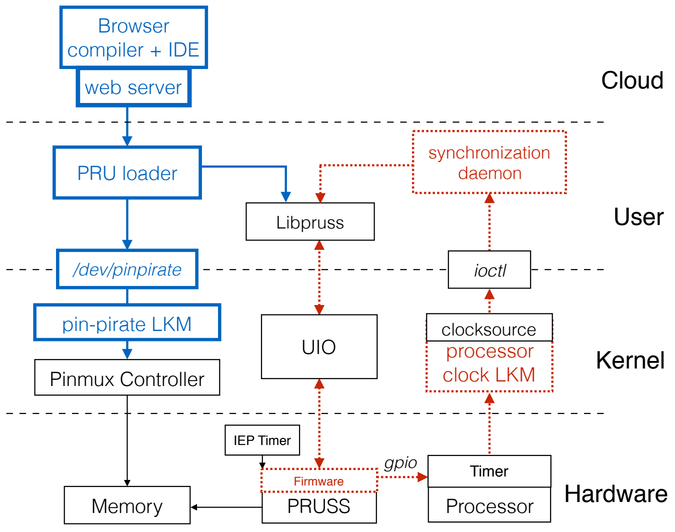


Fig. 2: Cyclops system architecture. Components built-in to the Beaglebone are outlined as regular black blocks, Cyclops components are bold blue, and the components associated with our PRU/Processor time-synchronization are dotted red. These components are described in detail in Section III.

configure the Beaglebone's pins for use with the PRU and peripherals like the ADC, GPIO, and PWM. The right-hand side of Figure 2 is related to PRU/CPU clock synchronization and is described in Section III-E. We now describe these main pieces of Cyclops in more detail.

A. Web-browser-based PRU IDE

We implement the IDE as a web-browser-based application, a screenshot of which appears in Figure 1. The user writes Cyclops code in the upper-left window. The "compile" button converts the Cyclops program into pasm, which is shown in the lower-left window. This PRU code can be edited in-place for minor corrections and tweaks which is a great feature in Cyclops to the make the generated code readable. Clicking "run" compiles the pasm to PRU machine code, sends it to the PRU's instruction memory, configures the pins as inputs or outputs as specified by the user's program, then starts the PRU. Any data from standard output or error is piped back into the IDE's terminal pane.

B. Programming Language, Compiler, and Loader

The Cyclops IDE can be programmed in pasm, or a custom language we developed: the Cyclops language. The syntax is similar to JavaScript; it is intended to have a minimal learning curve and serve as a way to quickly experiment with the PRU. The goal of the Cyclops language is to combine the convenience of C with the determinism of pasm. It achieves this by supporting a small feature set: variable assignment, arithmetic expressions, if/else conditional statements, and while-loops. Notably it does not support for-loops, functions, or floating-point arithmetic. Consequently, it forces the programmer to write simple code more appropriate for a real-time application, with more deterministic runtimes. Moreover, unlike the Texas Instruments PRU C compiler, we contribute the Cyclops language and compiler to the open-source software community for inspection and improvement. The source code for the Cyclops library is hosted here:

- https://github.com/nesl/Cyclops-PRU

For the Javascript-inspired Cyclops language, variables do not need to be explicitly declared, but rather are implicitly declared on first assignment. The number of variables in a program is limited to the size of the register file. Keeping all variables in registers avoids nondeterministic latencies due to register spilling, and helps keeping the compiler design simple. Cyclops reserves `r0` as the zero register and `r1` as a stack pointer for passing arguments to the print function. In the case of arithmetic expressions, intermediate results are assigned to temporary registers that are scoped to the current statement. The range of registers available for general and temporary use can be customized by the user.

