ST. JUDE MEDICAL:
AN OBJECT-ORIENTED SOFTWARE ARCHITECTURE FOR
EMBEDDED AND REAL-TIME MEDICAL DEVICES

A Thesis
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Biomedical Engineering

by
Atila Amiri

July 2010

© 2010

Atila Amiri

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	ST. JUDE MEDICAL: AN OBJECT-ORIENTED SOFTWARE ARCHITECTURE FOR EMBEDDED AND REAL-TIME MEDICAL DEVICES
AUTHOR:	Atila Amiri
DATE SUBMITTED:	July 2010
COMMITTEE CHAIR:	Lily Laiho, Dr. and Professor
COMMITTEE MEMBER:	Kristen O'Halloran Cardinal, Dr. and Professor
COMMITTEE MEMBER:	Robert Crockett, Dr. and Professor

ABSTRACT

AN OBJECT-ORIENTED SOFTWARE ARCHITECTURE FOR EMBEDDED AND REAL-TIME MEDICAL DEVICES

ATILA AMIRI

Medical devices used for surgical or therapeutic purposes require a high degree of safety and effectiveness. Software is critical component of many such medical devices. The software architecture of a system defines organizational structure and the runtime characteristic of the application used to control the operation of the system and provides a set of frameworks that are used to develop that. As such, the design of software architecture is a critical element in achieving the intended functionality, performance, and safety requirements of a medical device. This architecture uses object-oriented design techniques, which model the underlying system as a set of objects that interact to achieve their goals. The architecture includes a number of frameworks comprised of a set of classes that can be extended to achieve different functionality required for a medical device. The *Input/ Output (IO)* framework includes a number of core classes that implement periodic and a periodic input output with varying priority requirements, provides a hardware neutral interface to the application logic, and a set of classes that

can be extended to both meet the hardware IO specifics of a target platform and create new sensor and actuator types for client applications. The *Devices* framework provides a blueprint to develop the controller logic of the medical device in terms of abstractions that parallel the hardware components of the medical device. The *Configuration* framework allows creation and configuration of a medical device from an XML (Extensible Markup Specification) specification that specifies the configuration of the device based on abstract factories that can be extended to meet requirements of a specific medical device. The *Controller* is the component of the architecture that defines classes that implement reception of commands from and transmission of status and data to a local or remote client and dictate the structure of threads, thread priorities and policies for this purpose. The *Diagnostics* package of the architecture defines a framework for developing components that monitor the health of the system and detect emergency conditions. The architecture is implemented in C++ and runs on a real-time LINUX operating system. At this time, the architecture is used in development of two of the St. Jude Medical Atrial Fibrillation Division's medical devices; one of these has FDA class III and the other class II classification.

Keywords: St. Jude Medical, Medical Device, Software Architecture, Software Framework, Embedded, Real-time, Object-oriented Design/Programming.

TABLE OF CONTENTS

List of Figures	viii
Background	1
An informal introduction to OOD/P and UML	4
Classes and objects	5
Generalization relation	10
association relation	13
Composition relation	13
Aggregation relation	14
Stereotypes	14
Package diagram	15
Template classes	16
Objectives	3
Methods and Materials	4
Design Overview	5
Input/ Output Framework	9
Readers and Input Devices	12
Input Devices and Sensors	17
Generic Input	19
Extending Generic Input: Temperature Sensor Example	22
A note on the output	26
Periodic input/output	27
Bookkeeping, startup and shutdown	30

Software Device Framework	32
Construction	38
Startup	39
Shutdown.....	39
Request and reply scheme	40
Diagnostics.....	40
The emergency monitor state machine	43
Configuration	46
Defining the Configuration	47
IO Specification.....	50
Device Specification.....	53
Design of configuration.....	57
The Configurator class	57
Creation of device objects.....	61
Controller	63
Command objects	64
Message Queue.....	66
Memory pools.....	67
SystemController class	67
Results.....	69
Development times.....	69
Software problem reports	70
Performance results	71
Conclusion and future work.....	74
References.....	76

LIST OF FIGURES

	<i>Page</i>
Figure 1: UML class diagram, Circle class.....	6
Figure 2: C++ representation of <i>Circle</i> class	8
Figure 3: C++ implementation of <i>Circle</i> class.....	9
Figure 4: A Circle object, C++	9
Figure 5: Generalization relation, UML example	11
Figure 6: Figure abstract class	11
Figure 7: Circle class revised.....	11
Figure 8: Polymorphism example.....	12
Figure 9: Association relation.....	13
Figure 10: Composition relation	14
Figure 11: Stereotype example, singleton.....	15
Figure 12: Package diagram.....	16
Figure 13: A template class with two instantiations	16
Figure 14: Software packages and their dependencies	6
Figure 15: IO core classes and their relations	10
Figure 16: Input devices and readers	14
Figure 17: Template method design pattern in reader	16
Figure 18: Sensors and input devices.....	18
Figure 19: Generic input	20
Figure 20: Defining a Temperature Sensor.....	23
Figure 21: Thermocouple Reader	25

Figure 22: Application's view of a sensor.....	26
Figure 23: Device abstract class	34
Figure 24: Parent Device abstract class	35
Figure 25: A hypothetical device hierarchy.....	36
Figure 26: Collaboration across devices	38
Figure 27: Core diagnostics classes	41
Figure 28: UML State diagram of emergency monitor.....	44
Figure 29: High level schema definition for RF Generator	49
Figure 30: Schema describing a SPI controller.....	51
Figure 31: SPI generic input/reader	52
Figure 32: XML specification for a spi controller and a therocoupe.....	53
Figure 33: Schema definition for software devices	54
Figure 34 : XML spFigure 33ecification for Catheter device.....	55
Figure 35: Schema definition for parameters.....	56
Figure 36: The Configurator class	58
Figure 37: <i>IoControllerFactroy</i> and examples of its concrete classes.....	60
Figure 38: Software device factories example.....	62
Figure 39: Look up table for software devices	62
Figure 40: Controller classes	64
Figure 41: Distribution of Software Work Request.....	71

BACKGROUND

A review of over 100 European and U.S. companies, indicates that software is one of the most crucial elements in creating novel products and as the role of software in medical devices increases, so do the software related failures (1). The architecture of a software system is significant for meeting the quality and providing the framework to achieve the intended functionality (2).

