MODULARIZED PCA PUMP DESIGN FOR AN ICE-INFORMED MEDICAL DEVICE COORDINATION FRAMEWORK

by

SHIWEI LUAN

B.S., Tianjin University, China, 2008
M.S., Tianjin University, China, 2010

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Electrical & Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2015

Approved by:

Major Professor
Steven Warren

Abstract

Medical device interoperability and re-configurability continue to be important areas of research toward the realization of verifiable medical systems that can be rapidly assembled to meet the needs of specific patients. This thesis addresses the modularized design of a patient-controlled analgesia (PCA) pump to be used within the context of the Medical Device Coordination Framework (MDCF), an open source framework under development by Kansas State University and the University of Pennsylvania that is informed by the Integrated Clinical Environment (ICE) specification managed by the MD PnP program and its collaborators.

The thesis illustrates how to set up the MDCF development environment with Eclipse so that a developer can create software for both a remote MDCF console and a local PCA pump, where ICE channels are used for message transmission. Software development on the MDCF console side includes the development of apps that communicate with a local PCA pump through ICE channels and (b) the development of a GUI that can be launched from an MDCF console to configure, control, and monitor a PCA pump. Software development on the PCA pump side includes the creation of (a) ICE channels that can communicate with an MDCF console and (b) multiple threads for corresponding UART ports that support a modularized design.

Several hardware modules were implemented to demonstrate the modularized design approach: an alarm module, a patient button module, a pump module, and a control panel module. These modules employ BeagleBone, Arduino, and MSP430 boards. Status information is displayed on an MDCF console GUI, a PCA pump GUI, and a local LCD screen. An enhanced PCA pump or general medical sub-system with more modules can be developed using a similar method by connecting individual modules to UART ports and then creating the corresponding threads to support device-console communication.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I wish to acknowledge the people who helped me complete this work toward a Master of Science degree in Electrical Engineering from Kansas State University. First and foremost, I thank my advisor, Dr. Steven Warren, who helped with my thesis and graduation work. I can always get helpful advice from him for both research and life. I particularly thank Dr. John Hatcliff, KSU Computing & Information Sciences, who led the project that provided my funding and accepted me as a group member among many creative and passionate engineers. Thanks also to Yu Jin Kim, who helped me solve software problems during the design work, and to Dr. Don Gruenbacher, KSU ECE department head, who helped with my final graduation. I am also grateful to Mr. Steve Booth, an ECE staff technician who assisted me with my work on hardware design. This research was funded by the National Science Foundation (NSF) under grant CNS–1239543. Any opinions, findings and conclusions, or recommendations in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

Chapter 1 - Introduction

1.1 Integrated Clinical Environment

The Integrated Clinical Environment (ICE) standardization efforts are based on early work by the MD PnP program [1] and their collaboration partners. ICE efforts have been managed since 2006 through ASTM International, Subcommittee F29.21, chaired by Dr. Julian Goldman, Center for Integration of Medicine & Innovative Technology (CIMIT). The motivation of ICE is to solve the problem that most current medical devices in a patient's room are designed as stand-alone devices. Each device is unaware of other medical devices that may be connected in the same system, and they do not share information with each other, even though most of them are designed with data ports, such as Ethernet or RS-232, that allow them to interact with a host system. For example, a patient-controlled analgesia (PCA) pump is used to deliver morphine to a patient at their request, but the pump does not know if other devices (e.g., electrocardiographs or pulse oximeters) are connected to the patient that could monitor physical parameters that could be affected by the morphine delivered by the PCA pump. Conversely, physiological monitors may provide information regarding a patient's status, but they do not understand that these parameters are affected by morphine administered through a PCA pump. So, each medical device produces its own data, but these data are not shared or combined together in context, meaning that the clinician lacks information that should otherwise be available when making care decisions for a patient.

Again, the idea behind ICE is to provide a platform that can aggregate data and coordinate medical devices to work as a system, instead of working independently as stand-alone devices. In an ICE-compliant system, information from all medical devices is automatically combined and processed by the system, allowing it to produce more accurate and useful information or alarms

[2, 3]. The whole process can be viewed as a closed loop control to improve safety, with the inputs from medical devices fed into the ICE-compliant system, and the output from the system fed back to the medical devices [4, 5].

A sensible example of an ICE system with a PCA pump works as shown in Figure 1-1 [6]. In this application, a PCA pump is used to deliver medication, such as morphine, to the patient according to a prescribed bolus rate at the patient's request. A respiratory rate monitor and a pulse oximeter are also connected to track the patient's physical parameters, since these parameters could be affected by the morphine from the PCA pump. If all of the physical parameters are in a safe range, the ICE system will issue a "Ticket" so that the PCA pump can be used by the patient to deliver medication. If the PCA pump delivers too much morphine, the respiratory rate and heart rate could be affected, and if these parameters are out of limits, the ICE system will stop the PCA pump and send an alarm. This architecture reduces a clinician's work and improves safety.

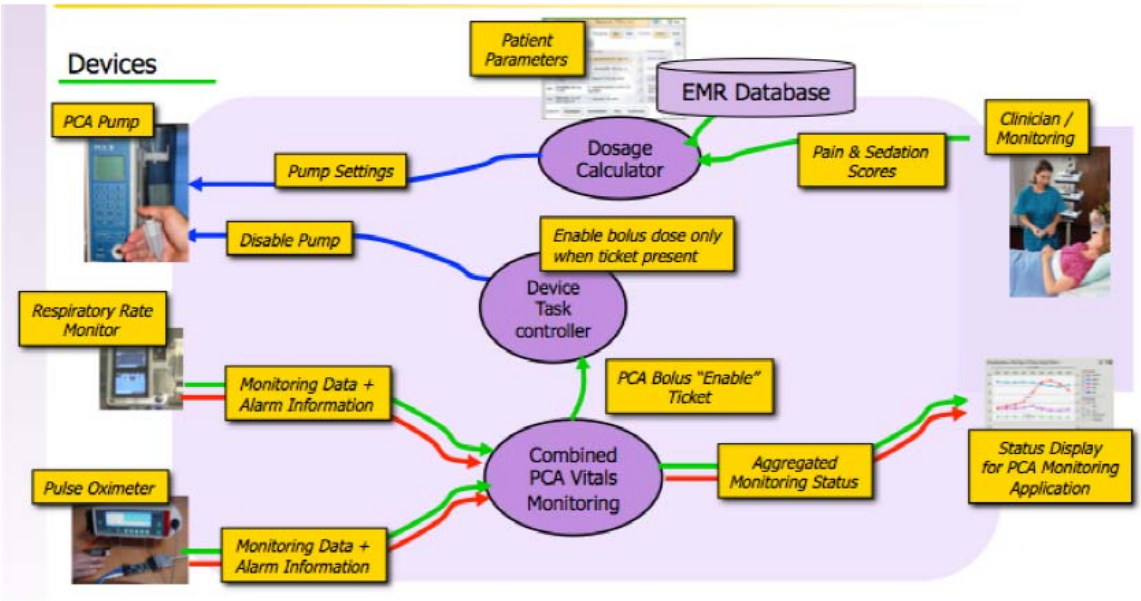

Figure 1-1. ICE system with multiple medical devices.

1.2 Medical Device Coordination Framework

The Medical Device Coordination Framework (MDCF), an implementation of ICE as an open source platform, was developed by the SAn-ToS Laboratory at Kansas State University and the University of Pennsylvania. The MDCF was developed to bring together academic researchers, industry vendors, and government regulators, and to demonstrate how ICE-compatible medical devices can be integrated and coordinated through a network to form a uniform system [7, 8]. The goals of the MDCF effort follow:

- Offer an open source infrastructure.
- Meet performance requirements of realistic clinical scenarios.
- Provide middleware with reliability, real-time operation, and security.
- Provide an effective app programming model and development environment with integrated verification/validation support and construction of regulatory artifacts.
- Support evaluation of device interfacing concepts.
- Illustrate how to support real and mock devices.
- Illustrate envisioned regulatory oversight and third party certification.

Figure 1-2 depicts a representative MDCF console, where multiple medical devices, such as a respiratory rate monitor, a pulse oximeter, and a PCA pump are represented by apps in the left column. All of the medical devices are connected through a virtual data bus, and all apps are coordinated by a virtual Coordination Supervisor. The clinician can launch the desired medical device when it is available.

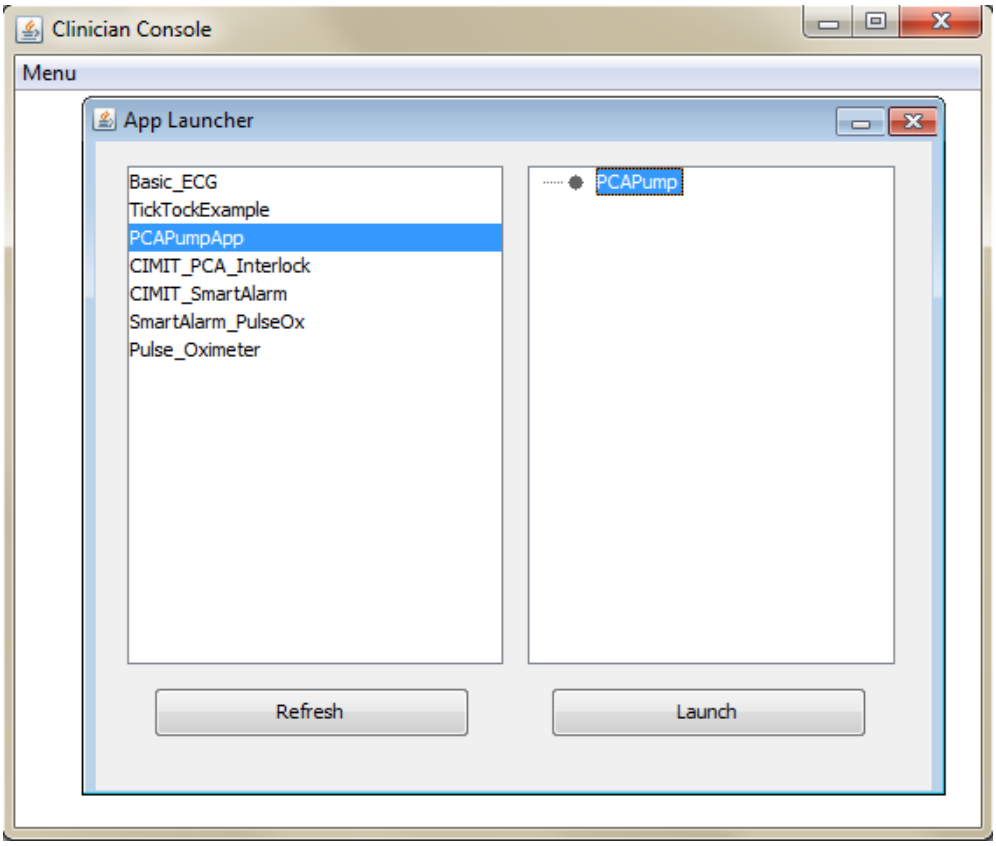

Figure 1-2. Representative MDCF console with multiple medical devices.

1.3 Programmer's Development Environment

The Programmer's Development Environment (PDE) is a tool provided in the MDCF toolset to develop component-based apps for ICE-compliant medical devices [9]. Two main software elements are required to implement a functional medical device:

ICE Device Interface that Runs on a Medical Device: The medical device manufacturer or another third party could implement code either directly on the device or on an external unit (such as a dongle) that would convert the native signals of the device to ICE protocols and data formats so that the device would be ICE Compatible.

ICE App that Runs within an MDCF console: This app can be launched in an MDCF console so that the clinician could launch, validate or configure a medical device that is attached to the ICE system.

Figure 1-3 illustrates an example workspace within the PDE. The PDE defines three component roles:

1. **Device:** Represents the actual medical device ("Medical_Device" in Figure 1-3).
2. **Logic:** Method to process data on the MDCF side ("APP_Logic" in Figure 1-3).
3. **AppPanel:** Specification to display the data and information on an MDCF console GUI (APP_in_MDCFConsole" in Figure 1-3).

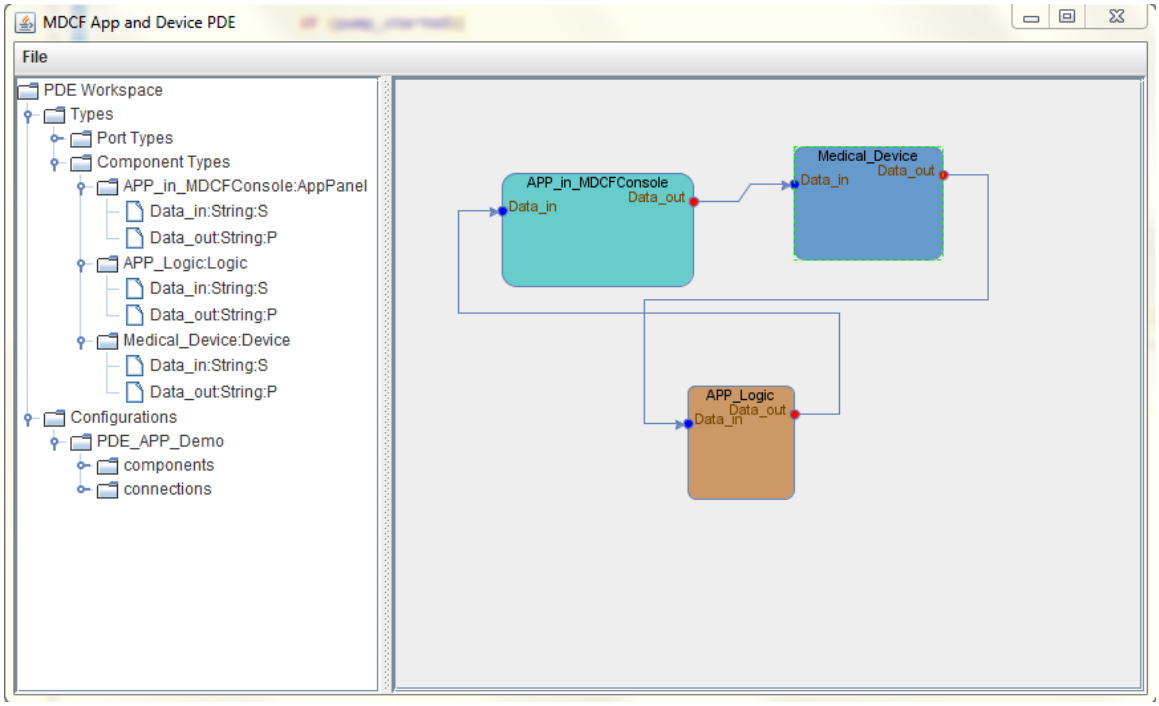

Figure 1-3. Example workspace within the programmer's development environment.

The code that implements the Device functionality will run on the medical-device side, while the code that implements the Logic and AppPanel roles will run on the MDCF (PC) side. The developer first defines the ports for each component and creates channels between these components. The PDE is then used to generate the Java code skeleton and the developer needs to finally fulfill the code for each role to implement the desired behavior. More PDE development details will be discussed in Chapter 5.

1.4 PCA Pump

A patient-controlled analgesia (PCA) pump is a medical device that contains a pain medication reservoir connected to a patient's intravenous line [10]. This type of pump is used to reduce patient pain, especially after surgery or as a result of cancer [11]. A PCA pump is set to deliver a small, constant flow of pain medication, such as morphine, Dilaudid, Demerol, or Fentanyl, into the blood at a prescribed dosage rate and interval. When the patient feels pain, he/she can, within limits, press a button to receive additional doses of pain medication besides the prescribed, continuously running amount. This avoids the need for a care provider to frequently administer additional medication; the patient can control the medication as needed based on their pain levels.

Modern PCA pumps are designed with many safety features and work within in a broader system of bedside medical devices [12-14]. The main parts of a PCA pump include actuators, sensors, and controllers, where a touch LCD, bar scanners, etc. are employed to improve operability.

A **controller** is the main processing unit that analyzes the information from sensors/operators and responds to the operations of a patient, clinician, or remote ICE operator.

An **actuator** is usually in the form of a pump that pumps medication from a drug reservoir through a tube and needle, then into the patient. There are different types of mechanical designs for actuator operation, but the most important factors are that the pump should deliver the medication at an accurate rate and that this dosage rate parameter should be verified by sensors and managed by a controller.

Sensors are used to measure various medication delivery parameters as well as the working status of a PCA pump. Several important parameters must be monitored:

- Pump rate: The prescribed constant flow rate and the requested additional bolus rate must be accurate. Back flow should also be detected and monitored when the pump is working.
- In-tube air bubble or occlusion: Air in the tube or an occlusion can be dangerous to the patient, so the PCA pump must stop when either is detected.
- Drug reservoir level: The medication level in the drug reservoir should be monitored, and an alarm should be triggered if the level is too low.

Some other parameters, such as hardware working status, should also be monitored. Details of these parameters are defined by requirement files and will be discussed in the following chapter.

Chapter 2 - PCA Pump Design Requirements

The ICE-informed PCA pump addressed in this thesis is designed following the "Open Patient-Controlled Analgesia Infusion Pump System Requirements" report by Brian Larson and John Hatcliff, a document which specifies the requirements for an ICE-compliant PCA pump [13]. The file defines the basic PCA pump structure, functionality, and safety and security requirements. An AADL model for an ICE-compliant PCA pump is also included in this requirement file, as are operation cases considering safety.

2.1 ICE-Compliant PCA Pump Structure

As described in section 1.4, a general PCA pump is used to infuse pain medication at a prescribed dose rate and upon the patient's request. Pain medication, prescribed by a licensed physician, is dispensed by the hospital's pharmacy. The drug is then placed into a vial labeled with the name of the drug, its concentration, the prescription, and the intended patient.

Figure 2-1 [15] is a typical block structure for a standalone PCA pump. It contains all of the basic functional parts (e.g., the pump, drug reservoir, patient request button, and control panel) as well as several featured parts such as a scanner to scan prescriptions, medication labels, clinician badges, etc.

Figure 2-2 [15] depicts the block structure of an ICE-compliant PCA pump defined by the requirements file. The main difference between these two PCA pumps is that the ICE-compliant PCA pump is designed with an ICE Interface so that it can be connected to a virtual ICE bus and integrated into an ICE system to work together with other medical devices. A remote ICE app will be developed to monitor the PCA pump status and to remotely control or set the parameters of the PCA pump.

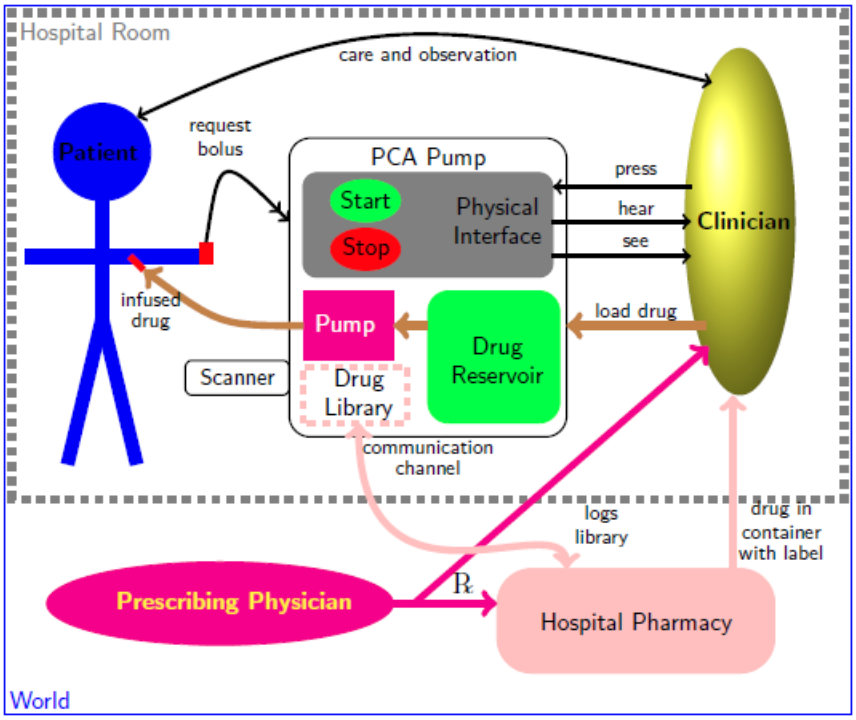

Figure 2-1. Typical PCA pump. Reprinted with permission from John Hatcliff, KSU Computing & Information Sciences.