The Cyclops language provides built-in variables that are linked to its single-cycle GPIO pins. The variables have the same names as the Beaglebone's P8 and P9 header pins, e.g., `P8_45`. The compiler infers whether or not a pin is for input or output based on its first use: A statement assigning a 1 or 0 to the pin variable implies that the GPIO pin should be configured an an output. Reading a pin variable, for example in a conditional statement like `if(P8_45){...}`, results in Cyclops auto-configuring the pin as an input with an internal pull-down resistor. Cyclops throws a compilation error if a pin is being used for both input and output in the same program.

The PRU Loader is a simple userspace application. It copies the PRU binary into the PRU and configures the BBB's pins as specified by the user's code. At load-time, it uses the Libpruss UIO driver to copy the binary to the PRU's instruction memory, and it uses our "pin-pirate" Linux kernel module to configure the SoC's pinmux appropriately. This LKM is described next.

C. Automatic Pinmux Configuration with Pin-Pirate

As described in Section II-B, it can be challenging to configure a Beaglebone's device tree overlay correctly. To address this, Cyclops provides an alternative method of configuring the hardware during runtime: a Linux kernel module named "pin-pirate." Pin-pirate is a simple LKM that configures pins by writing directly in the pinmux controller's hardware registers. Normally, the Beaglebone's memory-protection unit protects the SoC's control registers from userspace, so one cannot simply `mmap` the control registers from userspace; however, as an LKM, pin-pirate executes in privileged mode and can thereby bypass this protection. Pin-pirate is exposed to userspace as a simple character device located at `/dev/pinpirate`. If udev rules are configured properly on the system, then root permissions are not required to access this device. To change a pin configuration one can simply run `echo reg val > /dev/pinpirate`, where `reg` and `val` are the register offset and value of the pin for the pinmux controller, as described in the TI AM335x Technical Reference Manual, section 9.3.1.

D. PRU C library for I/O and Time Synchronization

The Cyclops IDE and Javascript-inspired Cyclops language provide a quick way for programmers to get started with PRU development. However, the conventional method for programming the PRU involves writing C and using the Texas Instruments PRU C compiler. For users who prefer this method of development, Cyclops includes a C library with the same convenient functions offered by the Cyclops language for I/O and time synchronization. A summary of these functions appears in Table I.

TABLE I: The PRU C library API functions.

Description	Example
Configure the ADC.	`init_ADC()`
Read the ADC.	`read_adc_data()`
Read GPIO.	`read_pin(P8_45)`
Write GPIO.	`assert_pin(P9_27)`
Toggle GPIO.	`toggle_pin(P8_46)`
PRU blocks for x μsec.	`WAIT_US(x)`
PRU reads the 32-bit IEP timer and converts that to nanoseconds.	`read_pru_time()`
Adjust PRU clock by x nanoseconds.	`adj_pru_time(x)`

E. Synchronization of PRU with Processor clock

Even though IEP timer and the main processor's timer are on the same SoC, they are driven by separate crystal oscillators. The hardware timer driving the processor time is driven by a 24 MHz oscillator, whereas the oscillator driving the IEP timer for the PRU clock is a separate 25 MHz oscillator. Therefore over time their clocks will drift, requiring re-synchronization. Cyclops includes the means to synchronize them, described next.

The red boxes in Figure 2 represent Cyclops components that synchronize the PRU time to processor time. This enables the PRU to time-stamp sensor measurements with processor time, instead of a PRU-specific time. The synchronization scheme works as follows: The synchronization daemon in Figure 2 initiates time synchronization by making the PRU assert pin P8_46. The PRU has single-cycle access to this GPIO pin, so this signal has very precise time resolution. At pin assertion, the PRU calls the Cyclops function `read_pru_time()`, which records its own timestamp in cycles using the 32-bit IEP timer and converts it to nanoseconds (T_1). Pin P8_46 is physically wired to pin P8_9, which is monitored by an input-capture channel of the processor timer. This channel records the cycles of the processor timer when pin P8_9 asserts. The "processor clock LKM" shown in Figure 2 is a loadable kernel module that presents the processor clock as a Linux PTP-based `clocksource` abstraction. This clocksource converts the processor timer's raw 32-bit cycles — captured upon pin assertion — to nanoseconds (T_2). The userspace synchronization daemon uses `ioctl` calls to read the PTP-based clocksource's value in nanoseconds (T_2), and communicates it to the PRU using the UIO LKM. The PRU defines the value `offset` to be ($T_1 - T_2$) and adjusts its own clock using the Cyclops C function `adj_pru_time(offset)`.