The software architecture discussed in this thesis has been used to construct the software that controls operations of two new medical devices developed by St. Jude Medical Atrial Fibrillation Division, hereafter referred to as AF1 and AF2, the descriptions of which remain company confidential at this time. While each product of these products has a unique function and clinical purpose, both of have overlapping requirements: they are both embedded systems with real-time, safety, and reliability requirements.

Architecture is the organizational structure of a system (3). Software architecture of a system arises from partitioning the system into units with well-defined objectives, relations, and interfaces. Arriving at the structure of a system is a matter of dealing with complexity and decomposing the system into subsystems of lesser complexity, each with its own charter and function. As Dijkstra (4) puts it: “The technique of mastering complexity is known since ancient times: Divide et impera (Divide and rule).” The method used in arriving at the structure of the software for both of the medical devices mentioned here has been object-oriented decomposition. In an object-oriented design

the software system is defined as collection of objects that interact to achieve the system goals (5). An object is a software representation of some entity, where the entity is represented as a packaging of data and procedures that act on that data (6). Objects interact by sending messages to one another, that is, they implement a given scenario by invoking the procedures of one another. The terms procedure, method, operation, and function are used interchangeably in this text. While the terms object and class are typically used interchangeably, there is a difference between the two: a class is a set of objects that have common structure and behavior (5). Object-oriented design promotes encapsulation, which allows reliable modification of programs with limited effort; it does so by hiding the data and implementation of methods of objects (5). The only means of communicating with an object is by invoking the operations that are made visible by the object (6). The benefits of using object-oriented analysis and design include: using the expressive capabilities offered by object oriented programming languages such as C++ (7), reuse of the design and application frameworks, and production of systems that are more resilient (5).

The architecture presented herein is comprised of a number of frameworks. A framework is a collection of classes that make up a set of services for a specific domain. A framework introduces a number of classes and the architectural guidance that the clients of the framework adapt to implement a specific application (5; 6). If we were to compare and contrast the software system of the two medical devices mentioned here, we would note that there are elements that are similar or identical across the two systems, among these are the input/output framework, configuration framework, and

core components of the device framework that are described later in subsequent sections. The two applications while entirely different, utilize these frameworks and their designs are directed by the blueprint of the architecture discussed herein. Design of the architecture is based on object-oriented design techniques but has evolved from the prior experiences of the author and observations of what has and has not worked in the past projects, especially in design and implementation of embedded and real-time software systems, particularly in semi-conductor and medical device industries.

The architecture discussed in this thesis is designed for medical devices that have embedded and real-time properties. An embedded system is one that the computer software, typically running on a microprocessor, is embedded in the product and is responsible for controlling the hardware and operation of the product. The software controlling operation of an embedded system is subject to constraints imposed by its physical platform and by the requirements arising from reaction to the physical world in which the system operates (8). For example, software of an implantable pacemaker is subject to battery longevity requirements and as such it has to be frugal with its computations and use of processor time. The constraints arising from reaction to the physical world constitute the real-time aspects of the software. A hard real-time system is one that must respond to the external stimuli coming from the surrounding world within a deadline consistently or suffer severe outcome (9; 10). For example, software controlling a medical device that is designed to cauterize heart tissue by delivering radio frequency energy, must consistently and within a fixed deadline respond to input from temperature sensors or otherwise risk injury to the subject.

The notation used to express the design in this paper is the Unified Modeling Language (11). UML provides a set of tools that allow one to express different aspects of a software design. In this article we have used package diagrams to express the structure and breakdown of the architecture, class diagrams to depict the static structure of a package design based on its classes and their relationships, state diagrams to express the lifecycle of an object and its response to events, and collaboration diagrams to depict realization of a scenario by depicting sequence of messages across objects that interact to realize the scenario.

In order to demonstrate applications of some aspects of the architecture, the text gives several examples in the context of radio frequency ablation. In our example context, radio frequency ablation refers to nonsurgical treatment of some types of recurrent tachyarrhythmias (fast heartbeats) that start in the heart's upper chambers or middle region. In an ablation procedure, a physician guides an ablation catheter using real-time imaging techniques inside the heart chamber and places the tip of the catheter on the tissue that causes the abnormal heart rhythms (12). Then, a medical device capable of generating alternating currents at high frequencies of 500 to 1000 KHz is used to deliver a current from the tip of the catheter to a patch placed on patient's skin. The current passing from the tip of the catheter through the tissue to the patch causes generation of heat at the tip of the catheter and cauterizes the contact site (13).

AN INFORMAL INTRODUCTION TO OOD/P AND UML

This section provides a brief introduction to object-oriented design (OOD), object-oriented programming (OOP) and Unified Modeling Language (UML) notation. The

material covered herein is intended to give the reader who is new to these concepts a basic understanding of the notation and terminology used throughout the text and it is by no means a complete description of OOD/P or UML subjects.

CLASSES AND OBJECTS

Object-oriented design (OOD) is a method that promotes decomposition of the design in terms of objects that interact to fulfill the intended goal of the design. An *object* is a runtime representation of some entity. By runtime we mean that the representation exists in a running computer program. For example a circle with radius of 3cm can be modeled as an object. In defining objects we use an abstraction of the target entity to represent what we need in our design. For example a circle may have several properties but in our problem domain we may only need to represent a circle with a radius and be able to compute its area, effectively ignoring other aspect of that entity. A *class* is an abstraction of some entity. If in our above example a specific circle was an object, then a circle class is the set of all such circle objects, defined by our circle abstraction. The terms class and object are often used interchangeably; however, more precisely an object is an instance of a class.

In object-oriented design a class is represented by a name and is comprised of a number of *attributes* and *operations*. For example, *radius* is an attribute and *area* is a function of our `Circle` class. The terms operation, member function, method, and procedure are often used interchangeably, so are the terms attribute and member variable.