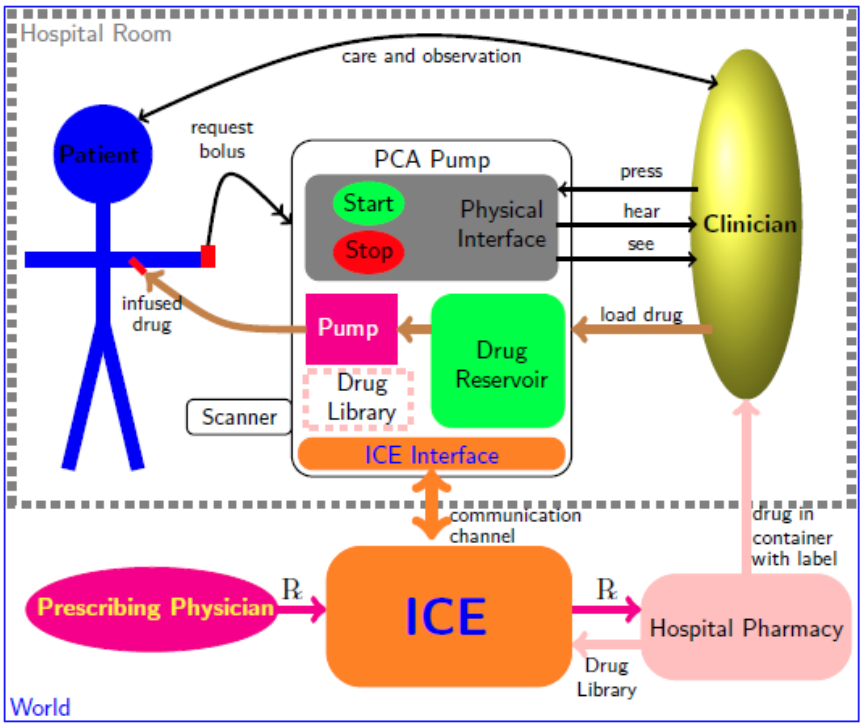

Figure 2-2. PCA pump with an ICE Interface. Reprinted with permission from John Hatcliff, KSU Computing & Information Sciences.

2.2 PCA Pump Requirements File

The "Open Patient-Controlled Analgesia Infusion Pump System Requirements" document lists detailed requirements for an ICE-compliant PCA pump. These requirements are provided as a public-domain example, because real requirements are highly confidential to medical device manufacturers, often using proprietary clinical data.

Overall, an ICE-compliant PCA pump implements the functionality of a stand-alone PCA pump, where communication and control are handled by the ICE system. This allows clinicians to remotely control and monitor the operation of the pump, and ICE apps can coordinate its operation with other ICE devices. These are the high level goals of the PCA pump:

- The pump should safely reduce pain.
- The patient should receive enough medication to reduce his pain.
- The patient should not receive so much medication that the patient becomes unaware or is harmed.
- Clinicians should be notified upon occurrence of hazardous conditions, unless alarms have been purposefully inactivated.
- The PCA pump should detect the smallest possible air-in-line embolism (bubble).
- The PCA pump should infuse safely when failures occur or hazards are detected.
- The patient should receive the drug as prescribed by their physician, and it should be administered by appropriate clinicians.
- A patient's health information should be available to those caring for the patient, and only those individuals.

As noted in this requirements file, the PCA pump should consist of several main functional elements:

Bolus Request Button: A patient presses a button to request a drug bolus in addition to a constant basal rate. The bolus request button is connected by a cable to the PCA pump and may have a clip to attach to a patient's bedding.

Delivery Tube and Needle: The drug is conveyed from the pump to a needle to be infused into the patient. The needle is placed into a vein, usually in an arm or hand.

Physical Pump: A physical pump forces the drug into the delivery tube at a specified rate. It also measures pressure and flow, and detects occlusions and air-in-line embolisms (bubbles).

Drug Reservoir: A drug reservoir holds the prescribed drug in a vial for extraction by the physical pump. Because the drugs administered are narcotic and may be abused, the drug reservoir must be tamper resistant. The drug reservoir also has an electronic means, such as optical code, to read the prescription from the vial labeled by the hospital's pharmacy.

Control Panel: A control panel allows the pump to be started and stopped. It allows a clinician to command delivery of a bolus. It also allows a clinician to specify the duration over which a prescribed volume-to-be-infused is delivered. Pump status and alarms are displayed or made audible by the physical interface.

Drug Library: A drug library containing information about drugs that may be used by the pump is stored in non-volatile memory. The drug library is determined by the hospital pharmacy.

Scanner: A scanner allows the entry of patient, clinician, and prescription information, automatically reducing both the work needed to operate the pump and possible harm to the patient through manual entry errors. The scanner may be optical or use radio-frequency identification (RFID).

ICE Interface: An ICE interface uses a communication channel to connect to the ICE system.

Safety Architecture: The system uses a safety architecture that separates normal operation from error and anomaly detection and response.

Security: Authentication of prescriptions, patients, and clinicians reduces the risk of malicious or accidental harm.

Chapter 3 - Modularized PCA Pump Design

PCA pumps on the market today are designed as stand-alone entities that are FDA approved as entire units and do not offer component-level upgradeability that would make them easily applicable to other usage environments. PCA pump functionality can be divided into several modules: central control unit, display, alarm, drug reservoir, power/battery, patient button, and scanner, etc. Ideally, each of these sub-systems could be individually upgraded to improve or alter the performance of the overall system, implying that medical device companies could then compete at the component level rather than the system level.

Another advantage of the reconfigurable modularized design is that if one module fails to work, one could easily unplug the damaged module and plug in another module with the same functionality. The base module would then automatically detect that a new module was plugged into the data bus and start to read information from that module, validating and configuring this module as needed to make it a part of the whole device. This approach would make it easier to maintain and upgrade the device module by module, either with hardware or firmware, as long as each component module is designed with a known hardware interface and data format [16, 17].

3.1 PCA pump Structure with Modules

The PCA pump design discussed in this thesis utilities a modularized design method that consists of several reconfigurable and replaceable peripheral modules. These peripheral modules are connected to a central module ("BaseBoard") through UART ports. This arrangement can be physically viewed as a set of peripheral hardware modules that are each attached to a core module, where the collection can be configured with more features as additional modules become available. From a software perspective, this system can be viewed as a group of software threads that are

each assigned to a hardware module. When a new hardware module is plugged into the system, the core module will verify it and configure it as part of the broader device.

All of the modules and the central BaseBoard share information using a fixed data package format which could be defined by, e.g., a manufacturer or comply with a standard. The BaseBoard does not care about the detailed physical design of each module, such as its type of MCU or its physical size; the BaseBoard only concentrates on the functionality supported by each module, which is aided by the pre-defined data format. The BaseBoard only needs to send/receive data to/from modules, while the modules themselves will focus on how to execute commands from the BaseBoard or how to send information to the BaseBoard through the data bus.

Figure 3-1 illustrates the basic structure of a modularized PCA pump. From a physical viewpoint, the PCA pump is embodied by a central board (BaseBoard) and several peripheral modules that implement essential functions. Each module can be attached to the central board through a wired UART port. More connection ports can be employed in the future that utilize, e.g., I^2C, SPI, a wireless Bluetooth link, etc. Regardless of the physical wired or wireless connection, each module communicates with the BaseBoard using the same type of protocol and data package, which is discussed in following section.

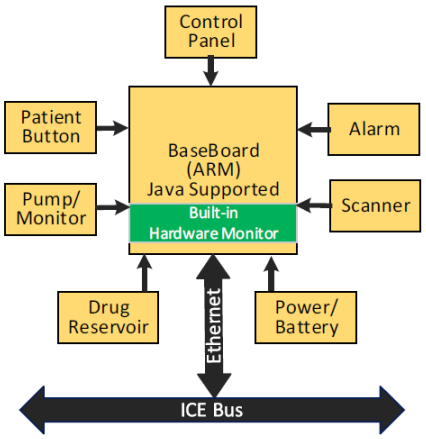

Figure 3-1. Modularized PCA pump structure.

The PCA pump in this thesis provides a concept demonstration for the design of ICE-informed medical devices using MDCF/PDE tools and a modularized design method. It is not a fully functional PCA pump: several hardware modules are designed to implement ICE concepts and a modularized design method, but the platform does not implement all of the features needed for use with a patient.

In this design, several hardware modules are designed: **an alarm module, a patient button module, a pump module, and a control panel module**. These modules are described in more detail in the sections that follow. A prototype of the PCA pump with these modules is shown in Figure 3-2. In this design, a Texas Instruments BeagleBone unit [18] is used as the BaseBoard, Arduino [19] and MSP430 [20] units are used to develop modules, and an LCD module with a built-in MCU is used as a control panel. All of the peripheral modules are powered by the BeagleBone and connected to the BeagleBone by UART Ports.

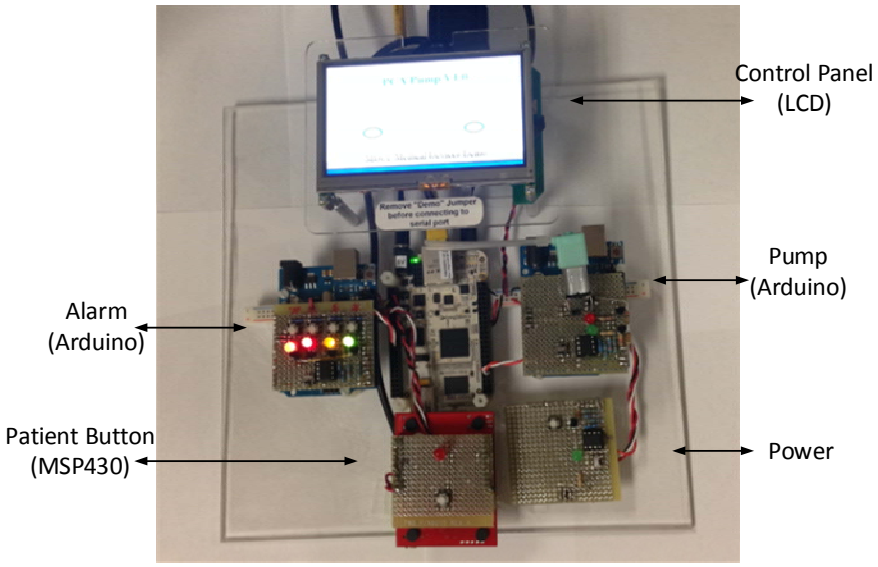

Figure 3-2. PCA pump prototype.

3.2 UART Connection between the BaseBoard and the Peripheral Modules

In the current design, all modules are connected to the BaseBoard through UART ports. Only TX and RX are used for the UART protocol [21]. Figure 3-3 illustrates a generic hardware connection based on UART between a module and the BaseBoard.

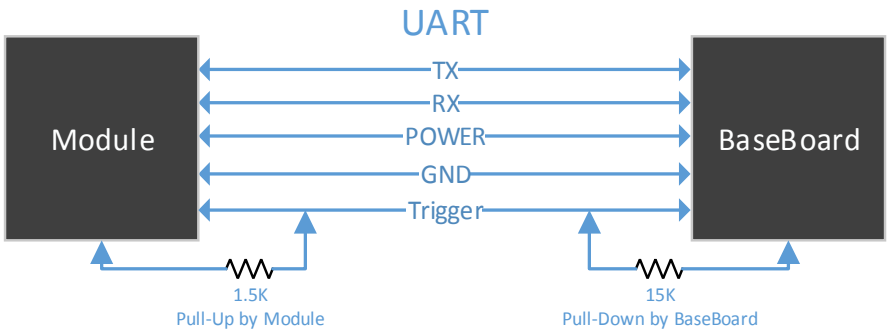

Figure 3-3. BaseBoard-to-module UART connection.

As noted above, all peripheral modules are powered by the BaseBoard. In the current design, the BeagleBone can provide +5 V and +3.3 V, the most commonly used voltages for MCUs, including those on the Arduino and MSP430 units. Since the BeagleBone can be powered by external power in addition to the USB port, it can provide enough power for modules that require more current, such as modules designed with LEDs and pumps.

A "trigger" wire is used by the BaseBoard to detect if a new module has been attached. This pin on the BaseBoard side is by default pulled down to logic 0 by a 15 kΩ resistor, whereas this pin on the module side is pulled up to a logic 1 by a 1.5 kΩ resistor. So, when a module is attached, the pin on the BaseBoard side is pulled up to a logic 1 since the voltage divided by these two resistors is logic 1.

For this design, the UART ports are working at 3.3 V, which requires the logic level on the module side to also be 3.3 V. Some modules designed with a 5 V MCU, such as the Arduino unit

in this design, need to include a voltage shifter to form a successful UART connection. The details of the voltage shifter will be discussed in the following hardware chapter.

3.3 Module Message Structure

This section defines the structure for the messages passed between modules and the host BaseBoard. All modules send messages to, and receive messages from, the BaseBoard. To meet plug-and-play needs, module messages need to have a consistent structure and adhere to the same protocol. For the current design, simple data fragment structures are designed as noted in Table 3-1, where 'BB' is short for BaseBoard. All messages begin with a "Start" and finish with an "End." The Direction column indicates whether a message is sent to the BaseBoard or received from the BaseBoard, since directionality affects the data structure and some modules may support only one direction. Different modules may have different data content and data lengths, and a module or the BaseBoard will execute its tasks according to the data packaged in the message it receives. Data standards, such as TEDS are recommended for future design [22, 23].

Table 3-1. Messages between the modules and the BaseBoard.

Module		Direction	Data 1	Data 2	Data 3	
Button	Start	To BB	Pressed			End
LCD Panel	Start	To BB	Page	Message		End
	Start	From BB	Page	Position	Message	End
Pump	Start	To BB	Pump rate	Air Bubble	Occlusion	End
	Start	From BB	Pump rate	Start/stop	Status	End
Alarm	Start	To BB	Alarm Type			End
	Start	From BB	Alarm Type	Priority	Action	End
Reservoir	Start	To BB	Level	Delivered	Remaining	End
	Start	From BB	Status Report			End
Battery	Start	To BB	Level	Remaining	Elapsed	End
	Start	From BB	Status Report			End

3.4 Modules Implemented in the Current Design

As a demonstration, several but not all modules are implemented in this design, including an **alarm module, a patient button module, a pump module, and a control panel module (LCD).** These modules are connected to UART ports of the BeagleBone as listed in Table 3-2. UART1, UART4 and UART5 are physical UART ports available from the BeagleBone, and the ttyUSB0 port is a virtual UART port provided by a USB-to-serial adapter. Since a BaseBoard can usually only provide a limited number of hardware UART ports, a designer can extend the available UART ports by using a USB-to-serial adapter in such a manner. All modules should be designed following the PCA pump requirements file [13]. Brief requirements for the 4 modules implemented in this thesis are noted in the following subsections.

Table 3-2. Specified UART ports for the modules.

Module	UART on BeagleBone
Alarm	UART1
Patient Button	UART4
Pump	UART5
Control Panel	ttyUSB0

3.4.1 Alarm Module

The alarm module shall issue alarms and warnings that require clinician attention, including visual and audible alarms. In the current design, the alarm module can indicate different types of alarms by illuminating LEDs, and an audible alarm can also be sounded by the control panel module (LCD).

3.4.2 Patient Button

The bolus request button is used by a patient to request a drug bolus in addition to a constant basal delivery rate that is locally or remotely preset by a clinician. The bolus request button is

connected by a cable to the PCA pump and may have a clip to attach to the patient's bedding. When the patient presses the request button, the patient button module will detect this request and send this request to the BaseBoard through the UART port. When the BaseBoard receives this signal, it then decides whether it will permit the pump to administrate a bolus, depending on the prescription and lapse time.

3.4.3 Pump Module

In the PCA pump, the pump itself is the most important actuator that should be accurately controlled to maintain the desired pump rate. As required, the pump must measure pressure, measure flow, and detect occlusions and air-in-line embolisms (bubbles). For the current prototype design, the pump monitor module is designed only with a pump rate controlled by pulse width modulation (PWM). No air or occlusion sensors are designed, although several push buttons have been implemented on the alarm module to simulate the associated alarms.

3.4.4 Control Panel Module

As required, a control panel allows the pump to be started and stopped. It allows a clinician to command delivery of a bolus and to specify the duration over which a prescribed volume-to-be-infused is delivered. Pump status and alarms are displayed or sounded by the physical interface. In this design, a touch LCD screen can work as a control panel instead of physical buttons, through which the clinician can operate the PCA pump. Pump status and alarms are also displayed and sounded by the LCD unit.

Chapter 4 - Modularized PCA Pump Hardware Design

4.1 Development Boards

Since the PCA pump is designed with one BaseBoard and several modules based on UART ports, the BaseBoard and all modules should all support UART ports. Ideally, for a modularized design with UART, any board supporting UART could be selected as a BaseBoard or a module board. In this design, a BeagleBone A6 is selected as a Baseboard; Arduino and MSP430 boards are selected as module development boards.

4.1.1 BeagleBone A6 as a BaseBoard Module

The PCA pump is intended to be an ICE-informed medical device through the use of the software development tools provided in the MDCF PDE. Some of the code running on the PCA pump is exported from the MDCF project, and the MDCF is developed with Java, which does not care about the low level hardware differences between the various platforms. The basic requirements for a development platform follow:

- Support Java development within an operating system such as Windows, Linux, or an embedded real-time operating system.
- Support Ethernet or Wi-Fi connections to implement an ICE interface.
- Support UART, which is the fundamental communication method for a modularized design method. Other communication protocols, such as I^2C, SPI, USB, Bluetooth wireless, etc. may be developed in the future.
- Support general purpose input/output (GPIO) for future functionality expansion.

A BeagleBone is an open source, credit-card-sized Linux computer that connects to the Internet and runs Angstrom (a Linux distribution that can support Java development). It offers plenty of I/O and processing power for real-time analysis through an AM335x 720 MHz ARM®

processor, as well as capabilities for high level Java development and low level hardware extension and development. Figure 4-1 and Table 4-1 illustrate the BeagleBone White A6 used for the current design and list its features.

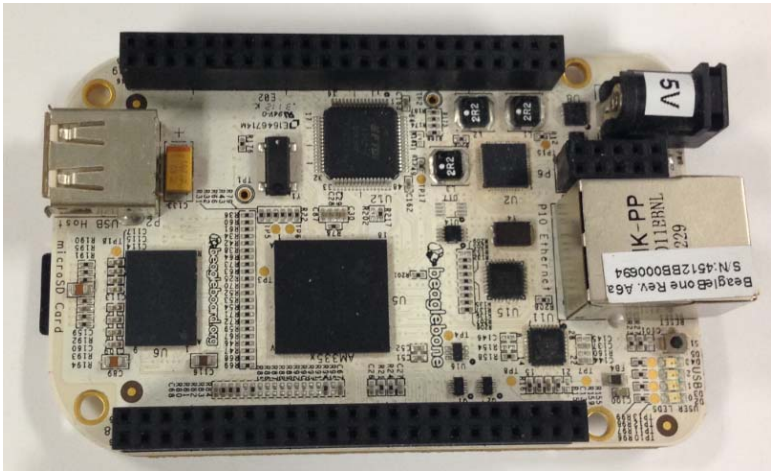

Figure 4-1. BeagleBone used as a BaseBoard.

Table 4-1. BeagleBone White A6 features.

Processor	ARM3359 500 MHz: Powered by USB 700 MHz: Powered by DC	
Memory	256 MB DDR2 at 400 MHz	
Debug Support	USB to Serial Adapter	Mini USB connector
	On Board JTAG vis USB	4 User LEDs
HS USB 2.0 Client Port	USB Client Mode	
HS USB 2.0 Host Port	Type A Socket, 500 mA LS/FS/HS	
Ethernet	10/100, RJ45	
Expansion Connectors	POWER 5 V, 3.3 V, 1.8 V UART, SPI, I2C, GPIO, LCD, MMC1, MMC2, Timers, CAN, PWM	

Besides the fast CPU and memory that can support the Linux OS, the BeagleBone A6 provides extensive expansion GPIOs which can be configured as I^2C, SPI, UART, PWM, or other functional GPIOs. These ports provide the capability to develop flexible devices with the

development board. Details for how to configure these GPIOs in Angstrom can be found on the BeagleBone datasheet. UART port configuration through Java will be discussed in the software development section.