TABLE II: Three methods of generating a PWM signal on pin P8_45 with a 1-second period and 50% duty cycle: Raw pasm assembly language (left), the Cyclops C library (middle) with functions for I/O, and the Cyclops language (right) with built-in variables for I/O operations.

pasm

```
START:
    MOV r0, 0xBEBC200 // count for
     0.5 seconds (2 * 10^8 cycles)
MAINLOOP:
   SET r30.t1   // set the output pin
     P8_45 high
    MOV r1, r0   // wait for 0.5
     seconds
HOLD_HIGH:
    SUB r1, r1, 1
    QBNE    HOLD_HIGH, r1, 0
    CLR r30.t1   // set the output
     pin P8_45 low
    MOV r1, r0   // wait for 0.5
     seconds
HOLD_LOW:
    SUB r1, r1, 1
    QBNE    HOLD_LOW, r1, 0
    QBA MAINLOOP
```

C with Cyclops C lib

```
pin =  P8_45;
// 1 second period
period_us =  1000000;
// 0.5 second  duty cycle
pulse_width_us =  500000;
while(true){
  // Output high
  assert_pin(pin);
  // Wait for  0.5 seconds
  WAIT_US(pulse_width_us);
 // Output low
  deassert_pin(pin);
  // Wait for  0.5 seconds
  WAIT_US(period_us-pulse_width_us);
}
```

Cyclops language

```
// Each tick is 5 nanoseconds.
while(true){
// reset the cyclecounter
tick = 0;
// Output high
P8_45 = 1;
// Wait for 0.5 seconds
// (2*10^8 cycles)
while (tick < 0xBEBC200) {}
  tick = 0;
  // Output low
  P8_45 = 0;
  // Wait for 0.5 seconds
  while (tick < 0xBEBC200) {}
}
```

Since the PRU runs very predictably, the runtimes of these function calls are deterministic and we account for them in the time-synchronization algorithm; their runtimes are tallied in Table III. Once the PRU is synchronized, all timestamps from the PRU are expressed in processor time.

It should be noted that this synchronization scheme only corrects the offset between the IEP timer and the main processor timer and does not attempt to estimate the skew between them. Higher-order synchronization methods to estimate clock skew and its temperature dependence are outside the scope of this paper.

TABLE III: Precise runtime of functions related to time-synchronization in the Cyclops C library.

Function	# cycles	# nanosec
`read_pru_time()`	124	620
`adj_pru_time(x)`	58	290
`read_raw_iep()`	23	115

IV. Evaluation

In this section we evaluate the Cyclops software in two ways. First we compare code for generating a PWM signal in pasm, Cyclops C library, and the Cyclops language, and show that both the Cyclops C library and the Cyclops language provide benefits over pasm. We then demonstrate how to use Cyclops to synchronize the PRU clock to the processor clock, so that PRU-measured sensors can be time-stamped with processor time, facilitating real-time applications that use both Linux and the PRU.

A. Comparison of Cyclops code

Table II demonstrates three implementations of a software-based PWM signal on pin P8_45 of the Beaglebone Black using the PRU. The implementation on the left uses pasm. The middle column implements PWM in C; the red-colored functions are provided by the Cyclops C library and facilitate I/O and waiting for precise times. The rightmost column in Table II is written in the Cyclops language using the Cyclops IDE. Our IDE converts this code directly into pasm. Note that the variables `tick` and `P8_45` are pre-defined by Cyclops, and resolve to the PRU's cycle counter and its single-cycle GPIO pin respectively. These built-in variables and the packaged API simplifies PRU configuration and reduce programming effort, leading to more readable and deterministic code.