Unified Modeling Language (UML) is a specification for modeling software system, standardized by Object Management Group (OMG). It includes an array of tools that are used to describe different aspects of an object-oriented design. Among these tool is *class diagram*, which is used to depict classes in our model and their relations. The simple class diagram of Figure 14 consists of the `Circle` class, which represents the abstraction of a circle that we discussed earlier:

FIGURE 1: UML CLASS DIAGRAM, CIRCLE CLASS

The top compartment in Figure 1 is the class name, the middle one display attributes, and finally the bottom compartment includes operations of the class. Often, unlike the above, labels of attributes and operations compartments are not shown. In some cases either of the compartments may be excluded from the view – e.g. for sake of brevity. In the above example, `Circle` has an attribute named `radius`, which is of type `float` (a floating point number) and two operations, one named `area` that has no arguments and returns a `float` value. The other operation that has the class name `Circle` is called a *constructor* because it is the operation which is used for constructing and

initializing instances of `Circle`. Here, the constructor accepts a `float` argument named `r` that is used to initialize the `radius` attribute when the object is constructed.

Object-oriented paradigm promotes *encapsulation*, this means that a class hides access to some or all of its attributes (and even operations) and provides access only through some of its operations; even the details of the implementation are unreachable to the users of the class. Another term for encapsulation is *data hiding*. The benefit of encapsulation is that other parts of the program will not be dependent on internals of a class and therefore do not need to change if the internal structure or implementation of the class changes. We can achieve encapsulation by changing visibility of attributes or operations of a class. `Circle` class defines the visibility of `radius` attribute to be private, which means the attribute is only accessible to the internal implementation of the class; this is demonstrated in the class diagram with the red class icon with the picture of lock superimposed on it. On the other hand the `area` function which is labeled with a small green class icon is public and may be accessed by any parts of the program.

Object-oriented programming (OOP) is a programming technique that the same as OOD utilizes the concept of objects. Object-oriented programming languages, such as C++, have constructs that allow us realize an OOD straightforwardly. The C++ code excerpt depicted in Figure 2 is roughly equivalent to that of UML diagram of Figure 1 (line numbers are not part of the source code). Typically, definition of a class and implementation of the class are organized into separate files. The source code in Figure

2 shows definition of class `Circle` from a header file (e.g. `Circle.h`). The other programs include the header file to work with `Circle` class. The `explicit` keyword prior to the declaration of the constructor (line 4) tells the C++ compiler not to automatically create a `Circle` object from a float in a context that such conversion is possible. The `const` keyword in front of the `area` function (line 5), is a promise that the implementation of the function will not change the state of the object, i.e. will not change values of any attributes. Note that Lines 4 and 5 only show the signatures of the constructor and the `area` function. Signature of a function is determined by the type of its arguments.

```
1.  class Circle
2.  {
3.  public:
4.      explicit Circle(float r);
5.      float area() const;

6.  private:
7.      float radius;
8.  };
```

FIGURE 2: C++ REPRESENTATION OF *CIRCLE* CLASS

In C++ implementation of the `Circle` class, which consists of definition of the operations is organized into a separate file (e.g. `Circle.cpp`). Figure 3 shows the C++ source code that implements the `Circle` class. Line 1, includes a standard C++ header file that has declarations of some mathematical functions and constants such as `M_PI` (the constant used to represent π) and the power function, both of which are used in line 9. Line 2, simply includes our previous definitions for `Circle`.

```

1.  #include <math.h>
2.  #include "Circle.h"

3.  Circle::Circle(float r)
4.  : radius(r)
5.  {
6.  }

7.  float Circle::area() const
8.  {
9.      return M_PI * pow(radius, 2);
10. }

```

FIGURE 3: C++ IMPLEMENTATION OF *CIRCLE* CLASS

Lines 3-4 are implementation of the `Circle` constructor, which simply initializes `radius` to the value of the argument `r`. Lines 7-10 are implementation of the `area` function. When this function is called on an instance of `Circle`, it computes and returns $\pi \times (\text{radius})^2$.

Figure 4 shows excerpts from a C++ program that creates a `Circle` object and sends a message to it to get its area. The program creates an instance of `Circle` and sets its `radius` to 2.5 and assigns the address of newly created instance to a pointer-to-circle (`Circle*`) named `p`. Note that the object itself does not have a name but we could use the pointer to send messages to the object.

```

1.  Circle* p = new Circle(2.5);

2.  std::cout << "Area of a: " << p->area();

```

FIGURE 4: A *CIRCLE* OBJECT, C++

Line 2 of the program is sending a message to the object to get and print its area. This piece of code is just to illustrate the point of creating an instance of a class and sending messages to an object of that class.

GENERALIZATION RELATION

Assume that in our example problem we intended to represent other geometric figures and be able to compute their areas. Object-oriented design allows us to define an abstraction such as `Figure` with an area function and subsequently refine in to other specialized classes such as `Circle` or `Square` with some common trait among all of those - for example all of them have an area function. This can be achieved by using *generalization* relation as depicted in UML class diagram of Figure 5. In this example, `Figure` class is called base class or super class and `Circle` and `Square` are called its subclasses or derived classes. We also say that `Circle` (or `Square`) extends `Figure`, specializes `Figure`, or inherits from `Figure`. The term inheritance is often used in C++ context because if the base class had any public (or protected) attributes or functions they will be accessible to the subclass (inherited from the base class) and can also be invoked on instance of the subclass. Note that the area function is inherited by both of the subclasses.