The PCA pump is designed based on UART ports, so the total number of modules that can be attached to the BaseBoard is limited by the available UART ports. There are four physical UART ports that can be directly accessed through BeagleBone expansion headers P8 and P9: UART1, UART2, UART4 and UART5. A USB-to-serial adapter can also be used to emulate a serial port. So, with the USB host port on the BeagleBone, we can accrue additional UART ports for more hardware modules.

4.1.2 MSP 430 MCU as a Module Board

MSP430 microcontrollers from Texas Instruments (TI) are 16-bit, RISC-based, mixed-signal processors designed for ultra-low power. Figure 4-2 illustrates an MSP430 LaunchPad development board with an MSP430G2553 mixed signal microcontroller. Besides the basic communication protocols that it supports, the MSP430 offers features that could benefit other hardware modules:

- 7 low-power modes,
- instant wakeup,
- autonomous peripherals,
- high performance analog and digital blocks, and
- flexible ports, including USB ports and ADCs.

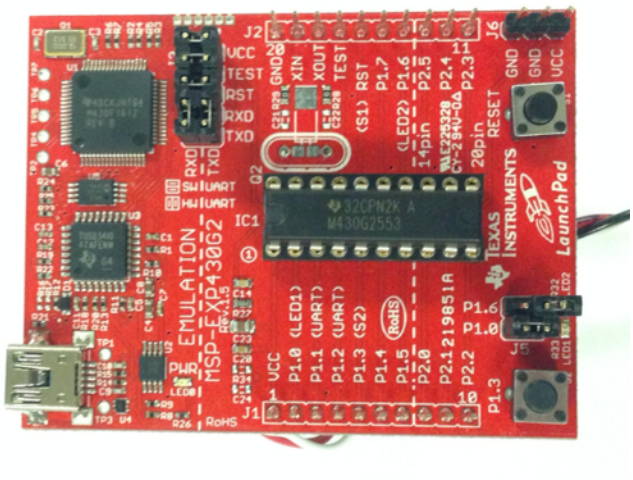

Figure 4-2. MSP430 as a module board.

4.1.3 Arduino UNO as a Module Board

An Arduino is an open source prototyping platform that helps one to develop embedded applications more easily using the provided libraries and flexible GPIOs. An Arduino UNO R3 development board shown in Figure 4-3 was selected for this PCA pump application. It provides a physical UART port working at a 5 V level. Several digital input/output pins and analog input pins can be used to expand this functionality.

Figure 4-3. Arduino UNO as a module board.

4.2 Pump Module Designed with an Arduino Board

4.2.1 Generic Voltage Shifter for UART Connections

The pins for the P8 and P9 expansion headers on the BeagleBone are working at a 3.3 V level, and the pins of the Arduino are working at a 5 V level. In order to set up a UART connection between a BeagleBone and an Arduino, the voltage shifter in Figure 4-4 was used to shift these levels from 3.3 V to 5 V.

For the TX pin (3.3 V) on the BeagleBone, an LM311 comparator (Figure 4-4, upper) with a reference voltage of 1.8 V at the negative input (Pin 3) is used so that when TX is 0 V (logic 0 for the BeagleBone), the output is 0 V (also logic 0 for the Arduino), and when TX is 3.3 V (logic 1 for the BeagleBone), the output is 5 V (logic 1 for the Arduino). For the TX pin (5 V) on the Arduino, a simple voltage divider with two resistors is used to get 3.3 V for the BeagleBone (Figure 4-4, lower).

For these TX and RX channels, two switches (SW1 & SW2) are placed on the Arduino side. Because the output from the comparator can interfere with the code-uploading process, the Arduino needs to be disconnected from the voltage shifter before code is uploaded to the Arduino; the Arduino is then reconnected to the voltage shifter circuit to form the UART channels.

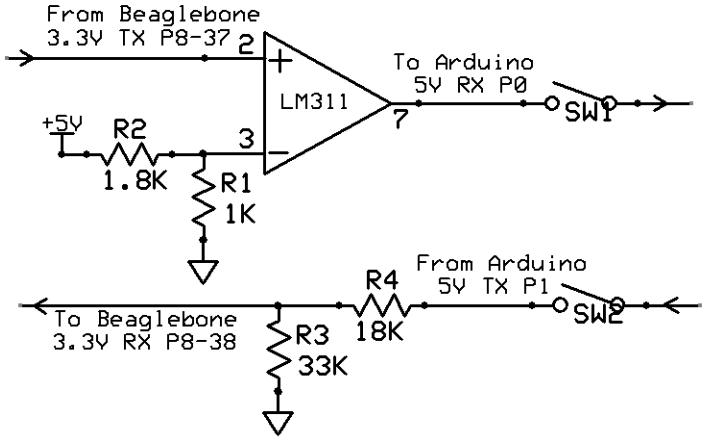

Figure 4-4. Voltage shifter for a UART port.

4.2.2 Pump Used for the Pump Module

For the current demonstration design, a miniature peristaltic pump (see Figure 4-5) is used to simulate the pump used in a commercial unit. This pump is powered at 3 V, and its speed can be controlled by PWM. Table 4-2 lists the pump specifications.

Figure 4-5. Pump used for design.

Table 4-2. Peristaltic pump parameters.

Model Number	RP-Q1-S-P45A-DC3V
Discharge Rate	0.45 ml/min ±15%
Discharge Pressure	50 kPa
Motor	DC Geared Motor

4.2.3 Pump Module Hardware Design

The pump module in Figure 4-6 was built to deliver fluid. The UART and power cables connect the pump module to the BeagleBone. The voltage level shifter was designed as in Figure 4-4. The pumping indicator is a green LED that illuminates if the pump is working. The bolus request indicator is a red LED that illuminates when the pump is working at a higher rate after the patient's bolus request is permitted.

Figure 4-7 depicts the driver schematic designed with a BJT (Fairchild PN4141) and a protection diode (IN4001). The pump can be driven with 3.3 V and sinks only around 45 mA, so the 3.3 V power pin on the Arduino can directly drive the pump. To control the pump speed, a PWM signal from P6 of the Arduino is applied to a BJT.

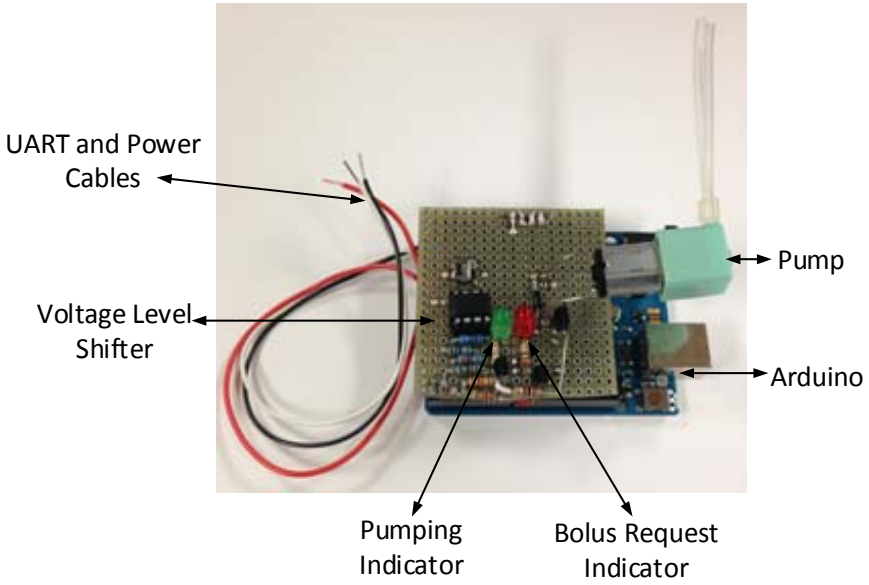

Figure 4-6. Pump module.

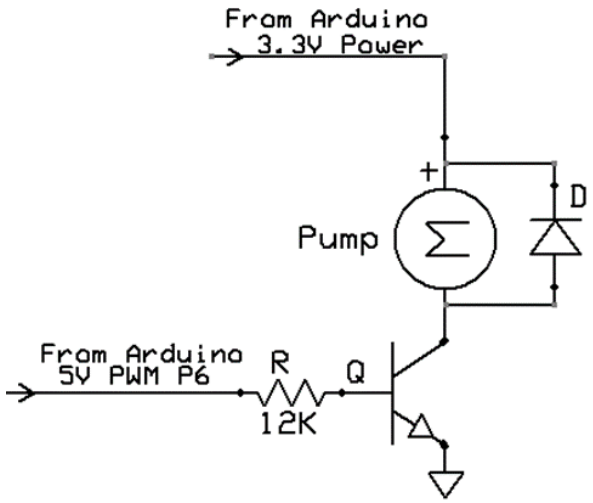

Figure 4-7. Pump PWM driver circuit.

4.2.4 LED Indicator

Green and red LEDs indicate the pump working status. When the pump is turned off, either by a local medical device operator or by a remote clinician through an MDCF console, these two LEDS are off. When the pump is delivering fluid normally, the green LED is on. When a bolus

request is permitted and the pump is working at a higher rate to delivery more fluid, both the red and green LED are on. The two LED drives are designed with BJTs and resistors as in Figure 4-8.

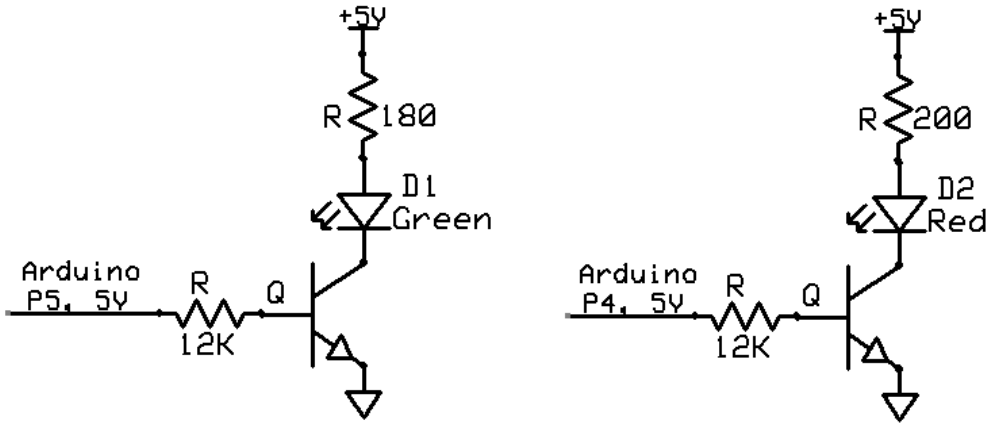

Figure 4-8. LED indicators for the pump module.

4.3 Alarm Module Designed with an Arduino Board

Figure 4-9 illustrates the alarm module hardware. For a fully functional PCA pump, the alarms are generated and sent by different sensors, such as an air/bubble sensor, occlusion sensor, or drug level sensor. Since the current PCA pump is designed without sensors, four push buttons are designed to simulate four types of alarms so that the user can press one button to send a simulated alarm to the BeagleBone through a UART port. For each button, there is a corresponding LED to indicate that an alarm has been generated and sent to the BeagleBone. These simulated alarms represent Battery Too Low, Drug Level Too Low, Air In Tube, and Occlusion conditions as noted in Figure 4-9. Figure 4-10 illustrates the LED driver circuits and push buttons that generate interrupts to the Arduino board when they are pressed.

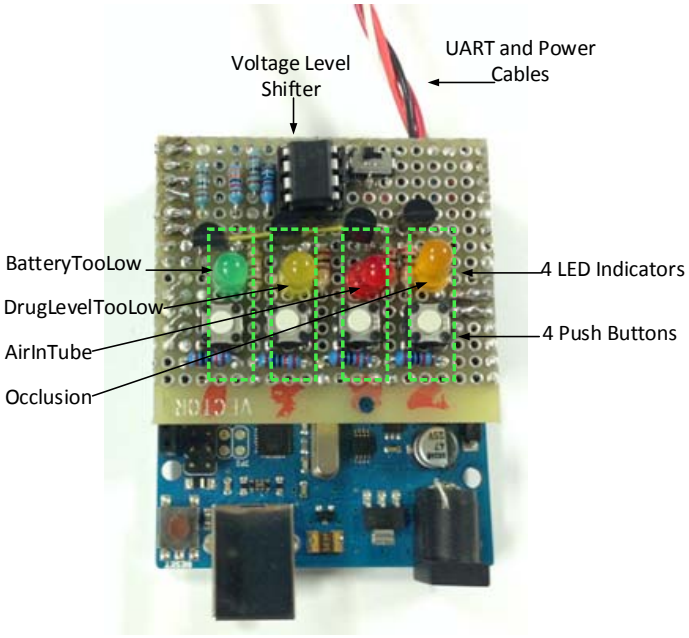

Figure 4-9. Alarm module.

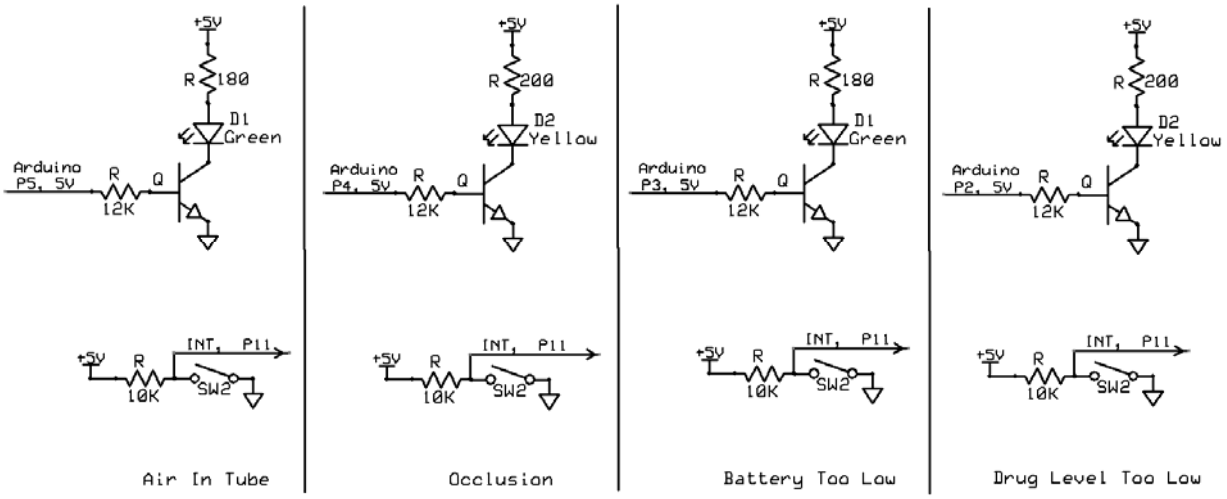

Figure 4-10. Alarm LED indicators and key circuits.

4.4 Patient Button Module Designed with an MSP430 Board

As discussed in Chapter 3, the modules can be designed with different platforms as long as these platforms can support UART communication. The patient button module is designed with an MSP430 G2553 board. Figure 4-11 includes a picture of the patient button module designed with a push button and a red LED indicator. When the button is pressed, an interrupt is generated

and sent to the MSP430 board (the backside of the picture), then the red LED indicator is turned on. A bolus request is then sent to the BeagleBone through a UART port.

Figure 4-12 depicts the push button and LED driver schematic, where P1.4 of the MSP430 G2553 is the interrupt pin and P1.6 is used to turn the LED indicator on and off.

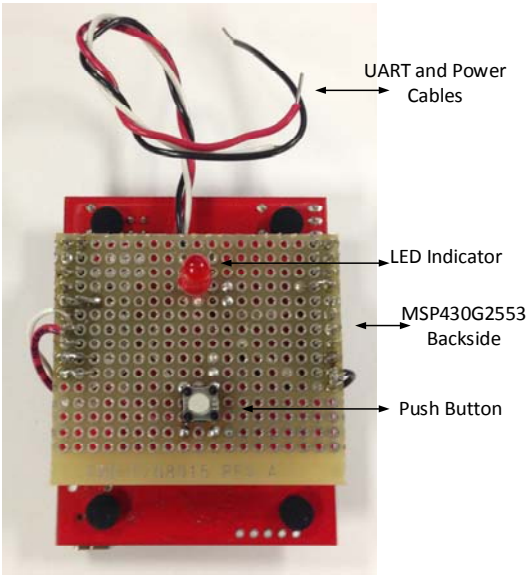

Figure 4-11. Patient button module.

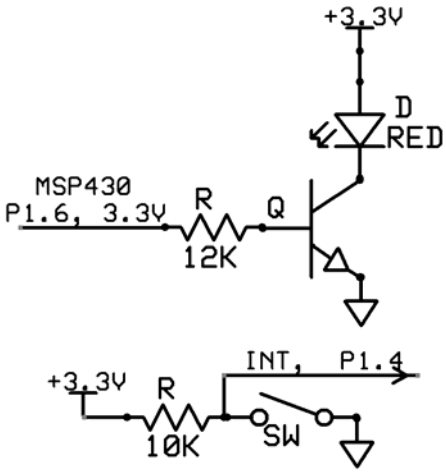

Figure 4-12. Patient button module circuit.

4.5 Control Panel Module Designed with a Resistive-Touch LCD

For the current design, a resistive-touch LCD is used as a control panel module to display information from the BaseBoard (BeagleBone) and to control the PCA pump. The LCD is a 4.3" display module from Innolux Inc. whose specifications are listed in Table 4-3.

Table 4-3. LCD specifications for the control panel module.

Board Model	LSCD43
Screen Resolution	480*272
Screen Type	WQVGA
Touch Panel	Resistive
Interface	RS232/3.3 V CMOS

There is a built-in microcontroller on the LCD module, and the developer can directly send commands to the LCD module through a 3.3 V UART port from the BeagleBone, so a control panel module was designed without any development board. Rather, it communicates directly with the BaseBoard (BeagleBone) through a USB-to-serial cable (/dev/ttyUSB0 on the BeagleBone). More details on the LCD GUI design will be discussed in the next chapter.

Chapter 5 - Modularized PCA Pump Software Development

5.1 Development Environment Setup

As discussed in Chapter 2, a medical device developer must address two types of Java code (defined as apps for purposes of the MDCF) in this environment. One relates to apps running within a remote MDCF console (PC side) that could be launched by a clinician to configure and monitor medical devices. A second relates to apps running on local medical devices (device side; PCA pump in this case) to implement desired functionality and communicate with the remote MDCF console. These two types of apps are connected through virtual ICE channels created and defined by the PDE.

Figure 5-1 illustrates these two types of software development with regard to this PCA pump design. The blue arrows are the ICE channels created by the PDE, together with Java skeleton code for "PCAPump.jar" and "PCAPumpDisplay.jar" files.

