Hardware and Software for a Power-Aware Wireless Microsensor Node

by

Nathan J. Ickes

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

May 2002

© Massachusetts Institute of Technology 2002. All rights reserved.
Hardware and Software for a Power-Aware Wireless Microsensor Node

by

Nathan J. Ickes

Submitted to the Department of Electrical Engineering and Computer Science on May 24, 2002, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis examines important issues in the design of hardware and software for microsensor networks, with particular attention paid to mechanisms for providing power awareness. The $\mu$AMPS Revision 1 microsensor node is used as an example. The design of this node implementation is described in detail, including, in particular, the design of the $\mu$AMPS processor board and its power-scalable architecture. The operating system and application programming interface for the node is described. Finally, an analysis is made of the power consumed by each of the node’s subsystems, and these results are used to predict the degree of power-awareness provided by the $\mu$AMPS Revision 1 node.

Thesis Supervisor: Anantha Chandrakasan
Title: Associate Professor
Acknowledgments

The author would like to thank Rex Min for his comments and critiques, especially early in the design of the $\mu$AMPS Revision 1 processor board, as well as Piyada Phanaphat and Fred Lee, who were instrumental in making $\mu$AMPS Revision 1 a fully functional node.

This research is sponsored by the Defense Advanced Research Project Agency (DARPA) Power Aware Computing/Communication Program and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0551; and by the Army Research Laboratory (ARL) Collaborative Technology Alliance, through BAE Systems, Inc. subcontract RK7854.
Contents

1 Microsensor Networks 15
   1.1 Introduction ........................................ 15
   1.2 Microsensor Attributes ................................ 15
   1.3 Power Awareness ...................................... 18
   1.4 The µAMPS Project .................................. 19
       1.4.1 Project Phases and Timeline ..................... 19
       1.4.2 Goals for the Revision 1 Node ................. 20

2 Microsensor Node Architecture 21
   2.1 Node Components .................................... 21
   2.2 µAMPS Revision 1 Node Architecture ................ 25

3 Implementation of the Revision 1 Node 31
   3.1 Overview ............................................. 31
   3.2 Processor Board ..................................... 31
       3.2.1 Processor and Memory .......................... 33
       3.2.2 Power Supplies ................................ 34
       3.2.3 Analog Power Supply ........................... 35
       3.2.4 Digital Logic Supply ............................ 36
       3.2.5 5V Supply ....................................... 37
       3.2.6 Microprocessor Core Supply .................... 37
       3.2.7 Acoustic Sensor ................................ 40
       3.2.8 I/O ............................................... 43
3.3 Processor Circuit Board ........................................ 45
3.4 Radio Board ......................................................... 45
3.5 Basestation board ................................................... 45
3.6 Four-Channel Acoustic Sensor Board ......................... 48
3.7 Battery ............................................................... 50
3.8 Summary of Power-Aware Design Techniques ............... 51

4 Microsensor Operating Systems .................................. 53
  4.1 Requirements ...................................................... 53
  4.2 eCos ................................................................. 54
    4.2.1 RedBoot ....................................................... 55
  4.3 Porting eCos to $\mu$AMPS ..................................... 56
  4.4 Power Management in eCos .................................... 58
    4.4.1 Idle Mode .................................................... 59
    4.4.2 Sleep Mode .................................................. 60
    4.4.3 Clock and Voltage Scaling ................................. 61
  4.5 Application Programming Interface ............................ 63

5 Performance Analysis of the $\mu$AMPS Revision 1 Node ...... 65
  5.1 Intent of the Analysis .......................................... 65
  5.2 Power Measurements ............................................ 66
    5.2.1 Subsystems and Their Modes of Operation ............... 66
    5.2.2 Frequency and Voltage Scaling of the StrongARM Processor . 69
    5.2.3 Node-Level Power Management ............................ 72
  5.3 Test Applications ............................................... 75

6 Conclusions ......................................................... 79

A The $\mu$AMPS Revision 1 Low-Level Node API .................. 83
  A.1 Overview ......................................................... 83
  A.2 Acoustic Sensor: <uapi/analog.h> ........................... 84
  A.3 Node Power Management: <uapi/power.h> .................... 87
List of Figures

1-1 Generic microsensor application ........................................ 16
2-1 Typical microsensor node architecture ............................... 22
2-2 $\mu$AMPS Revision 1 block diagram ................................ 25
2-3 $\mu$AMPS Revision 1 node physical architecture .................. 27
2-4 System connector pinout .................................................. 28
2-5 The $\mu$AMPS node. Clockwise from upper left: A complete basestation node, with basestation, processor, and radio boards; the radio board; the processor board; a U.S. quarter for size comparison, and an example external sensor board providing four microphone inputs for acoustic sensing and a battery pack. ........................................ 29
3-1 Block diagram of the processor board ............................... 32
3-2 Microprocessor core supply .............................................. 38
3-3 Sensor Circuitry .............................................................. 41
3-4 Photograph of the processor board showing important component locations .................................................. 46
3-5 Block diagram of basestation board ................................. 47
3-6 Block diagram of four-channel acoustic sensor board .......... 49
3-7 Photograph of four-channel acoustic sensor board .............. 50
4-1 $\mu$AMPS memory map .................................................... 57
5-1 Power management capabilities of the $\mu$AMPS node .......... 67
5-2 SA-1110 power consumption and minimum voltage versus frequency .............................................................. 73
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Comparison of the SA-1100 and SA-1110</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Summary of power supplies on the processor board</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Frequency, voltage, and memory timing combinations.</td>
<td>62</td>
</tr>
<tr>
<td>5.1</td>
<td>Power consumption of node subsystems in all supported modes of operation</td>
<td>68</td>
</tr>
<tr>
<td>5.2</td>
<td>Minimum voltage at each operating frequency, for three different SA-1110 processors.</td>
<td>71</td>
</tr>
<tr>
<td>5.3</td>
<td>Core power consumption for the SA-1110 at each operating frequency</td>
<td>72</td>
</tr>
</tbody>
</table>
Chapter 1

Microsensor Networks

1.1 Introduction

Microsensor networks are distributed collections of small, connected sensor nodes. Although each node may have very limited sensing abilities, the nodes collaborate to form a composite examination of their environment that may be richer than an examination obtainable with a single, large, complex macrosensor. The size, number, and distribution of the nodes depends highly on the purpose of the sensor network, and can range from a few nodes distributed in a small room, to hundreds of larger nodes spread across a mountainside, to thousands of nodes monitoring a factory assembly line.

This thesis discusses microsensor networks in general, but pays particular attention to the hardware and software developed for MIT’s μAMPS project. The details of the design of the μAMPS Revision 1 sensor node are discussed, and an analysis of the node’s measured performance is made.

1.2 Microsensor Attributes

Figure 1.2 illustrates a typical microsensor setup. Sensor nodes are distributed throughout the area under observation, and are connected via hierarchical communication network. Communication can be wired, but is generally wireless. At the root
of the communication network is the base-station node. This is the point where all data is collected from the network. The base-station may be connected to a mass storage system, or a special long-range communications system. In this generic example, the base-station is the only special node: all of the other nodes are identical in construction and programming. This generality facilitates deployment of the system, but requires the use of potentially complicated ad-hoc networking protocols, as each node must discover its role in the network. Network roles may even change over time, in the event that some individual nodes fail or run out of energy.

The primary advantage of microsensor networks derives from the spatial diversity of the data collected by the network as a whole. This diversity can be exploited to reveal details about the network’s environment that would not necessarily have been determinable using a single macrosensor. Such details might include the spatial location of some signal source, or at least the direction from which a particular signal is coming. Alternatively, the sensor network may be used to imitate a single very large sensor, one that might be impractically large to build or deploy.[1, 2]

A second advantage of networked microsensor is the potential for increased reliability through redundancy. In an ideally arranged network, the loss of any given node should have only minimal impact on the functionality of the sensor system as
a whole. Assuming the nodes are distributed densely, compared to the range of each communication link, and that the links between nodes are reconfigurable, the network should be able to reconfigure itself to maintain its connectedness. The missing data from a failed node is only a small portion of the total data collected by the network. Unfortunately, the single basestation node is an exception to this general redundancy. However, since there is only one basestation, it is frequently possible to take measures to increase its reliability, such as providing the basestation with an increased capacity power source, and positioning the node such that it is easily maintainable.

Since networking is what makes microsensor systems interesting, the design of inter-node communication mechanisms is critical. In some applications, it is possible to provide wired connectivity between nodes. Wired connections are simple to implement, reliable, and use only small amounts of power. However, microsensor applications requiring large numbers of nodes or fast deployment times necessitate wireless communication. Wireless communication adds several complications. First wireless transceivers are generally larger, more complex, and more power-hungry than their wired counterparts. Second, wireless communication links are generally limited to lower bandwidth and shorter distances than wired links. Limited bandwidth in turn limits the amount of collaboration and data sharing between nodes. Limited link distance either limits the maximum diameter of the network, or necessitates complex multi-hop routing protocols.

The small size of microsensor nodes may facilitate their deployment, but it severely complicates the problem of providing power to the nodes. Like wired networking, the ability to tap into a large, external power source is a rare luxury. For most applications, nodes must carry their own power supplies, and this generally means batteries. In some cases, nodes may scavenge energy from their environment. Example environmental energy sources include solar radiation, or mechanical vibration. Self-powered nodes have the potential to create sensor networks with near infinite lifetimes; however, generally only very small amounts of power can be extracted from the environment, requiring nodes to run on very little power, and to manage their energy consumption very carefully.
1.3 Power Awareness

A key concept in power management is the difference between low-power design, and power-aware design. Of course, if energy consumption is to be minimized, the design should be optimized to use as little power as possible. However, reducing power consumption frequently requires a corresponding decrease in some other performance metric, such as computational speed or radio range. A power-aware design recognizes that in practice, some performance metrics can occasionally be relaxed, without any decrease in real performance.

For example, a design specification may require a radio transmitter powerful enough to communicate over a distance of 100m. However, if, in some situation, the receiver is only 5m from the transmitter, it would be beneficial to be able to reduce the transmit power to just the level required to reach the receiver. Similarly, peak anticipated computational loads may require a 50MIPS processor. However, when the load is less than maximum, the additional processing performance is wasted, along with the power necessary to maintain that level of performance. A power-aware design would allow the processor clock speed to be varied, so that 50MIPS performance was available when needed, but power consumption could be decreased when the processor load is less.

Power-aware designs are characterized by multiple operating modes that permit power/performance tradeoffs to be made in real time. To be effective, it is necessary to intelligently select which mode to use in any given scenario. In practice, this is a difficult problem, often requiring feedback loops that compare measured performance with target performance, and gradually home in on the optimal operating mode. Sensor systems are inherently real-time systems, where data-processing deadlines must be met, or the data becomes stale. Even if it was possible to instantaneously determine the optimal operating mode for a given scenario, it is generally not possible to instantaneously change modes. In power-aware systems, the delay in changing operating modes is frequently greater when changing to a mode that offers increased performance than when changing to a lower performance, lower-power mode, espe-
cially when additional hardware must be brought online in order to generate the increased performance. It is therefore necessary to anticipate the need for increased performance, in order for it to be available exactly when needed.

1.4 The $\mu$AMPS Project

The MIT $\mu$AMPS (Micro, Adaptive, Multi-Domain Power-Aware Sensors) project is a four year effort to develop a flexible architecture for microsensors. $\mu$AMPS nodes are not intended for any specific application: rather, the $\mu$AMPS project focuses on hardware, protocol, and algorithm building blocks that are useful for many different applications. As implied by the project name, the $\mu$AMPS nodes are highly power-aware. The final goal of the project is to develop a highly-miniaturized, self-powered sensor node on a single chip.

1.4.1 Project Phases and Timeline

$\mu$AMPS is a three-phase project, with phase 0 presently complete, and phase 1 nearing completion.

Phase 0 Phase 0 was an exploratory, proof-of-concept project, in which various microsensor node components were prototyped. To facilitate debugging, the prototypes were based on commercial development board designs. The prototypes were therefore neither small, nor highly power efficient, and there was little integration between the different modules of the system at this point. These early prototypes, however, served as a useful testbench and software development platform, and identified problems that were avoided in the next design iteration. Phase 0 was completed in the Spring of 2000.

Phase 1 This thesis centers on the design of the Revision 1 $\mu$AMPS node, for phase 1 of the $\mu$AMPS project. This is the first fully functional $\mu$AMPS node. It is designed to be as small and power-efficient as possible, while still being base on
off-the-shelf components. Phase 1 will be complete after the Summer of 2002.

**Phase 2** The final phase of the \( \mu \)AMPS project will be a custom system-on-a-chip implementation of the sensor node, which will build upon the successful ideas from phase 0 and 1, and will also take advantage of additional opportunities afforded by a chip-level design, including ultra-low-power VLSI design techniques, custom power-aware computing architectures, and dedicated circuitry for low-power network protocol processing. Phase 2 is expected to be completed during 2004.

### 1.4.2 Goals for the Revision 1 Node

The primary purpose of the Revision 1 node is to implement a fully-functional microsensor node, and to demonstrate its functionality and its power-aware design in a few assorted applications. Approximately ten nodes are to be built, in order to allowing small networks to be deployed and tested.

Secondary goals were that the revision 1 node be as small and power efficient as possible. Restricting the design to off-the-shelf components limits the degree to which these secondary goals can be achieved. However, based on the performance attained by modern cellular phones and personal digital assistants, lifetimes of about a week should be attainable from a sensor node with a volume of about 100cm\(^3\), given relatively low duty cycles (active for 1–5% of the time, as typical of many actual microsensor applications).
Chapter 2

Microsensor Node Architecture

2.1 Node Components

Figure 2.1 identifies the essential components of a wireless microsensor node, and indicates some of the design considerations associated with each component.

Sensors  A sensor node must include some form of sensor. The \( \mu \)AMPS project has so far implemented only seismic and acoustic sensors, although many other sensing mechanisms can be used effectively in a microsensor system. With COTS components, as used in the current iterations of the \( \mu \)AMPS nodes, the sensor system (including transducers, amplifiers, filters, and analog-to-digital converters) can be built to run on only a few milliwatts, making it a very minor power consumer compared with the radio or microprocessor. Therefore, the ability to turn off a sensor (lowering its power consumption to a few microwatts) generally provides sufficient power scalability. The development of custom micropower processors and radios dedicated to distributed sensor applications promises to reduce the power consumption of these systems from hundreds of milliwatts to a few milliwatts, or even microwatts. When this occurs, the ability to make dynamic power/performance tradeoffs in the sensor hardware will be much more important. Mechanisms for doing this might include changing the bias current applied to a transducer, changing the number of gain stages in an amplifier chain, or changing the order of a switched-capacitor filter.
Processors  Power-awareness implies some rudimentary form of intelligence, and that intelligence is invariably provided by a microprocessor. How much computational capability is required depends highly on the type of sensors used and the amount of post-processing of the raw sensor signal required, the architecture of the sensor network and how computational tasks are distributed among the nodes, and latency, resolution, or other quality of service constraints placed on the overall network. Achieving power-awareness in a processing element is a matter of dynamically adjusting the computational capacity of the processor to exactly match its workload. This can be achieved by varying the frequency of the clock along with the power supply voltage, or by shutting down unused functional units.

Software  Any system that includes a microprocessor necessarily also includes software. In a typical microsensor network, software running on each node controls the manner in which data is collected, the mechanisms for sharing data between nodes and for combining data streams from multiple nodes (data aggregation), and the algorithms for optimizing power consumption. Besides controlling power aware hardware, software can be a means of power awareness in itself. An example would be the
inclusion of multiple, scalable algorithms for the same task. When available power is low, a computationally intensive, but highly optimized data aggregation algorithm might be replaced by a much simpler but coarser algorithm. Another example would be varying the number of taps in an FIR filter. The concept of low-power, or power-aware software, is relatively new, and the tools necessary for creating it are only now being developed. Traditional compilers have usually optimized for execution speed, which is usually roughly equivalent to optimizing for lowest energy. However, the use of specialized hardware, such as is found in a highly power-aware processor, may change that.

**Radios** Once data has been collected by an individual sensor node, it must be transported to the central collection point for the network, a job for the radio system. Radio power consumption is fundamentally controlled by range and bandwidth requirements. Receiving a signal can be more difficult than transmitting one: in the low-power, short-range radios typical of microsensors, the receiver may consume more power than the transmitter. In physically dense networks, nodes must avoid interfering with each other’s radio transmissions. Radio systems present a variety of opportunities for power aware design. Scaling transmitter power is but a simple example. Realizing that a radio system encompasses more than just the RF circuitry, power awareness can be achieved in the use of multiple error correction protocols and multiple or variable higher-level networking protocols.

**Power Sources** Every node must have a power source. It is fundamentally the finite capacity of common power sources that limits the lifetime of a sensor network and motivates low-power or power-aware design. Size constraints frequently rule out the possibility of extending the lifetime by simply enlarging each node’s power source.

For most small, mass-produced sensor nodes, the only currently practical power source is batteries. Primary (non-rechargeable) batteries provide the highest energy density. Secondary (rechargeable) batteries may be appropriate in applications where the sensor nodes can be collected afterwards and reused, however, the luxury of
reusable batteries brings with it a significant reduction in energy density.

As the power consumption of sensor nodes continues to decrease, it is becoming possible to build “self-powered” nodes that extract all of the energy they need from their environment and can therefore run indefinitely without maintenance. Energy can be extracted from sunlight using photovoltaic cells, from other forms of electromagnetic waves using appropriately sized antennas, or from mechanical vibration using MEMs (micro-electromechanical systems) transducers. Direct sunlight provides up to 1000W/m². Using modern high-efficiency solar cells, which achieve efficiencies in the range of 18–25%, a 5cm square node could therefore extract up to about 500mW of power, depending on the time of day. This is certainly sufficient to power even a node based on off-the-shelf components at a low duty cycle.