FIGURE 5: GENERALIZATION RELATION, UML EXAMPLE

Furthermore, the signature of `area` function in the UML class diagram of Figure 5 is italicized. In UML speak the `area` function is abstract. Abstract functions do not have an implementation, only declaration (signature) of the function is present. A class that has any abstract functions is itself abstract. An abstract class cannot be instantiated, for example, we will not be able to create an instance of `Figure` class.

```

class Figure
{
public:
    virtual float area() const = 0;
};
  
```

FIGURE 6: FIGURE ABSTRACT CLASS

```

class Circle: public Figure
{
public:
    explicit Circle(float r);
    virtual float area() const;

private:
    float radius;
};
  
```

FIGURE 7: CIRCLE CLASS REVISED

Figure 6 shows the C++ representation of `Figure` class. Note the keyword `virtual` in the declaration of `area` function. In C++ functions that are supposed to be inherited and implemented by subclasses are called virtual functions, the `=0` characters in front of

the function make the function abstract or in C++ speak: pure virtual function. Figure 7 demonstrates the revised C++ representation of `Circle` class as a subclass of `Figure`. Note that the definition of `area` function is preceded by `virtual` keyword; this tells us that `Circle` is assigning its own meaning to the `area` function. Another term for this is function overloading. This allows us to have code that uses functions defined by `Figure` without even having to know what the actual figure is, consider the example of giving an array of `Figure` pointers to some part of the code and asking it to compute the sum total of areas, this can be achieved as depicted in code excerpt of Figure 8:

```
1.  float areas(Figure* figures[], int n_elements)
2.  {
3.      float sum = 0;
4.      for (int i = 0; i < n_elements; i++)
5.      {
6.          sum += figures[i]->area();
7.      }
8.      return sum;
9.  }
```

FIGURE 8: POLYMORPHISM EXAMPLE

Each element of `figures` is a pointer to `Figure`; the code iterates through the input array and increment the sum by the area of each element. The array could have been populated by some other component and may be comprised of pointers to `Circle`, `Square`, or any other subclass of `Figure`; all these subclasses are still a `Figure` and as such all our code needs to do is to call the `area` function to get the area of an underlying figure. This ability to assign different meaning to the same function and determine its behavior during the run time is a powerful feature of object-oriented design and programming. If we were to use a method that lacked generalization and

polymorphism we would have to resort to complicated, difficult to read and difficult to change code made up of nested if-then-else statements.

ASSOCIATION RELATION

An *association* depicts relation between two classes and implies a structural relation between the two. An association specifies the cardinality of the participating objects and if one can navigate to the other object (accesses it). Consider the UML class diagram of Figure 9. It implies that an instance of `Canvas` is associated with zero or more instances of `Figure` and that `Canvas` can navigate (access) `Figure` instances. The `*` indicates the cardinality of zero or more, other notation for cardinality are: `1` (mandatory), `0..1` (optional), `1..*` (at least one), or any positive number. The arrow between `Canvas` and `Figure` specifies the direction of navigation, and the label called `figures` signifies a role name.

FIGURE 9: ASSOCIATION RELATION

When translating design into code, the role name is an indication that `Canvas` class has an attribute called `figures` which could be used to access the `Figure` objects (much like the array of pointers in Figure 8)

COMPOSITION RELATION

A composition relation is similar to association relation with respect to information that it carries but depicts the part-whole relation between the two objects. Assume that we want to depict that a circle has a center; this can be demonstrated with the UML class diagram of Figure 10:

FIGURE 10: COMPOSITION RELATION

The filled diamond shape specifies the lifetime time of the object at the other end of the arrow, here it tells that lifetime of `Point` object that is in association with `Circle` will end when the `Circle` object is destroyed.

AGGREGATION RELATION

An aggregation relation is similar to the composition relation but is depicted with an empty diamond; it implies that the object on the diamond side is an aggregate of object(s) on the arrow side. The life time of the object on the arrow side does not depend on the containing object.

STEREOTYPES

A stereotype is a mechanism for extending existing UML constructs, often for use of a project. A stereotype is represented as a text enclosed between « and » symbols. There are a number of standard stereotypes, for example «singleton» above a class name

denotes the singleton design pattern, which states that there is going to be only one instance of the class per application (process) that is accessed through a static function supplied by the class. A static function or class function is one that is invoked on its own, that is, it does not need to be invoked against an instance of the class. The UML class diagram of Figure 14 shows a singleton class named `SystemController`. The static function `instance` can be used to get the reference to the only single instance of this class. Note that the constructor is deliberately made private so that no one would be able to create other instances.

FIGURE 11: STEREOTYPE EXAMPLE, SINGLETON

PACKAGE DIAGRAM

A package is an arbitrary collection of classes; however, classes are packaged together to fulfill a specific purpose. For example the IO package, in the main text, includes classes that are designed to handle input/output operations. A package may depend on another package, that is, it may use services (classes) of another package in implementation of its classes. In the UML package diagram of Figure 12 the Devices package depends on the IO package, the dependency relation is depicted with a dashed arrow, with the head of the arrow pointing to the package that renders the service.

FIGURE 12: PACKAGE DIAGRAM

The dependency relation is also used between two classes, usually to demonstrate that a class is using the pointed class as an argument or return value of one of its functions.

TEMPLATE CLASSES

Template classes are power tools for developing data structures and algorithms that may be applicable to many data types. As a simple example consider the example of Figure 13:

FIGURE 13: A TEMPLATE CLASS WITH TWO INSTANTIATIONS

The `MessageQueue<T>` is a template class, where `T` is the template parameter. The template parameter signifies a type but is not a real type; it allows us to program the message queue in a generic way. This class defines a `put` function that allows

object of type `T` to be placed in its internal data structures and a `get` function retrieves the value of the element in front of the internal representation of the queue. Later the template argument `T` can be substituted for some specific data type to create message queues of that type. This is called template instantiation, for example we can instantiate the example template to create message queues that accept integers, `MessageQueue<int>`, or to create message queues of `Figure` class that we discussed earlier, `MessageQueue<Figure>`. Figure 13 depicts the UML notation for the two instantiated types.