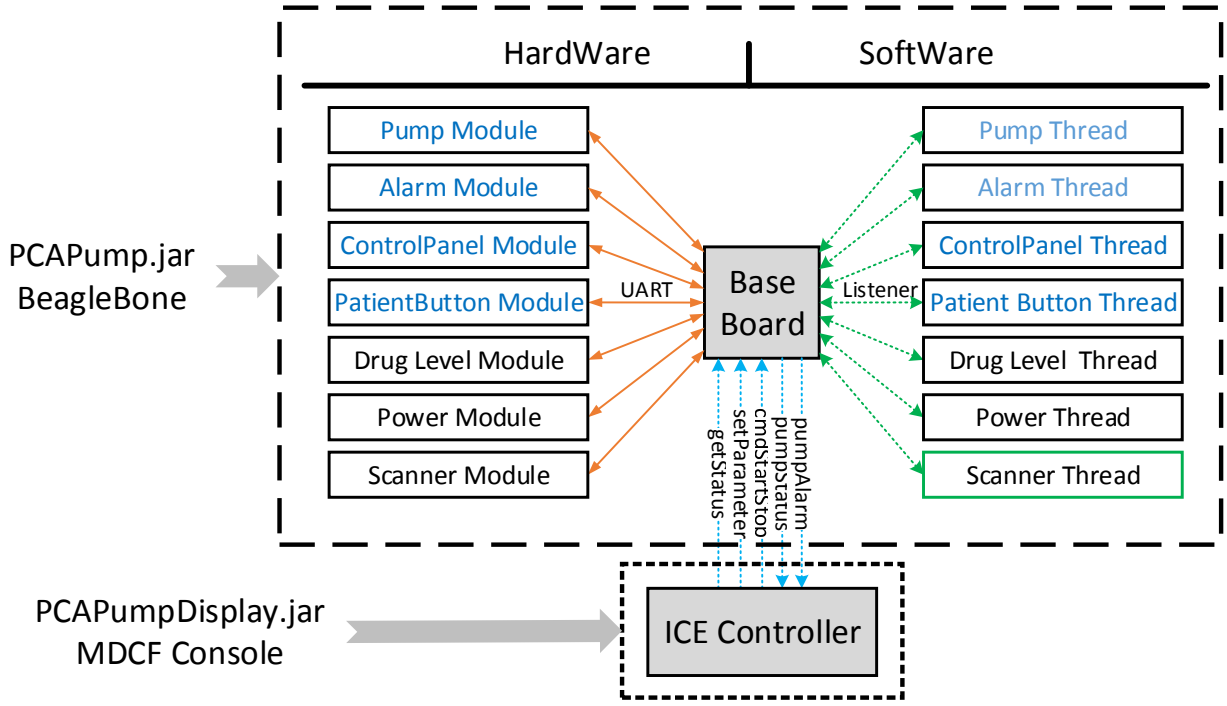

Figure 5-1. Overview of the software structure for the PCA pump.

The **"PCAPump.jar"** file is created within the MDCF to implement PCA pump functionality, then it is exported and transferred to the BeagleBone to run as a PCA pump. For this particular modularized PCA pump design, there are several hardware modules and corresponding software threads implemented in the "PCAPump.jar" code as indicated by the green and orange arrows. For different medical devices, the developer or manufacturer should flesh out the Java skeleton to implement desired functionality.

The "**PCAPumpDisplay.jar**" file runs within the MDCF to transfer the "PCAPump.jar" code to the BeagleBone through a physical Ethernet port, and a GUI can be launched to display information from the PCAPumpDiplay.jar code on an MDCF console.

5.1.1 Installation of the MDCF Developer Environment and PDE on a PC

For this PCA pump design, "**MDCF2.0withXStream1.4.3_BufferedConfiguration.zip**" is imported into Eclipse as a project package. Details on how to install the MDCF and the PDE can be found in several tutorial files on the MDCF Project web site [5]. Figure 5-2 is the MDCF environment for the PCA pump development after creating ICE channels and Java skeleton code using the PDE.

The "myPCAPumpApp" package created by the PDE includes the code running within an MDCF console (PC side). The included "PCAPumpDisplay.java" file is created by the PDE to implement ICE channels, and another "PCAPumpGUI.java" file must be developed and be launched in an MDCF console to display important information from "PCAPumpDisplay.java" and interact with the clinician.

The "PCAPump" package includes the code that will run on the local PCA pump (BeagleBone side). This includes auto-generated Java skeleton code to implement ICE channels created by the PDE as well as code developed to implement PCA pump functionality. This package

is exported as a runnable Java file ("PCAPump.jar" in this design) then uploaded to the PCA pump (BeagleBone). More software development details will be discussed in section 5.2.

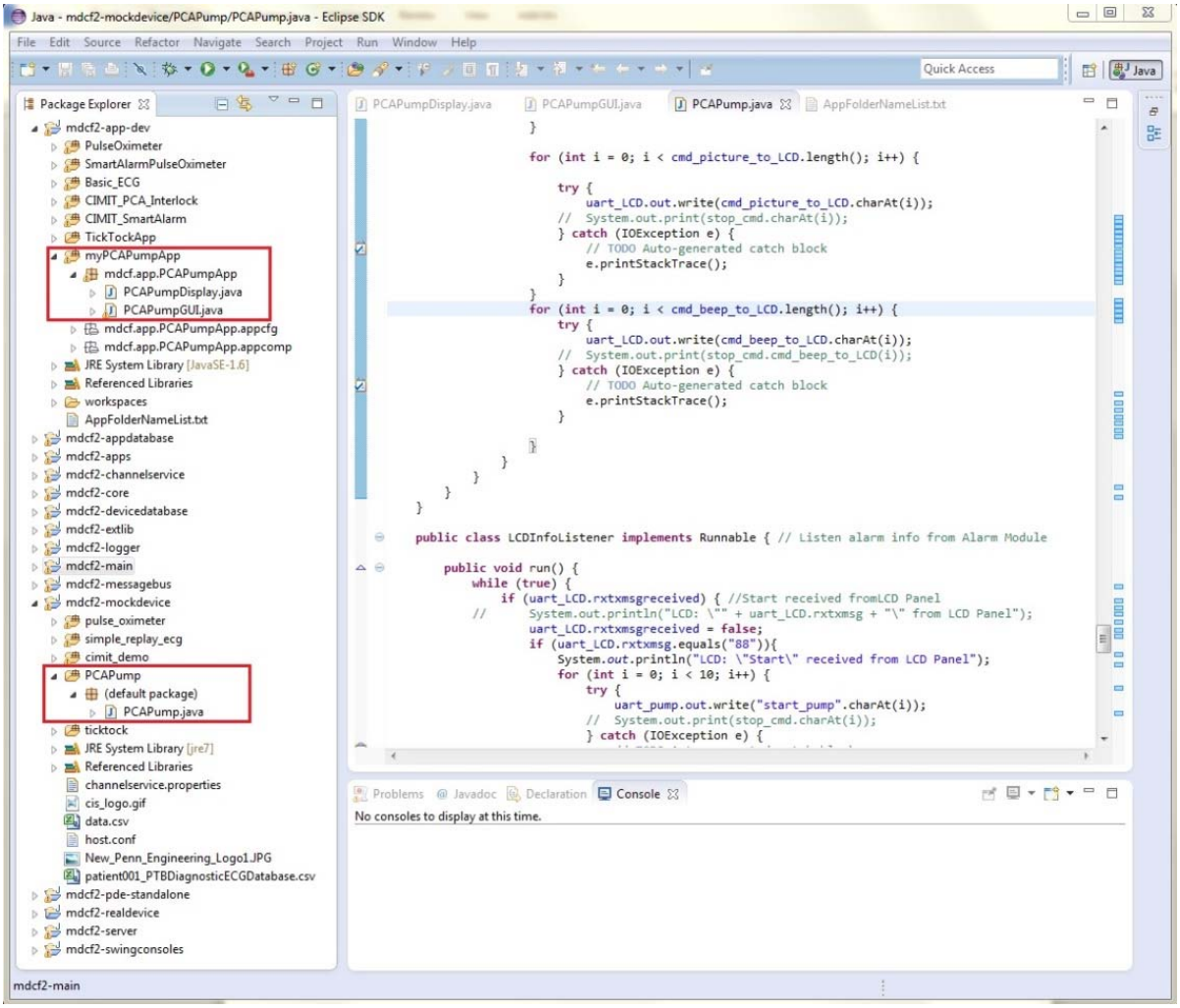

Figure 5-2. Overview of PCA pump development within the MDCF.

5.1.2 BeagleBone Development Environment Setup

Since the exported **"**PCAPump.jar" file is executed on a BeagleBone that is connected to a remote MDCF console, we need to set up the BeagleBone development and running environment in order to monitor local system information, run java files, and connect to the MDCF console.

I. **BeagleBone Setup:** Since there is no monitor connected to a BeagleBone, a PC monitor can be used to visualize system or debugging information from the BeagleBone. This requires a USB-to-serial cable connected to BeagleBone.

i. Install the BeagleBone driver on the PC (a Windows 7 system in this design; other operating systems that support RS232 will also work), then use a USB-to-serial adapter to plug the BeagleBone into the USB port on the PC. This adapter will be recognized by the PC system as a COM port. A different system may recognize it with a different name. Assume the port is "COM3" on a Windows system.

ii. Open TeraTerm (a free Serial Port monitor tool), connect "COM3," then set the baud rate to 115200 and choose an N-8-1 data format to match the BeagleBone settings. One can log into the BeagleBone with a username of 'root,' and the command shell displayed on the PC monitor allows the developer to operate the BeagleBone.

II. **Network Setup:** Set the PC (MDCF) and the BeagleBone (PCA pump) to work at the same network scope. In this design, both the PC and the BeagleBone are connected to a router (e.g., the PC IP is 192.168.1.9 and the BeagleBone IP is 192.168.1.7).

III. **File Transfer Setup:** Set up "FileZilla FTP Client" on the PC so that the developer can transfer files between the PC and the BeagleBone, mainly to transfer the exported Java file (PCAPump.jar) to the BeagleBone. Set the FTP connection parameters to Host: 192.168.1.7 (BeagleBone working as an FTP Host), Username: root, Password: none (left blank), and Port: 22.

5.1.3 Java Installation and Configuration

For the current version of the MDCF, the JDK 1.7 is required for both the MDCF (PC side) and the PCA pump (BeagleBone side).

I. **Java installation in Eclipse on the MDCF/PC side:** For Java installation and configuration on a PC, the developer needs to check that Eclipse is using JDK 1.7 just in case other Java versions are installed on the same PC.

II. **Java installation on the BeagleBone side:** The JRE or JDK is required on the BeagleBone to run the Java file (PCAPump.jar). The following steps lay out the JDK1.7 installation:

i. Download JDK 7 for Linux/ARM with Soft Floating Point numbers (SoftFP) to the PC from the Oracle website. For the current design, "jdk-7u60-linux-arm-vfp-sflt.tar.gz" is downloaded from the Oracle Java website. The BeagleBone is designed with Linux/ARM tools, so Java for other platforms will not work.

ii. Copy the "jdk-7u60-linux-arm-vfp-sflt.tar.gz" file to the BeagleBone via FileZilla.

iii. In this design, JDK 1.7 is installed in the file folder path of"/user/java," although it can be installed in any location. Unzip the "jdk-7u60-linux-arm-vfp-sflt.tar.gz" file and then copy the individual files into /usr/java.

   a. $tar xzf jdk-7u60-linux-arm-vfp-sflt.tar.gz
   b. $mkdir /usr/java
   c. $mv jdk1.7.0_60 /usr/java

   Check to ensure that all Java files, including lib, jre, include, etc., are under path "/usr/java/jdk1.7.0_60."

iv. Configure the Java path on the BeagleBone. Edit the file with following commands:

   a. $vi /etc/profile
   b. At the end of this "profile" file, add the following lines:

```
JAVA_HOME=/usr/java/jdk1.7.0_60
PATH=$JAVA_HOME/jre/bin:$PATH
CLASSPATH=.:$JAVA_HOME/lib

export JAVA_HOME
```

exprot PATH

export CLASSPATH

c. Check the Java installation:

$java -version //The output should indicate Java 1.7 is installed.

$which java //The output indicates /usr/java/jdk1.7.0_60/jre/bin/java

5.1.4 UART Register Configuration on the BeagleBone

As discussed before, UART1, UART2, UART4 and UART5 can be directly accessed by expansion headers. But, these pins are used for different functions, such as PWM, GPIO, SPI, etc., and the default configuration is not set to UART. The developer needs to configure the related registers to set these pins to UART mode.

The pins for each UART port on expansion headers are shown in Table 5-1. In this case, only RX and TX are used with an N-8-1 data format, so CTS and RTS are not used even though they also appear in this table. All of the modules designed with an Arduino board are powered by 5 V from the BeagleBone, and the modules designed with an MSP430 are powered by 3.3 V from the BeagleBone.

Table 5-1. BeagleBone UART ports and power pins.

UART	**RX**	**TX**	**CTS**	**RTS**
UART1	P9-26	P9-24	P9-20	P9-19
UART2	P9-22	P9-21	P8-37	P8-38
UART4	P9-11	P9-13	P8-35	P8-33
UART5	P8-38	P8-37	N/A	N/A
PWR +5 V	P9-5, P9-6			
PWE +3.3 V	P9-3, P9-4			
GND	P9-1, P9-2, P8-1, P8-22			

The developer can execute the UART configuration commands on the BeagleBone before executing the “PCAPump.jar” file in the Linux Command Shell, but the configuration will be lost

and registers will be reset to default after rebooting the system. Instead, several ***ProcessBuilders*** are used to do the configurations at the start of the "PCAPump.jar" file so that every time this file is executed, the configuration commands will be executed first. More details will be discussed in section 5.3.1.

5.1.5 RXTX Library Installation on PC and BeagleBone

As a high level development language, Java needs special library files to communicate with hardware. In order to talk with BeagleBone UART ports, RXTX library files are installed on both the PC and the BeagleBone. In the current design, version "rxtx-2.2" is installed. For installation on both the PC and the BeagleBone, the developer needs to point Eclipse to the "RXTXcomm.jar" file, which contains the libraries for UART ports. For RXTX installation on the BeagleBone, these library files are installed in "/usr/lib/jni."

Also, RXTX tries to detect available UART ports by scanning /dev (on the BeagleBone Linux system) for files matching any of a set of known prefixes, such as 'ttyS', 'ttyO', and so on. A UART port may be enumerated with different names by different systems. For example, a UART port is named COM# by a Windows system and usually named /dev/ttyS# in Linux. For Angstrom on a BeagleBone, UART is named /dev/ttyO#. The developer can set the system properties of "*gnu.io.rxtx.SerialPorts"* and *"gnu.io.rxtx.ParallelPorts."* If either of these is set to the name style by the system, then no scanning will be carried out and only the specified ports will be available. So, the following command is executed to run the "PCAPump.jar" file:

```
java -jar -Djava.library.path=/usr/lib/jni -
Dgnu.io.rxtx.SerialPorts=/dev/ttyO1:/dev/ttyO2:/dev/ttyO4:/dev/ttyO5:/dev/ttyUSB0
PCAPump.jar
```

where `-Djava.library.path` points to the RXTX library file location and

`-Dgnu.io.rxtx.SerialPorts` points to the UART names specified by the BeagleBone.

To avoid typing this command in the Linux shell every time to run the "PCAPump.jar," the command is put in an executable bash file named "PCAPump_java_command" in the same file location with the following content:

```
#~/bin/bash

java -jar -Djava.library.path=/usr/lib/jni -Dgnu.io.rxtx.SerialPorts=/dev/ttyO1:
/dev/ttyO2:/dev/ttyO4:/dev/ttyO5:/dev/ttyUSB0 PCAPump.jar
exit 0
```

The developer can execute the command "./PCAPump_java_command" in the Linux Shell to execute the "PCAPump.jar" file.

5.2 PCA pump App Development with the PDE

As discussed in Chapter 2, the PDE is a development tool provided in the MDCF distribution for app building, including device, display and logic components that are connected by virtual ICE channels. The medical device manufacturer or developer can use the PDE to create ICE channels and Java skeleton code to develop apps running within an MDCF console and apps running on a medical device.

5.2.1 Definition of Components, Ports and ICE Channels for the PCA pump

Figure 5-3 lists the components, ports, and ICE channels for the PCA pump created within the PDE:

Components: One is "PCAPumpDisplay" (green block in Figure 5-3), which has a display role and runs within an MDCF console on the PC to monitor or control the PCA pump. Another is "PCAPump" (blue block in Figure 5-3), which has a device role and runs on the BeagleBone as a remote medical device. Figure 5-2 depicts the related file package in the MDCF project. Since we focus on the operation but not on the logical control of the PCA

pump for this design, the Logical role is not defined here. The developer can easily add a Logic role and create ICE Channels in a same way.

Ports: For the "PCAPump" device app, five ports are defined, including three Publish ports and two Subscribe ports. For the "PCAPumpDisplay" display app, five ports are defined, including three Subscribe ports and two Publish ports.

Connections: There are five connections to connect these five ports to form MDCF Message channels. Each connection is from a Publish port to a Subscribe port.

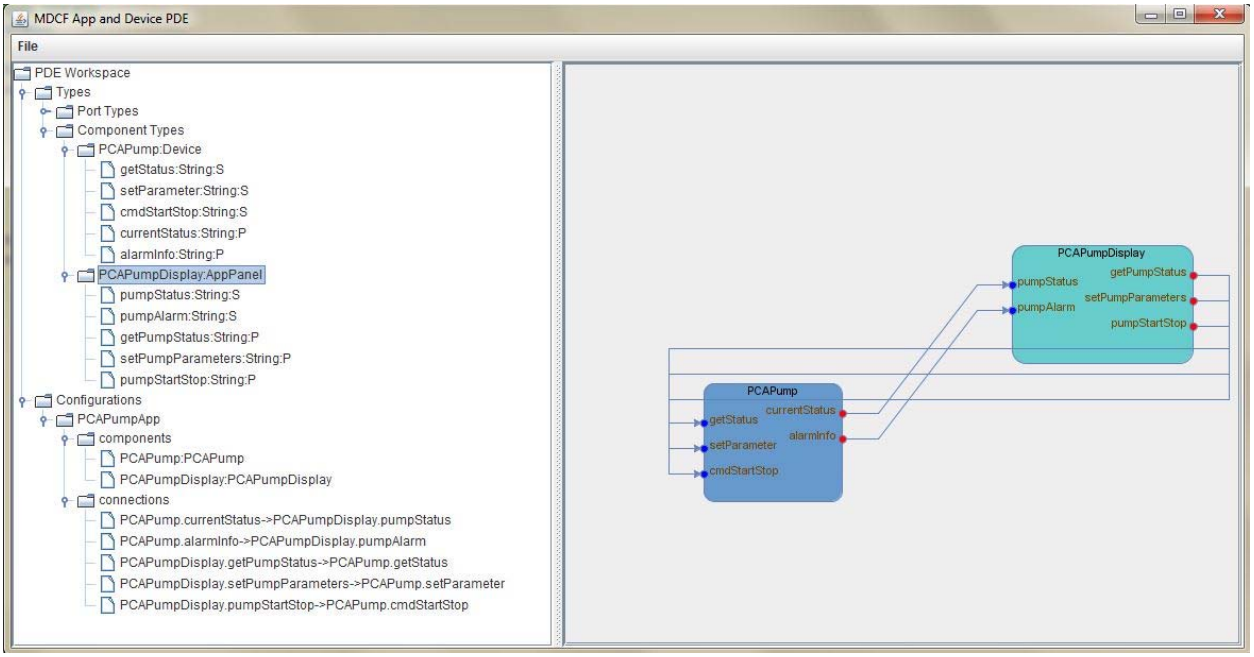

Figure 5-3. PCA pump components, ports, and channels.

5.2.2 Java Code for the ICE Channels Created Through the PDE

As discussed before, the Java skeleton code implementing these five ICE channels is automatically generated by the PDE and installed within the MDCF. Table 5-2 lists these channels with the auto-generated Java class and function names.

Table 5-2. PCA pump ICE channels.

Channel	PCA Pump (BeagleBone Side)	MDCF Console (PC Side)
1	class getStatusListener implements IMdcfMessageListener	public void send_getPumpStatus(String data)
2	class setParameterListener implements IMdcfMessageListener	public void send_setPumpParameters(String data)
3	class cmdStartStopListener implements IMdcfMessageListener	public void send_pumpStartStop(String data)
4	private void send_currentStatus(String data)	class pumpStatusListener implements IMdcfMessageListener
5	private void send_alarmInfo(String data)	class pumpAlarmListener implements IMdcfMessageListener

Channel 1: From an MDCF console to a PCA pump. Commands are sent through this channel to get the current status of PCA pump.

Channel 2: From an MDCF console to a PCA pump. Commands and parameters are sent through this channel to set PCA pump parameters.

Channel 3: From an MDCF console to a PCA pump. Commands are sent through this channel to start or stop the PCA pump.

Channel 4: From a PCA pump to an MDCF console. The current status of the PCA pump is sent to the MDCF console through this channel as a response to a command from Channel 1.

Channel 5: From a PCA pump to an MDCF console. Alarm information for the PCA pump is sent through this channel to the MDCF console.

Figure 5-4 depicts the ICE channel overview in Eclipse. Three blue arrows indicate channel #1, #2, #3, and two orange arrows indicate channels #4 and #5 in Table 5-2.