Note that the code in Table II is merely an example; one would normally use the built-in peripherals for PWM instead of dedicating a PRU to it.

B. Using Cyclops for Time-Synchronization

To show the performance of synchronizing the PRU's clock with the processor clock, we first configured the system for time-synchronization as described in Section III-E. Then we captured simultaneous clock values at 250-ms intervals from the PRU and processor clocks in the following way. We triggered clock-sampling instants via an external 250-ms-period square wave fed to GPIOs at pins P8_45 and P8_10. The PRU monitored pin P8_45; upon a transition, it read the IEP timer, converted it to nanoseconds according to the disciplining parameters provided by the clock synchronization, and sent the result to a listening userspace thread. On the processor side, an input-capture channel monitored pin P8_10; upon a transition, it recorded the value of processor timer, which our processor clock LKM then converted to nanoseconds. This procedure results in a list of pairs of 64-bit integers of nanosecond clock values: one from the PRU clock, and one from the processor clock. The difference between these two numbers is the clock-synchronization error.

We plot histograms of this time-synchronization error in Figure 3. To explore the sensitivity of synchronization error to the synchronization interval, we ran three experiments, each for 30 minutes, and each with a different rate of re-synchronization (resync). In Figure 3a, the synchronization daemon re-synchronizes the PRU clock every 8 seconds,

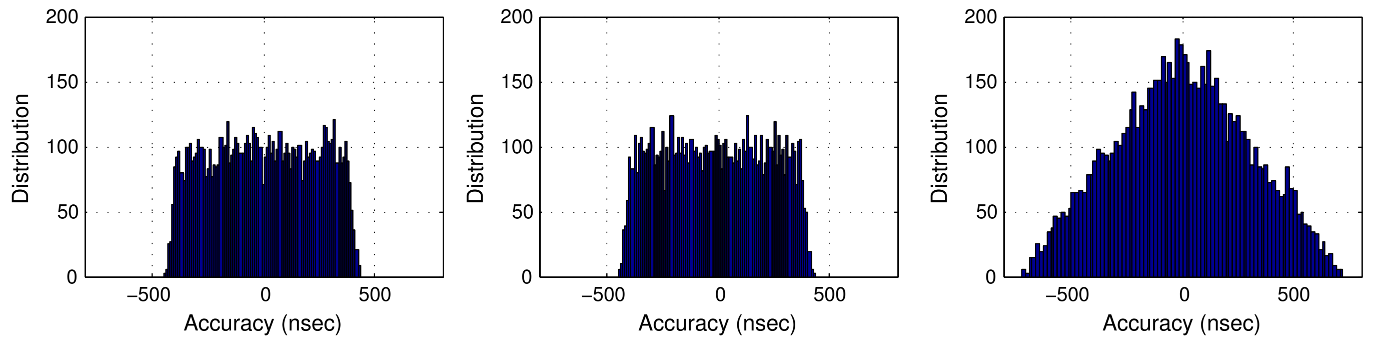


Fig. 3: Synchronization Accuracy between PRU and Processor Clock.

(a) Synchronization Period = 8 seconds (b) Synchronization Period = 32 seconds (c) Synchronized only once at beginning of test

resulting in a maximum synchronization error of 432 ns with an RMS accuracy of 233 ns. Similar results appear in Figure 3b, in which the resync period was increased to 32 seconds. The uniform distributions are due to small fixed resync periods and fixed sampling resolution. Figure 3c shows the result of synchronizing the clocks only once, then letting the two clocks drift apart for 30 minutes. In this case, the error is bounded by 720ns, with RMS error of 297ns. Note that this error distribution shows the effect of clock skew which was not visible in Figure 3a and 3b due to smaller resync periods. We conclude from these results that we can achieve sub-microsecond synchronization accuracy between the PRU and processor time with only a modest resync period.