OBJECTIVES

The overall objective for software of medical device AF1 and AF2 have been virtually the same as many other class II and class III medical devices: implement software that meets the clinical requirements set forth for the product reliably in a shortest duration of time with the least amount of resources. By reliably, we mean that software is subject to design reviews and inspections defined by Standard Operating Procedures (SOP) of St. Jude Medical in compliance with FDA guidelines. This includes passing formal verification and validation tests that are designed, implemented and executed by the software verification team to validate that the implantation meets product requirements. The objective of the architecture has been to use sound principals of object-oriented design and lessons learned from prior experiences to achieve the following objects:

- The architecture has the ability to evolve by extending the basic core classes to support future modifications or hardware variations over different medical devices
- Provides foundation classes that shorten the duration of developing application logic for a new medical device
- Encapsulates the details of the hardware input/output from the application code and incorporates the complexities of concurrency design within the architecture, relieving the application developers from these concerns and thereby reducing the possibility of errors on these accounts

We limit our design and discussion of the architecture to what we dub as the controller. Loosely defined, the controller is part of the system that controls the operation of the medical device, or so to speak the embedded part. This includes software components that interface with hardware as well as the bulk of the application logic minus the Graphical User Interface (GUI). The GUI is the component that provides the clinician with an interface to interact with the system. While design of the GUI is excluded from this architecture, the means of receiving commands and sending status and data to the GUI or remote client are important components and are described in the Controller section.

METHODS AND MATERIALS

The following is an enumeration of the tools that were used in design and construction of the architecture and the hardware and operating platforms of AF1 and AF2 devices:

The software designs were expressed in UML, class diagrams were used to document the static structure of the system and state diagrams were extensively used to capture the dynamic behavior of the objects that react to system events. The designs were hand coded in C++ (7), a programming language that supports procedural as well as object-oriented programming. Standard Template Library (STL) (14) and BOOST (15), which are libraries of foundation classes that are useful in developing any substantial C++ code, were used in construction of the architecture. *Xerces* (16) validation parser and class libraries were used in constructing the components that require manipulation of files described in Extensible Markup Language (XML) (17). XMLSpy (18) and Eclipse

(19) were used to develop the XML schema (20) and populate configuration files described in XML. Additionally Eclipse was used as the development environment for developing C++ source code. The source code was compiled using GCC compiler (21). The operating system for development of both of the products is Red Hat Linux (22). The target hardware platform of AF1 device is currently AmproCore™ Module 620 with an AMD 500 MHz Geode™ LX800 processor and 1GB RAM running under MontaVista 6 (23) operating system. The target hardware platform AF2 is currently phyCORE-imx27® with ARM926EJ-S™ processor and 128MB RAM running under MontaVista Pro 5 (23) operating system.

It should be noted that the package/ class diagrams and code excerpts presented herein only depict a subset of the actual classes, their relations, attributes, and operations from the actual design and implementation.

DESIGN OVERVIEW

The UML package diagram of Figure 14 depicts the organization and decomposition of the of the controller software architecture. Packages that are labeled with and «*external*» stereotype are the 3rd party components that have been used in development of the architecture and application logic but are not part of the architecture. The dotted arrows are dependency relationships, the arrow points to the package that provides a service and the other end of the dotted line emanates from the dependent, i.e. the package that uses offered services. Packages with a «*global* » stereotype provide services that are used by other architecture components.

FIGURE 14: SOFTWARE PACKAGES AND THEIR DEPENDENCIES

The *IO* package abstracts the details of hardware input/output and is a framework that encapsulates the policies and details of periodic and aperiodic input/output mechanisms; it defines the sensor and actuator classes for use of the application software and includes a number of classes that are used to develop input/output mechanisms suitable for the target platform hardware. These are depicted as packages that are framed under *IO Extensions*.

The *Devices* package introduces an object-oriented framework for developing the application program. The framework encourages decomposition of the application logic to parallel that of the underlying hardware. The application is organized as a tree of software devices. A *software device* is an abstraction representing the very basic hardware device in the overall equipment. It defines *startup*, *shutdown* and message reporting protocols. Other software devices are defined by extending the software device abstraction. As an example consider the *Catheter* software device in the RF Generator, it contains temperature sensors and a state machine that detect the type of hardware catheter connected to the device and monitors temperature at the tip of the hardware catheter by considering only sensors that are at a given time connected as a result of the result of the specific hardware catheter.

The *Diag* (Diagnostics) package is comprised of classes that allow building diagnostics monitors for observing health of the system by monitoring conditions that may be considered emergency. Emergency conditions are specific to requirement and setup of a target system and are often an indication of potential hardware problems, such as a CANopen emergency message (24) from a node on a CAN bus that indicates possible failure in that node. A monitor detects and publishes emergency conditions but does not take reactive measures towards the detected emergency. A monitor class provides interfaces that allow the medical device application code to subscribe to the identified emergency and respond according the requirements of the application.

The *Configuration* package consists of a set of classes that are used to build and configure the software system from a configuration file described in XML (Extensible

Markup Language) (17). As a framework, the package defines a set of abstract factory classes (6) that are extended to create IO, software device, and diagnostics monitor objects for the medical device at hand. The package allows defining arbitrary number of parameters for these objects - such as upper and lower temperature limits for the catheter device mentioned earlier, or system level parameters such as those that specify priorities of different threads.

The *Metrics* package is comprised of a number of classes and C++ macros (7; 25) that can be conditionally compiled to instrument the code to collect performance data from the code running on the target platform. The collected data is analyzed for detecting bottlenecks or anomalies and to verify that real-time constraints are met during target testing.

The *Common* package includes common data types, memory pool templates to avoid burden of dynamic memory allocation and heap fragmentation in a real-time system, and filters. These are used by any packages that may require their services.

The architecture uses a message queuing pattern (26) to synchronize and share information between the device thread that executes incoming commands and cyclic IO notifications and the threads that receive commands from a communication channel, cyclic IO threads, and threads that response/status on a communication channel. The incoming commands and their associated responses are implemented as command design pattern (6) within the *Controller* package. *Controller* also includes the logic for utilizing *Configure* to construct and setup the runtime objects and is the entity that

initiates and coordinates startup and shutdown operations across *IO*, *Devices*, and *Monitors* on power up and on shutdown requests from the client.

Metrics and *Common* packages are not detailed any further in this text.