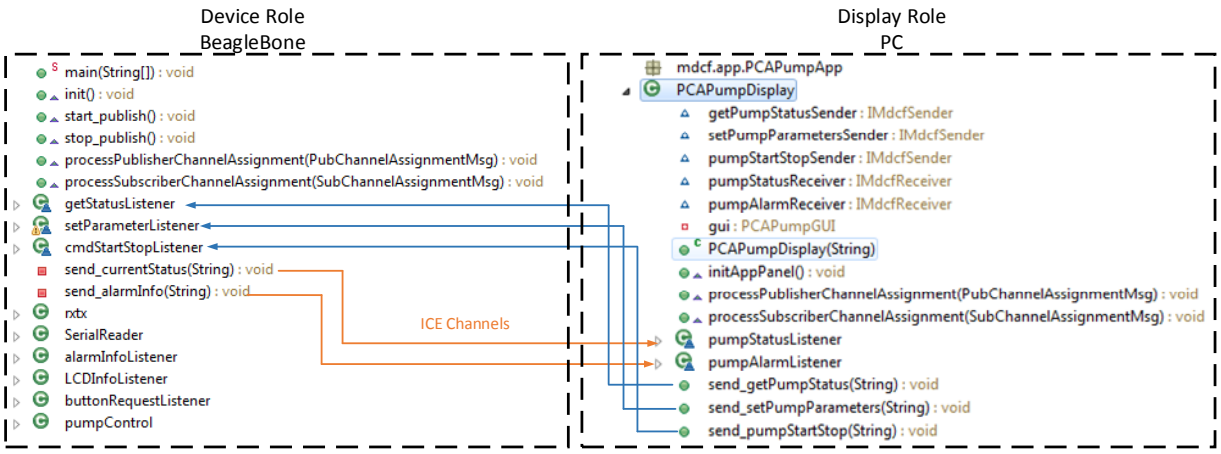

Figure 5-4. ICE channels in Eclipse.

5.3 Multi-Threaded Software Development for the Hardware Modules

As discussed in previous sections, "PCAPump.jar" is the file that will run on BeagleBone with pre-defined ICE channels and Java skeleton by PDE. These channels are used to send messages to an MDCF console. Another important part of "PCAPump.jar" is to implement the desired functionalitiy as a PCA pump. This section will focus on how to develop multi-thread software to coordinate all the hardware modules.

5.3.1 UART Ports Configuration by Java

As discussed before, all the UART ports should be configured before the code try to connect modules through UART ports.

To configure hardware registers in Java, several ***ProcessBuilder*** are defined as following:

```
// set for UART1
static String uart1_rx = "echo 20 > /sys/kernel/debug/omap_mux/uart1_rxd";
static String uart1_tx = "echo 0 > /sys/kernel/debug/omap_mux/uart1_txd";
// set for UART2
static String uart2_rx = "echo 21 > /sys/kernel/debug/omap_mux/spi0_sclk";
static String uart2_tx = "echo 1 > /sys/kernel/debug/omap_mux/spi0_d0";
// set for UART4
static String uart4_rx = "echo 26 > /sys/kernel/debug/omap_mux/gpmc_wait0";
```

```
static String uart4_tx = "echo 6 > /sys/kernel/debug/omap_mux/gpmc_wpn";
// set for UART5
static String uart5_rx = "echo 24 > /sys/kernel/debug/omap_mux/lcd_data9";
static String uart5_tx = "echo 4 > /sys/kernel/debug/omap_mux/lcd_data8";
// ProcessBuilders
static ProcessBuilder uart1_rx_set = new ProcessBuilder("bash", "-c", uart1_rx);
static ProcessBuilder uart1_tx_set = new ProcessBuilder("bash", "-c", uart1_tx);
static ProcessBuilder uart2_rx_set = new ProcessBuilder("bash", "-c", uart2_rx);
static ProcessBuilder uart2_tx_set = new ProcessBuilder("bash", "-c",uart2_tx);
static ProcessBuilder uart4_rx_set = new ProcessBuilder("bash", "-c", uart4_rx);
static ProcessBuilder uart4_tx_set = new ProcessBuilder("bash", "-c", uart4_tx);
static ProcessBuilder uart5_rx_set = new ProcessBuilder("bash", "-c", uart5_rx);
static ProcessBuilder uart5_tx_set = new ProcessBuilder("bash", "-c", uart5_tx);
```

At the beginning of "main" function of PCAPump.jar, the following commands should be executed and the registers will be configured.

```
uart1_rx_set.start();
uart2_rx_set.start();
uart4_rx_set.start();
uart5_rx_set.start();
```

Since the UART ports are connected by RXTX library files (discussed in next section) that will try to remove UART lock files before connecting UART ports. For BeagleBone, following commands are executed to remove all these lock files so that the UART ports can be successfully connected:

```
static String uart1_lockfile = "rm /var/lock/LCK..ttyO1";
static String uart2_lockfile = "rm /var/lock/LCK..ttyO2";
static String uart4_lockfile = "rm /var/lock/LCK..ttyO4";
static String uart5_lockfile = "rm /var/lock/LCK..ttyO5";

static ProcessBuilder uart1_lockfile_clear = new ProcessBuilder("bash", "-c",
uart1_lockfile);
```

```
static ProcessBuilder uart2_lockfile_clear = new ProcessBuilder("bash", "-c",
uart2_lockfile);
static ProcessBuilder uart4_lockfile_clear = new ProcessBuilder("bash", "-c",
uart4_lockfile);
static ProcessBuilder uart5_lockfile_clear = new ProcessBuilder("bash", "-c",
uart5_lockfile);
```

At the beginning of "main" function of PCAPump.jar, the following commands should be executed to remove related lock files.

```
uart1_lockfile_clear.start();
uart2_lockfile_clear.start();
uart4_lockfile_clear.start();
uart5_lockfile_clear.start();
```

If more UART Ports are required for more hardware modules, the developer can follow the same method to configure the UART Ports.

5.3.2 Software Threads for the Hardware Modules

From a hardware viewpoint, all of the modules are physically connected to UART ports as noted in Table 5-3. From a software viewpoint, each UART port is assigned to a thread that will send or receive data to or from the assigned UART port.

Table 5-3. Module UART port assignments.

Module	UART Port
Alarm Module	UART1
Patient Button Module	UART4
Pump Module	UART5
Control Panel Module (LCD)	/dev/ttyUSB0

All of the threads for UART ports are objects of a generic Java class defined as follows:

```
public class rxtx {
	volatile boolean rxtxmsgreceived = false;
		volatile String rxtxmsg;
	OutputStream  out;
	InputStream  in;

	public rxtx() {
		super();
	}
	void connect(String portName) throws Exception {
	}
}
```

Here, "`rxtxmsgreceived`" indicates if a message is received from a UART port, "rxtxmsg" is used to store the message received from the UART port, "out" is the stream to write data to the UART port, "in" is the stream to read data from the UART port, and the function "connect ()" is used to create the thread and assign the desired UART port to this thread when initializing an object of "rxtx" class. In the current design, four module threads are implemented, including an alarm module, a control panel module (LCD), a patient button module, and a pump module. Four objects of class rxtx are defined as follows:

```
rxtx uart_alarm = new rxtx();    //For Alarm Module
rxtx uart_button = new rxtx();   //For Patient Button Module
rxtx uart_LCD = new rxtx();      //For ControlPanel Module
rxtx uart_pump = new rxtx();     //For Pump Module
```

Four threads will be created and assigned to the UART ports so that the hardware module can execute the following lines of code, where each line calls the "**`connect ()`**" function:

```
uart_alarm.connect("/dev/ttyO1");
uart_button.connect("/dev/ttyO1");
uart_LCD.connect("/dev/ttyO1");
uart_pump.connect("/dev/ttyO1");
```

After execution, each object (uart_alarm for the alarm module, uart_button for the patient button module, uart_LCD for the control panel module, and uart_pump for the pump module) has its own "out" and "in" streams so that we can use the "`out.write()`" **`function`** to write data to its own UART port, i.e. to the hardware module, and we can also build a "*Listener*" to receive data by the "**`in`**" stream from the UART port, i.e., from the hardware module.

Figure 5-5 depicts the definition of these threads in Eclipse, where "rxtx" is the base class for all of these objects and alarmInfoListener, LCDInfoListener buttonRequestListener, and pumpControl are "*Listeners*" used to listen if messages are received from the hardware modules. If messages are received, an ICE channel can be used to send data to a remote MDCF console.

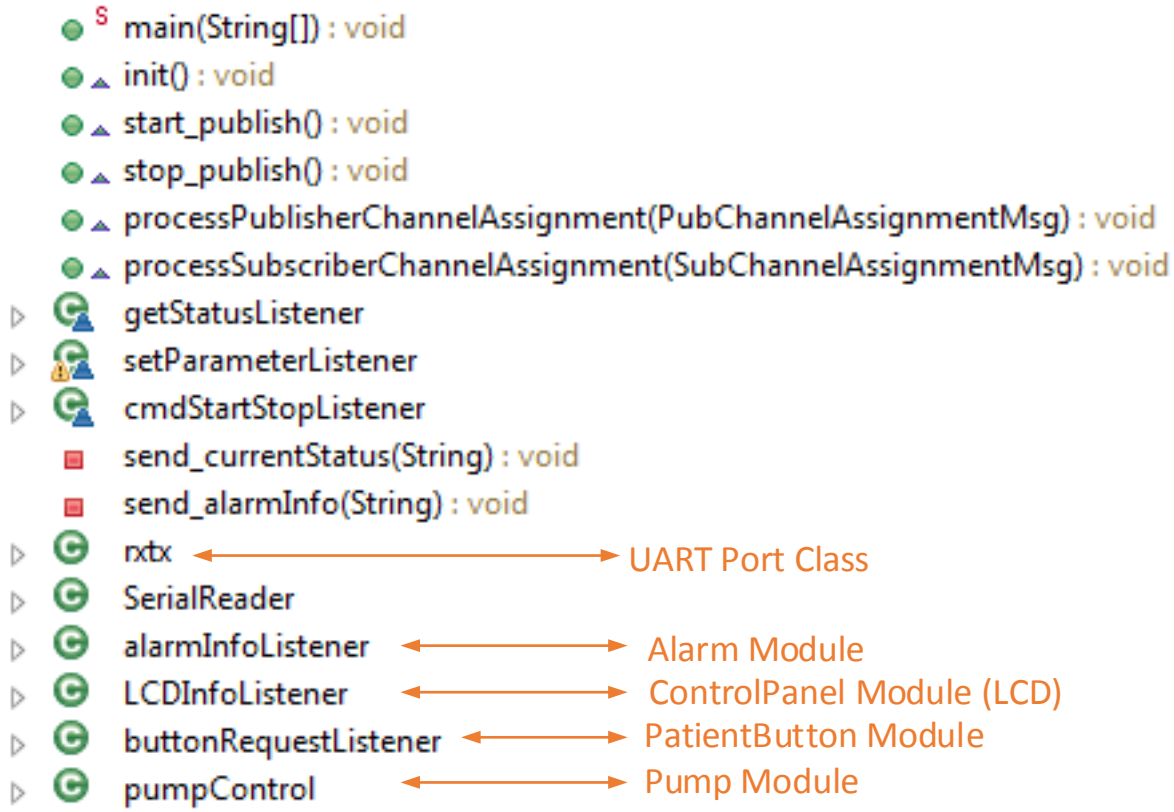

Figure 5-5. Module object definitions.

Assume alarmInfoListener to be an example for how the messages are transmitted between the alarm module, the BeagleBone and a remote MDCF console. After a thread is created as discussed in this section, a Listener for the alarm module on UART tort 1 is defined as follows:

```
public class alarmInfoListener implements Runnable {
	public void run() {
		while (true) {
			if (uart_alarm.rxtxmsgreceived) {
				uart_alarm.rxtxmsgreceived = false;
				send_alarmInfo(uart_alarm.rxtxmsg);
				for (int i = 0; i < 10; i++) {
					try {

		uart_pump.out.write("stop_pump_".charAt(i));
					} catch (IOException e) {
						e.printStackTrace();
					}
				}
			pump_started = false;
		}
	}
}
```

The “alarmInfoListner” thread can be started with

```
(new Thread(new alarmInfoListener())).start();
```

For this alarm module and thread, there are three types of message communications:

1. **Communication Between a Hardware Module and the BeagleBone**

   After the alarm thread is started, a `SerialReader()` defined by “rxtx” listens to the UART port that is connected to the alarm module. When the alarm module sends an alarm message to the BeagleBone through this UART port, the `SerialReader()` function sets `uart_alarm.rxtxmsgreceived` to true.

2. **Communication Between the BeagleBone and an MDCF Console**

After `uart_alarm.rxtxmsgreceived` is set to true, the alarmInforListener will call the `send_alarmInfo(uart_alarm.rxtxmsg)` function that is defined as an ICE channel to send this received alarm message to a remote MDCF console.

3. Communication between Hardware Modules

In this design, if any alarm information is detected, the current design will stop the pump by calling `uart_pump.out.write("stop_pump_".charAt(i))` to write the "stop_pump" command to the pump module so that the pump will stop pumping.

5.4 MDCF Console App development

Another important part of software development is to develop an app running within an MDCF console on the PC side so that the clinician can launch, configure, and monitor the desired medical device.

As noted in Figure 5-2, there are two Java files in the "myPCAPumpApp" package. The file "PCAPumpDisplay.java" is generated by the PDE in a display role. The file "PCAPumpGUI.java" defines the GUI that could be launched within the MDCF console. The GUI launched in an MDCF console can be quite flexible depending on the manufacturer. A GUI for the PCA pump in this thesis is shown in Figure 5-6. This GUI provides five tabs to show the status of the PCA pump, including pump rate, drug level, battery, pump parameters, patient information, nurse information, pump setting, and alarm. The clinician can also start or stop the PCA pump by clicking the bottom two buttons. The bottom status label indicates the working status of the PCA pump or alarm info. More details on the PCA pump operation with the GUI will be discussed in section 5.6.

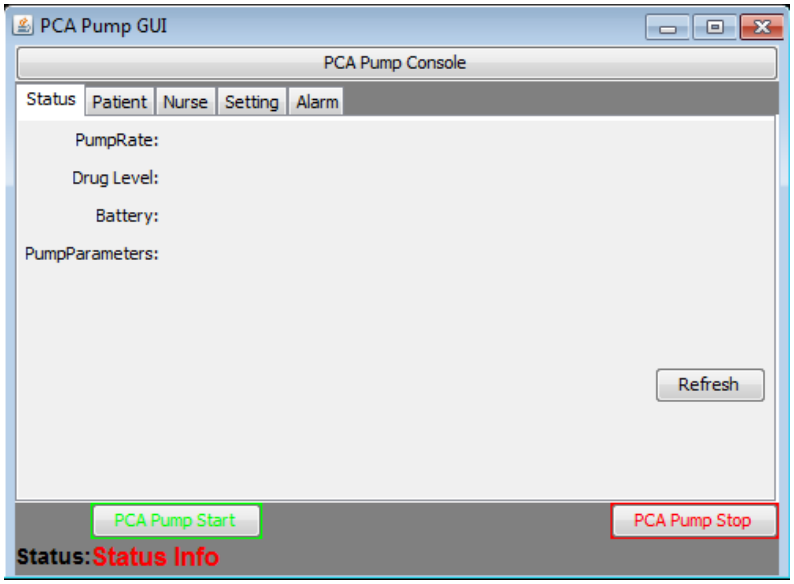

Figure 5-6. PCA pump GUI launched in an MDCF console.

5.5 Export Runnable Java File to Device

After finishing software development, the manufacturer or developer should export the software code from the MDCF project as a runnable Java file to the BeagleBone. In this design, all necessary files are stored in "/home/root/MDCF_PCAPump" on the BeagleBone.

The PCAPump.java file is exported as a runnable "PCAPump.jar" file with the option "Package required libraries into generated JAR" as depicted in Figure 5-7. The file is then uploaded to the BeagleBone by FileZilla to the path "/home/root/MDCF_PCAPump."

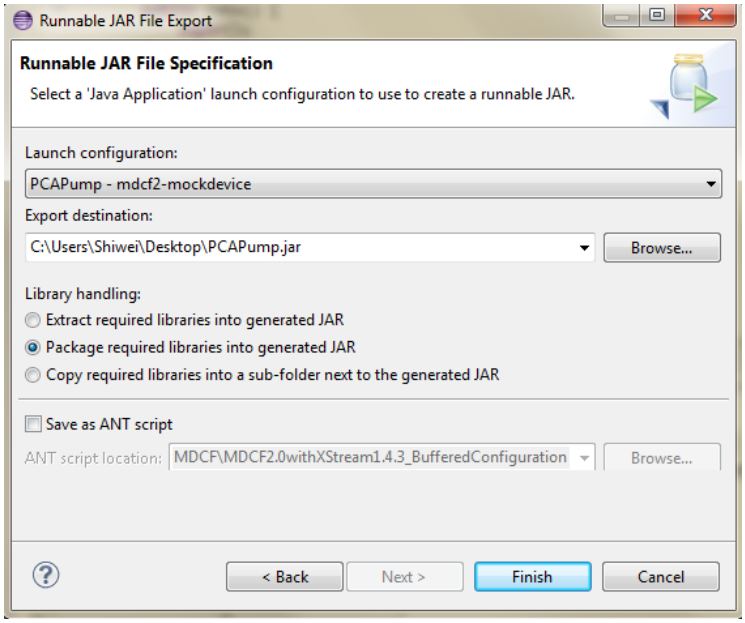

Figure 5-7. Settings to export a runnable PCAPump.jar file.

As configured before, the IP of the MDCF on PC side is "192.168.1.9," and a "host.conf" file should contain this IP in the same file path as PCAPump.jar. A "channelservice.properties" file is also required to connect the BeagleBone to MDCF console. Figure 5-8 notes the content of the "/home/root/MDCF_PCAPump" folder.

```
root@beaglebone:~/MDCF_PCAPump# ls
PCAPump.jar  PCAPump_java_command  channelservice.properties  host.conf
```

Figure 5-8. Files contained in the MDCF_PCAPump folder.

On the MDCF (PC side), the developer should specify the app by putting "PCAPumpApp" in the file "AppFolderNameList.txt" so that the PCAPump GUI will appear after it is launched in an MDCF console.

5.6 PCA Pump Module Operations Within an MDCF Console

After finishing the configurations, software development, and runnable Java file exportation, the PCA pump should work as an ICE-informed device.

5.6.1 MDCF Console and PCA Pump GUI Launching

The main MDCF console on the PC side should be launched first by right clicking on "ExecuteServerAndConsole.java" in the MDCF to run it as a Java Application. There are several devices in the App Launcher, and "PCAPumpApp" is the app developed for this PCA pump, as noted in the left part of Figure 5-9.

At this time, the PCA pump is not available, since the "PCAPump.jar" file has not been executed on the BeagleBone, so "PCAPumpAPP" is in a gray font and cannot be launched. As discussed before, the "PCAPump.jar" file can be launched by executing the bash command

```
root@beaglebone:~/MDCF_PCAPump# ./PCAPump_java_command
```

The output information from the PCA pump (BeagleBone) is shown in Figure 5-10, including MDCF configuration messages and PCA pump working messages.