**Power Converters**  The output of energy storage devices is seldom directly compatible with microsensor electronics. The output voltage of a storage device is generally determined by the physics or chemistry of the storage mechanism. Furthermore, the voltage may change, depending on the amount of energy available. Interfacing the storage devices with the electronics is the task of power converters. Generally, these converters take the form of switch mode dc/dc converters. Such converters are never perfectly efficient, although conversion efficiencies of 95% are not uncommon with carefully designed converters. Conversion efficiency is generally a function of the load on the converter, with efficiencies dropping as the load current drops. The mobile electronics industry has resulted in a abundance of high efficiency dc/dc converter controller chips for converters with outputs of 1–5V, and anywhere from 100mA to 50A. Unfortunately, the time-varying current draw of a power-aware microsensor makes it difficult to select a single, optimal controller from among the commercial offering.
2.2 µAMPS Revision 1 Node Architecture

The µAMPS node is designed for flexibility. Since µAMPS is not dedicated to any one particular microsensor application, it must be easy to equip the node with virtually any kind of sensor. The node is, however, optimized for acoustic sensing, since acoustic sensors are easy to test in the laboratory.

The electrical block diagram of the µAMPS Revision 1 node, shown in Figure 2.2, closely matches the typical node block diagram in Figure 2.1. The sensor block consists of a microphone, amplifier, anti-aliasing filter, and analog-to-digital converter. Power control signals from the processor allow each of the sensor components to be shut down when not needed. Although the sensor is only active when all of its components are powered, the ability to power down only selective portions of the sensor circuitry creates additional power scalability by creating standby modes where power consumption is intermediate between the full active and full shutdown states, but the delay required to transition back to the active state from idle is less than the delay to transition to the active state from the full shutdown state.

The processor block consists of a StrongARM microprocessor, along with low-power SRAM and a flash ROM. The StrongARM processor was chosen because of its high performance/power ratio, and its built-in variable frequency (59–221MHz) core clock generator. Varying the processor clock speed, in real time, is an important
part of the $\mu$AMPS power-awareness strategy. In the $\mu$AMPS Revision 1 node, the processor voltage is varied along with the clock frequency. This is accomplished with a special dc/dc converter built into the processor board. Reducing the voltage applied to the processor core to the lowest level possible to support the current operating frequency increases power savings at low clock frequencies by reducing both switching and leakage currents.

The radio block is subdivided into a digital baseband component (implemented on an FPGA) and an RF component (implemented with discrete components and a Linear $\mu$). The digital component is responsible for encoding, decoding, and error detection/correction, as well as controlling the timing of transmitter and receiver according to the TDMA scheme employed by the network protocol. The RF circuitry consists of an Linear Semiconductor LMX3162 2.4GHz radio, along with the variable power amplifier for transmitting, low noise amplifiers for receiving, a VCO, and an antenna. As with the sensor circuitry, power to the various components of the radio is controlled by the processor, allowing components to be shutdown when idle.[9][16]

Figure 2.2 shows the physical architecture of the $\mu$AMPS Revision 1 node. The node consists of a stack of three or four printed circuit boards. Each board is 55mm square. A system connector, present on each board, links the boards electrically, creating a common bus of control signals. The top-most board contains the radio, including the RF circuitry and an FPGA used for digital coding and decoding. The second board contains an Intel StrongARM processor, and associated RAM and flash ROM. Also on the processor board are an acoustic sensor (microphone, amplifier, and analog-to-digital converter) and a collection of dc/dc power converters that service the entire node. The optional third board down in the stack is an additional sensor module, to replace the acoustic sensor on the processor board. The $\mu$AMPS Revision 1 node can be easily adapted to different applications by designing an appropriate sensor board. The bottom board in the stack contains the power source. For a typical node, this consists of a battery pack, containing either four AAA cells, or (not yet implemented) two lithium-ion rechargeable cells.

To convert a typical sensor node into a basestation node, the battery board is
replaced with a PC interface board. This board provides standard connectors (RS-232 and USB) for interfacing to a larger computer. The interface board also contains voltage regulators that allow the basestation node to be powered from ac wall power using a 6–12V plug-in wall transformer. Special connectors on the interface board allow easy connection of a logic analyzer to facilitate debugging of the node.

The $\mu$AMPS Revision 1 component boards plug together by means of a standardized connector. With the exception of the radio, power supply, and basestation boards, which only have connectors on one surface, all other boards in the node are equipped with a male, surface-mount connector on the top of the board, and a corresponding female connector directly underneath on the bottom of the board. Vias connect the pins of the top connector directly to the corresponding pins on the bottom connector, so that when the boards are stacked, a common bus of signals is formed between all boards in the node. The bus carries portions of the processor’s address and data busses, RS-232, USB, and SPI serial busses, and both regulated 3.3V and raw battery power. Physically, the connectors are IEEE1386 “Mezzanine” connectors, each of which have two rows of 32 pins, on 1mm centers. Figure 2.2 shows the pinout of the connector.
<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>D0  Data bus bit 0</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>2</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>D1  Data bus bit 1</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>4</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>D2  Data bus bit 2</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>6</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>D3  Data bus bit 3</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>8</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>D4  Data bus bit 4</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>10</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>D5  Data bus bit 5</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>12</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>D6  Data bus bit 6</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>14</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>D7  Data bus bit 7</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>16</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>D8  Data bus bit 8</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>18</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>D9  Data bus bit 9</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>20</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>D10 Data bus bit 10</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>22</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>D11 Data bus bit 11</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>24</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>D12 Data bus bit 12</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>26</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>D13 Data bus bit 13</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>28</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>D14 Data bus bit 14</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>30</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>D15 Data bus bit 15</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>32</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>A0  Address bus bit 0</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>34</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>A1  Address bus bit 1</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>36</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>A2  Address bus bit 2</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>38</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>A3  Address bus bit 3</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>40</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>CS2 Chip select 2</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>42</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>CS3 chip select 3</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>44</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>OE  Output enable</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>46</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>WE  Write enable</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>48</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>RS232_RX RS-232 receive</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>50</td>
<td>RS232_TX RS-232 transmit</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>51</td>
<td>USB_D- USB data -</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>52</td>
<td>USB_D+ USB data +</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>53</td>
<td>RXD_C SPI receive</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>54</td>
<td>TXD_C SPI transmit</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>55</td>
<td>SCLK_C SPI clock</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>56</td>
<td>SFRM_C SPI chip select</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>57</td>
<td>P0_PWREN Power enable 0</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>58</td>
<td>P1_PWREN Power enable 1</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>59</td>
<td>V3.3 +3.3V power</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>60</td>
<td>IRQ Interrupt request</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>61</td>
<td>VBATT Battery power</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>62</td>
<td>GND Ground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>VBATT Battery power</td>
<td>GND</td>
<td>Ground</td>
</tr>
</tbody>
</table>

Figure 2-4: System connector pinout
Figure 2-5: The $\mu$AMPS node. Clockwise from upper left: A complete basestation node, with basestation, processor, and radio boards; the radio board; the processor board; a U.S. quarter for size comparison, and an example external sensor board providing four microphone inputs for acoustic sensing and a battery pack.
Chapter 3

Implementation of the Revision 1 Node

3.1 Overview

This chapter describes in detail the implementation of the component boards of the \( \mu \)AMPS Revision 1, node, particularly the processor board. The processor board is the core of the Revision 1 node, and contains a StrongARM microprocessor, several power supplies (including the supply for all digital logic on the node), and also the node’s default acoustic sensor element. Full schematics for all of the boards can be found in Appendix B. Drawings of the printed circuit board layout for each board are shown in Appendix C.

3.2 Processor Board

A block diagram of the circuitry contained on the processor board is shown in Figure ???. The processor board contains a 32-bit microprocessor and its associated ROM and RAM, a collection of dc/dc converters, a small acoustic sensor, and assorted I/O interface circuitry, including RS-232 drivers and a USB port.
Figure 3-1: Block diagram of the processor board
(Data is collected from [8] and [7].)

<table>
<thead>
<tr>
<th>Part no.</th>
<th>SA-1100</th>
<th>SA-1110</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADES1100EF</td>
<td>GDS1110BB</td>
<td></td>
</tr>
<tr>
<td>Max. Clock Speed (MHz)</td>
<td>206</td>
<td>206</td>
</tr>
<tr>
<td>Performance (MIPS)</td>
<td>220</td>
<td>235</td>
</tr>
<tr>
<td>Core Voltage (V)</td>
<td>1.50</td>
<td>1.75</td>
</tr>
<tr>
<td>Power @ Max. Speed (mW)</td>
<td>&lt;330</td>
<td>&lt;400</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of the SA-1100 and SA-1110

### 3.2.1 Processor and Memory

The preliminary µAMPS Revision 0 node utilized an Intel StrongARM SA-1100 processor. The StrongARM processor is a good match for the design goals of the µAMPS project: it provides a significantly above average balance of computational performance and power consumption (220MIPS and <330mW at 206MHz), a programmable PLL for core clock generation (59–206MHz), and many important peripherals (three UARTs, SPI interface, USB peripheral controller) on-chip. Its fully-static CMOS design permitted the development of a dynamically variable core power supply, which further reduces the power consumption of the CPU core by up to 60%.[14]

For the µAMPS Revision 1 node, the updated StrongARM SA-1110 processor was selected. This chip uses the same ARM V4 CPU core as the SA-1100, and is thus code-compatible. Compared to the SA-1100, the SA-1110 consumes more power, but provides slightly higher performance, as shown in 3.2.1.

The µAMPS Revision 0 node used the SA-1100 in a 208-pin LQFP package, which facilitated printed circuit board design and assembly. The SA-1110 is only available in a 256-ball BGA package, which forced the use of an expensive, 8-layer PCB with 5mil traces, 5mil between traces, and 10mil holes, but helped significantly reduce the overall size of the board.

The µAMPS Revision 1 node was designed with 1MB of random access memory, and 1MB of program storage memory. Random access memory is implemented using two Cypress CY62146V 4Mbit (16×256k) low-power SRAMs. The devices are used in parallel to match the StrongARM’s 32-bit data bus. The Cypress parts were chosen
for their low operating current, 7mA (at 3.6V and maximum access rate). Program storage is implemented using two Sharp LH28F300BVE flash ROMs. Again these parts are 4Mbit (16×256k) and used in parallel to match the StrongARM 32-bit data bus. The flash is programmed in system by using the SA-1110’s JTAG port to manually toggle the flash’s data, address, and control lines. A Java program was written which reads the s-record files generated by the GNU compiler toolchain and uses a PC’s parallel port to drive the StrongARM’s JTAG interface. The contents of flash memory can be write-protected by a switch on the μAMPS Revision 1 processor board which controls the write-protect pin on the flash components.

The SA-1110 provides 28 general purpose I/O (GPIO) pins, which can be individually be configured as inputs or outputs, and can even generate interrupt signals. The μAMPS node primarily uses these pins for power management signals. Two GPIO pins are used to drive LEDs, which are useful for debugging purposes. GPIO pin 0 serves as a general purpose interrupt request line (IRQ) for the node. The IRQ input present in the main system connector, so any interrupts can be signaled by any of the node’s board. IRQ is a wired-or signal, with a pullup resistor located on the processor board.

### 3.2.2 Power Supplies

The μAMPS Revision 1 node was designed to operate from a wide range of battery voltages, allowing flexibility in the choice of batteries used to power the node and ensuring that the node can continue to operate at decreasing battery voltages as the battery is discharged. Jumpers are provided on the processor board to configure the node for one of two input voltage ranges: 3.3–5V or 5–14V. The lower voltage range is well matched to a battery pack containing three alkaline cells in series, as it allows the individual cell voltage to range from 1.6V down to 1.1V.

The intended rechargeable battery configuration for the node is two lithium ion (or lithium polymer) batteries in series. The safe operating voltage range for lithium rechargeable cells is 2.7–4.1V, giving a pack voltage of 5.4–8.2V, which fits neatly into the 5–14V node configuration.
Table 3.2: Summary of power supplies on the processor board

<table>
<thead>
<tr>
<th>Supply</th>
<th>Input Source</th>
<th>Output (V)</th>
<th>Circuitry powered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital</td>
<td>Battery</td>
<td>3.3</td>
<td>All digital circuitry on node (RAM, ROM, RS-232 transceiver, digital portions of radio and sensor boards)</td>
</tr>
<tr>
<td>+5V</td>
<td>Battery or digital power supply</td>
<td>5.0</td>
<td>Controller for processor core supply</td>
</tr>
<tr>
<td>Processor core</td>
<td>Battery</td>
<td>0.9–2.0</td>
<td>StrongARM core logic (ARM execution core and most on-chip peripherals)</td>
</tr>
<tr>
<td>Analog</td>
<td>Battery</td>
<td>3.3V</td>
<td>Built-in acoustic sensor circuitry (microphone, amplifier, filter, ADC)</td>
</tr>
</tbody>
</table>

*a in which case battery must be < 5V, because the 5V supply is a boost-mode dc/dc converter

The \( \mu \)AMPS Revision 1 processor board carries four different power supplies which convert the variable voltage from the battery into multiple regulated power busses used by various node components. The three main supplies provide +3.3V for all digital logic on the node, +0.9–2.0V for the StrongARM core, and +3.3V (with low noise) for the analog circuitry in the sensor portion of the board. A fourth supply provides the +5V needed by the the StrongARM core supply controller chip. Table 3.2.2 summarizes the roles of the four power supplies.

### 3.2.3 Analog Power Supply

To minimize noise, the 3.3V analog supply is generated directly from the battery by a low-dropout linear regulator. A Linear LT1521 device was chosen, due to its small quiescent current (12\( \mu \)A) and shutdown input.[4] Even though the linear regulator is not very efficient (40% if the battery voltage is 8.2V), the current drawn by the analog circuitry is very small (<5mA), and therefore the total power wasted by the regulator is reasonably small (<25mW).

Due to the close proximity of the microprocessor and the three switching power supplies to the analog circuitry, minimizing the noise coupled into the analog circuitry is a challenging problem. The analog circuitry is confined to a small area of the...
processor board. The power and ground planes for this area are isolated from those of the rest of the board, and the analog ground plane is tied to the digital ground plane in one location, using a ferrite bead. A ferrite bead is also used in the connection between the battery and the LT1521 regulator.

### 3.2.4 Digital Logic Supply

The processor board contains the node’s main digital logic supply, which powers all of the 3.3V logic in the node. This includes the microprocessor’s I/O, memory, other glue logic on the processor board, and the digital portion of the radio board. For efficiency, a switching regulator is used to generate the digital supply. The regulator is implemented using a Maxim MAX1685 buck-mode dc/dc controller, which can supply up to 1A, and has a fast 300kHz switching frequency, which enables the use of a small inductor.[10] This component also includes a built-in MOSFET switch, the only critical components that need to be chosen to complete the design are the inductor and output capacitors. The MAX1685 was chosen based on its small size, internal switch (which further reduces the size of the overall supply) and high efficiency at low load current (95% at as low as 1mA load, verified by testing with Maxim’s evaluation board).

Equations given in the datasheet for this part suggest using a 22µH inductor, which sets the inductor ripple current at 400mA. A Toko inductor of type D62CB was selected. This inductor is very small (6.0×6.3mm) with a low dc resistance (170mΩ). This saturation current for this inductor is 700mA, limiting the output of the supply to 300mA, which is more than adequate for this application. A 150µF, Panasonic specialty-polymer aluminum electrolytic output capacitor was used to minimize size, while maintaining a low (15mΩ) series resistance (which reduces output ripple and increases efficiency).
3.2.5 5V Supply

The five volt power supply exists only to power the MAX1717 controller for the microprocessor core supply (discussed in Section 3.2.6). The 5V power supply is generated by a Maxim MAX1675 boost-mode dc/dc converter. This part includes a built-in power MOSFET. Based on recommendations in the datasheet, a 22µH inductor was chosen.

Jumpers on the processor board allow the input to the 5V supply to be either the raw battery voltage, or the output of the 3.3V supply. Powering the 5V supply directly from the battery limits the battery voltage to between 3.3V and 5V. Powering the 5V supply from the 3.3V supply increases the maximum battery voltage to 14V (the maximum input voltage for the 3.3V supply), but decreases the efficiency of the 5V supply, since the overall efficiency for this supply is then the product of the efficiencies of the MAX1684 and MAX1675.

The MAX1675 includes a low-battery detector function. This is a voltage comparator with a fixed threshold of 1.3V. The output of this comparator is connected to a StrongARM GPIO pin which can be programmed to generate an interrupt when the comparator changes state. The input to the low-battery detector is derived from the battery voltage by means of a voltage divider (R19 and R20). The specific resistances of R19 and R20 must be chosen based on the type of battery being used.

3.2.6 Microprocessor Core Supply

Because the StrongARM microprocessor is the second largest power consumer on the µAMPS Revision 1 node (after the radio power amplifier), the design of its power supply is especially critical. Because this voltage of this supply is variable, it is the most complicated supply on the processor board.