INPUT/ OUTPUT FRAMEWORK

The *IO* package abstracts the details of hardware IO and provides a framework that allows application to: perform input/output operations in a consistent manner irrespective of the hardware, make extensions to the core classes of the package to accommodate project specific hardware, perform periodic and aperiodic input output, and to encapsulate complex multi-threading aspect of timing and periodic IO into the framework and out of the application code. The UML class diagram of Figure 15 depicts the high level static design of the IO package.

IoDevice is an abstract base class that its subclasses correspond to some physical input/output device; it allows its subclasses to establish type of the input/output data. *InputDevice* and *OutputDevice* abstract classes further the generalization hierarchy originating from *IoDevice* for input and output purposes.

IoReader class is the base abstraction used to implement read operations of an *InputDevice* and is used to encapsulate the details of input mechanisms for a hardware or device driver. A reader may be associated with one or many *InputDevice* objects. This is to accommodate cases like CANopen standard (24) where a peripheral node can transmit status of several data items, such as hardware sensors connected to it, in a

single Transmit Process Data Object (TPDO) message (24) . In such cases the reader has the knowledge of how to unpack the data and how to make it available to multiple *InputDevice* objects. Likewise specializations of *IoWriter* encapsulate detail of output operations.

FIGURE 15: IO CORE CLASSES AND THEIR RELATIONS

IoController class is the base abstraction for a hardware IO controller; its derived concrete classes implement the startup and shutdown operations for initialization and cleanup of a target IO controller. An *IoReader* or *IoWriter* is always owned by a specific *IoController*. This property is used in finding readers/writers and binding them to input/output devices during construction, additionally it allows any special handling of the readers/ writers during startup or shutdown operations of the *IoController*.

Collectively *InputDevice*, *OutputDevice*, *IoReader*, *IoWriter*, and *IoController* define the interface requirements for extending the framework to cover different input/output on different platform, later on we will present an example of defining temperature readings in a hardware neutral manner and define a type of *IoReader* that is capable of retrieving input from a SPI (Serial Peripheral Input) bus for this purpose. These abstractions and their subclasses are rarely ever used by the application code; instead, the *IoObserver* and its derived classes including *Sensor* and *Actuator* are used for this purpose.

A *Sensor* is the interface used by application for receiving cyclic input notifications and for performing on demand input through and receive cyclic or change notifications from an *InputDevice*. Note the one-to -many relation between *IoObserver* and *IoDevice* in Figure 14, this allows different parts of applications to receive the same sensory feedback from the same input device. Likewise an *Actuator* is the interface used by the application to access an *OutputDevice*.

The *IoCycle* class encapsulates details of periodic input/ output operations. An *IoCycle* instance has a deadline timer that is programmed to alert the object periodically; on expiry of this timer all the *IoReader/IoWriter* instances associated with the *IoCycle* instance are commanded to perform their read/write operations. The readers update their input device instance and the writers use the data values of their corresponding output device instance and eventually the notification for completion of cyclic IO is conveyed through sensors and actuators. There may be any number of *IoCycle* instances in a medical device application depending on the required sampling frequencies or cyclic output periods required by the application.

The *IoService* singleton (6) provides entry point to IO package for starting up and shutting down components such as *IoController* and *IoCycle* instances, and providing lookup services to locate other components such as *IoDevice* by its unique name for setup and testing purposes.

READERS AND INPUT DEVICES

The UML class diagram of Figure 16 illustrates the generalization hierarchy of some of the input device and reader cases and the relationship between the two. The purpose of an *IoReader* is to encapsulate the details involved in fetching data from a hardware device, typically through some device driver. The other important function a reader is interfacing with an *IoCycle* object to perform sampling of input data periodically. A reader retrieves data related to one or more *InputDevice* objects. An *InputDevice* serves the following purposes: it receives and stores input data for access of the application until the next IO cycle, participates as publisher in publish and subscribe (6) relation

with associated *Sensor* instances, as such it notifies sensor objects, hence the application, of availability of periodic input data, and fulfills on demand input on behest of any associated *Sensor*.

IoReader and *InputDevice* are abstract classes, as such, they cannot be instantiated directly, in other words, one cannot declare an instance of *IoReader* or *InputDevice*; their purpose is to define the interface requirements for the reader and input device abstractions by declaring abstract functions, or in C++ speak, pure virtual functions, that ultimately must be implemented by some subclass to make it a concrete class that can be instantiated. Abstract functions are italicized in UML class diagrams, for example, *refresh* is an abstract function of *IoReader* as depicted in Figure 16.

FIGURE 16: INPUT DEVICES AND READERS

Then the process of creating input capability for a particular medical device starts by creating appropriate subclasses of *IoDevice* and subclasses of *IoReader* to handle a particular hardware. An appropriate *IoDevice* is one that ultimately provides the data requirements needed by the application. The IO framework supplies a number of built-in *InputDevice* subclasses that are commonly used in embedded systems, these include *DigitalInputDevice* and *AnalogInputDevice* depicted in Figure 16. The *nameM* attribute in *IoDevice* is used to assign a unique name to an instance of input (or output) device, e.g. “*Thermocouple_1*”. The primary use of the unique name is to locate the input device and associated it with sensors.

At some level in the generalization hierarchy, a subclass of *InputDevice* is associated with a subclass of *IoReader* that includes the instructions for performing the input operation against the hardware or the device driver and updates the status and data of that associated *InputDevice*. As an example, consider a digital input pin on a PCA9555, which is a device with General Purpose Input Output (GPIO), and another digital input pin connected to a CAN node residing on a CAN bus, these hardware pins can be both represented as *DigitalInputDevice* objects, one associated with an instance of *Pca9555GpioReader* and the other with a *CanIoRtrReader*, as depicted in Figure 16, each implementing the details of *refresh* and *immediateRefresh* operations, thus enabling cyclic and on demand input against the said hardware. The benefit of abstractions of *InputDevice* and *IoReader* is that a mechanism such as *IoCycle* that initiates cyclic reads need not be concerned with the type of input device or specifics of reading mechanisms; in this case all an *IoCycle* instance needs to do is to call the *read* operation on all the *IoReader* objects associated with that *IoCycle*. The *read* operation in turn calls *refresh* operation on itself, where the detailed semantics of *refresh* being ultimately handled in runtime by the concrete class of *IoReader* such as *Pca9555GpioReader* or *CanRtrReader*. In turn these readers, after performing input against the hardware, call *update* functions on their related input devices, in this case through their relation with *WordInputDevice*, to update its state with the value and of input and/or success/failure of the operation. Note diagram of Figure 16, the *immediateRead* operation is introduced at the level of *WordInputDevice*. This operation unlike the read operation of the *IoReader* that is periodically initiated by *IoCycle* is meant to fetch input data on demand. A call to this operation results in invoking a call to