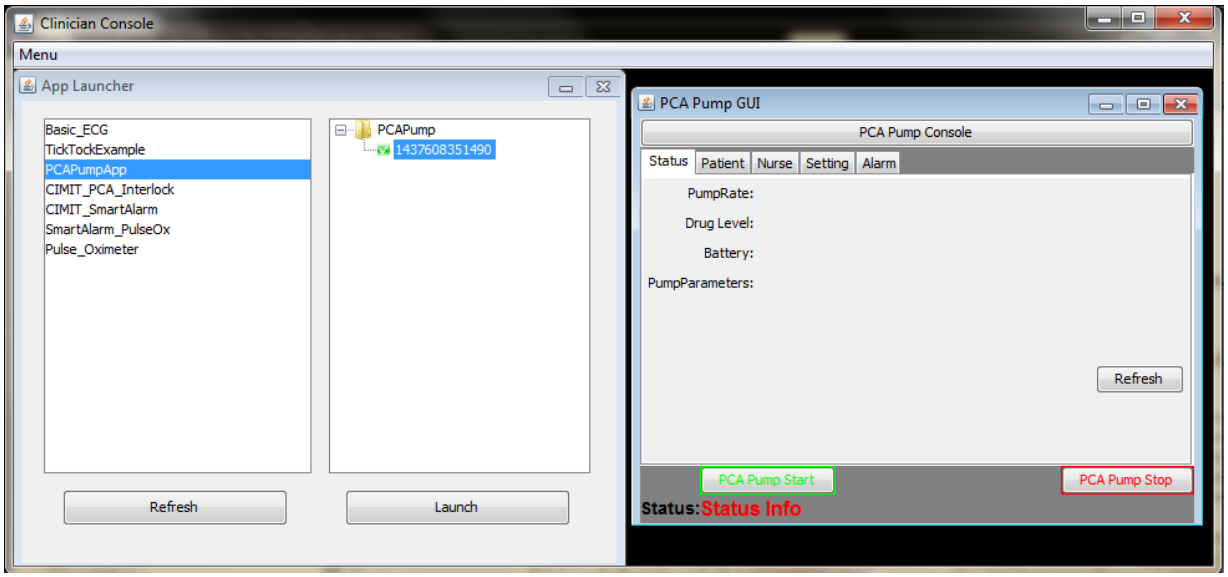

Figure 5-9. Launching a PCA pump app in an MDCF console.

A. Main start: Iindicates when the "PCAPump.jar" file starts running and starts several configurations, including the UART configurations as discussed in section 5.3.1.

B. Com connected successfully: These outputs indicate that UART threads for UART1, UART2, UART4, UART5 and /dev/ttyUSB0 have been created, initialized and successfully connected to use.

C. (OP:1437608351490)<IDLE>==><PUBLISHING>

When the device goes into its PUBLISHING state as defined by the MDCF, it is successfully connected to an MDCF console and can be launched from the MDCF console.

When the device goes into a "PUBLISHING" state, "PCAPumpApp" goes from a gray to a green font, indicating that it can be launched. After clicking the "Launch" button, the "PCAPumpApp"GUI developed in the "myPCAPump" package is launched as shown in Figure 5-9. This GUI provides several operations that will be discussed in section 5.6.3.

```
Main start
(OP:1437608351490)<Init>==><IDLE>
(SM:1437608351490:PCAPump)
(SM:1437608351490)[Init]-->[DISCONNECTED]
(MSG:1437608351490)Auth Attempt
(SM:1437608351490)[DISCONNECTED]-->[AUTHENTICATING]
Main end
(MSG:1437608351490)Authenticated:fromManagerPriv1437608351490,toManagerPriv14376
08351490
(SM:1437608351490)[AUTHENTICATING]-->[AUTHENTICATED]
WARNING:  RXTX Version mismatch
        Jar version = RXTX-2.2-20081207 Cloudhopper Build rxtx.cloudhopper.net
        native lib Version = RXTX-2.2pre2
RXTX Warning:  Removing stale lock file. /var/lock/LCK..ttyUSB0
++++++++++++++Com connected successfully++++++++++++++
++++++++++++++Com connected successfully++++++++++++++
++++++++++++++Com connected successfully++++++++++++++
++++++++++++++Com connected successfully++++++++++++++
++++++++++++++Com connected successfully++++++++++++++
(SM:1437608351490)[AUTHENTICATED]-->[ASSOCIATING]
(MSG:1437608351490)HBChanReceived:HBAckChannel1437608351490,HBReportChannel
(MSG:1437608351490)PublisherAssignment:-GUID:1437608351490-AppID:null-ChannelMap
:{alarmInfo=alarmInfo1437608351490, currentStatus=currentStatus1437608351490}
(SM:1437608351490)[ASSOCIATING]-->[ASSOCIATED]
(OP:1437608351490)<IDLE>==><PUBLISHING>
```

Figure 5-10. System output from the PCA pump (BeagleBone).

5.6.2 PCA pump GUI Overview

5.6.2.1 Status Tab

The Status Tab, depicted in Figure 5-11, lists several basic parameters and the working status of the PCA pump. The clinician can click the Refresh button, and then a "refresh" command will be sent to the PCA pump through Channel #1 in Table 5-2. Command Line #1 in Figure 5-15 indicates

that the PCA pump receives a “refresh” command from the MDCF console. It will then gather the current PCA pump working status and send the most updated status back to the MDCF through Channel #4 in Table 5-2.

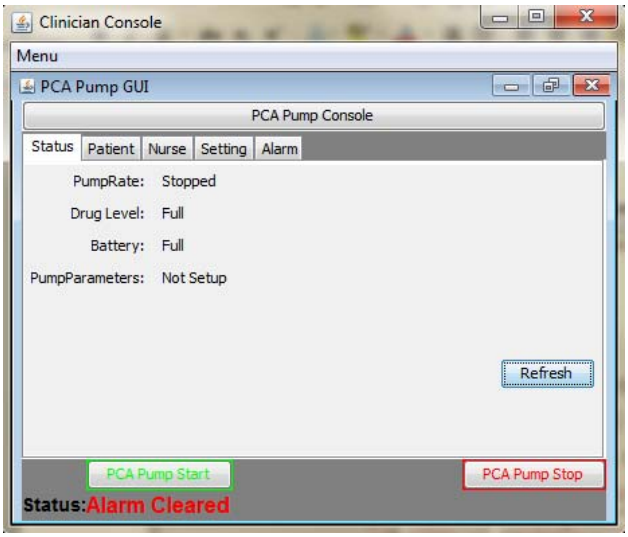

Figure 5-11. PCA pump GUI Status tab.

5.6.2.2 Patient Tab

The patient tab lists basic information about the patient: Sex, Age and Weight as noted in Figure 5-12.

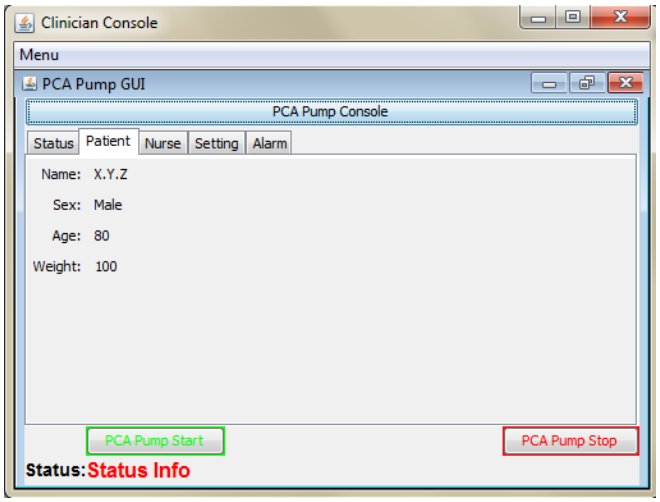

Figure 5-12. PCA pump GUI Patient tab.

5.6.2.3 Nurse Tab

The nurse tab lists basic information about the nurse that locally operates the PCA pump: name and ID, as noted in Figure 5-13.

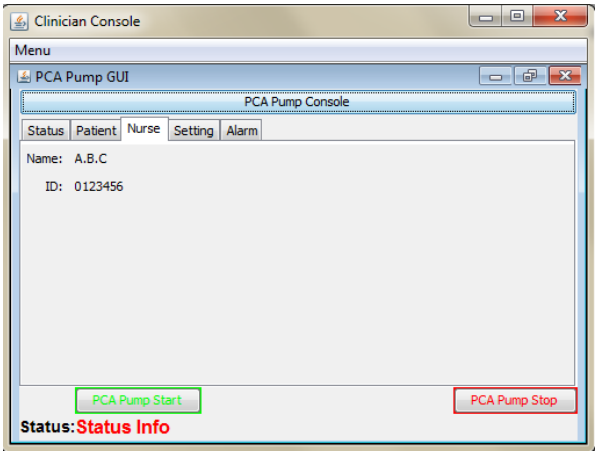

Figure 5-13. PCA pump GUI Nurse tab.

5.6.2.3 Setting Tab

The nurse or clinician can set PCA pump parameters in the Setting tab: pump rate, max rate, min rate and lapse time. After parameters are set, clicking the 'Confirm' button sends a command to the PCAPump to set up the parameters. That is, a String including the parameters listed in Figure 5-14 is sent to the PCA pump through Channel 2 in Table 5-2. Command Line #2 in Figure 5-15 notes the parameters received from the MDCF console.

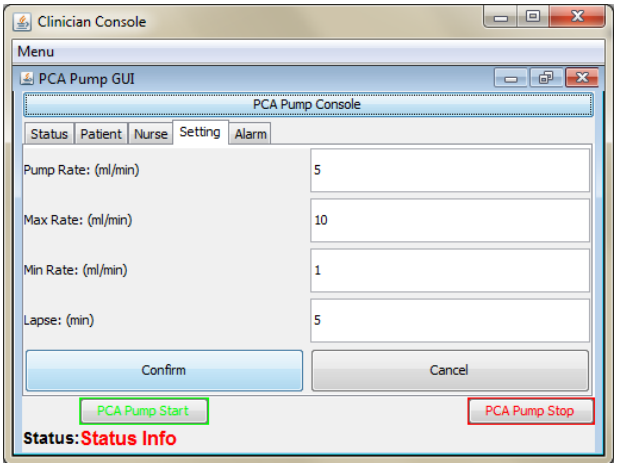

Figure 5-14. PCA pump GUI Setting tab.

5.6.3 PCA Pump Operation

The output lines in Figure 5-15 echo the information from the PCA pump (BeagleBone), the information received from an MDCF console, and the information sent to the MDCF console when operating the PCA pump.

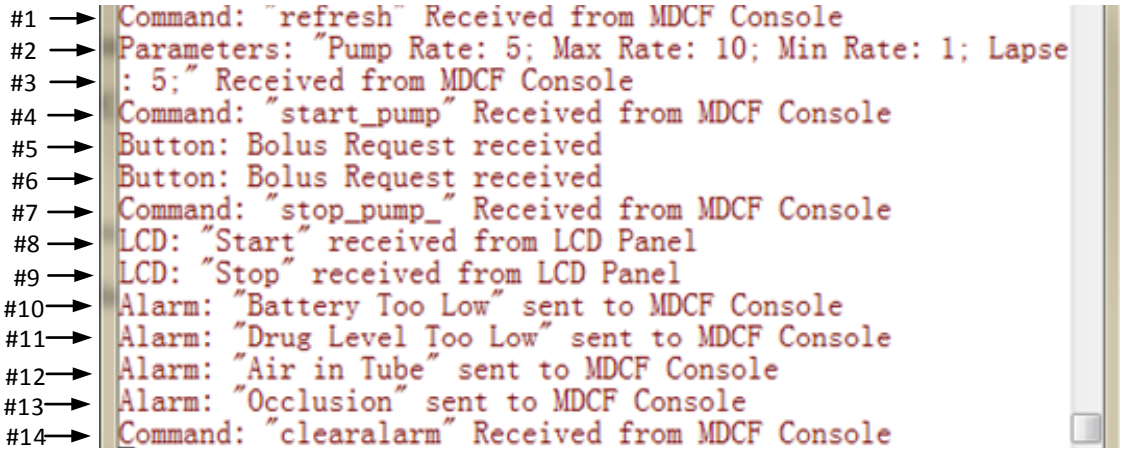

Figure 5-15. System output from the PCA pump (BeagleBone).

5.6.3.1 Starting and Stopping the PCA Pump

After the PCA pump is connected to an MDCF console, the PCA pump can be started or stopped remotely by the MDCF console or locally by the control panel (LCD).

A. In the MDCF console, the clinician clicks the "PCA Pump Start" button on the GUI, and then a "start_pump" command is sent to the PCA pump through Channel #3 in Table 5-2. The Status bar will show that the current status is 'PCA Pump Started' as in Figure 5-16. Command Line #4 in Figure 5-15 indicates that the "**start_pump**" command is received from the MDCF console.

Figure 5-16. Status bar after the PCA pump is started.

After the PCA pump is started, the green LED on the pump module is turned on to indicate the pump status as shown in Figure 5-17.

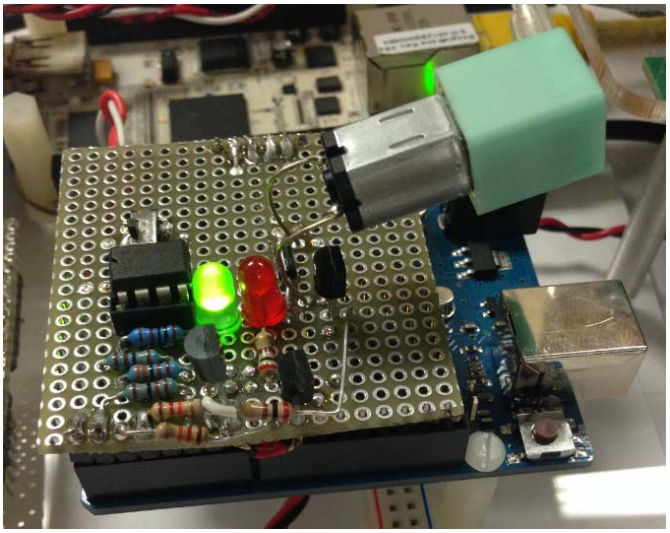

Figure 5-17. Green LED that illuminates after the pump is started.

B. The clinician can also click the "PCA Pump Stop" button to send a "stop_pump" command through the same channel to stop the pump. The Status bar will show that the PCA pump is Stopped as in Figure 5-18. Command Line #7 in Figure 5-15 indicates that the command "stop_pump" is received from MDCFConsole. After the PCA pump is stopped, the green and red LEDs are turned off to indicate the stopped status.

Figure 5-18. Status after the PCA pump is stopped.

C. The PCA pump can also be started by the local control panel (LCD). The local clinician can press the Start button on the LCD to start the PCA pump. The flashing "Pumping" text indicates that the pump is started, as depicted in Figure 5-19. The same status bar is displayed as shown in Figure 5-16. Command Line #8 in Figure 5-15 indicates that PCA pump is started by LCD.

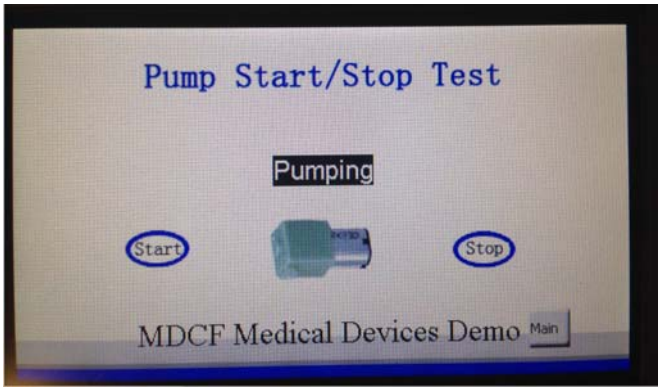

Figure 5-19. LCD interface after the PCA pump is started.

D. The PCA pump can also be stopped by touching the Stop button on the LCD. The "Pump Stopped" text indicates that the pump is stopped, as depicted in Figure 5-20. The same Status bar is displayed as shown in Figure5-18. Command Line #9 in Figure 5-15 indicates that the PCA pump has been stopped by the LCD.

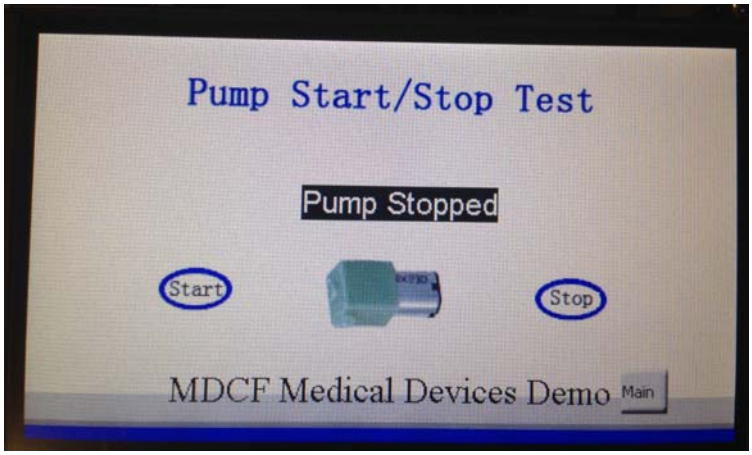

Figure 5-20. LCD interface after the PCA pump is stopped.

5.6.3.2 Patient Bolus Request

When a patient presses the Bolus Request button, the red LED turns on, as in Figure 5-21. A "Bolus Request" message is sent from the patient button module to the BeagleBone at the same time. Command Line #5 in Figure 5-15 indicates that a "Bolus Request" has been received.

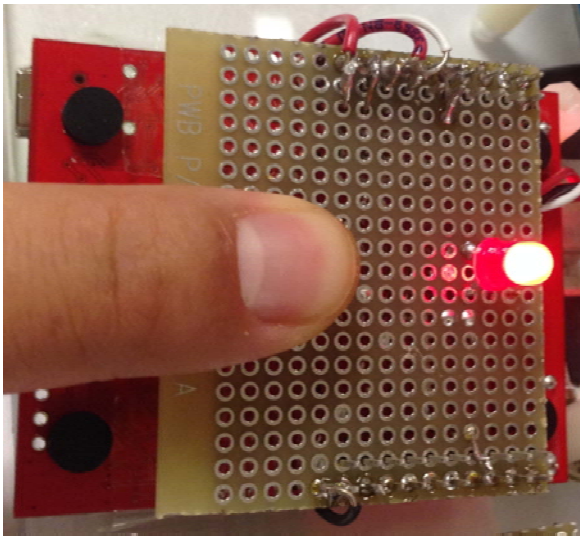

Figure 5-21. Red LED that illuminates after the patient button is pressed.

Even though a "Bolus Request" is sent every time the button is pressed, whether or not this request will be permitted depends on the preset lapse time, the PCA pump working status, etc. Several conditions follow:

A. If the PCA pump has not started, it will not permit a "Bolus Request" and will send an alarm to the MDCF console through channel #5 – this alarm is displayed in Figure 5-22.

Figure 5-22. Status message when a bolus request is not permitted.

B. If the PCA pump is started, either remotely through an MDCF console, or locally though the LCD, the patient can press the button to request an additional bolus. If the request meets the time lapse requirement, the request will be permitted and the PCA pump will deliver

an additional bolus at a higher pump rate. A Bolus Request Permitted message is sent to the MDCF console and displayed as in Figure 5-23. The red LED on the Pump module will be 'on' to indicate that the pump is delivering additional medication at a higher rate (see Figure 5-24), and be 'off' when the pump returns to its normal pumping rate.

Figure 5-23. Status message when a bolus request is permitted.

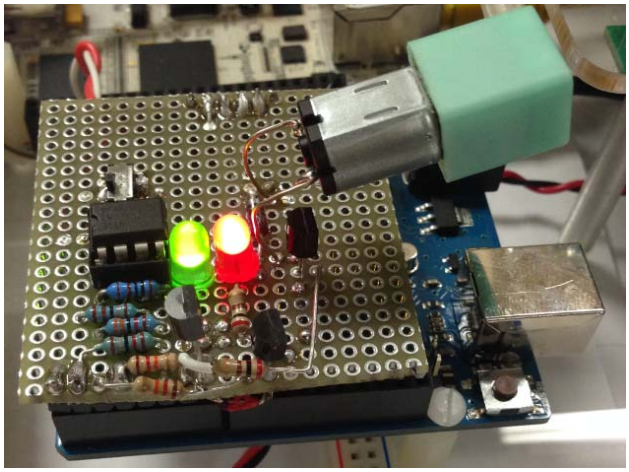

Figure 5-24. Red LED that indicates when a bolus request is permitted.

C. If the Bolus Request is received before the allowable lapsed time, the request will not be permitted and an alarm will be sent to the MDCF console and displayed as in Figure 5-25.

Figure 5-25. Status message when bolus requests occur too frequently.

5.6.3.3 Alarm Simulation

As discussed in previous chapter, four physical buttons on the alarm module can be pressed to simulate different alarms. When a button is pressed to simulate an alarm, the corresponding

LED turns on and an alarm is sent from the alarm module to the PCA pump, and then to the MDCF console by an ICE channel.

Figure 5-26 illustrates that the "Battery Too Low" and "Drug Level Too Low" buttons are pressed and the LEDs are on, so these two alarms are sent from the alarm module to the BeagleBone. Command lines #10 and #12 indicate that "Battery Too Low" and "Drug Level Too Low" alarms have been sent to the MDCF console over Channel #5.