The microprocessor core supply is generated by a Maxim MAX1717 buck-mode dc/dc controller. This part was designed for powering mobile Pentium processors, so it is capable of supplying much more current than necessary (40A!) for the µAMPS node. The MAX1717 includes a built-in, 5-bit digital-to-analog (DAC) converter
which can be used to program the output voltage from 0.9V to 2.0V. In its intended application of powering Pentium processors, the DAC inputs are wired to pins on the socket for the processor. Inside the processor packaging, these pins are hard-wired to power or ground, programming the appropriate voltage for each particular processor. This allows motherboards to be designed to support a variety of processor models, each of which might require a different core supply voltage. In the µAMPS Revision 1 node, the DAC inputs are wired to GPIO pins on the StrongARM processor, giving the processor dynamic control over its own supply voltage.

The MAX1717 also supports Intel’s SpeedStep technology, which allows a mobile processor to run at one voltage and frequency while on AC power and a different voltage and frequency while on battery power. This is accomplished by providing two sets of DAC inputs, and an external pin to switch between them. In the µAMPS Revision 1 node, this feature is used to ensure that the StrongARM core voltage is always set to a safe value on startup. Figure 3.2.6 shows a simplified schematic of the microprocessor core supply.

As with the 3.3V digital and 5V switching supplies, the choice of inductor and output capacitors is critical to the performance of this supply. Since the MAX1717 does not include an internal power MOSFET, that component must also be chosen.

Due to the critical nature of this supply, a test board was constructed. This board was used to try out different combinations of transistors and inductors. The Maxim
evaluation board for the MAX1717 uses IRF7811 MOSFETs from International Rectifier, with a on resistance of 12mΩ. However, this part was not readily available. Based on experimentation with several different transistors on the test board, it was determined that minimizing on resistance gives maximum efficiency. Switching losses due to gate capacitance are less significant than conduction losses, for this supply. The lowest resistance transistor available, the IRF7413A (13.5mΩ) was therefore selected.

As with the 3.3V and 5V switching supplies, to minimize size, a Toko D62CB inductor was used. Experimentation showed that a 1µH inductor provided the greatest efficiency. This inductor has a dc resistance of 11mΩ, and saturates at 3.48A. The low inductance means that the supply is operated in the discontinuous conduction mode, but the MAX1717 employs a pulse-skip algorithm that provides high efficiency in this regime.

The most complicated portion of the processor core supply is the logic for selecting a safe voltage on startup. As described previously, the MAX1717 permits switching between two voltages, via its A/B pin. The A/B pin selects between the A input of the DAC, which is specified by the state of the five voltage select pins, and the B input to the DAC, which is a register internal to the MAX1717. The B register is automatically loaded whenever the A/B pin goes low (and at startup). The value loaded is determined by checking the impedance of the voltage select pins. When the A/B pin goes low, the MAX1717 weakly drives the voltage select pins low and then high. If a particular pin follows the MAX1717’s drive, then it is considered high impedance, and a 1 is loaded into the corresponding bit of the B register. If the pin does not follow the MAX1717’s drive, then a 0 is loaded into the B register.

When the StrongARM is in reset, its GPIO pins are tristated, and therefore the MAX1717’s voltage select pins are not driven. Resistors R25–29 pull down the select pins. These resistors have a high value (1MΩ) so that they do not cause significant power drain when the StrongARM is driving the select pins. The resistors on voltage select pins D3 and D4 have a 4700pF capacitor wired in parallel with them. This capacitor serves to reduce the apparent impedance of these pins when the StrongARM is not driving them. Therefore, on startup, when the StrongARM is in reset, the B
register of the MAX1717 is loaded with 0b00111, which programs a voltage of 1.650V in B mode. NAND gates U23c and U23d form an S-R latch which is reset whenever the StrongARM reset signal is active (low). The latch holds the MAX1717 in B mode until the StrongARM has started up. Once the operating system has started and the StrongARM has configured its GPIO pins to drive the voltage select bus, the set input of the latch is driven low by another GPIO pin, giving the StrongARM direct control over the voltage produced by the MAX1717.

Additional logic is necessary to ensure proper power supply sequencing on startup. When power is first applied to the board, the digital 3.3V and 5V supplies start automatically. A reset monitor (U2) holds the StrongARM reset input low until the 3.3V supply stabilizes. While in reset, the StrongARM asserts its reset output, resetting the S-R latch described above. When its reset input is released, the StrongARM begins its initialization process. One of the first steps in this sequence is to assert its PWR\_EN pin to turn on the power supply for its core. This power enable is ANDed (by U22) with a ready signal from the 5V supply and the result is used to drive the shutdown pin of the MAX1717. The 5V power ready signal (generated by voltage monitor U20) prevents the MAX1717 from starting before the 5V supply has reached at least 4.1V. The core power supply must reach operating voltage within 10ms after the StrongARM asserted its PWR\_EN pin, or the processor will fail to come out of reset.

### 3.2.7 Acoustic Sensor

The acoustic sensor built into the processor board consists of an electret microphone, variable-gain amplifier, an analog-to-digital converter and anti-aliasing prefilter, and a threshold detector. The sensor was designed for a 2kHz sampling rate, for analyzing sounds from 20Hz to 1kHz. A simplified schematic of this circuitry is shown in Figure 3.2.7.

The microphone element is a Panasonic WM-54BT. A 2.2kΩ pullup to 3.3V provides the bias current necessary to power the microphone’s internal amplifier. The output of the microphone is extremely small, on the order of a few tens of microvolts.
Figure 3-3: Sensor Circuitry
A high-gain amplifier boosts this signal to a few volts in amplitude. The amplifier is built from three stages, each constructed around a Burr-Brown OPA244 op-amp. The first two stages have a fixed gain of 20; the gain of the final stage is variable from unity to 31, set by a Maxim MAX5160 digital potentiometer.

To avoid the necessity of a negative power supply rail, all ac signals are biased with a 1.22V dc offset. This 1.22V reference is generated by a Zetex ZXRE1004EFCT bandgap reference, buffered with another OPA244 wired as a voltage follower.

A 10µF capacitor couples intermediate stages of the amplifier, and couples the output of the microphone to the input of the first amplifier stage. The feedback network in each amplifier stage was designed with a one-pole roll-off at 1kHz, to reduce the amount of noise coupled in at each stage. An eight-pole elliptic 1kHz low-pass filter prevents aliasing in the analog-to-digital converter. This filter is implemented with a Maxim MAX7404, and provides 82dB of stopband rejection.[13] A 330pF capacitor sets the cutoff frequency of the filter to 1kHz, according to the equation found in the datasheet for this part.[13] The MAX7404 includes a shutdown input, which is hard wired to the third of the node’s four peripheral power enable signals (labeled P2_PWREN in the schematics), allowing the filter to be shutdown without powering off the remainder of the analog circuitry. This is particularly useful when the sensor is operating in threshold detection mode.

Analog to digital conversion of the bandlimited filter output is performed by an AD7887 from Analog Devices. This 12-bit converter dissipates only 3.5mW at 125ksps.[3] Its power consumption can be reduced substantially at the 2ksps conversion rate required by µAMPS by placing the converter in standby mode between conversions. The AD7887 communicates with the StrongARM processor via an SPI-compatible synchronous serial interface.

The microphone, amplifier, filter, and analog converter require very little power—no more than a few milliwatts—but in order to trigger conversions and examine the output of the analog-to-digital converter, the StrongARM must be active, which consumes almost 100mW even at low clock frequencies. Given the expected 1% active duty cycle of the node, it is unacceptable to require the processor to continue running
while the node is waiting for the microphone to hear something. The addition of a fully-analog threshold detector allows the processor to sleep whenever no external stimulus is present, but to be automatically awakened by an interrupt if the ambient noise level reaches a programmable level.

The threshold detector is a peak detector and a voltage comparator, built from op-amps U15 and U16 and diode D1. The diode allows C24 to charge whenever the input to U15 is lower in voltage than the negative side of C24. Resistor R10 slowly discharges C24, causing the capacitor voltage to decrease at a fixed rate whenever the envelope of the acoustic signal decreases. The output of the peak detector is fed into a voltage comparator, formed from a OPA244 op-amp (U16). The inverting input of U16 is driven by digital potentiometer U17, which is wired as a voltage divider, effectively implement a 5-bit DAC. The output of the voltage comparator is connected to a StrongARM GPIO pin, which can be configured to generate an interrupt, waking the processor from sleep when a sufficiently loud sound is detected by the microphone.

3.2.8 I/O

As described in Section 2.2, the system connector provides access to the StrongARM memory buss, serial communication via RS-232, USB, and SPI, peripheral power enable signals, and an interrupt request (IRQ) line.

Only a portion of the StrongARM memory bus signals are included in the system connector. The connector carries the low 16 bits of the data bus, four address lines (A1 through A3), two chip selects, in addition to the WE and OE control lines. This permits the processor to address up to $2^4 = 16$ 16-bit registers in up to two different peripherals. (The two power enable signals available through the system connector are intended to serve these two peripherals.)

In order to minimize loading of the StrongARM’s pins, the memory bus signals are buffered before the connector. A Texas Instruments SN74ALVCH16245DG 16-bit bidirectional transceiver (U26) is used to buffer the data bus. The direction input of the transceiver is driven by the StrongARM’s WE signal, so that during a write, the StrongARM signals are driven on to the connector, and during a read,
the connector drives the StrongARM signals. The enable input of the transceiver is
driven by a logical OR (implemented with an AND gate, due to inverted logic) of
the two chip select signals available on the system connector. This reduces power
consumption, because the connector’s data bus is only driven during access to an
off-board peripheral. An SN74ALVCH244PWR 8-bit buffer from Texas Instruments
(U27) buffers the address and control signals going to the system connector. To
ensure that a peripheral sees valid levels on these signals during the rising edge of the
write enable signal, this buffer drives the connector at all times.

The µAMPS Revision 1 processor board is equipped with two RS-232 serial ports.
One is carried by the main system connector; the other is brought out to a small
three pin connector. This second port is intended for debugging, particularly when
the processor is not connected to a basestation board that would provide convenient
access to the main serial port.

The StrongARM’s serial ports are 0–3.3V signals. To convert these to the ±12V
signals required by the RS-232C standard, a MAX3223 RS-232 transceiver is used.
The MAX3223 incorporates an automatic shutdown system, which powers down the
charge pumps used to generate the ±12V supplies whenever the serial input and
output lines are idle for more than approximately 30 seconds. The charge pumps
consume about 70mW when on.

The system connector provides a USB port for connecting to the StrongARM’s
built-in USB peripheral controller. Resistors R40 and R41 are 22Ω terminators,
required by the USB specification, while resistors R43 and R44 are pull-downs used
by the USB protocol to detect device attachment and detachment.[15]

According to the USB standard, a peripheral signals its presence on the bus by
pulling the non-inverted data line of the bus to +3.3V through a 1.5kΩ resistor (R42).
Linear regulator U28 is used to provide the 3.3V for the pullup. By controlling the
shutdown input of U28 (through a StrongARM GPIO signal), the StrongARM can
effectively detach itself from the USB by shutting off the the regulator. This is useful
for ensuring that any necessary initialization code has a chance to run before the USB
host starts sending messages to the node.
The final I/O capability of the processor board is a bicolor LED. The LED is controlled by two of the StrongARM’s GPIO pins. Two spare NAND gates in a 74HC00 (U23) buffer the GPIO signals to drive the LED. Since turning on each color (red or green) of the LED consumes 54mW, it is important not to overuse the LEDs in power-sensitive applications. Software automatically turns off the LEDs if the processor is put into sleep mode.

3.3 Processor Circuit Board

The processor board is fabricated on a 55mm square 8-layer circuit board, made from FR-4 material. Since minimizing the size of the node was a major design goal, integrated circuits were used in their smallest available package, usually SO8, SOT23, or TSSOP. Resistors and small capacitors were used in 0603 form, with 1206 packages used for medium (1µF) capacitors. Large capacitors are tantalum. Figure 3.3 shows a photo of the processor board, and identifies the location of select components described in the preceding sections.

3.4 Radio Board

The details of the radio implementation are beyond the scope of this thesis. A comprehensive discussion of the RF portion of the radio board can be found in [9]. The digital base-band portion of the radio, as well as the µAMPS radio network protocol are discussed in [16].

Figure ?? shows a block diagram of the radio board.

3.5 Basestation board

The principle task of the basestation board is to facilitate connection of the µAMPS node to a personal computer by providing industry standard connectors for the node’s communication ports. A block diagram of this board is shown in ??
Figure 3-4: Photograph of the processor board showing important component locations
The basestation board provides a male DB9 connector for RS-232 communications, as well as a USB “series B” peripheral connector. Both of these connectors are wired directly to the standard µAMPS system connector: all of the hardware associated with these communications ports is located on the processor board.

The interface board provides a coaxial power socket that allows a standard wall transformer to be used to power the node. The transfer can be either ac or dc, but must provide at least 6V at 600mA. A linear regulator on the interface board is used to generate a constant 3.4V power supply that powers the node. Additional linear regulators provide 3.3V and 5V to power a small amount of logic on the interface board.

A terminal strip provides a connection where power can be fed directly to the node, bypassing the regulator. The main power switch located on the interface board is a DPDT, center-off type. In the center position, the node is off. When the switch is pushed one way, the node is powered by the ac adapter. Pushed the other way, the switch allows the regulated external power supply to power the node, while the 3.3V and 5V circuitry on the interface board is powered by the ac adapter. A jumper
allows easy insertion of a ammeter into the power supply to the node.

Four rows of 8 × 2 header pins provide easy access to all of the signals on the µAMPS system connector. The headers are specifically designed to connect to probe pods for a Tektronix TLA700-series logic analyzer.

Additional logic was designed into the interface board, but was never used. (These parts were never populated.) A Cypress CY37128V 128-macrocell CPLD is present on the interface board. The CPLD interfaces with the node via the StrongARM memory bus and emulates an IEEE1284 parallel port. The parallel port was intended to allow communication with the µAMPS Revision 0 radio board, while the revision 1 radio was developed. The CPLD also provides an interface to a Hewlett Packard HDSP-2111 eight-character alphanumeric LED display. The display was intended to facilitate debugging during the process of porting the eCos operating system to the new processor board.

3.6 Four-Channel Acoustic Sensor Board

As part of a vehicle tracking demonstration, a four-channel acoustic sensor board was designed for the µAMPS Revision 1 node. This board is an example of an external sensor that replaces the processor board’s built-in acoustic sensor. Figure 3.6 shows a block diagram of this board.

The four-channel sensor connects to up to four pre-amplified microphones via 1/8in headphone jacks. Four Burr-Brown OPA244 op-amps, one per channel, further amplify the incoming signals. The gain of each of these amplifiers is set by an individual potentiometer. Four Maxim 7404 eight-pole elliptic 1kHz low-pass filter ICs perform anti-alias filtering. The cutoff frequency of these filters is set at 500Hz. A Texas Instruments TLV2544 four-channel, 12-bit analog-to-digital converter samples the output of the filters. The TLV2544 connects to the processor via the StrongARM’s SPI port. Since the processor board does not contain any hardware to multiplex SPI signals from different ADCs, the ADC component of the processor board’s built-in sensor must be removed before adding the four-channel sensor board to the node.
Figure 3-6: Block diagram of four-channel acoustic sensor board
The sensor board includes a space for a 4-cell AAA battery holder, which can be used to power the node. This battery is intended as an interim power source for the node until the real µAMPS battery board is designed.

Figure 3.6 shows a photograph of the sensor board, identifying the location of important components.

### 3.7 Battery

The battery board is still in the design stage. When completed, this component of the node will include options for powering the node from either 4 AAA alkaline batteries, or two lithium-polymer rechargeable cells. A charger for the lithium-polymer cells will be included on the battery board (but will not be powered unless the node is connected to an external power supply for charging).
3.8 Summary of Power-Aware Design Techniques

In summary, the \(\mu\)AMPS Revision 1 node employs several key ideas for power-aware design.

- Almost every system provides a means of completely removing power from that system. This coarse, on/off control creates maximum power savings when particular systems will be idle for long periods of time.

- Some systems cannot be turned off, such as RAM for the processor and the power supply for the RAM. These systems are responsible for the residual power consumption when the node is in its deepest sleep state. An effort has been made to minimize this power by using generally low-power devices, and selecting static devices which draw very little power when not clocked.

- Most systems have multiple power controls. An example is the built-in acoustic sensor. The entire sensor can be turned completely off by shutting down the analog power supply. Additionally, the anti-aliasing filter and analog-to-digital converter can be individually shut down. Having multiple power controls creates additional states inbetween full on and full off. These states are useful because they allow a faster return to the fully operational state than is possible from the full-off state.

- The largest power consumers, the processor and radio power amplifier, provide scalable performance when in active mode. This allows dynamic quality/energy tradeoffs.
Chapter 4

Microsensor Operating Systems

4.1 Requirements

A multitude of embedded operating systems are available today. Some of them are more suited to the $\mu$AMPS project than others, but ultimately, none of them is a perfect match. Effectively exploiting the power management control implemented in the $\mu$AMPS hardware requires an operating system that is thin and lightweight. The operating system must not provide too much isolation between application code and the hardware, and it must not add excessive overhead, as every instruction cycle not used to collecting, analyzing, or sharing sensor data represents wasted energy.

In the context of $\mu$AMPS, the role of the operating system is to provide a framework for multitasking and power management. There are three components to this framework: support for multiple threads of execution, support for interrupts, and support for changing global power states.

Threads are an excellent model for complicated, long-duration task, whose running times might range from tens of microseconds to several seconds. Examples of such tasks include the analysis or compression of raw sensor data or high-level network protocol (such as routing, in a multi-hop network). There may be hard, real-time deadlines on these tasks, but the deadlines are long: on the order of seconds, or so. Because these tasks are long running, there are frequently more than one of them
that need to run more or less simultaneously. In a low-power system, minimizing
the time required for a context switch is important.\textsuperscript{1} Time spend saving or restoring
state represents wasted energy.