immediateRead and ultimately is translated to a calling *immediateRefresh* of the corresponding *WordIoReader*. Again similar to case of *refresh* which was ultimately handled in a concrete subclass of *IoReader*, the full semantics of *immediateRefresh* is also implemented in a concrete reader. The reason for the operation being defined not at the level of *IoDevice* but at the level of some derived class, such as *WordInputDevice*, is that the signature of the call introduces the input data specific to the subclass, for *WordInputDevice*; this is presented as the *value* argument in *immediateRead* , which is an unsigned 32 bit integer .

It is worth pointing out that why, for cyclic input, a call is made to the *read* operation of *IoReader* which is then translated to its *refresh* operation. As depicted in code excerpt of Figure 17, the *read* function performs assertion to ensure structural correctness and collects metrics on duration of time it takes to perform the *refresh* operation, which is the duration of input against the hardware. As such, the *read* operation implements a template method design pattern (6) that in this case it relieves the subclasses from implementing assertion and metrics collection, and leaves the *refresh* in the subclass to implement details of the cyclic input operation.

```
inline void IoReader::read( )
{
    assert(!empty());
    METRICS_RUN(refresh());
}
```

FIGURE 17: TEMPLATE METHOD DESIGN PATTERN IN READER

Note that both the assertions and the metrics collection are enabled only during debugging and performance testing. In the actual production code, both of these

constructs would be eliminated and the in line function would be optimized as direct call to *refresh* by the C++ compiler. Similar characteristics exists between *immediateRead* and *immediateRefresh* functions.

INPUT DEVICES AND SENSORS

The UML class diagram of Figure 18 depicts the relations between *Sensor* and *InputDevice* and their generalization hierarchies. *Sensor<I>* is a template class (25) where the template argument *I*, is bound to a specific *InputDevice* type at some level of generalization hierarchy. The mechanisms necessary to connect a *Sensor* to an *InputDevice* are implemented in its base class, *IoObserver<K>*, where K eventually will bind to a subclass of *IoDevice*. Reason of including these in the base class is that for this the same mechanisms are common between *Actuator*, another subclass of *IoObserver*, and *OutputDevice*, which is another subclass of *IoDevice*, for output operations.

FIGURE 18: SENSORS AND INPUT DEVICES

The relation between *IoObserver* and *IoDevice* in Figure 18 is implemented as an observer design pattern (6). This allows instances of *IoObserver* to subscribe to notifications published by *IoDevice*. With respect to *InputDevice*, there are two such notification depicted in Figure 18, these are: *cyclicSignalM*, an attribute of *IoDevice*, and the other is *changeSignalM*, an attribute of *InputDevice*. These are both *boost* signal types, the former is used to signal availability of cyclic input and the latter is used to signal change notifications, that is, if there has been a change in the input value of *InputDevice* from the most recent cycle to the preceding one. A client program uses *open* method of *Sensor* type to subscribe to these notifications. The operations

disconnect, *block*, and *unblock* defined in *IoObserver* can be used to unsubscribe permanently from, or block and unblock these signals temporarily. These signals are conveyed through callback functions that are registered with the signal when *open* method of the *Sensor* (inherited from *IoObserver*) is called. The callback functions have a signature of *void (bool)*, the Boolean argument is an indication of success/failure of the input operation. It is the responsibility of the handler to disposition a failure status.

The *value* function in *AnalogSensor* is used to access the most recent sampled value stored in the *WordInputDevice*. This value is represented as an unsigned 32 bit integer. It is the responsibility of the application code to decipher this value in term of intended units. The *isActive* function in *DigitalSensor* is used to access the most recent input value of a digital device. The *modeM* attribute of *DigitalInputDevice* (Figure 16) specifies the logic level of the digital input device, which specifies if the pin is active when high or active when low, where low/high often refer to voltage with respect to some threshold. The implementation of *isActive* function consults this specification to determine active/inactive status from the 0/1 value of the underlying input device.

GENERIC INPUT

The framework defines a number of template classes that allow creation of sensor and input device types other than the digital and analog. These are *TypedInputDevice*<*T*, *C*>, *TypedIoReader*<*T*>, *InputDevicePairedReader*<*T*>, and *TypedSensor*<*T*>. These classes and their relations are depicted in UML class diagram of Figure 19.

FIGURE 19: GENERIC INPUT

The purpose of template class *TypedInputDevice*<*T*, *C*>, beyond that inherited from *InputDevice*, is to associate a user defined data type with the input. The parameter *T* is substituted for a user specified data type, for example *T* could be defined to represent a Celsius quantity from a temperature reading. The template parameter *C* of is a trait (27) and in this context is used in deciding if the input values of a current cycle differ from that of the preceding cycle, in order to publish the change-signal if they do. The default value of *C* is defined to be *DefaultComparator*<*T*>, which defines the *equal* function

that accepts two variable of type T and returns the result of $a == b$. This requires that any user type, such as Celsius that is substituting for T , to intrinsically have or define the equality operator. The operator overloading of C++ can be used to achieve this for user defined types(7).