The MDCF console will display these alarms on the GUI, and the most recent alarm will also be displayed on the Status Bar. In this case, "Drug Level Too Low" is displayed as in Figure 5-27. The PCA pump will also make audible alarms, display a red "Light," and flash the alarm on the LCD. A "Battery Too Low" text flashes, and the pump is stopped – see Figure 5-28.

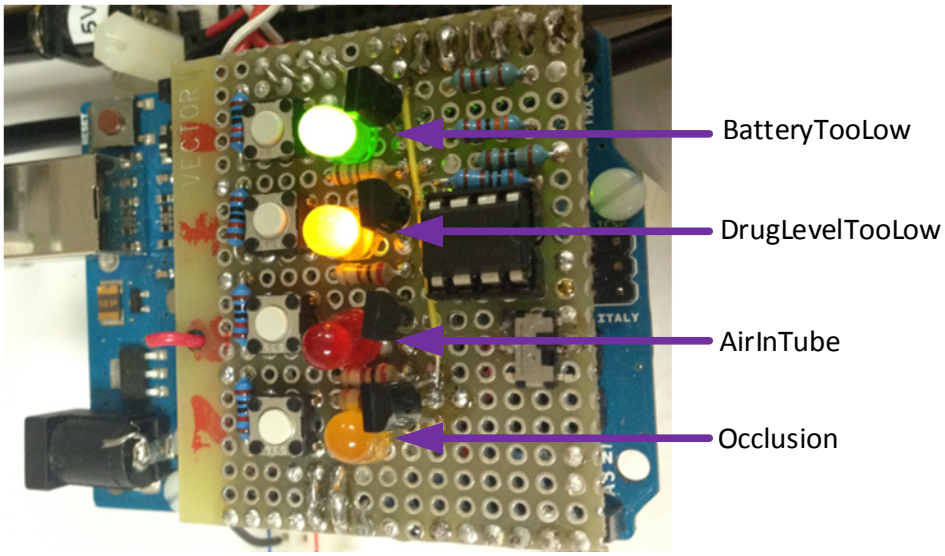

Figure 5-26. Alarms assigned to LEDs and keys.

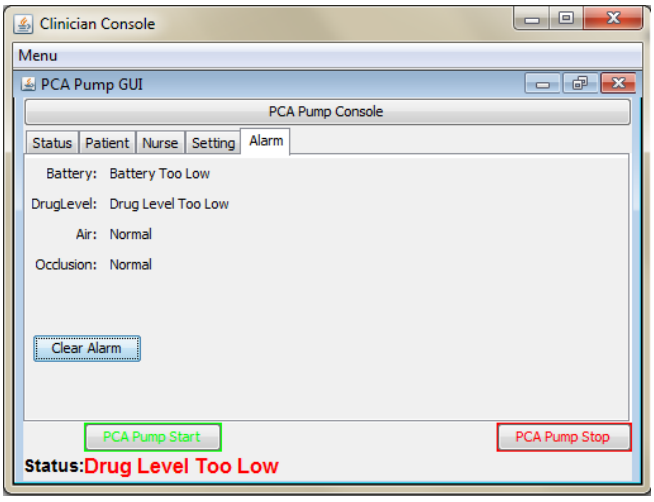

Figure 5-27. Alarms displayed on the MDCF console.

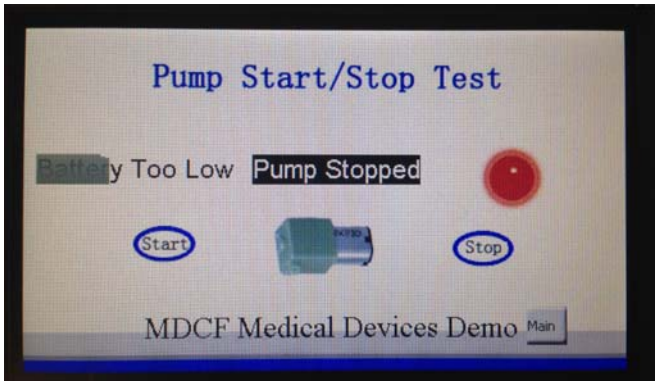

Figure 5-28. Alarm information displayed on the LCD.

If any alarm is received, the clinician should handle this alarm and return the PCA pump to normal working status. Then, the clinician can click the "Clear Alarm" button on the MDCF console and a "clearalarm" command will be sent to the PCA pump through Channel 1. After receiving the "clearalarm" command, the PCA pump turns off all of the LEDs and returns to its normal status. The Status bar then shows that all alarms are cleared, as in Figure 5-29.

Figure 5-29. Status message once the alarm is cleared by the MDCF console.

Chapter 6 - Summary and Future Work

6.1 Summary

In this thesis, a modularized PCA pump was designed with the use of the MDCF PDE as a prototype for an ICE-informed medical device. This design illustrates how to define device and display components and how to create ICE channels using the PDE. ICE channels are implemented by two generated Java file packages. One package implements the display component to communicate with a PCA pump and implements a GUI running within an MDCF console to interact with a clinician. Another package implements a device component to communicate with the MDCF console and implements PCA pump functionality running on a BeagleBone.

The PCA pump in this thesis particularly illustrates a modularized design method for both the hardware modules and the software threads. This modularized PCA pump is designed based on UART ports. Four hardware modules are designed, and four corresponding UART threads are created to communicate with these UART ports. The BeagleBone can send messages to an MDCF console through ICE channels, and it can also receive messages from an MDCF console through ICE channels prior to sending messages to the hardware modules.

This modularized design method can be used to develop medical devices based on UART ports and other communication protocols, such as I^2C, SPI or Bluetooth. The developer can follow the same procedures to design more hardware modules and create corresponding software threads.

6.2 Future Work

The current PCA pump is designed as a simple prototype. Future work can be done:

- Implement more hardware modules and corresponding software threads to build a fully functional PCA pump. The current PCA pump is designed with four hardware modules as a demonstration. More function parts (e.g., sensors and a scanner).

- Use a standard data format, such as IEEE 1451 TEDS for the messages between BaseBoard and modules. Message security should also be considered for future design.
- Select a platform with more features as a BaseBoard, such as an Edison unit that offers a more compact size for wearable medical device design.
- Expand the modularized design method to additional communication protocols, such as I^2C, SPI, and even wireless Bluetooth. The current modularized design method is based on UART, and this method can be more flexible with additional communication protocols.
- Create more ICE channels so that the messages between the PCA pump and an MDCF console could be more organized and meet PCA pump requirements. There are five ICE channels in the current design, yet more ICE channels should be defined with a fully functional PCA pump.
- Develop a software processing unit on the MDCF side to process data from multiple medical devices to demonstrate the features of MDCF and ICE.

References

[1] Goldman, Julian M., S. Whitehead, Sandy Weininger, and M. Rockville. "Eliciting clinical requirements for the medical device plug-and-play (MD PnP) interoperability program." Anesthesia & Analgesia 102, S1-54, 2006.

[2] Etches, Richard C. "Respiratory depression associated with patient-controlled analgesia: a review of eight cases." Canadian journal of anaesthesia, pp. 125-132, 1994.

[3] Hicks, Rodney W., Vanja Sikirica, Winnie Nelson, Jeff R. Schein, and Diane D. Cousins. "Medication errors involving patient-controlled analgesia." American Journal of Health-System Pharmacy, pp. 429-440, 2008.

[4] King, Andrew, Dave Arney, Insup Lee, Oleg Sokolsky, John Hatcliff, and Sam Procter. "Prototyping closed loop physiologic control with the medical device coordination framework." In Proceedings of the 2010 ICSE Workshop on Software Engineering in Health Care, pp. 1-11, ACM, 2010.

[5] Cortès, Philippe-Antoine, Shankar M. Krishnan, Insup Lee, and Julian M. Goldman. "Improving the safety of patient-controlled analgesia infusions with safety interlocks and closed-loop control." In High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability, HCMDSS-MDPnP. Joint Workshop, pp. 149-150, IEEE, 2007.

[6] Kansas State University, SAn-ToS Laboratory. Medical Device Coordination Framework Documentations, http://mdcf.santos.cis.ksu.edu/download.

[7] Andrew King, Sam Procter, Dan Andresen, John Hatcliff, and Steve Warren, “Demonstration of a Medical Device Integration and Coordination Framework,” Proc. 31st Int. Conf. In Software Engineering Companion Volume, pp. 433–434, 2009.

[8] King, Andrew, Sam Procter, Dan Andresen, John Hatcliff, Steve Warren, William Spees, Raoul Jetley, Paul Jones, and Sandy Weininger. "An open test bed for medical device integration and coordination." In Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference, pp. 141-151. IEEE, 2009.

[9] Li, Kejia, Steve Warren, and John Hatcliff. "Component-based app design for platform-oriented devices in a medical device coordination framework." In Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium, pp. 343-352. ACM, 2012.

[10] Grass, Jeffrey A. "Patient-controlled analgesia." Anesthesia & Analgesia 101, no. 5S S44-S61, 2005.

[11] Checketts, M. R., C. J. Gilhooly, and G. N. Kenny. "Patient-maintained analgesia with target-controlled alfentanil infusion after cardiac surgery: a comparison with morphine PCA." British journal of anaesthesia 80, no. 6: 748-751, 1998.

[12] Henriksen, Kerm, James B. Battles, Margaret A. Keyes, Mary L. Grady, Ray R. Maddox, Harold Oglesby, Carolyn K. Williams, Marianne Fields, and Sherry Danello. "Continuous Respiratory Monitoring and a “Smart” Infusion System Improve Safety of Patient-Controlled Analgesia in the Postoperative Period," Advances in Patient Safety: New Directions and Alternative Approaches, Vol. 4, p.9, 2008.

[13] Rudolph, Heiko, John S. Packer, John F. Cade, Brian Lee, and Peter Morley. "Pain Relief Using Smart Technology: An Overview of a New Patient-Controlled Analgesia Device," IEEE Trans Information Technology in Biomedicine, Vol. 3, pp. 20–27, March 1999.

[14] Henriksen, Kerm, James B. Battles, Margaret A. Keyes, Mary L. Grady, Ray R. Maddox, Harold Oglesby, Carolyn K. Williams, Marianne Fields, and Sherry Danello. "Continuous Respiratory Monitoring and a “Smart” Infusion System Improve Safety of Patient-Controlled Analgesia in the Postoperative Period," Advances in Patient Safety: New Directions and Alternative Approaches, Vol. 4, p.9, 2008.

[15] Larson, Brian R., John Hatcliff, and Patrice Chalin. “Open Source Patient-Controlled Analgesic Pump Requirements Documentation,” SEHC 2013, 5th International Workshop on Software Engineering in Health Care, pp. 28–34, May 20–21, 2013

[16] Yang, Kai-Chao, Yu-Tsang Chang, Chien-Ming Wu, and Chun-Ming Huang. "Modularized development platform for hardware/software design." In Integrated Circuits (ISIC), 2011 13th International Symposium, pp. 492-495. IEEE, 2011.

[17] LIU, Jian-jun, Xiao-dong ZHENG, Hong-guang CHEN, Jing LI, and Shu-xin LIU. "Application study on modularization design of operation instrument set used in field medical point [J]." Chinese Medical Equipment Journal 11, 041, 2006.

[18] Wikipedia, “BeagleBoard”, https://en.wikipedia.org/wiki/BeagleBoard.

[19] Wikipedia, “Arduino”, https://en.wikipedia.org/wiki/Arduino.

[20] Wikipedia, “TI MSP430”, https://en.wikipedia.org/wiki/TI_MSP430

[21] Wikipedia, “UART”, https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter.

[22] “IEEE Standard for a Smart Transducer Interface for Sensors and Actuators-Digital Communication and Transducer Electronic Data Sheet (TEDS) Formats for Distributed Multidrop Systsms,” IEEE Std 1451.3-2003, pp. 1–175, 2004.

[23] Markovic, Dejan, Uros Pesovic, and Sinisa Randic. "TEDS specification for IEEE 1451.0 smart Transducer." In Telecommunications Forum (TELFOR), pp. 1532-1535. IEEE, 2012.