Some tasks must be completed virtually instantaneously, possibly at a very precise
moment in time. Examples include the periodic, low-jitter triggering of an analog-to-
digital conversion, or the transfer of data from a radio FIFO before overflow occurs.
It is important that the handling of interrupts be as swift as possible, not so much
because of the energy consumed by extra execution cycles, but because precisely timed
interrupts make possible not only higher quality data collection, but also enable more
aggressive power management strategies (such as powering up an analog-to-digital
converter only an instant before it is needed).

Power management tasks performed by the operating system range from as simple
as toggling a GPIO pin to control power to a peripheral, to as complex as restarting
the node after exiting from sleep mode. In $\mu$AMPS, the operating system does not
include any power management policies. Policies are left up to application code to
implement, so that they can be customized for each application. What the $\mu$AMPS
operating system does provide is simple procedures for changing power states when
the application’s power management policy decides it is efficient to do so.

The only way to obtain an optimal operating system for a power-aware microsen-
sor node would be to develop one from scratch. Practically speaking, however, the
benefits of a developed and tested operating system and a comprehensive set of de-
bugging tools are enormous.

\subsection*{4.2 eCos}

For the first and second phases of the $\mu$AMPS project, the eCos operating system has
been used. First released in 1998, eCos (embedded Configurable Operating System)
is an open-source real-time operating system which provides compatibility with the
EL/IX Level 1 subset of the POSIX API (which means it looks vaguely like UNIX)\cite{6}.

\footnote{The same is true in high-performance, high-power systems!}
eCos is a lightweight operating system with no file system, memory protection, or multiple process support (though multiple threads in a common memory space are supported). eCos development is managed by RedHat and is written in C and C++ for the GNU toolchain. eCos supports many different processors, including ARM, x86, MIPS, and PowerPC. eCos applications are statically linked, which means the operating system itself is merely a library against which application code is linked. Static linking eliminates the run-time overhead of dynamic linking, and permits the detection of memory allocation errors at compile time.

The greatest strength of eCos is its extreme degree of configurability. The operating system is divided into modules, and modules unnecessary for a given application can be removed. Associated with each module of the operating system is a configuration script, which specifies the configuration options available for that module. The configuration scripts also contain rules about how the module (and the values of its configuration options) interact with other modules, thus allowing module dependencies to be resolved automatically.

The process of porting eCos to a new system has been simplified by the effective use of these module interfaces. Like most operating systems, eCos separates peripheral driver code for peripherals from code responsible for the core functionality of the system. The core code is further divided into architecture-independent code (such as the thread scheduler) and architecture-dependent code (such as interrupt handlers). The architecture-dependent portion is defined by an interface known as the HAL (hardware abstraction layer). The HAL itself is composed of three layers: architecture (e.g. ARM), variant (e.g. StrongARM), and platform (e.g., the Assabet development board for the StrongARM). Porting eCos to a new system generally only requires writing a new platform-level HAL module, assuming the system uses a processor for which architecture- and variant-level modules have already been written.

### 4.2.1 RedBoot

Closely associated with eCos is the RedBoot bootloader. RedBoot is actually a very simple eCos application, which means that the process of porting eCos and the pro-
cess of porting RedBoot are virtually one and the same. Once the bootloader has been ported to a platform, the underlying operating system is ready to be used for application development.

RedBoot includes stubs for the GNU Debugger (GDB), which provides remote debugging capability over a serial or ethernet link. RedBoot also implements a simple flash file system, which allows multiple application images to be stored, loaded, and run.

4.3 Porting eCos to μAMPS

For the μAMPS Revision 1 node, a custom platform-port of eCos was developed. The port is based on the Intel SA-1110 development board known as “Assabet”.

The principle difference between the Assabet and μAMPS boards is the use of SDRAM in Assabet and SRAM for μAMPS. To accommodate this, code used to initialize the SA-1110’s internal DRAM controller was removed and replaced with code to setup the correct number of wait cycles for SRAM accesses. Since memory timings are derived from the processor clock, the timing values must be changed whenever the processor clock frequency is changed. The mechanism for accomplishing this is described in Section 4.4. SRAM and DRAM banks are hard-wired to different physical addresses by the StrongARM’s internal memory controller, but this was compensated for using the memory management unit (MMU). Figure 4-1 shows a complete memory map for the μAMPS Revision 1 node. eCos makes use of the StrongARM’s built-in MMU to map peripherals to convenient addresses, and to map discontinuous memory segments into contiguous address ranges. The memory map is static: it is not changed after startup. The MMU is not used to provide protected memory spaces.

On startup, the processor automatically begins executing code at address 0x00000000, corresponding to static memory bank 0, which is occupied by flash ROM on the μAMPS processor board. A jump instruction at this location transfers control to the RedBoot bootloader. The bootloader first checks the state of the sleep status
Figure 4-1: µAMPS memory map

bit in the StrongARM’s power manager sleep status register to see if the processor is recovering from sleep. If the processor is recovering from sleep, control is transferred to code responsible for restoring the state of the processor and resuming normal operation. Otherwise, the bootloader initializes the memory controller, setting up the appropriate memory timings for accessing the SRAM and flash banks given the default processor clock speed of 59MHz. (As part of the reset process, on power-up, before any code was executed, the timings for static memory bank 0 were automatically set to the slowest setting possible. They are now set to match the minimum timings for the flash ROM, so that code can be read as fast as possible.) With the memory timings initialized, it is now possible to access SRAM. The direction and default state of the GPIO pins is set. Next, in preparation for enabling the MMU, the memory map is constructed in SRAM, with its base at physical location 0x08004000. Note from Figure 4-1, that with the MMU enabled, the base of SRAM is mapped to location 0x00000000 and the base of flash to location 0x50000000. This is an eCos standard configuration that helps code compatibility between ports to different
boards. Because the location of flash changes when the MMU is enabled, the processes of enabling the MMU is tricky. The eCos designers dealt with this by taking advantage of the three cycle delay between the execution of the coprocessor directive that enables the MMU and the first instruction that is fetched using the MMU mappings. In the first of these three delay slots, a branch instruction changes the program counter register to point to the new virtual memory address for the location in flash where execution should continue. The remaining two delay slots (which are fetched without translation) are filled with no-ops.

Once the MMU is enabled, control is returned to the standard eCos ARM startup code, which performs the standard ARM interrupt initialization and starts the Red-Boot bootloader.

4.4 Power Management in eCos

The official eCos distribution does not include provisions for high-level power management. Nor does the standard StrongARM variant package include support for the power management features built into the StrongARM chip. Since power-awareness is central to the μAMPS project, adding support for these features was important.

The StrongARM’s internal power manager provides mechanisms for manually switching between the chip’s three operating modes: run, idle, and sleep. Run mode is the only mode in which code is executed. In this mode, the full processor is active. The idle and sleep modes differ in the amount of power savings they offer and in the delay associated with returning to run mode.

In idle mode, the clock to the processor core is gated, so no execution takes place. The PLL remains locked, however, and the assertion of any unmasked interrupt signal causes the processor to immediately resume execution.

Sleep mode provides significantly greater power savings, at the expense of significantly greater complexity in returning to normal operation. In sleep mode, the core power supply is shut down. (A dedicated I/O pin is used to provide a shutdown signal to the dc/dc converter). A very limited amount of state is preserved using the I/O
power supply. The I/O supply also runs a 32kHz oscillator which clocks the power management logic and a real-time clock. Return from sleep mode is initiated by either an external interrupt signal, or by a real-time alarm. The processor performs a “warm-start” upon returning from sleep: after the power supply is re-enabled, the chip performs its standard reset procedure, and begins execution at the reset vector location (0x00000000). A flag in the power manager status register allows the operating system to determine that the chip is returning from sleep.

For the µAMPS project, it was also important to take advantage of the StrongARM’s programmable clock frequency. The core clock for the StrongARM is generated by an on-chip PLL, based on a 3.6864MHz reference oscillator. The PLL multiplier is programmable to twelve values, from 16 (to give 59MHz) to 60 (to give 221MHz). The multiplier value can be reprogrammed at any time, but doing so introduces a 150µs interruption that may interfere with some of the StrongARM’s on-chip peripherals. Proper precautions must therefore be made to ensure the clock frequency change occurs cleanly. The µAMPS platform additionally requires that the core voltage supply be adjusted for the new clock frequency. If the frequency is being increased, the voltage must be increased before the clock is adjusted. If the frequency is being decreased, the voltage must be decreased after the clock is adjusted. Memory access timings must be adjusted using a similar strategy.

The code for changing the processor clock speed, as well as for supporting idle and sleep modes, is part of the µAMPS platform support module that was developed for eCos, a full listing of which is found in Appendix D.

### 4.4.1 Idle Mode

Idle mode is initiated with the coprocessor instruction

```
mcr p15, 0, r0, c15, c2, 2
```

Idle mode is exited when an unmasked interrupt occurred, which in practice means that no further code is needed to support idle mode. By putting the above coprocessor instruction in the eCos idle thread, the processor is automatically placed in idle
mode whenever no other thread is ready for execution. The processor is automatically awakened every 10ms by the operating system’s time-slicing timer so that the scheduler can run. Should any other interrupt occur, the processor is woken an the interrupt serviced immediately.

4.4.2 Sleep Mode

Supporting sleep mode is significantly more difficult than idle mode, because all important state must be saved and then restored when the processor is awakened. The state in question here includes not only the processor registers, but also configuration registers for most on-chip peripherals. While the StrongARM is in sleep mode, the core power supply is disabled. Only the real-time clock, power management, and I/O driver circuitry remains powered (by the I/O power supply). Since the external SRAM on the µAMPS board remains powered while the processor is asleep, state can be saved in this memory.

Preparation for sleep mode involves three steps:

1. Save the configuration of on-chip peripherals

2. Safely shutdown on-chip peripherals

3. Disable and flush memory caches

4. Save the state of processor registers

The first three steps are performed by the procedure `uamps_sleep`. This procedure begins by disabling interrupts, and then saving the necessary peripheral configuration registers. Next, the procedure waits for the serial port transmit FIFOs to empty, then disables the serial ports. If the serial ports are not disabled, they will transmit several garbage characters as the core voltage supply falls. This could confuse the A/D converter (attached to the SPI port) or a remote debugger (attached to one of the RS-232 ports). At this time, the GPIO sleep state registers are programmed. These registers specify the state of the GPIO pins while the processor is asleep. They are
programmed to match the current state of the GPIO pins, except where the state of these pins is incompatible with power saving nature of sleep mode. (For example, the LEDs are forced off, and the 3.3V digital supply is forced to standby in sleep mode.) The `uamps_sleep` procedure next proceeds to drain and disable the write-back cache, synchronize and disable the data cache, and then calls the `uamps_suspend_processor` procedure. When `uamps_suspend_processor` returns (after the processor has been awoken out of sleep mode), `uamps_sleep` re-enables the caches, restores the configuration of on-chip peripherals, re-enables the serial ports, and re-enables interrupts.

The `uamps_suspend_processor` procedure is written entirely in assembly, and performs the process of saving all of the processor’s registers (including coprocessor registers) to memory, and then setting the sleep bit in the power manager register to force entry into sleep mode.

When the processor is awakened (via a real-time clock alarm, which must be set prior to calling `uamps_sleep` or an external interrupt signal), the StrongARM performs an internal reset and begins executing code at address 0x00000000. The eCos startup code detects that the processor is returning from sleep mode and jumps to the `uamps_resume_processor` procedure. This procedure restores the processor’s registers, re-enables the MMU (using the same technique described in Section ??), and then performs a return-from-subroutine to return control to the `uamps_sleep` procedure, which finishes wake-up procedure.

### 4.4.3 Clock and Voltage Scaling

The µAMPS Revision 1 node makes maximal use of the StrongARM’s programmable core clock PLL by changing the core clock frequency dynamically, while simultaneously scaling the core power supply voltage. Changing the clock frequency is as simple as writing to the PLL configuration register. However, scaling the supply voltage and ensuring that the frequency and voltages changes occur smoothly requires somewhat more complexity.

Clock speed changes are performed using the `uamps_set_speed` procedure, which takes one argument: an integer from 0 to 11 indicating which of the 12 clock speeds
Table 4.1: Frequency, voltage, and memory timing combinations.
Memory read timing is the minimum access time, and is 70ns for RAM and 100ns for ROM. Write timing is the minimum write enable assertion time, and is 50ns for RAM and 100ns for ROM. Memory timings are specified in cycles of the StrongARM’s MCLK, which runs at half the processor clock frequency.

<table>
<thead>
<tr>
<th>Speed (1–11)</th>
<th>Freq. (MHz)</th>
<th>Voltage (V)</th>
<th>SRAM Timing</th>
<th>Flash ROM Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Read Cycles/ns</td>
<td>Write Cycles/ns</td>
</tr>
<tr>
<td>0</td>
<td>59.0</td>
<td>1.125</td>
<td>3/100</td>
<td>2/67</td>
</tr>
<tr>
<td>1</td>
<td>73.7</td>
<td>1.125</td>
<td>3/81</td>
<td>2/54</td>
</tr>
<tr>
<td>2</td>
<td>88.5</td>
<td>1.125</td>
<td>4/90</td>
<td>3/68</td>
</tr>
<tr>
<td>3</td>
<td>103.2</td>
<td>1.125</td>
<td>4/78</td>
<td>3/58</td>
</tr>
<tr>
<td>4</td>
<td>118.0</td>
<td>1.175</td>
<td>5/85</td>
<td>3/51</td>
</tr>
<tr>
<td>5</td>
<td>132.7</td>
<td>1.250</td>
<td>5/75</td>
<td>4/60</td>
</tr>
<tr>
<td>6</td>
<td>147.5</td>
<td>1.300</td>
<td>6/81</td>
<td>4/54</td>
</tr>
<tr>
<td>7</td>
<td>162.2</td>
<td>1.350</td>
<td>6/74</td>
<td>5/62</td>
</tr>
<tr>
<td>8</td>
<td>176.9</td>
<td>1.450</td>
<td>7/79</td>
<td>5/57</td>
</tr>
<tr>
<td>9</td>
<td>191.7</td>
<td>1.550</td>
<td>7/73</td>
<td>5/52</td>
</tr>
<tr>
<td>11</td>
<td>221.2</td>
<td>1.750</td>
<td>8/72</td>
<td>6/54</td>
</tr>
</tbody>
</table>

is to be selected. If the processor speed is being increased, the voltage and memory delay cycles are first increased in order to support the faster clock speed. The serial ports are then disabled (as in preparation for sleep), and the PLL is changed. The processor will stall until the PLL relocks. When execution resumes, the serial ports are re-enabled. If uamps_set_speed is being used to decrease the processor speed, then the voltage and memory delay cycles are now decreased to the appropriate, most efficient levels for new processor speed.

Appropriate frequency, voltage, and memory delay cycle combinations are stored in a table, illustrated in Table 4.1. Memory timings for each possible clock speed were computed from the memory component data sheets. Intel does not, however, specify a minimum core voltage for each speed, so the voltage column of the table had to be derived experimentally, as described in Section 5.2.2.
4.5 Application Programming Interface

Low-level driver code not essential to the operation of eCos or RedBoot has been separated from the eCos source tree and placed in a separate code library known as the µAMPS API. This was done to streamline the process of developing the radio, sensor, and power management code by not requiring everyone involved in the development process to learn the details of the eCos source tree.

The API consists of a library, against which application code that uses the API must be linked, and a collection of C header files. The contents of each header files is described in detail in Appendix A.

The core modules of the API define power management procedures and implement a high-resolution timer system.

The power management module itself includes only a few procedures, but it defines a set of standard procedures that every power-scalable subsystem of the node implements. Each system is allowed to define a set of power states. Generic states, such as ACTIVE or SHUTDOWN are defined by the power module of the API, and can be shared by any API module. Modules may also define their own, specialized states where necessary. The power API module defines a class of procedures for changing and monitoring the state of each subsystem. When checking the state of a subsystem, the subsystem may return any of its static states (such as SHUTDOWN), or a dynamic state (for example, CHANGING_TO_SHUTDOWN). Dynamic states encapsulate the idea that state changes are not instantaneous. There is a delay, for example, between when a subsystem is switched into its ACTIVE state, and when the subsystem can actually be used. When a subsystem is ordered into a given state $X$, the subsystem should immediately change its reported state to CHANGING_TO_$X$. Once the subsystem has actually reached state $X$, it should automatically begin reporting that it is in state $X$. This is facilitated by the timer module.

The timer module makes use of one of the StrongARM’s timer peripherals to provide accurate time measurements and programmed delays, with a resolution of about 270s. The timer module supports the simultaneous use of multiple software
timers, so each subsystem may use a timer object to keep track of its current power state, or perform other system-specific operations. Software timers can be used for measurement, or for generating delays, or for setting up callbacks. A callback is an indication that a specific piece of code is to be run at a certain time in the future.

Based on the power and timer modules, API modules have been developed for the built-in acoustic sensor and the radio. The sensor module supports requests for blocks of periodic samples from the microphone. Multiple, simultaneous requests are queued, making for a simple implementation of double buffering schemes. Since, for each block of samples, all future sampling times can be computed in advance, the analog-to-digital converter can be automatically placed in standby mode whenever possible. The radio implements a TDMA scheme, mostly in hardware, but relies on the timer module for automatic frame synchronization.
Chapter 5

Performance Analysis of the \( \mu \)AMPS Revision 1 Node

5.1 Intent of the Analysis

The previous four chapters described the design of the Revision 1 \( \mu \)AMPS node. This chapter will show that the design described meets the original design goals, as defined in Section 1.4.2. Those goals were

**Compactness** The node is to be as small as possible.

**Power-Awareness** The node is to consume a little power as possible and is to provide viable opportunities for making dynamic energy/quality tradeoffs.