V. Conclusion

Single-board computers with real-time co-processors offer a platform for real-time applications that can combine the advantages of operating systems and bare-metal platforms. This paper presented a suite of software — Cyclops — to facilitate the development of real-time applications on the Programmable Real-time Unit within the Beaglebone Black platform. Cyclops provides automated configuration of the PRU, support for a high-level programming language, and a library to interface the PRU with several I/O peripherals on the Beaglebone. We demonstrated how one can use these tools to synchronize the PRU clock with the processor clock, achieving sub-microsecond synchronization accuracy with a low re-synchronization frequency. This lays the foundation for embedded applications which enjoy the features of a full OS like Linux and use the "bare metal" PRU for time-critical operations. Future work may explore time-synchronizing PRUs across multiple nodes, enabling microsecond-accurate coordination in a distributed control system.

Acknowledgment

This research is funded in part by the National Science Foundation under awards # CNS-1329755 and CNS-1329650, and by the King Abdullah University of Science and Technology (KAUST) through its Sensor Innovation research program. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the funding agencies.

References

[1] D. Molloy, *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux*. John Wiley & Sons, 2014.

[2] B. L. J. E. Karl-Erik, A. A. Cervin, and D. Henriksson, "How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime," *Control Systems Magazine*, vol. 23, pp. 16–30, 2003.

[3] A. McPherson and V. Zappi, "An environment for submillisecond-latency audio and sensor processing on beaglebone black," in *Audio Engineering Society Convention 138*. Audio Engineering Society, 2015.

[4] A. M. Anand, B. Raveendran, S. Cherukat, and S. Shahab, "Using pruss for real-time applications on beaglebone black," in *Proceedings of the Third International Symposium on Women in Computing and Informatics*. ACM, 2015, pp. 377–382.

[5] K. Kepa and N. Abaid, "Development of a frequency-modulated ultrasonic sensor inspired by bat echolocation," in *SPIE Smart Structures and Materials+ Nondestructive Evaluation and Health Monitoring*. International Society for Optics and Photonics, 2015, pp. 942 913–942 913.

[6] B. Travaglione, "Using a single-board microcontroller and adc to perform real-time sonar signal processing."

[7] M. Götz, M. W. Gobetti, and F. B. Libano, "A grid-tie micro-inverter software development based on a low cost multiprocessor platform," in *Computing Systems Engineering (SBESC), 2015 Brazilian Symposium on*. IEEE, 2015, pp. 122–127.

[8] J. Chaoui, K. Cyr, S. de Gregorio, J. P. Giacalone, J. Webb, and Y. Masse, "Open multimedia application platform: enabling multimedia applications in third generation wireless terminals through a combined risc/dsp architecture," in *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*, vol. 2, 2001, pp. 1009–1012 vol.2.

[9] F. Anwar, S. Dsouza, A. Symington, A. Dongare, R. Rajkumar, A. Rowe, and M. Srivastava, "Timeline: An operating system abstraction for time-aware applications," in *Real-Time Systems Symposium (RTSS), 2016 IEEE*. IEEE, 2016, pp. 191–202.

[10] A. Alanwar, F. Anwar, J. P. Hespanha, and M. Srivastava, "Realizing uncertainty-aware timing stack in embedded operating system," in *Proc. of the Embedded Operating Systems Workshop*, Oct. 2016.

[11] D. L. Mills, "Internet time synchronization: the network time protocol," *Communications, IEEE Transactions on*, vol. 39, no. 10, 1991.

[12] K. Lee, J. C. Eidson, H. Weibel, and D. Mohl, "Ieee 1588-standard for a precision clock synchronization protocol for networked measurement and control systems," in *Conference on IEEE*, vol. 1588, 2005, p. 2.

[13] R. Cochran and C. Marinescu, "Design and implementation of a ptp clock infrastructure for the linux kernel," in *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2010 International IEEE Symposium on*. IEEE, 2010, pp. 116–121.