Appendix A - PCAPump.jar Full Code

This appendix is the code of PCAPump.jar that implements the functionality discussed in this thesis, including ICE channels to communicate with an MDCF console and multi-threads to communicate with hardware modules.

```
/*
The PCAPump.jar file implement the ICE channel code that is automatically generated by MDCF/PDE, and the functionality code for PCA pump.
The developer can call the ICE channel functions to send messages to remote MDCF console, and receive messages from MDCF console.
For the modularized design, multiple threads are created by defining Object of rxtx. These threads are listening to the hardware modules to detect messages by inputstream, and use outputstream to write messages to hardware modules.

Author Shiwei Luan, Electrical and Computer Engineering Department, Kansas State University
Aug.12 2015
Developed with Java 1.7, Eclipse Juno on Windows 7.
*/
import mdcf.channelservice.common.IMdcfMessageListener;
import mdcf.channelservice.common.IMdcfReceiver;
import mdcf.channelservice.common.IMdcfSender;
import mdcf.channelservice.common.MdcfMessage;
import mdcf.channelservice.common.ReceiverFactory;
import mdcf.channelservice.common.SenderFactory;
import mdcf.core.ctypes.device.DeviceComponent;
import mdcf.core.messagetypes.devicemgmt.PubChannelAssignmentMsg;
import mdcf.core.messagetypes.devicemgmt.SubChannelAssignmentMsg;

import gnu.io.CommPort;
import gnu.io.CommPortIdentifier;
import gnu.io.SerialPort;
import gnu.io.SerialPortEvent;
import gnu.io.SerialPortEventListener;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public final class PCAPump extends DeviceComponent {

	/*
	 * AUTO-GENERATED CODE OF THE DATAINTERFACES
	 */
	IMdcfSender currentStatusSender = null;
	IMdcfSender alarmInfoSender = null;

	IMdcfReceiver getStatusReceiver = null;
```

```
IMdcfReceiver setParameterReceiver = null;
IMdcfReceiver cmdStartStopReceiver = null;

/*
 * USER DEFINED FIELDS
 */
rxtx uart_alarm;
rxtx uart_power;
rxtx uart_button;
rxtx uart_LCD;
rxtx uart_pump;

String cmd_string_to_LCD; // t "Alarm Info" 10 100 N. \n required by SLCD
                                                                    // cmd
String cmd_picture_to_LCD; // display red alarm
String cmd_beep_to_LCD; // display alarm beep
String cmd_to_LCD;
String cmd_clear_to_LCD; // clear flashing text, beep, alarm etc.

public boolean pump_started = false;

// set for PWM
static String cmdset1 = "echo 6 > /sys/kernel/debug/omap_mux/gpmc_a2";
static String cmdset2 = "echo 6 > /sys/kernel/debug/omap_mux/gpmc_a3";

static String cmdstop = "echo 0 > /sys/class/pwm/ehrpwm.1:0/run";
static String cmdcleardutypercent = "echo 0 > /sys/class/pwm/ehrpwm.1:0/duty_percent";
static String cmdperiodhighfreq = "echo 10 > /sys/class/pwm/ehrpwm.1:0/period_freq";


static String cmdperiodlowfreq = "echo 1 > /sys/class/pwm/ehrpwm.1:0/period_freq";
static String cmddutypercent = "echo 50 > /sys/class/pwm/ehrpwm.1:0/duty_percent";
static String cmddutypercent_10 = "echo 10 > /sys/class/pwm/ehrpwm.1:0/duty_percent";
static String cmddutypercent_50 = "echo 50 > /sys/class/pwm/ehrpwm.1:0/duty_percent";
static String cmddutypercent_80 = "echo 80 > /sys/class/pwm/ehrpwm.1:0/duty_percent";
static String cmdrun = "echo 1 > /sys/class/pwm/ehrpwm.1:0/run";

static ProcessBuilder pbset1 = new ProcessBuilder("bash", "-c", cmdset1);
static ProcessBuilder pbset2 = new ProcessBuilder("bash", "-c", cmdset2);
static ProcessBuilder pbstop = new ProcessBuilder("bash", "-c", cmdstop);
static ProcessBuilder pbcleardutypercent = new ProcessBuilder("bash", "-c",
                cmdcleardutypercent);
static ProcessBuilder pbperiodhighfreq = new ProcessBuilder("bash", "-c",
                cmdperiodhighfreq);
static ProcessBuilder pbperiodlowfreq = new ProcessBuilder("bash", "-c",
                cmdperiodlowfreq);
static ProcessBuilder pbdutypercent = new ProcessBuilder("bash", "-c",
                cmddutypercent);
static ProcessBuilder pbdutypercent_10 = new ProcessBuilder("bash", "-c",
                cmddutypercent_10);
static ProcessBuilder pbdutypercent_50 = new ProcessBuilder("bash", "-c",
                cmddutypercent_50);
static ProcessBuilder pbdutypercent_80 = new ProcessBuilder("bash", "-c",
                cmddutypercent_80);
static ProcessBuilder pbrun = new ProcessBuilder("bash", "-c", cmdrun);
```

```
// clear UART lock files. Different platform may have different lock files
static String uart1_lockfile = "rm /var/lock/LCK..ttyO1";
static String uart2_lockfile = "rm /var/lock/LCK..ttyO2";
static String uart4_lockfile = "rm /var/lock/LCK..ttyO4";
static String uart5_lockfile = "rm /var/lock/LCK..ttyO5";

static ProcessBuilder uart1_lockfile_clear = new ProcessBuilder("bash",
                        "-c", uart1_lockfile);
static ProcessBuilder uart2_lockfile_clear = new ProcessBuilder("bash",
                        "-c", uart2_lockfile);
static ProcessBuilder uart4_lockfile_clear = new ProcessBuilder("bash",
                        "-c", uart4_lockfile);
static ProcessBuilder uart5_lockfile_clear = new ProcessBuilder("bash",
                        "-c", uart5_lockfile);

// set for UART1
static String uart1_rx = "echo 20 > /sys/kernel/debug/omap_mux/uart1_rxd";
static String uart1_tx = "echo 0 > /sys/kernel/debug/omap_mux/uart1_txd";

// set for UART2
static String uart2_rx = "echo 21 > /sys/kernel/debug/omap_mux/spi0_sclk";
static String uart2_tx = "echo 1 > /sys/kernel/debug/omap_mux/spi0_d0";

// set for UART4
static String uart4_rx = "echo 26 > /sys/kernel/debug/omap_mux/gpmc_wait0";
static String uart4_tx = "echo 6 > /sys/kernel/debug/omap_mux/gpmc_wpn";
// set for UART5
static String uart5_rx = "echo 24 > /sys/kernel/debug/omap_mux/lcd_data9";
static String uart5_tx = "echo 4 > /sys/kernel/debug/omap_mux/lcd_data8";

static ProcessBuilder uart1_rx_set = new ProcessBuilder("bash", "-c",
                        uart1_rx);
static ProcessBuilder uart1_tx_set = new ProcessBuilder("bash", "-c",
                        uart1_tx);
static ProcessBuilder uart2_rx_set = new ProcessBuilder("bash", "-c",
                        uart2_rx);
static ProcessBuilder uart2_tx_set = new ProcessBuilder("bash", "-c",
                        uart2_tx);
static ProcessBuilder uart4_rx_set = new ProcessBuilder("bash", "-c",
                        uart4_rx);
static ProcessBuilder uart4_tx_set = new ProcessBuilder("bash", "-c",
                        uart4_tx);
static ProcessBuilder uart5_rx_set = new ProcessBuilder("bash", "-c",
                        uart5_rx);
static ProcessBuilder uart5_tx_set = new ProcessBuilder("bash", "-c",
                        uart5_tx);

/*
 * AUTO-GENERATED CONSTRUCTOR
 */
public PCAPump() {
        super("1437608351490", "PCAPump");
}

/*
 * AUTO-GENERATED DEVICE DRIVER'S ENTRY FUNCTION
```

```
 */
public static void main(String[] args) {
	System.out.println("Main start");
	// configure PWM and UART
	try {
		pbset1.start();
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
	try {
		pbset2.start();
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
	try {
		pbcleardutypercent.start();
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}

	// clear UART lock files
	try {
		uart1_lockfile_clear.start();
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
	try {
		uart2_lockfile_clear.start();
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
	try {
		uart4_lockfile_clear.start();
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
	try {
		uart5_lockfile_clear.start();
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}

	// configure UART
	try {
		uart1_rx_set.start();
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
```

```
		try {
			uart1_tx_set.start();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		try {
			uart2_rx_set.start();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		try {
			uart2_tx_set.start();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		try {
			uart4_rx_set.start();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		try {
			uart4_tx_set.start();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		try {
			uart5_rx_set.start();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		try {
			uart5_tx_set.start();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		PCAPump dev = new PCAPump();
		dev.connectToNetworkController();

		System.out.println("Main end");
}

/*
 * AUTO-GENERATED CODE FOR INIT USER NEEDS TO FILL IN THE METHOD FOR
 * DEVICE'S INIT
 */
public void init() {
		uart_alarm = new rxtx();
		uart_power = new rxtx();
```

```
		uart_button = new rxtx();
		uart_pump = new rxtx();
		uart_LCD = new rxtx();

		try { // connect alarm to UART1;
			uart_alarm.connect("/dev/ttyO1");
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		try { // connect power to UART2;
			uart_power.connect("/dev/ttyO2");
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		try { // connect button to UART4;
			uart_button.connect("/dev/ttyO4");
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		try { // connect pump to UART5;
			uart_pump.connect("/dev/ttyO5");
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		try { // connect LCD to USB/Serial Adapter;
			uart_LCD.connect("/dev/ttyUSB0");
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		(new Thread(new alarmInfoListener())).start();
		(new Thread(new LCDInfoListener())).start();
		(new Thread(new buttonRequestListener())).start();
}

/*
 * AUTO-GENERATED CODE FOR START PUBLISHING USER NEEDS TO FILL IN THE METHOD
 * FOR START PUBLISHING
 */
public void start_publish() {

}

/*
 * AUTO-GENERATED CODE FOR STOP PUBLISHING USER NEEDS TO FILL IN THE METHOD
 * FOR STOP PUBLISHING
 */
```

```
public void stop_publish() {

}

/*
 * AUTO-GENERATED CODE FOR PUBLISHER CHANNEL ASSIGNMENT
 */
public void processPublisherChannelAssignment(
                PubChannelAssignmentMsg pubAssign) {
        currentStatusSender = SenderFactory.createSender();
        currentStatusSender.connect(pubAssign.channelMap.get("currentStatus"));
        alarmInfoSender = SenderFactory.createSender();
        alarmInfoSender.connect(pubAssign.channelMap.get("alarmInfo"));
}

/*
 * AUTO-GENERATED CODE FOR SUBSCRIBER CHANNEL ASSIGNMENT
 */
public void processSubscriberChannelAssignment(
                SubChannelAssignmentMsg subAssign) {
        if (subAssign.channelMap.get("getStatus") != null) {
                getStatusReceiver = ReceiverFactory.createReceiver();
                getStatusReceiver.connect(subAssign.channelMap.get("getStatus"));
                getStatusReceiver.registerMessageListener(new getStatusListener());
        }
        if (subAssign.channelMap.get("setParameter") != null) {
                setParameterReceiver = ReceiverFactory.createReceiver();
                setParameterReceiver.connect(subAssign.channelMap
                                .get("setParameter"));
                setParameterReceiver
                                .registerMessageListener(new setParameterListener());
        }
        if (subAssign.channelMap.get("cmdStartStop") != null) {
                cmdStartStopReceiver = ReceiverFactory.createReceiver();
                cmdStartStopReceiver.connect(subAssign.channelMap
                                .get("cmdStartStop"));
                cmdStartStopReceiver
                                .registerMessageListener(new cmdStartStopListener());
        }

}

/*
 * AUTO-GENERATED MESSAGE HANDLER
 */
class getStatusListener implements IMdcfMessageListener {
        @Override
        public void onMessage(MdcfMessage message) {
                /*
                 * AUTO-GENERATED DATA CONVERTER
                 */
                String getStatusData = null;
                try {
                        getStatusData = message.getTextMsg();
                } catch (NumberFormatException e) {
                        System.err.println("getStatusListener:NumberFormatException:"
```

```
                        + message.getTextMsg());
            e.printStackTrace();
}

/*
 * USER DEFINED HANDLING
 */

if (getStatusData.equals("refresh")) {
            System.out.println("Command: \"" + getStatusData
                        + "\" Received from MDCF Console");
}

if (getStatusData.equals("clearalarm")) {
            for (int i = 0; i < 10; i++) {
                        try {
                                    uart_alarm.out.write(getStatusData.charAt(i));
                        } catch (IOException e) {
                                    // TODO Auto-generated catch block
                                    e.printStackTrace();
                        }
            }
            // clear LCD alarm
            cmd_clear_to_LCD = "tfd 1 1";
            for (int i = 0; i < cmd_clear_to_LCD.length(); i++) {
                        try {
                                    uart_LCD.out.write(cmd_clear_to_LCD.charAt(i));
                        } catch (IOException e) {
                                    // TODO Auto-generated catch block
                                    e.printStackTrace();
                        }
            }
            cmd_clear_to_LCD = "tfx 1";
            for (int i = 0; i < cmd_clear_to_LCD.length(); i++) {
                        try {
                                    uart_LCD.out.write(cmd_clear_to_LCD.charAt(i));
                        } catch (IOException e) {
                                    // TODO Auto-generated catch block
                                    e.printStackTrace();
                        }
            }
            cmd_clear_to_LCD = "tfx 0";
            for (int i = 0; i < cmd_clear_to_LCD.length(); i++) {
                        try {
                                    uart_LCD.out.write(cmd_clear_to_LCD.charAt(i));
                        } catch (IOException e) {
                                    // TODO Auto-generated catch block
                                    e.printStackTrace();
                        }
            }
            cmd_to_LCD = "m 5" + "\r\n";
            for (int i = 0; i < cmd_to_LCD.length(); i++) {
                        try {
                                    uart_LCD.out.write(cmd_to_LCD.charAt(i));
                        } catch (IOException e) {
                                    // TODO Auto-generated catch block
```

```
                                        e.printStackTrace();
                                }
                        }
                        cmd_to_LCD = "rb 0 0" + "\r\n"; // stop beep
                        for (int i = 0; i < cmd_to_LCD.length(); i++) {
                                try {
                                        uart_LCD.out.write(cmd_to_LCD.charAt(i));
                                } catch (IOException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                        }
                        cmd_to_LCD = "beep 500" + "\r\n"; // beep once
                        for (int i = 0; i < cmd_to_LCD.length(); i++) {
                                try {
                                        uart_LCD.out.write(cmd_to_LCD.charAt(i));
                                } catch (IOException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                        }
                        System.out.println("Command: \"" + getStatusData
                                        + "\" Received from MDCF Console");
                }
        }
}

/*
 * AUTO-GENERATED MESSAGE HANDLER
 */
class setParameterListener implements IMdcfMessageListener {
        @Override
        public void onMessage(MdcfMessage message) {
                /*
                 * AUTO-GENERATED DATA CONVERTER
                 */
                String setParameterData = null;
                try {
                        setParameterData = message.getTextMsg();
                } catch (NumberFormatException e) {
                        System.err
                                        .println("setParameterListener:NumberFormatException:"
                                                        + message.getTextMsg());
                        e.printStackTrace();
                }

                /*
                 * USER DEFINED HANDLING
                 */
                System.out.println("Parameters: \"" + setParameterData
                                + "\" Received from MDCF Console");
        }

}

/*
```

```
 * AUTO-GENERATED MESSAGE HANDLER
 */
class cmdStartStopListener implements IMdcfMessageListener {

	String start_cmd = "start_pump";
	String stop_cmd = "stop_pump_";

	@Override
	public void onMessage(MdcfMessage message) {
		/*
		 * AUTO-GENERATED DATA CONVERTER
		 */
		String cmdStartStopData = null;
		try {
			cmdStartStopData = message.getTextMsg();
		} catch (NumberFormatException e) {
			System.err
					.println("cmdStartStopListener:NumberFormatException:"
							+ message.getTextMsg());
			e.printStackTrace();
		}

		/*
		 * USER DEFINED HANDLING
		 */

		if (cmdStartStopData.equals("start")) {
			System.out.println("Command: \"" + start_cmd
					+ "\" Received from MDCF Console");
			for (int i = 0; i < 10; i++) {
				try {
					uart_pump.out.write(start_cmd.charAt(i));
				} catch (IOException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				}
			}
			pump_started = true;
		}
		if (cmdStartStopData.equals("stop")) {
			System.out.println("Command: \"" + stop_cmd
					+ "\" Received from MDCF Console");
			for (int i = 0; i < 10; i++) {
				try {
					uart_pump.out.write(stop_cmd.charAt(i));
				} catch (IOException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				}
			}
			pump_started = false;
		}
	}
}

/*
```

```
	 * AUTO-GENERATED MESSAGE TYPE CONVERTER METHOD
	 */
	private void send_currentStatus(String data) {
		currentStatusSender.sendMessage(data);
	}/*
	 * AUTO-GENERATED MESSAGE TYPE CONVERTER METHOD
	 */

	private void send_alarmInfo(String data) {
		alarmInfoSender.sendMessage(data);
	}

	public class rxtx {

		volatile boolean rxtxmsgreceived = false; // to indicate if there's msg

	// from serial port;
		volatile String rxtxmsg; // used to save msg from serial port;
		OutputStream out; // used to write msg to serial port;
		InputStream in;

		public rxtx() {
			super();
		}

		void connect(String portName) throws Exception {
			CommPortIdentifier portIdentifier = CommPortIdentifier
					.getPortIdentifier(portName);
			if (portIdentifier.isCurrentlyOwned()) {
				System.out.println("Error: Port is currently in use");
			} else {
				CommPort commPort = portIdentifier.open(this.getClass()
						.getName(), 2000);

				if (commPort instanceof SerialPort) {
					SerialPort serialPort = (SerialPort) commPort;
					serialPort.setSerialPortParams(9600, SerialPort.DATABITS_8,
							SerialPort.STOPBITS_1,
SerialPort.PARITY_NONE);

					InputStream in = serialPort.getInputStream();
					OutputStream out = serialPort.getOutputStream();

					this.out = out;
					this.in = in;

					SerialReader serialreader = new SerialReader(in, this);
					serialPort.addEventListener(serialreader);
					serialPort.notifyOnDataAvailable(true);
					System.out
							.println("++++++++++++++Com connected
successfully++++++++++++++");
				} else {
					System.out
							.println("Error: Only serial ports are handled by this
example.");
```

```
                        }
                    }
                }
            }

public class SerialReader implements SerialPortEventListener {
        private InputStream in;
        private byte[] buffer = new byte[1024];
        rxtx uart_x;

        public SerialReader(InputStream in, rxtx uart_x) {
            this.in = in;
            this.uart_x = uart_x;
        }

        public void serialEvent(SerialPortEvent arg0) {
            int data;

            try {
                int len = 0;
                while ((data = in.read()) > -1) {
                    if ((data == '1') | (data == '\r') | (data == '\n')) { // end

                        // with

                        // 1
                        break;
                    }
                    buffer[len++] = (byte) data;
                }

                uart_x.rxtxmsg = (new String(buffer, 0, len));
                uart_x.rxtxmsgreceived = true;
            } catch (IOException e) {
                e.printStackTrace();
                System.exit(-1);
            }
        }
}

public class alarmInfoListener implements Runnable {

        public void run() {
            while (true) {
                if (uart_alarm.rxtxmsgreceived) {
                    System.out.println("Alarm: \"" + uart_alarm.rxtxmsg
                            + "\" sent to MDCF Console");
                    uart_alarm.rxtxmsgreceived = false;
                    send_alarmInfo(uart_alarm.rxtxmsg);
                    // Stop PCA pump
                    for (int i = 0; i < 10; i++) {
                        try {
                            uart_pump.out.write("stop_pump_".charAt(i));
                        } catch (IOException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
```

```
		}
	}
	pump_started = false;

	cmd_string_to_LCD = "tf 1 200 \"" + uart_alarm.rxtxmsg
			+ "\" 10" + " 100" + " R" + "\r\n";
	cmd_picture_to_LCD = "xi 12 360 90" + "\r\n";
	cmd_beep_to_LCD = "rb 100 500" + "\r\n";

	// send alarm info to LCD panel
	cmd_clear_to_LCD = "tfd 1 1";
	for (int i = 0; i < cmd_clear_to_LCD.length(); i++) {
		try {
			uart_LCD.out.write(cmd_clear_to_LCD.charAt(i));
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
	cmd_clear_to_LCD = "tfx 1";
	for (int i = 0; i < cmd_clear_to_LCD.length(); i++) {
		try {
			uart_LCD.out.write(cmd_clear_to_LCD.charAt(i));
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
	cmd_clear_to_LCD = "m 7" + "\r\n";
	for (int i = 0; i < cmd_clear_to_LCD.length(); i++) {
		try {
			uart_LCD.out.write(cmd_clear_to_LCD.charAt(i));
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
	for (int i = 0; i < cmd_string_to_LCD.length(); i++) {
		try {
			uart_LCD.out.write(cmd_string_to_LCD.charAt(i));
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
	for (int i = 0; i < cmd_picture_to_LCD.length(); i++) {

		try {
			uart_LCD.out.write(cmd_picture_to_LCD.charAt(i));
			// System.out.print(stop_cmd.charAt(i));
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
	for (int i = 0; i < cmd_beep_to_LCD.length(); i++) {
```

```
                                                try {
                                                        uart_LCD.out.write(cmd_beep_to_LCD.charAt(i));
                                                        // System.out.print(stop_cmd.cmd_beep_to_LCD(i));
                                                } catch (IOException e) {
                                                        // TODO Auto-generated catch block
                                                        e.printStackTrace();
                                                }
                                        }
                                }
                        }
                }
        }

        public class LCDInfoListener implements Runnable {

                public void run() {
                        while (true) {
                                if (uart_LCD.rxtxmsgreceived) { // Start received fromLCD Panel
                                        uart_LCD.rxtxmsgreceived = false;
                                        if (uart_LCD.rxtxmsg.equals("88")) {
                                                System.out
                                                                .println("LCD: \"Start\" received from LCD
Panel");
                                                for (int i = 0; i < 10; i++) {
                                                        try {

        uart_pump.out.write("start_pump".charAt(i));
                                                        } catch (IOException e) {
                                                                // TODO Auto-generated catch block
                                                                e.printStackTrace();
                                                        }
                                                }
                                                pump_started = true;
                                                send_alarmInfo("PCA Pump Started");
                                        }
                                        if (uart_LCD.rxtxmsg.equals("99")) {// Start received

        // fromLCD Panel
                                                System.out
                                                                .println("LCD: \"Stop\" received from LCD
Panel");
                                                for (int i = 0; i < 10; i++) {
                                                        try {

        uart_pump.out.write("stop_pump_".charAt(i));
                                                        } catch (IOException e) {
                                                                // TODO Auto-generated catch block
                                                                e.printStackTrace();
                                                        }
                                                }
                                                pump_started = false;
                                                send_alarmInfo("PCA Pump Stopped");
                                        }
                                }
                        }
                }
```

```
}

public class buttonRequestListener implements Runnable {

	String tooFrequentRequest = "Too Frequent Bolus Request";
	SimpleDateFormat tempDate = new SimpleDateFormat("MM/dd/yyyy HH:mm:ss");
	SimpleDateFormat format = new SimpleDateFormat("MM/dd/yyyy HH:mm:ss");

	Date previous_request_time_seconds = null;
	Date current_request_time_seconds = null;

	String previous_request_time = null;
	String current_request_time = null;

	long time_lapse;

	public void run() {
		previous_request_time = tempDate.format(new java.util.Date());
		try {
			previous_request_time_seconds = format
					.parse(previous_request_time);
		} catch (ParseException e1) {
			// TODO Auto-generated catch block
			e1.printStackTrace();
		}
		while (true) { // pump_started = true;
			if (uart_button.rxtxmsgreceived) {

				if (pump_started) {

					System.out.println("Button: " + uart_button.rxtxmsg
							+ " received");
					uart_button.rxtxmsgreceived = false;

					current_request_time = tempDate
							.format(new java.util.Date());

					try {
						current_request_time_seconds = format
								.parse(current_request_time);

					} catch (Exception e) {
						e.printStackTrace();
					}

					time_lapse = current_request_time_seconds.getTime()
							/ 1000
							- previous_request_time_seconds.getTime()
							/ 1000;

					if (time_lapse > 10) { // define time lapse in seconds
						send_currentStatus(uart_button.rxtxmsg);
						previous_request_time_seconds =
	current_request_time_seconds;

						for (int i = 0; i < 10; i++) {
```

```
                                                                try {

                                                        uart_pump.out.write("button_req".charAt(i));
                                                                } catch (IOException e) {
                                                                        // TODO Auto-generated catch
block
                                                                        e.printStackTrace();
                                                                }
                                                        }
                                                        send_alarmInfo("Bolus Request Permitted");
                                                } else
                                                        send_alarmInfo(tooFrequentRequest);
                                        }

                                        else
                                                send_alarmInfo("PCA Pump not started. Bolus request
disabled.");
                                }
                        }
                }
        }

        public class pumpControl implements Runnable {
                public void run() { // All other threads can directly write command to
                                                        // Pump module to start or stop pump.

                }
        }
}
```

Appendix B - BeagleBone Expansion Headers

Table B-1. Pinout for the P8 expansion header.

Signal Name	Pin numbers	Pin numbers	Signal Name
GND	1	2	GND
GPIO1_6	3	4	GPIO1_7
GPIO1_2	5	6	GPIO1_3
TIMER4	7	8	TIMER7
TIMER5	9	10	TIMER6
GPIO1_13	11	12	GPIO1_12
EHRPWM2B	13	14	GPIO0_26
GPIO1_15	15	16	GPIO1_14
GPIO0_27	17	18	GPIO2_1
EHRPWM2A	19	20	GPIO1_31
GPIO1_30	21	22	GPIO1_5
GPIO1_4	23	24	GPIO1_1
GPIO1_0	25	26	GPIO1_29
GPIO2_22	27	28	GPIO2_24
GPIO2_23	29	30	GPIO2_25
UART5_CTSN	31	32	UART5_RTSN
UART4_RTSN	33	34	UART3_RTSN
UART4_CTSN	35	36	UART3_CTSN
UART5_TXD	37	38	UART5_RXD
GPIO2_12	39	40	GPIO2_13
GPIO2_10	41	42	GPIO2_11
GPIO2_8	43	44	GPIO2_9
GPIO2_6	45	46	PIO2_7

Table B-2. Pinout for the P9 expansion header.

Signal Name	Pin numbers	Pin numbers	Signal Name
GND	1	2	GND
VDD_3V3EXP	3	4	VDD_3V3EXP
VDD_5V	5	6	VDD_5V
SYS_5V	7	8	SYS_5V
PWR_BUT*	9	10	SYS_RESETn
UART4_RXD	11	12	GPIO1_28
UART4_TXD	13	14	EHRPWM1A
GPIO1_16	15	16	EHRPWM1B
I2C1_SCL	17	18	I2C1_SDA
I2C2_SCL	19	20	I2C2_SDA
UART2_TXD	21	22	UART2_RXD
GPIO1_17	23	24	UART1_TXD
GPIO3_21	25	26	UART1_RXD
GPIO3_19	27	28	SPI1_CS0
SPI1_D0	29	30	SPI1_D1
SPI1_SCLK	31	32	VDD_ADC(1.8V)
AIN4	33	34	GNDA_ADC
AIN6	35	36	AIN5
AIN2	37	38	AIN3
AIN0	39	40	AIN1
CLKOUT2	41	42	GPIO0_7
GND	43	44	GND
GND	45	46	GND

Appendix C - Hardware Parts List

Table C-1. Hardware Parts List

Part	Value	Figure
R1	1K	Figure 4-4
R2	1.8K	Figure 4-4
R3	33K	Figure 4-4
R4	18K	Figure 4-4
D	IN4001	Figure 4-7
R	12K	Figure 4-7, 4-8, 4-10, 4-12
Q	PN4141	Figure 4-7, 4-8, 4-10, 4-12
D1	GREEN LED	Figure 4-8
D2	RED LED	Figure 4-8
Rp	10K	Figure 4-10, 4-12