**System Integration** The components of the system should connect cleanly and form a fully functional sensor node.

The degree to which the first of these goals, compactness, was achieved can be determined by inspection. At 55mm square, and 30mm high (not including the battery), the Revision 1 node is much smaller than the three Revision 0 boards. The reduction is due to the use of smaller components, and tighter circuit board layouts. Compactness was the least important of the three design goals.
The capability for power-awareness can be measured in the lab. Jumpers were incorporated into the PCBs to allow for inserting an ammeter into the power supply to each of the node’s subsystems. A detailed profile of the node’s power consumption can thus be constructed. The profile will show not only which parts of the node consume the most power, and how widely the power consumption of each part can be varied, but also how the various modes of operation of each component of the node can be combined to provide energy scalability at the system level.

Finally, demonstrating that the system is fully functional is an important part of showing that the design is feasible, and that the project is ready to move on to the next level.

5.2 Power Measurements

5.2.1 Subsystems and Their Modes of Operation

Figure 5.2.1 illustrates the power management controls provided by the \( \mu \)AMPS node hardware. Chapter 3 described how the controls were implemented. Now, the effectiveness of these controls is evaluated.

The Revision 1 processor and radio boards incorporate several power measurement jumpers in their designs. These jumpers, shown as zero ohm resistors on the schematics in Appendix B, can be desoldered and replaced with ammeters in order to measure the current consumption of various major subsystems of the node. Table 5.1 shows the power consumed by each of the individually measurable subsystems, in various modes of operation.

The processor has two power supplies: one for the core, and one for I/O. The core power varies greatly, depending on the processor clock speed in use. The two extreme’s of the StrongARM’s frequency range are shown in Table 5.1; a more detailed profiling of the processor core’s power consumption will be discussed later. In idle mode, portions of the core are still clocked, though the ARM core’s execution units are not. Power consumption is therefore less than in active mode, but still non-
Figure 5-1: Power management capabilities of the µAMPS node
Table 5.1: Power consumption of node subsystems in all supported modes of operation

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Mode</th>
<th>Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor (core)</td>
<td>Active (59MHz)</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>Active (221MHz)</td>
<td>552</td>
</tr>
<tr>
<td></td>
<td>Idle (59MHz)</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Idle (221MHz)</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>Sleep</td>
<td>0</td>
</tr>
<tr>
<td>Processor (I/O)</td>
<td>Active (Caches on)</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Active (Caches off)</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Idle</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Sleep</td>
<td>2</td>
</tr>
<tr>
<td>RAM</td>
<td>Active</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Idle</td>
<td>0</td>
</tr>
<tr>
<td>ROM</td>
<td>Active</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Idle</td>
<td>0</td>
</tr>
<tr>
<td>On-board Sensor</td>
<td>Active</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Idle</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Shutdown</td>
<td>0</td>
</tr>
<tr>
<td>Radio (RF)</td>
<td>Transmit (0dBm)</td>
<td>204</td>
</tr>
<tr>
<td></td>
<td>Transmit (15dBm)</td>
<td>1224</td>
</tr>
<tr>
<td></td>
<td>Receive</td>
<td>364</td>
</tr>
<tr>
<td></td>
<td>Standby</td>
<td>24</td>
</tr>
<tr>
<td>Base(^b)</td>
<td></td>
<td>28</td>
</tr>
</tbody>
</table>

\(^a\)Also see Table 5.3 for power measurements at other frequencies

\(^b\)Residual power consumption when all other systems are shut down.
zero. In sleep mode, the processor’s core power supply is turned off, resulting in zero power consumption. The processor I/O power is predominantly affected by the frequency of off-chip memory accesses. Table 5.1 shows values for operating with the StrongARM's instruction and data caches on (and no cache misses occurring)—resulting in very low I/O power—and with the caches off—resulting in higher I/O power, since every instruction fetch results in a read from ROM. In a real application, the I/O power would lie between these two extremes.

The caches were also toggled on and off while measuring the power consumed by the memory circuits. With the caches on, and therefore no memory accesses, both RAM and ROM consume extremely little power. Turning the caches off automatically results in frequent ROM accesses due to instruction fetching. A short test program was used to generate rapid RAM accesses. The ROM power is slightly higher than that consumed by the RAM, probably because the flash chip used for ROM is an older design than the SRAM, and because the program used for testing RAM did not generate memory accesses quite as quickly as possible.

The acoustic sensor built into the processor board consumes almost 30mW in active mode. About half of this is due to the switched capacitor anti-aliasing filter, which is turned off in standby mode. Much of the remaining power is accounted for by the necessary microphone bias current.

Radio power consumption varies from a few milliwatts in standby mode to more than a watt when the transmitter’s power amplifier is at its highest setting. Most of the radio’s power consumption in transmit and receive modes is due to the RF circuitry. Radio standby mode power represents power consumed by the FPGA, with the RF circuits powered down.

5.2.2 Frequency and Voltage Scaling of the StrongARM Processor

A key component of the µAMPS power scaling strategy is varying the operating frequency of the processor to exactly match the processing demand placed upon it
at any given time. To further optimize power consumption, the core power supply voltage is dynamically varied to match the current clock frequency. The StrongARM clock frequency can be scaled in twelve steps from 59 to 221MHz, while the MAX1717-based core voltage supply can range from 0.9 to 2.0V in 30 steps. The twelve most optimal frequency/voltage pairs are stored in a table that is used by the operating system to update the voltage whenever the clock frequency is changed.

Intel specifications for the SA-1110BB require a core voltage between 1.58 and 1.93V [7]. No comment is made about dynamically changing the voltage, although the specifications do discourage dynamically changing the clock speed. Certainly, Intel makes no guarantee that the processor will work reliably with changing clock speed, and some decrease in stability is to be expected, as is variation from sample to sample of the SA-1110.

Determining the optimal frequency voltage pairs for the SA-1110 first requires characterizing several samples of the component. Table 5.2 shows the minimum voltage required at each frequency, for two different SA-1110 samples. Data for the table was generated using the test program listed in Appendix E. The program works by performing a series of tasks at each possible core voltage setting. The tasks are designed to determine if the processor is functioning correctly, and consist of cached and uncached memory accesses, and simple arithmetic operations. The values in Table 5.2 are the minimum voltages at which the processors correctly performed all of the tasks. The two samples in Table 5.2 match rather closely, allowing operation at as little as 0.925V at 59MHz, and requiring 1.550V at 221MHz.

Accepting the fact that \(\mu\)AMPS Revision 1 is an experimental platform, and that some potential instability can be tolerated in order to show the full potential of a power-awareness technique, the following frequency/voltage pairs were chosen, based on the two SA-1110 samples in Table 5.2.
Table 5.2: Minimum voltage at each operating frequency, for three different SA-1110 processors.

<table>
<thead>
<tr>
<th>Frequency (MHz)</th>
<th>Processor 1 (V)</th>
<th>Processor 2 (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>0.925</td>
<td>0.925</td>
</tr>
<tr>
<td>74</td>
<td>0.925</td>
<td>0.925</td>
</tr>
<tr>
<td>88</td>
<td>0.925</td>
<td>0.925</td>
</tr>
<tr>
<td>103</td>
<td>0.925</td>
<td>0.925</td>
</tr>
<tr>
<td>118</td>
<td>0.925</td>
<td>0.975</td>
</tr>
<tr>
<td>132</td>
<td>1.025</td>
<td>1.050</td>
</tr>
<tr>
<td>147</td>
<td>1.100</td>
<td>1.100</td>
</tr>
<tr>
<td>162</td>
<td>1.150</td>
<td>1.175</td>
</tr>
<tr>
<td>177</td>
<td>1.225</td>
<td>1.250</td>
</tr>
<tr>
<td>192</td>
<td>1.300</td>
<td>1.350</td>
</tr>
<tr>
<td>206</td>
<td>1.400</td>
<td>1.450</td>
</tr>
<tr>
<td>221</td>
<td>1.500</td>
<td>1.550</td>
</tr>
</tbody>
</table>

These values were arrived at by adding 200mV to the minimum voltage from Table 5.2 for whichever of the sample parts had the higher voltage requirement at that frequency. This table of frequency/voltage pairs was used to generate Table 5.3, which shows the core power consumption for the SA-1110 at each of its operating frequencies, in both active and idle modes. Active mode power measurements were taken while running a test program which takes samples from the processor board’s microphone and prints them to the serial port. For comparison, Table 5.3 also shows
Table 5.3: Core power consumption for the SA-1110 at each operating frequency

<table>
<thead>
<tr>
<th>Frequency (MHz)</th>
<th>Voltage (V)</th>
<th>Variable V&lt;sub&gt;core&lt;/sub&gt;</th>
<th>Fixed V&lt;sub&gt;core&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(mW)</td>
<td>Active Power</td>
<td>Idle Power</td>
</tr>
<tr>
<td>59</td>
<td>1.125</td>
<td>60</td>
<td>9.1</td>
</tr>
<tr>
<td>74</td>
<td>1.125</td>
<td>74</td>
<td>10.6</td>
</tr>
<tr>
<td>88</td>
<td>1.125</td>
<td>88</td>
<td>12.5</td>
</tr>
<tr>
<td>103</td>
<td>1.125</td>
<td>102</td>
<td>14.1</td>
</tr>
<tr>
<td>118</td>
<td>1.175</td>
<td>127</td>
<td>17.1</td>
</tr>
<tr>
<td>132</td>
<td>1.250</td>
<td>162</td>
<td>21.5</td>
</tr>
<tr>
<td>147</td>
<td>1.300</td>
<td>195</td>
<td>25.6</td>
</tr>
<tr>
<td>162</td>
<td>1.350</td>
<td>233</td>
<td>30.2</td>
</tr>
<tr>
<td>177</td>
<td>1.450</td>
<td>294</td>
<td>38.1</td>
</tr>
<tr>
<td>192</td>
<td>1.550</td>
<td>368</td>
<td>47.7</td>
</tr>
<tr>
<td>206</td>
<td>1.650</td>
<td>453</td>
<td>58.7</td>
</tr>
<tr>
<td>221</td>
<td>1.750</td>
<td>552</td>
<td>71.4</td>
</tr>
</tbody>
</table>

power measurements for the same sample application, but with a fixed 1.750V power core power supply.

Figure 5-2 shows a plot of the information from Table 5.3. With a fixed core voltage, power varies linearly with operating frequency. This means there is no incentive to operate the processor at less than maximum speed. The optimal power management scheme in a fixed-voltage application is to run the processor as fast as possible, so that tasks are finished as quickly as possible, and put the processor into sleep mode as frequently as possible. With a variable core voltage, active mode power consumption is reduced by more than 60% at 59MHz. In a system with variable core voltage, the processor should be run at the slowest speed that still allows all tasks to complete before their deadline.

5.2.3 Node-Level Power Management

Tabulating the power consumption of individual node components, as in Table 5.1, illustrates the degree of power scaling permitted by the µAMPS Revision 1 hardware. It is also important to consider the expected total power consumption of the node.
Figure 5-2: SA-1110 power consumption and minimum voltage versus frequency
This is a difficult prediction to make for a power-aware system, because the average total power consumption varies not just with the type of sensor application, but also with the characteristics of the actual signals recorded. The problem can be simplified by from among the node’s many possible operating modes a few important characteristic modes.

During the design of the revision 1 node, four canonical modes of operation for the node were envisioned. These global modes and the respective states of each subsystem in each mode, are shown in Table ??.

The first mode is a deep-sleep mode. The node would spend most of its time in this lowest possible power mode. In this mode, all systems are shutdown. The only way that the node can transition out of this mode is through an alarm generated by the StrongARM’s real time clock. Deep sleep mode must be used carefully, because once entered, the node remains asleep for a fixed amount of time. There is no way for the node to know if it is missing an important event.

The second lowest power mode was invented as an attempt to avoid the problem of complete unresponsiveness in sleep mode, without significantly increasing power consumption. In threshold mode, only the sensor circuitry is powered up. Instead of measuring the sensor output using an analog-to-digital converter, which would require putting the processor in active mode, an analog threshold comparator is used to detect if the sensor signal exceeds some programmable threshold. If the threshold is exceeded, an interrupt signal wakes the processor, which can begin recording the event. The peak detector circuit described in Section 3.2.7 implements this functionality for the sensor built into the µAMPS Revision 1 processor board. The same

<table>
<thead>
<tr>
<th>Mode</th>
<th>Processor</th>
<th>Sensor</th>
<th>Radio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deep sleep</td>
<td>Sleep</td>
<td>Shutdown</td>
<td>Shutdown</td>
</tr>
<tr>
<td>Threshold</td>
<td>Sleep</td>
<td>Threshold</td>
<td>Shutdown</td>
</tr>
<tr>
<td>Sensing</td>
<td>Active</td>
<td>Active</td>
<td>Shutdown</td>
</tr>
<tr>
<td>Receive</td>
<td>Active</td>
<td>—</td>
<td>Receive</td>
</tr>
<tr>
<td>Transmit</td>
<td>Active</td>
<td>—</td>
<td>Transmit</td>
</tr>
</tbody>
</table>
concept can be used, however, with any sensor built for the \( \mu \)AMPS Revision 1 node.

In the third canonical operating mode, the sensor and processor are both active. This is the lowest power mode in which the node is capable of actually doing something.

The fourth mode adds the use of the radio. Since the current radio design consumes far more power than the built-in sensor, no distinction is made between whether the sensor is on or off in this mode. When active, the radio can either be receiving or transmitting data. Transmit/receive switching is under the control of hardware responsible for implementing the TDMA protocol, and, since the duty cycles generated by this power-aware protocol are application and scenario dependent, it is simplest to consider transmit and receive modes separately.

Figure 5-4 illustrates the total node power consumption, and its breakdown into consumption by individual components, for the four canonical operating modes. Because the variable processor clock speed and variable transmitter power amplifier induce wide power consumption variations in each of the three active modes, both minimum and a maximum power variations of each of these modes are illustrated.

Power consumption ranges from 50mW in deep sleep, to almost 2W while transmitting at maximum power. Assuming the processor and power amplifier are seldom operated at their maximum power levels, power consumption in any of the active modes should seldom be more than 400mW. Assuming the node spends 99% of its life in deep-sleep mode (a typical duty cycle for a microsensor node), the average power consumption is about 55mW, which produces a lifetime of about 100 hours using a battery pack composed of four high-capacity AAA alkaline cells.[5]

### 5.3 Test Applications

A number of simple demonstration applications have been developed for the \( \mu \)AMPS Revision 1 node.

The first test application was a single-channel acoustic recorder. The microphone built into the processor board was sampled at 2kS/s. Roughly every second, 500
Figure 5-4: Power consumption in canonical node operating modes
consecutive samples worth of data was transmitted to a PC via the interface board’s RS-232 port. On the PC, a Java program displayed the recorded waveform. Figure 5-5 shows a screenshot of the Java application. This application was eventually extended to utilize the µAMPS radio. Two nodes were used: one collected data from its microphone, and sent them to the other node via radio. The second node then transmitted the data to a PC via it’s serial port.

A second demonstration application was developed using the four-channel acoustic sensor board. In this application, three of the four microphone channels were sampled at 1kS/s, and the data was again sent over a serial link to a PC. The 12-bit analog-
to-digital converter results were truncated to seven bits in order to not exceed the bandwidth of the serial link. On the PC, a Java program displayed the data and recorded it to a file. This application was actually deployed at the U.S. Army’s Aberdeen Proving Grounds where it was used to record the sounds of tanks and other military vehicle moving around a track. The data collected during this event was later analyzed using a beamforming algorithm to determine a line-of-bearing from the sensor node to the tank.

This three-microphone application was modified to perform the beamforming analysis on the sensor node itself. The radio was also incorporated, so that the sensor node transmitted the line-of-bearing information to a base station node, which communicated with a PC-based Java program that displayed the line-of-bearing on the PC’s display.
Chapter 6

Conclusions

The revision 1 node described in this thesis represents a significant advancement of the μAMPS project. A complete, fully functional μAMPS node has been demonstrated, and has been shown to have the capacity for a significant degree of power-awareness. The revision 1 node will serve as a testbench for the development and analysis of network protocols and complete microsensor applications. Furthermore, lessons learned from the analysis of this node based on off-the-shelf components will be applied to the next phase of the μAMPS project: creating a node on a chip.

The revision 1 node achieves its goals of small size and reduced power consumption, as compared to the revision 0 node. The new node hardware provides many power management mechanisms, allowing total node power to be scaled by a factor of $40 \times$, from almost 2W down to just 50mW. This power-scalability is a good basis for power-awareness, although the software necessary to intelligently select operating modes has not been written yet.

Simple, sample applications developed for the node have demonstrated its ability to collect, process, and communicate sensor data, at least in relatively well-controlled lab environments. In the near future, additional applications will be developed. An extension of the simple beamforming application to automotive vehicle tracking is planned. This application will utilize at least two sensor nodes, each equipped with three microphones and communicating with a third basestation node. Beamforming, performed at both sensor nodes, will establish two lines-of-bearing, from which the
location of vehicles will be determined through triangulation. This will be the first µAMPS application involving more than two nodes, and will thus be a test of the nodes’ network protocol.

Analysis of the revision 1 reveals limitations that will need to be carefully considered during the development of future µAMPS nodes. Power consumption by the node is currently dominated by the processor and radio power amplifier. Each of these consume an order of magnitude more power than the sensor subsystem. Much research has already been done on high-efficiency RF amplifiers. It is likely, therefore, that the solution to reducing the power required by the µAMPS radio, lies not in a better amplifier, but in a more robust signal encoding and a more flexible network protocol. Similarly, reducing power consumption in the processor will likely be done not by developing a new processor that magically provides the same performance as the StrongARM with less power, but by developing a microsensor specific processor architecture with exactly the amount of performance required by microsensor applications, and offers hardware support for common tasks which are particularly difficult to implement in software.

Besides reducing active mode power, the design of the next µAMPS node will need to focus on sleep state power. Due to the use of commercial dc/dc converters, which must remain on during sleep mode in order to keep the contents of memory intact, the revision 1 node dissipate to draw 50mW when in sleep mode. Only a few microwatts are required to keep the SRAM alive; the remainder of this power is wasted by inefficient power supplies. Given the high percentage of time the nodes are expected to spend in deep sleep mode, minimizing power consumption in this mode has a huge effect on the lifetime of the nodes.

Of course, the low-power and power-aware design techniques used in the µAMPS Revision 1 node implementation are not limited to the domain of microsensor nodes. Any energy-constrained application can potentially benefit from power-aware hardware and software design. The relatively new concept of networked microsensors is particularly challenging energy-constrained problem due to the inherent real-time requirements, necessary long node lifetimes, and complicated, potentially dynamic
networking. Through the revision 1 node, the $\mu$AMPS project demonstrates solutions to all of these challenges.
Appendix A

The $\mu$AMPS Revision 1 Low-Level Node API

A.1 Overview

The $\mu$AMPS application programming interface is a collection of header files and an archive of functions that extend the eCos operating system and provide basic access to the radio, sensor, and power management components of the $\mu$AMPS node.

The API is implemented in C, and consists of a collection of header files containing declarations of the procedures and data types defined by the API. User programs are built by linking the user’s code against a single archive file containing the object files for each module of the API. The $\mu$AMPS API does not include eCos; therefore the eCos library file must also be included in the linking process.

The API contains the following components
Functions for controlling the acoustic sensor built into the processor board.

Definitions and functions for power management.

Functions for directly accessing the \(\mu\)AMPS radio.

Functions for microsecond-resolution timing.

System-level API components, including system-level initialization and last-minute eCos patches.

A.2 Acoustic Sensor: <uapi/analog.h>

This header file defines the interface for the acoustic sensor built into the \(\mu\)AMPS processor board. The interface provides procedures for making one-time measurements of the output of the microphone amplifier or the output of the envelope detector. These simple commands are useful for debugging, but generally are not sufficient for real applications, where many precisely spaced, sequential samples must be collected. The primary sensor interface, therefore, is through the analog_submit_request procedure, which enqueues a request for a specified number of samples, to be taken with a specified inter-sample delay. Requests are processed sequentially, in the order received. analog_request_completion_check is used to check the completion of a request. Procedures for setting the microphone gain, and the threshold for the envelope detector, are also provided.

The sensor system follows the standard \(\mu\)AMPS power management interface (discussed in Section A.3). The following power modes are defined:

**SHUTDOWN** The power supply to all of the analog circuitry is turned off.

**STANDBY** The analog power supply is enabled, but the anti-aliasing prefilter is in its shutdown state.
CHANGING TO STANDBY The analog power supply has just been enabled. The power state will automatically switch to STANDBY once the power supply has had a chance to stabilize.

ACTIVE The analog power supply is enabled, and the prefilter is on.

CHANGING TO ACTIVE The prefilter has just been enabled. The power state will automatically switch to ACTIVE once the prefilter has stabilized.

Changing and checking the sensor power state is accomplished with the procedures analog_get_pwr_state, analog_get_stable_pwr_state, and analog_set_pwr_state.

DATA TYPES AND CONSTANTS

analog_request_t

Used to hold the parameters for a request for measurement data from the analog system. The structure of this type is not intended to be visible to the user.

ENVCHANNEL

Constant used to identify the envelope detector A/D channel.

MICCHANNEL

Constant used to identify the microphone A/D channel.

PROCEDURES

cyg_uint32 analog_get_pwr_state(void)

Returns the current power state of the sensor.

cyg_uint32 analog_get_stable_pwr_state(void)

Returns the power state of the sensor after waiting, if necessary, for the sensor to reach one of the stable power states: SHUTDOWN, STANDBY, or ACTIVE).
**Cyg_ErrNo analog_set_pwr_state(cyg_uint32 state)**

Changes the sensor power state to the specified state. If **state** is not one of **SHUTDOWN**, **STANDBY**, or **ACTIVE**, then the error code **ENOSUP** ("operation not supported") is returned. Otherwise, **ENOERR** ("no error") is returned.

**cyg_uint16 analog_measure_mic(void)**

Returns one sample, taken immediately, from the microphone.

**cyg_uint16 analog_measure_envelope(void)**

Returns one sample, taken immediately, from the envelope detector.

**void analog_set_mic_gain(cyg_uint8 gain)**

Sets the microphone gain. The **gain** argument can range from 0 to 31.

**void analog_set_signal_threshold(cyg_uint8 thresh)**

Sets the envelope detector threshold. The argument **thresh** can range from 0 to 31.

**void analog_init_request(analog_request_t *request, cyg_uint16 *buffer, cyg_uint32 length, cyg_uint8 channel, cyg_uint32 period)**

Initializes an **analog_request_t** object. The array **buffer** will be used to store the samples, and must be at least **length** entries long. The **channel** argument can be either **MIC_CHANNEL** or **ENV_CHANNEL**. **period** specifies the delay between samples. Macros for specifying **period** in standard time units are found in **timer.h**.

**void analog_submit_request(analog_request_t *request)**

Submits **request** for processing. **request** must have been initialized previously.

**cyg_bool analog_request_completion_check(analog_request_t *request)**

Returns **false** if sensor is currently collecting samples for **request**, or if
is in the request queue. Otherwise, returns true.

A.3 Node Power Management: <uapi/power.h>

The power header file defines constants used by power management routines defined by each of the \( \mu \)AMPS subsystems, and also provides procedures for placing the CPU in its idle or sleep modes. Basic power management of the \( \mu \)AMPS subsystems is enabled by procedures with names of the form

\[
\begin{align*}
&\text{cyg\_uint32 } xxxx\_get\_pwr\_state(\text{void}) \\
&\text{cyg\_uint32 } xxxx\_get\_stable\_pwr\_state(\text{void}) \\
&\text{Cyg\_ErrNo } xxxx\_set\_pwr\_state(\text{cyg\_uint32 state})
\end{align*}
\]

where \( xxxx \) represents the name of the subsystem in question. These routines are defined in the drivers for the individual subsystems: see the API documentation for each subsystem.

Procedures of the form \( xxxx\_get\_pwr\_state \) return the instantaneous power management state of a subsystem. This can be the constant for one of the standard modes defined in the \texttt{power.h} header file (and described below), or a constant for a special mode defined in the subsystems own header file. The \texttt{power.h} header file defines constants for a generic subsystem with three stable states: shutdown, standby, and active. Constants are also defined to describe transient states: that is, when the system is transitioning to any of the three stable states. Subsystems with more than three stable states (or which are not well described by the generic states) define their own power management states in their own header files.

\( xxxx\_get\_stable\_pwr\_state \) procedures are similar to \( xxxx\_get\_pwr\_state \) procedures, but do not return transition states. If the subsystem is in a transition state, \( xxxx\_get\_stable\_pwr\_state \) will block until the subsystem finishes the transition. The final, stable state will be returned.

The \( xxxx\_set\_pwr\_state(\text{state}) \) procedures are used to command a subsystem to enter a specific power management state. Any transition may be commanded. For example, if the active state can only be reached from the standby state, then if
the subsystem is currently shutdown when a command to go to the active state is received, the subsystem’s driver must automatically transition the subsystem through the standby state. Repetitive calls to `xxxxx.get_pwr_state` would therefore reveal the following series of modes:

1. SHUTDOWN
2. CHANGING_TO_STANDBY
3. STANDBY
4. CHANGING_TO_ACTIVE
5. ACTIVE

**DATA TYPES AND CONSTANTS**

**SHUTDOWN**

Generic power management mode indicating that a subsystem is in its lowest-power and least active state.

**CHANGING_TO_SHUTDOWN**

Generic power management mode indicating that a subsystem is in the process of changing to the SHUTDOWN mode.

**STANDBY**

Generic power management mode indicating that a subsystem is in a stable, intermediate state between SHUTDOWN and ACTIVE. Returning to the ACTIVE state from STANDBY should take less time than transitioning to ACTIVE from SHUTDOWN.

**CHANGING_TO_STANDBY**

Generic power management mode indicating that a subsystem is in the process of changing to the STANDBY state.
**ACTIVE**

Generic power management mode indicating that a subsystem is in its most active mode.

**CHANGING_TO_ACTIVE**

Generic power management mode indicating that a subsystem is in the process of changing to the ACTIVE mode.

**PROCEDURES**

```c
void idle(void)
```

Puts the processor in idle mode. The processor will wake up when the next interrupt is signaled.

```c
void pause(cyg_uint32 duration)
```

Puts the processor in idle mode, and also sets an alarm to bring the processor out of sleep duration units later. (Macros for specifying `duration` in standard units of time are defined in `timer.h`).

```c
void sleep(cyg_uint32 duration)
```

Puts the processor in sleep mode, after setting an alarm to wake up the processor in `duration` time units.

**A.4 Radio: <uapi/radio.h>**

The final radio API is still under development. The current, complicated interface will be reduced to just the components described here, plus procedures for setting configuration options in the radio’s low-level protocol.

In normal operation, the radio is in TDMA mode, and switching between transmit and receive occurs automatically, under hardware control. The radio supports the generic **SHUTDOWN**, **STANDBY**, and **ACTIVE** power modes, as well as the custom modes
TRANSMIT, RECEIVE, and TDMA. In SHUTDOWN, all controllable radio circuits are powered down, including the FPGA. Transitioning to STANDBY mode therefore requires waiting for the FPGA to finish loading its configuration from EEPROM. In STANDBY mode, the FPGA is active, but all RF circuits remain powered down. TRANSMIT and RECEIVE modes are for debugging, and lock the radio into either full-time transmission (sending packets whenever they are queued) or full-time reception (receiver is always listening for packets). The ACTIVE and TDMA modes are synonyms, and indicate that the radio is to synchronize itself with surrounding nodes and automatically switch between transmit, receive, and idle states according to a TDMA schedule.

### Data Types and Constants

**PACKET_LENGTH**

The length (in bytes) of the data portion of a radio packet.

**RECEIVE**

Power management mode where the receiver is active.

**CHANGING_TO_RECEIVE**

Power management mode indicating that the radio is in the process of changing to the RECEIVE mode.

**TRANSMIT**

Power management mode where the transmitter is active.

**CHANGING_TO_TRANSMIT**

Power management mode indicating that the radio is in the process of changing to the TRANSMIT mode.

**TDMA**

Synonym for ACTIVE power management state. Indicates the use of a TDMA schedule.
protocol for automatic switching between transmit and receive.

**PROcedures**

*Cyg_ErrNo radio_init(void)*

Initializes the radio.

*cyg_uint32 radio_get_pwr_state(void)*

Returns the current power management state of the radio.

*cyg_uint32 radio_get_stable_pwr_state(void)*

Waits until the radio power state is SHUTDOWN, STANDBY or ACTIVE, and then returns that state.

*Cyg_ErrNo radio_set_pwr_state(cyg_uint32 state)*

Sets the radio power management state. Returns ENOSUP (“operation not supported”) if *state* is not one of SHUTDOWN, STANDBY, ACTIVE, or TDMA. Otherwise, ENOERR (“no error”) is returned.

*Cyg_ErrNo radio_send(cyg_uint8 *buf)*

Queues the data pointed to by *buf* for transmission in the next available TDMA transmit slot (or immediately, if the radio is in TRANSMIT mode).

*Cyg_ErrNo radio_get(cyg_uint8 *buf)*

Copies the next available packet from the radio receive FIFO into *buf*.

**A.5 Timer: <uapi/timer.h>**

The timer header file provides high-precision timing procedures for event duration measurement and generating short time delays. The StrongARM’s 3.6864MHz OS clock serves as the base clock for all of the procedures defined in this header file. The basic unit of time for all these procedures is therefore $1/3686400s = 271.267\text{ns}$. 

91
Macros are provided for converting to and from more standard measures of time, such as milliseconds or microseconds. Times are stored as unsigned 32-bit integers, making the maximum representable time about 19 minutes.

Timer operations are all performed relative to a reference timestamp known as the timer object’s mark point. The mark point can be read and written using the get_mark and set_mark procedures. Additionally, a timer’s mark point can be set to the present instant using the mark procedure. The initial value of a timer’s mark point is the OS timer value when the timer_create procedure was called to initialize that timer.

**DATA TYPES AND CONSTANTS**

`timer_t`

Container for timer objects.

`void timer_fn_t(timer_t *t, cyg_uint32 d)`

Procedure prototype for timer callbacks. Any user-supplied procedure matching this prototype can be used as a callback procedure. When the callback occurs, *t will contain the timer object that triggered the callback, and d will contain the data value that was specified when the timer_callback procedure was called to setup the callback.

**PROcedures**

`void timer_init(void)`

Initializes the timer system. This procedure must be called before any timer objects can be created.

`void timer_create(timer_t *timer)`

Initializes *timer and sets its mark point to the current value of the OS timer.
cyg_uint32 timer_mark(timer_t *timer)

Returns the current value of the OS timer and also sets the mark point of *timer to that value.

cyg_uint32 timer_measure(timer_t *timer)

Returns the number of OS clock ticks since the mark point of *timer

void timer_delay(timer_t *timer, cyg_uint32 duration)

Delays until duration OS clock ticks after the mark point for *timer before returning.

void timer_callback(timer_t *timer, timer_fn_t *func, cyg_uint32 data, cyg_uint32 duration)

Arranges for the function func to be called with the argument data after duration OS clock ticks have passed since timer’s mark point. The call to func will occur in a DSR, so func may not do anything that requires the scheduler to run.

cyg_uint32 time_s(cyg_uint32 duration)

Converts duration from seconds to OS clock ticks. The argument duration must be between 0 and 1,165.

cyg_uint32 time_to_s(cyg_uint32 duration)

Converts duration from OS clock ticks to seconds.

cyg_uint32 time_ms(cyg_uint32 duration)

Converts duration from milliseconds to OS clock ticks. The argument duration must be between 0 and 116,508.

cyg_uint32 time_to_ms(cyg_uint32 duration)

Converts duration from OS clock ticks to milliseconds.
cyg_uint32 timer_us(cyg_uint32 duration)

Converts duration from microseconds to OS clock ticks. The argument duration must be between 0 and 116,508.

cyg_uint32 time_to_us(cyg_uint32 duration)

Converts duration from OS clock ticks to microseconds.

cyg_uint32 time_ns(cyg_uint32 duration)

Converts duration from nanoseconds to OS clock ticks. The argument duration must be between 272 and 429,496.

cyg_uint32 time_to_ns(cyg_uint32 duration)

Converts duration from OS clock ticks to nanoseconds.

A.6 General: <uapi/uamps.h>

This header file provides a few small, miscellaneous interfaces which do not fit into any other modules.

DATA TYPES AND CONSTANTS

cchar *version

A text string describing the version of the uamps API, including the date and time at which the API was compiled.

PROCEDURES

void uamps_init(void)

Performs any initialization tasks not automatically completed by eCos on startup. This procedure will call the initialization procedures from the analog, radio, and timer API modules, so frequently, calling uamps_init is the
only initialization the user needs to perform. After this procedure is run, all systems (analog, radio, …) will be in their **SHUTDOWN** state.

```c
void uamps_green_led(cyg_uint32 state)

    If `state` is 0, turns off the green LED on the processor board. Otherwise, turns on the LED.

void uamps_red_led(cyg_uint32 state)

    If `state` is 0, turns off the red LED on the processor board. Otherwise, turns on the LED.
```
Appendix B

Schematics

This chapter contains full schematics for the \( \mu \)AMPS Revision 1 processor, basestation, and fou-channel acoustic sensor boards. For schematics of the radio board, see [9].
B.1 Processor Board
B.2 Basestation Board
B.3 Four Channel Acoustic Sensor
Appendix C

PCB Layouts

All layouts are shown from the top (component) side of the board and at actual size.
C.1 Processor Board

Layer 1 (Top)

Layer 2 (Ground Plane)

Layer 3

Layer 4
C.2 PC Interface

Layer 1 (Top)
Layer 2 (Ground Plane)
Layer 3 (Power Plane)
Layer 4 (Bottom)
C.3 4-Channel Acoustic Sensor

Layer 1 (Top)

Layer 2 (Bottom)
Appendix D

Implementation of Power Management Additions to eCos

This chapter contains a listing of the important pieces of the power management support that was added to eCos for the $\mu$AMPS project.

D.1 Platform Macros

File: packages/hal/arm/sa11x0/uamps/current/include/uamps.h

This is a portion of the $\mu$AMPS platform header file, showing the definitions of macros used to manipulate the various hardware power management signals on the Revision 1 processor board.

//GPIO masks
#define UAMPS_VCORE_D0 0x00000020 /* Core voltage select 0 */
#define UAMPS_VCORE_D1 0x00000040 /* Core voltage select 1 */
#define UAMPS_VCORE_D2 0x00000080 /* Core voltage select 2 */
#define UAMPS_VCORE_D3 0x00000100 /* Core voltage select 3 */
#define UAMPS_VCORE_D4 0x00000200 /* Core voltage select 4 */
#define UAMPS_VCORE_MASK 0x00000370
#define UAMPS_DVS_EN 0x00000400 /* DVS enable */
#define UAMPS_VCORE_STABLE 0x00000800 /* Core voltage stable */

// Set the core voltage. @volt_id@ is a five-bit code. Not all
// values are allowed: see MAX1717 datasheet.
#define UAMPS_SET_VOLTAGE(volt_id) 
  CYG_MACRO_START 
  *SA11X0_GPIO_PIN_OUTPUT_SET = ((volt_id)%32) << 5; 
  *SA11X0_GPIO_PIN_OUTPUT_CLEAR = (((volt_id)%32) ^ 31) << 5; 
  CYG_MACRO_END

// Enable processor voltage scaling. Voltage scaling is disabled
// whenever the processor’s ~RESET_OUT pin is asserted (which
// happens during hard reset, or sleep).
#define UAMPS_ENABLE_DVS() 
  CYG_MACRO_START 
  *SA11X0_GPIO_PIN_OUTPUT_CLEAR = UAMPS_DVS_EN; 
  *SA11X0_GPIO_PIN_OUTPUT_SET = UAMPS_DVS_EN; 
  CYG_MACRO_END

// Check whether the core voltage supply has stabilized. Returns
// true (not zero) if the supply is stable, false (zero) otherwise.
#define UAMPS_CHECK_VCORE_STABLE() 
  (*SA11X0_GPIO_PIN_LEVEL & UAMPS_VCORE_STABLE)

D.2 Power Management Procedures

File: packages/hal/arm/sa11x0/uamps/current/src/uamps.misc.c

This is a portion of the file containing assorted procedures for controlling the
µAMPS hardware. This file contains procedures for changing the processor speed
and voltage, and for setting up for sleep mode.

// Structure for keeping track of combinations of processor speed,
// processor voltage, and memory timings
typedef struct { 
  cyg_uint32 pll_val; 
  cyg_uint32 frequency; 
  cyg_uint32 dvs_val; 
  cyg_uint32 msc0; 
  cyg_uint32 msc1; 
} uamps_speed_record;

// This table defines the frequency/voltage/memory timing
// combinations.
uamps_speed_record uamps_speed_table[] = {

    // PLL Frequency DVS Voltage MSC0 MSC1
    // (Hz) code (mV) (RAM/ROM) (Radio)
    { 0x0, 59900000, 22, /* 1125 */ 0x01110210, 0xFFFCFFFC },
    { 0x1, 73700000, 22, /* 1125 */ 0x01110318, 0xFFFCFFFC },
    { 0x2, 88500000, 22, /* 1125 */ 0x02190420, 0xFFFCFFFC },
    { 0x3, 103200000, 22, /* 1125 */ 0x02190528, 0xFFFCFFFC },
    { 0x4, 118000000, 20, /* 1175 */ 0x02190528, 0xFFFCFFFC },
    { 0x5, 132700000, 17, /* 1250 */ 0x03210630, 0xFFFCFFFC },
    { 0x6, 147500000, 14, /* 1300 */ 0x03290738, 0xFFFCFFFC },
    { 0x7, 162200000, 13, /* 1350 */ 0x04290840, 0xFFFCFFFC },
    { 0x8, 176900000, 11, /* 1450 */ 0x04310840, 0xFFFCFFFC },
    { 0x9, 191700000, 9, /* 1550 */ 0x04310948, 0xFFFCFFFC },
    { 0xA, 206400000, 7, /* 1650 */ 0x05390a50, 0xFFFCFFFC },
    { 0xB, 221200000, 5, /* 1750 */ 0x05390b58, 0xFFFCFFFC }
};

// This table is used to convert the MAX1717 settings (a number
// from 0-31) to the respective voltages (in milливольts).

cyg_uint32 dvs_table[] = {
    2000, // 0
    1950, // 1
    1900, // 2
    1850, // 3
    1800, // 4
    1750, // 5
    1700, // 6
    1650, // 7
    1600, // 8
    1550, // 9
    1500, // 10
    1450, // 11
    1400, // 12
    1350, // 13
    1300, // 14
    0, // 15
    1275, // 16
    1250, // 17
    1225, // 18
    1200, // 19
    1175, // 20
    1150, // 21
    1125, // 22
    1100, // 23
    1075, // 24
    1050, // 25
};
// Keep track of the current voltage
cyg_uint32 current_voltage = 7; // Voltage after reset is 1.650V

// This macro performs the actual task of changing the processor
// clock frequency in a safe manner.
#define UAMPS_SET_PROC_CLOCK(speed) 
  CYG_MACRO_START 
  /* Disable clock switching */ 
  asm volatile("mcr p15,0,r0,c15,c2,0x2"); 
  /* Force core clock to MCLK by reading from uncacheable memory */ 
  { int temp; temp = *((volatile unsigned int *)0x08000000); } 
  /* Set the PLL. Processor will stall until PLL locks again. */ 
  *((int *)SA11X0_PWR_MGR_PLL_CONFIG) = speed; 
  /* Enable clock switching */ 
  asm volatile("mcr p15,0,r0,c15,c1,0x2"); 
  CYG_MACRO_END

/** Set the core clock frequency. The speed parameter is a value between
* 0x0 and 0xA.
*/
void uamps_set_processor_clock(cyg_uint32 val)
{
  if (val == uamps_get_processor_clock() || val > 12) {
    // Do nothing if speed is out of range, or the same as the
    // current speed.
  } else if (val > uamps_get_processor_clock()) {
    // We are about to increase the clock speed. It is important to
    // change the memory timings first, as we may not be able to read
    // the table at the faster processor speed otherwise.
    *SA11X0_STATIC_CONTROL_0 = uamps_speed_table[val].msc0;
    *SA11X0_STATIC_CONTROL_1 = uamps_speed_table[val].msc1;
    UAMPS_SET_PROC_CLOCK(val);
else {
    // We are about to decrease the clock speed. We should not change
    // the memory timings until after the clock has been slowed.
    UAMPS_SET_PROC_CLOCK(val);
    *SA11X0_STATIC_CONTROL_0 = uamps_speed_table[val].msc0;
    *SA11X0_STATIC_CONTROL_1 = uamps_speed_table[val].msc1;
}

/** Get the current clock setting, as a value from 0-11. */

cyg_uint32
uamps_get_processor_clock(void)
{
    return (*SA11X0_PWR_MGR_PLL_CONFIG % 32);
}

/** Get the current clock frequency, in hertz. */

cyg_uint32
uamps_get_processor_clock_frequency(void)
{
    return uamps_speed_table[uamps_get_processor_clock()].frequency;
}

/** Set the processor voltage. The val parameter is a number from
   * 0-31. (See the MAX1717 voltage table above.) */

void
uamps_set_dvs(cyg_uint32 val)
{
    if (val == current_voltage || val > 31 || dvs_table[val] == 0) {
        // Do nothing if requested voltage is out of range, the same as the
        // current voltage, or zero.
    } else {
        UAMPS_ENABLE_DVS();
        UAMPS_SET_VOLTAGE(val);
        current_voltage = val;
        while (! UAMPS_CHECK_VCORE_STABLE());
    }
}

/** Return the current processor voltage as a value from 0-31. */

119
uamps_get_dvs(void)
{
    return current_voltage;
}
/** Return the current processor voltage in millivolts. */
cyg_uint32
uamps_get_dvs_voltage(void)
{
    return dvs_table[current_voltage];
}
/** Set the speed of the processor by changing the frequency and the *
* voltage. The val parameter is a number form 0 to 11. */
void
uamps_set_speed(cyg_uint32 val)
{
    // We can't assume that the current voltage is optimal for current
    // frequency!

    if (val > 12) {
        // Do nothing if value is out of range.
        return;
    }

    if (dvs_table[current_voltage] < dvs_table[uamps_speed_table[val].dvs_val]) {
        // Current voltage is not adequate for new frequency. Change
        // voltage now.
        uamps_set_dvs(uamps_speed_table[val].dvs_val);
    }

    // It is now safe to change the processor clock frequency
    uamps_set_processor_clock(val);

    // Reduce the voltage, if possible
    uamps_set_dvs(uamps_speed_table[val].dvs_val);
}

cyg_uint32
uamps_get_speed(void)
{
    return uamps_get_processor_clock();
}
/** This procedure is responsible for actually putting the
 * processor in sleep mode. When the processor wakes up,
 * it will eventually perform a return from this subroutine,
 * so execution resumes in the code that called this procedure.
 *
 * This procedure is defined in Suspend.S
 */

extern void uamps_suspend_processor(void);

// Structure for storing the state of the peripheral
// configuration registers
typedef struct {
    cyg_uint32 gplr;
    cyg_uint32 grer;
    cyg_uint32 gfer;
    cyg_uint32 gafr;
    cyg_uint32 icmr;
    cyg_uint32 iclr;
    cyg_uint32 iccr;
    cyg_uint32 rtsr;
    cyg_uint32 rttr;
    cyg_uint32 oscr;
    cyg_uint32 osmr0;
    cyg_uint32 osmr1;
    cyg_uint32 osmr2;
    cyg_uint32 osmr3;
    cyg_uint32 ower;
    cyg_uint32 ossr;
    cyg_uint32 oier;
    cyg_uint32 uart1_utcr0;
    cyg_uint32 uart1_utcr1;
    cyg_uint32 uart1_utcr2;
    cyg_uint32 uart1_utcr3;
    cyg_uint32 uart2_utcr0;
    cyg_uint32 uart2_utcr1;
    cyg_uint32 uart2_utcr2;
    cyg_uint32 uart2_utcr3;
    cyg_uint32 uart3_utcr0;
    cyg_uint32 uart3_utcr1;
    cyg_uint32 uart3_utcr2;
    cyg_uint32 uart3_utcr3;
} uamps_sys_registers_t;
static uamps_sys_registers_t uamps_system_regs;

/** Shut down the serial ports cleanly.
 * Must be run with interrupts disabled.
 */
void uamps_serial_suspend(void)
{
    while (*SA11X0_UART1_STATUS1 & SA11X0_UART_TX_BUSY);
    uamps_system_regs.uart1_utcr0 = *SA11X0_UART1_CONTROL0;
    uamps_system_regs.uart1_utcr1 = *SA11X0_UART1_CONTROL1;
    uamps_system_regs.uart1_utcr2 = *SA11X0_UART1_CONTROL2;
    uamps_system_regs.uart1_utcr3 = *SA11X0_UART1_CONTROL3;
    *SA11X0_UART1_CONTROL3 |=
        (0xFF ^ (SA11X0_UART_RX_ENABLED | SA11X0_UART_TX_ENABLED));

    while (*SA11X0_UART2_STATUS1 & SA11X0_UART_TX_BUSY);
    uamps_system_regs.uart2_utcr0 = *SA11X0_UART2_CONTROL0;
    uamps_system_regs.uart2_utcr1 = *SA11X0_UART2_CONTROL1;
    uamps_system_regs.uart2_utcr2 = *SA11X0_UART2_CONTROL2;
    uamps_system_regs.uart2_utcr3 = *SA11X0_UART2_CONTROL3;
    *SA11X0_UART2_CONTROL3 |=
        (0xFF ^ (SA11X0_UART_RX_ENABLED | SA11X0_UART_TX_ENABLED));

    while (*SA11X0_UART3_STATUS1 & SA11X0_UART_TX_BUSY);
    uamps_system_regs.uart3_utcr0 = *SA11X0_UART3_CONTROL0;
    uamps_system_regs.uart3_utcr1 = *SA11X0_UART3_CONTROL1;
    uamps_systemregs.uart3_utcr2 = *SA11X0_UART3_CONTROL2;
    uamps_system_regs.uart3_utcr3 = *SA11X0_UART3_CONTROL3;
    *SA11X0_UART3_CONTROL3 |=
        (0xFF ^ (SA11X0_UART_RX_ENABLED | SA11X0_UART_TX_ENABLED));
}

/** Re-enable the serial ports.
 */
void uamps_serial_resume(void)
{
    *SA11X0_UART1_CONTROL0 = uamps_system_regs.uart1_utcr0;
    *SA11X0_UART1_CONTROL1 = uamps_system_regs.uart1_utcr1;
    *SA11X0_UART1_CONTROL2 = uamps_system_regs.uart1_utcr2;
    *SA11X0_UART1_CONTROL3 = uamps_system_regs.uart1_utcr3;

    *SA11X0_UART2_CONTROL0 = uamps_system_regs.uart2_utcr0;
    *SA11X0_UART2_CONTROL1 = uamps_system_regs.uart2_utcr1;
    *SA11X0_UART2_CONTROL2 = uamps_system_regs.uart2_utcr2;
void uamps_sleep(void) {
    cyg_uint32 int_state, dcache, icache;

    HAL_DISABLE_INTERRUPTS(int_state);

    // GPIO registers
    uamps_system_regs.grer = *SA11X0_GPIO_RISING_EDGE_DETECT;
    uamps_system_regs.gfer = *SA11X0_GPIO_FALLING_EDGE_DETECT;
    uamps_system_regs.gafr = *SA11X0_GPIO_ALTERNATE_FUNCTION;

    // Interrupt registers
    uamps_system_regs.icmr = *SA11X0_ICMR;
    uamps_system_regs.iclr = *SA11X0_ICLR;
    uamps_system_regs.iccr = *SA11X0_ICCR;

    // RTC registers. The RTC runs even when the processor is asleep:
    // saving these registers might not be necessary.
    uamps_system_regs.rttsr = *SA11X0_RTTSR & 0x03; // Mask flag bits
    uamps_system_regs.rttr = *SA11X0_RTTR;

    // OS timer
    uamps_system_regs.oscr = *SA11X0_OSCR;
    uamps_system_regs.osmr0 = *SA11X0_OSMR0;
    uamps_system_regs.osmr1 = *SA11X0_OSMR1;
    uamps_system_regs.osmr2 = *SA11X0_OSMR2;
    uamps_system_regs.osmr3 = *SA11X0_OSMR3;
    uamps_system_regs.otcr = *SA11X0_OTCR;
    // Save flags for sorting out OS timer later
    uamps_system_regs.ossr = *SA11X0_OSSR;
    uamps_system_regs.oier = *SA11X0_OIER;
// Serial ports
uamps_serial_suspend();

// Do not stop 3.6864MHz osc.
// Let 32kHz osc. stabilize on reset.
*SA11X0_PWR_MGR_GENERAL_CONFIG = 0x00;
*SA11X0_PWR_MGR_WAKEUP_ENABLE = 0x80000000; // RTC alarm enabled

// Set GPIO sleep states
uamps_system_regs.gplr = *SA11X0_GPIO_PIN_LEVEL;
*SA11X0_PWR_MGR_GPIO_SLEEP_STATE = *SA11X0_GPIO_PIN_LEVEL;
// Ensure LEDs are off
*SA11X0_PWR_MGR_GPIO_SLEEP_STATE
    |= UAMPS_USB_ATTACH | UAMPS_RED_LED | UAMPS_GREEN_LED;
// Put 3.3V power supply in standby mode
*SA11X0_PWR_MGR_GPIO_SLEEP_STATE &= 0xFFFFFFFF ^ UAMPS_V3_STBY;

// Disable caches
HAL_DCACHE_IS_ENABLED(dcache);
if (dcache) {
    HAL_DCACHE_SYNC();
    HAL_DCACHE_DISABLE();
    HAL_DCACHE_SYNC();
    HAL_DCACHE_INVALIDATE_ALL();
}
HAL_ICACHE_IS_ENABLED(icache);
if (icache) {
    HAL_ICACHE_DISABLE();
    HAL_ICACHE_INVALIDATE_ALL();
}

// This procedure will actually put the processor to sleep
uamps_suspend_processor();

// Begin the process of restoring the peripherals and caches.
if (dcache) HAL_DCACHE_ENABLE();
if (icache) HAL_ICACHE_ENABLE();
uamps_serial_resume();

// GPIO registers
*SA11X0_GPIO_PIN_OUTPUT_SET = uamps_system_regs.gplr;
*SA11X0_GPIO_PIN_OUTPUT_CLEAR = 0xFFFFFFFF ^ uamps_system_regs.gplr;
*SA11X0_GPIO_RISING_EDGE_DETECT = uamps_system_regs.grer;
*SA11X0_GPIO_FALLING_EDGE_DETECT = uamps_system_regs.gfer;
*SA11X0_GPIO_ALTERNATE_FUNCTION = uamps_system_regs.gafr;

// Interrupt registers
*SA11X0_ICMR = uamps_system_regs.icmr;
*SA11X0_ICLR = uamps_system_regs.iclr;
*SA11X0_ICCR = uamps_system_regs.iccr;

// RTC registers.
*SA11X0_RTSR = uamps_system_regs.rtsr;
*SA11X0_RTTR = uamps_system_regs.rttr;

// OS timer
*SA11X0_OSCR = uamps_system_regs.oscr;
*SA11X0_OSMR0 = uamps_system_regs.osmr0;
*SA11X0_OSMR1 = uamps_system_regs.osmr1;
*SA11X0_OSMR2 = uamps_system_regs.osmr2;
*SA11X0_OSMR3 = uamps_system_regs.osmr3;
*SA11X0_OWER = uamps_system_regs.ower;
*SA11X0_OSSR = 0xFFFFFFFF;
*SA11X0_OIER = uamps_system_regs.oier;

HAL_RESTORE_INTERRUPTS(int_state);
}

D.3 Saving and Restoring Processor Registers for Sleep Mode

File: packages/hal/arm/sa11x0/uamps/current/src/Suspend.S

The following assembly consists of two components which together make up the uamps_suspend_processor procedure. The first component performs the task of saving the processors registers and entering sleep mode. The second component is branched to when the processor is awakened from sleep. This component restores the processor’s registers, enables the MMU, and returns to whatever code called uamps_suspend_processor.

#include <cyg/hal/hal_sa11x0.h>
// Macros for converting between physical and virtual addresses.
#define PHYS_TO_VIRT(x) ((x)-0x50000000)
#define VIRT_TO_PHYS(x) ((x)+0x50000000)

// CPSR settings for each of the ARM processors
// modes
#define ARM_USR_MODE 0x10
#define ARM_FIQ_MODE 0x11
#define ARM_IRQ_MODE 0x12
#define ARM_SVC_MODE 0x13
#define ARM_ABT_MODE 0x15
#define ARM_UND_MODE 0x1B
#define ARM_SYS_MODE 0x1F
#define ARM_MODE_MASK 0x1F

// Storage area for CPU registers during sleep

    .section "_.fixed_vectors"
sleep_saved_regs:

// uamps_suspend_processor procedure
// =================================
// The following code saves all processor registers and
// enters sleep mode.

    .text

    .global uamps_suspend_processor
    .align

uamps_suspend_processor:

    // Save (most) supervisor mode registers on stack
    stmfd  sp!, {r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, lr}

    ldr r0, =sleep_saved_regs

    // Save abort mode registers
    mrs r1, cpsr
    bic r1, r1, #ARM_MODE_MASK
    orr r1, r1, #ARM_ABT_MODE
    msr cpsr, r1
str r13, [r0], #4 // Save ABT_R13 (00)
str r14, [r0], #4 // Save ABT_R14 (04)
mrs r1, spsr  // Save ABT_SPSR (08)
str r1, [r0], #4

// Save undefined mode registers
mrs r1, cpsr
bic r1, r1, #ARM_MODE_MASK
orr r1, r1, #ARM_UND_MODE
msr cpsr, r1
str r13, [r0], #4 // Save UND_R13 (0C)
str r14, [r0], #4 // Save UND_R14 (10)
mrs r1, spsr  // Save UND_SPSR (14)
str r1, [r0], #4

// Save interrupt mode registers
mrs r1, cpsr
bic r1, r1, #ARM_MODE_MASK
orr r1, r1, #ARM_IRQ_MODE
msr cpsr, r1
str r13, [r0], #4 // Save IRQ_R13 (18)
str r14, [r0], #4 // Save IRQ_R14 (1C)
mrs r1, spsr  // Save IRQ_SPSR (20)
str r1, [r0], #4

// Save fast interrupt mode registers
mrs r1, cpsr
bic r1, r1, #ARM_MODE_MASK
orr r1, r1, #ARM_FIQ_MODE
msr cpsr, r1
str r8, [r0], #4 // Save FIQ_R8  (24)
str r9, [r0], #4 // Save FIQ_R9  (28)
str r10, [r0], #4 // Save FIQ_R10 (2C)
str r11, [r0], #4 // Save FIQ_R11 (30)
str r12, [r0], #4 // Save FIQ_R12 (34)
str r13, [r0], #4 // Save FIQ_R13 (38)
str r14, [r0], #4 // Save FIQ_R14 (3C)
mrs r1, spsr  // Save FIQ_SPSR (40)
str r1, [r0], #4

// Return to supervisor mode and save SPSR and SP
mrs r1, cpsr

bic r1, r1, #ARM_MODE_MASK
orr r1, r1, #ARM_SVC_MODE
msr cpsr, r1

mrs r1, spsr // Save SVC_SPSR (44)
str r1, [r0], #4
str sp, [r0], #4 // Save SVC_SP (R13) (48)

// Save coprocessor registers
// C1 has MMU enable, other CPU configuration controls
mrc p15, 0, r1, c1, c0, 0 // Save C1 (48)
str r1, [r0], #4

// C2 has MMU table base
mrc p15, 0, r1, c2, c0, 0 // Save C2 (4C)
str r1, [r0], #4

// C3 has domain access control bits
mrc p15, 0, r1, c3, c0, 0 // Save C3 (50)
str r1, [r0], #4

// C5 has fault status
mrc p15, 0, r1, c5, c0, 0 // Save C5 (54)
str r1, [r0], #4

// C6 has last fault address
mrc p15, 0, r1, c6, c0, 0 // Save C6 (58)
str r1, [r0], #4

// C13 has process virtual ID
mrc p15, 0, r1, c13, c0, 0 // Save C13 (5C)
str r1, [r0], #4

// Disable clock switching
mcr p15, 0, r1, c15, c2, 2

// Put resume address in power manager scratchpad
ldr r0, =uamps_resume_processor
ldr r1, =0x4FFFFFFF
cmp r0, r1
b 1f
sub r0, r0, #0x50000000
b 2f
1:
add r0, r0, #0x08000000
2:
    ldr r1, =SA11X0_PWR_MGR_SCRATCHPAD
    str r0, [r1]

// Set force sleep bit in power manager control register
ldr    r0, =SA11X0_PWR_MGR_CONTROL
mov    r1, #0x1
str    r1, [r0]

// Loop forever (or at least until the processor goes to sleep).
1:
    b  1b

// uamps_resume_processor procedure
// ================================
// // This is the second half of the uamps_suspend_processor
// procedure. When the processor awaken from sleep, the startup
// code will detect that a return from sleep is in progress, and
// will branch here. This code restores the state of the
// processor’s registers, enables the MMU, and performs a
// return-from-subroutine, returning control to whatever code
// originally called uamps_suspend_processor.

uamps_resume_processor:

    // Release peripheral hold (set by RESET)
    ldr    r1, =SA11X0_PWR_MGR_SLEEP_STATUS
    ldr    r2, =SA11X0_PERIPHERAL_CONTROL_HOLD
    str    r2, [r1]

    ldr    r0, =sleep_saved_regs
    add    r0, r0, #0x08000000

    // Restore abort mode registers
    mrs    r1, cpsr
    bic    r1, r1, #ARM_MODE_MASK
    orr    r1, r1, #ARM_ABT_MODE
    msr    cpsr, r1

    ldr    r13, [r0], #4 // Restore ABT_R13 (00)
    ldr    r14, [r0], #4 // Restore ABT_R14 (04)
    ldr    r1, [r0], #4
    msr    spsr, r1     // Restore ABT_SPSR (08)

    // Restore undefined mode registers
    mrs    r1, cpsr
    bic    r1, r1, #ARM_MODE_MASK
    orr    r1, r1, #ARM_UND_MODE
    msr    cpsr, r1
ldr r13, [r0], #4 // Restore UND_R13 (0C)
ldr r14, [r0], #4 // Restore UND_R14 (10)
ldr r1, [r0], #4
msr spsr, r1 // Restore UND_SPSR (14)

// Restore interrupt mode registers
mrs r1, cpsr
bic r1, r1, #ARM_MODE_MASK
orr r1, r1, #ARM_IRQ_MODE
msr cpsr, r1

ldr r13, [r0], #4 // Restore IRQ_R13 (18)
ldr r14, [r0], #4 // Restore IRQ_R14 (1C)
ldr r1, [r0], #4
msr spsr, r1 // Restore IRQ_SPSR (20)

// Restore fast interrupt mode registers
mrs r1, cpsr
bic r1, r1, #ARM_MODE_MASK
orr r1, r1, #ARM_FIQ_MODE
msr cpsr, r1

ldr r8, [r0], #4 // Restore FIQ_R8 (24)
ldr r9, [r0], #4 // Restore FIQ_R9 (28)
ldr r10, [r0], #4 // Restore FIQ_R10 (2C)
ldr r11, [r0], #4 // Restore FIQ_R11 (30)
ldr r12, [r0], #4 // Restore FIQ_R12 (34)
ldr r13, [r0], #4 // Restore FIQ_R13 (38)
ldr r14, [r0], #4 // Restore FIQ_R14 (3C)
ldr r1, [r0], #4
msr spsr, r1 // Restore FIQ_SPSR (40)

// Return to supervisor mode
mrs r1, cpsr
bic r1, r1, #ARM_MODE_MASK
orr r1, r1, #ARM_SVC_MODE
msr cpsr, r1

ldr r1, [r0], #4
msr spsr, r1 // Restore SVC_SPSR (44)
ldr sp, [r0], #4 // Restore SVC_SP (48)

// Restore most coprocessor registers (don’t turn on MMU yet)
ldr r2, [r0], #4
ldr  r1, [r0], #4
mcr   p15, 0, r1, c2, c0, 0
ldr  r1, [r0], #4
mcr   p15, 0, r1, c3, c0, 0
ldr  r1, [r0], #4
mcr   p15, 0, r1, c5, c0, 0
ldr  r1, [r0], #4
mcr   p15, 0, r1, c6, c0, 0
ldr  r1, [r0], #4
mcr   p15, 0, r1, c13, c0, 0

// Enable MMU
ldr  r3, =resume_mmu_on
b    resume_mmu

.align 5
resume_mmu:
  mcr   p15, 0, r2, c1, c0, 0
  mov   pc, r3
  nop
  nop
  nop

.resume_mmu_on:

// Return to caller
ldmfd  sp!, {r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, lr}
mov   pc, lr
Appendix E

StrongARM Voltage Test Program

The following code is for an eCos application which tests the range of voltages required by the StrongARM SA-1110 processor at each operating frequency. The code consists of a series of procedures—which each perform some test of the correct functionality of the processor—and a main loop. The main loop sets the processor voltage, then runs each of the test procedures. If all tests pass, the voltage is reduced and the tests are run again.

```c
#include <cyg/infra/cyg_type.h>
#include <cyg/hal/hal_cache.h>
#include <cyg/infra/diag.h>
#include <cyg/hal/uamps.h>
#include <cyg/hal/hal_sa11x0.h>

#include <uapi/power.h>
#include <uapi/timer.h>
#include <uapi/uamps.h>

extern void uamps_set_dvs(cyg_uint32 val);
extern cyg_uint32 uamps_get_dvs_voltage(void);
extern cyg_uint32 uamps_get_processor_clock_frequency(void);
```
#define SET_LENGTH (8*1024)

cyg_uint32 temp[SET_LENGTH];
cyg_uint32 *uctemp
    = ((cyg_uint32 *) (((unsigned int) temp) + 0x08000000));

/** Write and read all zeros from a block of memory */
cyg_bool
test_memory_zeros(cyg_uint32 *addr, cyg_uint32 length)
{
    cyg_uint32 i;
    for (i=0; i<length; i++) { addr[i] = 0; }
    for (i=0; i<length; i++) {
        cyg_uint32 d = addr[i];
        if (d != 0) {
            diag_printf("FAILED: read 0x%08x at 0x%08x.\n", d, ((unsigned int) (i+addr)));
            return 0;
        }
    }
    return 1;
}

/** Write and read all ones from a block of memory */
cyg_bool
test_memory_ones(cyg_uint32 *addr, cyg_uint32 length)
{

cyg_uint32 i;
for (i=0; i<length; i++) { addr[i] = 0xFFFFFFFF; }
for (i=0; i<length; i++) {
    cyg_uint32 d = addr[i];
    if (d != 0xFFFFFFFF) {
        diag_printf("FAILED: read 0x%08x at 0x%08x.\n", d, ((unsigned int) (i+addr)));
        return 0;
    }
}
return 1;

/** Write and read a checkerboard pattern from a block of memory */

cyg_bool

test_memory_checkerboard(cyg_uint32 *addr, cyg_uint32 length)
{
    cyg_uint32 i, d;
    for (i=0; i<length; i++) {
        addr[i] = i%2 ? 0xAAAAAAAA : 0x55555555;
    }
    for (i=0; i<length; i++) {
        d = addr[i];
        if (d != (i%2 ? 0xAAAAAAAA : 0x55555555)) {
            diag_printf("FAILED: read 0x%08x at 0x%08x.\n", d, ((unsigned int) (i+addr)));
            return 0;
        }
    }
}

135
return 1;
}

/** Test the process of flushing the data cache. This procedure
* first writes a set of sequential numbers to a block of cached
* memory pointer to by *addr. It then writes zeros directly to
* the physical memory correpsong to the same locations as
* *addr. This is accomplished by having *ucaddr point to an
* uncached portion of the virtual memory space which is mapped
* to the same physical memory locations as *addr. The data
* cache is flushed and disabled, and the contents of *addr
* and *ucaddr are read back. They should both contain the data
* that was originally written to *addr.
*/

bool
test_dcache_flush(cyg_uint32 *caddr,
                 cyg_uint32 *ucaddr,
                 cyg_uint32 length)
{
    cyg_uint32 i, tc, tuc, dcache;
    // Clear physical memory
    for (i=0; i<length; i++) {
        ucaddr[i] = 0;
    }
    // Put count in cache
    for (i=0; i<length; i++) {
        caddr[i] = i;
    }
    // Flush cache
    HAL_DCACHE_IS_ENABLED(dcache);
if (dcache) {
    HAL_DCACHE_SYNC();
    HAL_DCACHE_DISABLE();
    HAL_DCACHE_SYNC();
    HAL_DCACHE_INVALIDATE_ALL();
}

// Read physical memory
for (i=0; i<length; i++) {
    tuc = ucaddr[i];
    tc = caddr[i];
    if (((tuc != tc) || (tuc != i))) {
        diag_printf("FAILED: expected 0x%08X, "
            "got 0x%08X at 0x%08X (cached) "
            "and 0x%08X at 0x%08X (uncached).\n",
            i, tc, ((unsigned int) (caddr+i)),
            tuc, ((unsigned int) (ucaddr+i)));
        return 0;
    }
}

if (dcache) HAL_DCACHE_ENABLE();
return 1;

/** Test the adder by adding a long series of numbers and
 * comparing with a precomputed result.
 */
cyg_bool
test_add(void)
{
    cyg_uint32 a, b, i, t;
a = 1;
b = 0;
for (i=0; i<10000; i++) {
    t = a+b;
    b = a;
    a = t;
}
t = 0x7F1BE43D;
if (a != t) {
    diag_printf("got 0x%08x, expected 0x%08x.\n", a, t);
    return 0;
}
return 1;
}

/** Test the multiplier by multiplying a long series of
 * numbers and comparing with a precomputed result.
 */
cyg_bool
test_multiply(void)
{
    cyg_uint32 a, i, t;
    a = 1;
    for (i=0; i<10000; i++) {
        a = a*i+1;
    }
t = 0x9E4301CC;
    if (a != t) {
        diag_printf("FAILED: got 0x%08x, expected 0x%08x.\n", a, t);
        return 0;
    }
    return 1;
}
int
main(void)
{
    cyg_uint32 i, f, v, e, int_state;

    HAL_DISABLE_INTERRUPTS(int_state);

    v = uamps_get_processor_clock_frequency();
    diag_printf("Clock frequency is %d.%03dMhz.\n", v/1000000, (v/1000)%1000);

    for (i=3; i<31; i++) {
        if (i==15) i++;
        uamps_set_dvs(i);

        v = uamps_get_dvs_voltage();
        diag_printf("Voltage is %d.%03dV\n", v/1000, v%1000);
        diag_printf("------------------------------------------\n");
        f = *SA11X0_OSCR;
        while (*SA11X0_OSCR < f+368640);
        diag_printf("    Write zeros.....................\n");
        f = test_memory_zeros(temp, 1024);
        if (f) { diag_printf("passed\n"); } else { e++; }
        
        diag_printf("    Write ones......................\n");
        f = test_memory_ones(temp, 1024);

    }

    return 1;
}
if (f) { diag_printf("passed\n"); } else { e++; }

diag_printf(" Write checkerboard..............");
f = test_memory_checkerboard(temp, 1024);
if (f) { diag_printf("passed\n"); } else { e++; }

diag_printf(" (Long) write zeros..............");
f = test_memory_zeros(temp, SET_LENGTH);
if (f) { diag_printf("passed\n"); } else { e++; }

diag_printf(" (Long) write ones..............");
f = test_memory_ones(temp, SET_LENGTH);
if (f) { diag_printf("passed\n"); } else { e++; }

diag_printf(" (Long) write checkerboard........");
f = test_memory_checkerboard(temp, SET_LENGTH);
if (f) { diag_printf("passed\n"); } else { e++; }

diag_printf(" Flush cache.....................");
f = test_dcache_flush(temp, uctemp, 1024);
if (f) { diag_printf("passed\n"); } else { e++; }

HAL_DCACHE_SYNC();
HAL_DCACHE_DISABLE();
HAL_DCACHE_SYNC();
HAL_DCACHE_INVALIDATE_ALL();

diag_printf(" Uncached write zeros............");
f = test_memory_zeros(temp, SET_LENGTH);
if (f) { diag_printf("passed\n"); } else { e++; }

140
diag_printf(" Uncached write ones.............");
f = test_memory_ones(temp, SET_LENGTH);
if (f) { diag_printf("passed\n"); } else { e++; }

diag_printf(" Uncached write checkerboard.....");
f = test_memory_checkerboard(temp, SET_LENGTH);
if (f) { diag_printf("passed\n"); } else { e++; }

HAL_DCACHE_ENABLE();

diag_printf(" Addition.........................");
f = test_add();
if (f) { diag_printf("passed\n"); } else { e++; }

diag_printf(" Multiplication.....................");
f = test_multiply();
if (f) { diag_printf("passed\n"); } else { e++; }

while (e != 0);
}

while (1);
Bibliography