The purpose of template class *TypedIoReader*< T >, beyond that inherited from *IoReader* is to associate data type to the input supplied by the reader. The template class *InputDevicePairedReader*< T, I > extends *TypedIoReader*< T > and in addition to defining the input data type through template argument T , binds the reader to a single input device type designated by the template argument I . This class defines a new abstract function *readData*($T\&$) that its semantics must be defined by concrete subclasses that are designed for reading data from the hardware and representing it in terms of data type T , for example in the next section we will demonstrate how a specialized reader could acquire data from a Serial Peripheral Interface (SPI) bus and convert it to an integer valued Celsius quantity. The other aspects of *InputDevicePairedReader*< T, I > are finalization of all other abstract functions inherited from its base classes and introducing filters for cases such as noise reduction. The *immediateRefresh* function of *InputDevicePairedReader*< T, I > is defined to calls *readData* and applies any applicable filters, if any, before returning its results. The *refresh* function simply calls *immediateRefresh* and updates the associated input device with the fetched value and then notifies completion of read operation to its associated *IoCycle* object. It should be noted that *InputDevicePairedReader*< T, I > can only be used when the reader is associated with a single input device, for cases that a reader

caters to more than one input device, the technique used in the example of *Pca9555GPIOReader* and *CanIoRtrReader* which were depicted in Figure 18 can be used.

The template class *TypedSensor*<*T*, *I*>, depicted in Figure 19, extends *Sensor*<*I*> to allow input data type generalization. The template argument *I*, same as in *Sensor*<*I*>, is a determinant of the input device type while the argument *T* determines the type of data that is accessed by *value* and *immediateRead* functions of this class. Implementations of these methods simply forward the calls to the same function names in the associated *TypedInputDevice*<*T*, *C*>.

EXTENDING GENERIC INPUT: TEMPERATURE SENSOR EXAMPLE

In the hardware design of the example RF Generator, there are five sources that can provide temperature reading from an ablation catheter, four of these are thermocouples and one is a thermistor. Different catheter types have connections that come into contact with one or more of these sources. These input sources are connected to a Serial Peripheral Interface (SPI) bus, which allows full-duplex synchronous communication between the master and other slave devices (28) . The example of Figure 20 is shows the lines of code that are required to define the temperature sensor, a temperature input device, and a temperature reader using the generic input mechanisms that were discussed in this section.

```

1.  class TemperatureInputDevice; // forward declaration
2.  typedef InputDevicePairedReader<Celsius,
    TemperatureInputDevice> TemperatureReader;
3.  class TemperatureInputDevice:
4.      public TypedInputDevice<Celsius>
5.  {
6.      public:
7.          TemperatureInputDevice(const std::string& name,
8.                                  TemperatureReader& reader)
9.              : TypedInputDevice<Celsius>(name, reader)
10.             {
11.                 readerM.add(*this);
12.             };
13.
14. typedef TypedSensor<Celsius, TemperatureInputDevice>
    TemperatureSensor;

```

FIGURE 20: DEFINING A TEMPERATURE SENSOR

TemperatureReader (line 2) simply is a type definition that binds *T*, *I* of *InputPairedReader*<*T*, *I*> to *Celsius*, and *TemperatureInputDevice* respectively, where *Celsius* is defined in Common as an integer data type. *TemperatureReader* is still an abstract because it does not define the *readData* method; we will create two separate subclasses, one that defines this method to read from thermocouple and another to read from thermistor on the SPI bus.

The class *TemperatureInputDevice* (lines 3-11) extends *TypedInputDevice*<*T*>, where *T* is substituted with *Celsius*. The constructor of this class (lines 6-10) simply calls the constructor of the base class (line 7). The arguments of this call are *name*, which is used to identify the device (e.g. “*thermocouple_1*”) and the reader, which is used to deposit cyclic input values in this instance of *TemperatureInputDevice* and carryout on demand input requests. The constructor completes the necessary relation by passing the reference of the created device to the reader (line 9).

TemperatureSensor is also a simple type definition (line 12) that binds *T*, *I* of *TypedSensor*<*T*, *I*> to *Celsius* and *TemperatureInputDevice* respectively. The application layer can now use an instance of this class to receive cyclic or change notifications from a temperature source such as “*thermocouple_1*”, or can call the object’s *value* function to get the most recent temperature reading in *Celsius* from the source, or call its object’s *immediateRead* function to get a temperature reading on demand.

As mentioned earlier, *TemperatureReader* is an abstract class and we need to define mechanisms for reading data from the physical thermocouple and thermistor on the SPI bus. We will discuss one mechanism for reading thermocouple input. The *ThermocoupleReader* depicted in Figure 21, extends temperature *TemperatureReader* in order to assign appropriate definition to its abstract method *readData*; we use *Spi::BaseReader* class to achieve this. We could have directly implemented details of reading from SPI in the *readData* but the details of SPI read were abstracted in *Spi::BaseReader* because the project requires SPI access for context other than temperature reading.

FIGURE 21: THERMOCOUPLE READER

The *readWrite* method in the *BaseReader* class implements the details of accessing SPI bus to retrieve data from a designated number of Analog to Digital Convertor (ADC) channels, the data is then stored in the receive buffer of *SpiData* object. Implementation of *readData* in *ThermocoupleReader* calls *Spi::BaseReader*'s *readWrite* and uses a formula supplied by the manufacturer of the thermocouple to convert the ADC counts to *Celsius*.

Figure 22 depicts a sample code that illustrates how application code declares and uses a sensor. Here the *TemperatureMonitor* class declares a *TemperatureSensor* and in its constructor, registers to the corresponding input physical device to receive cyclic callbacks for temperature readings:

```

class TemperatureMonitor
{
    Io::TemperatureSensor t1;

public:
    TemperatureMonitor() {
        t1.open("SPI_T1", Io::onEveryCycle,
            boost::bind(&TemperatureMonitor::check, this,
                _1));

        void check(bool ioSucceeded) {
            static const Celsius threshold = 80;

            if (ioSucceeded) {
                if (t1.value() > threshold) {
                    // do something
                }
            }
        }
    };
};

```

Sensor declaration

Binding to physical IO by name and registering cyclic callback

Getting sensor value

Callback function called on every cycle

FIGURE 22: APPLICATION'S VIEW OF A SENSOR

The *check* method is the callback function that uses the sensor's *value* function to access the latest temperature reading communicated to the callback.

A NOTE ON THE OUTPUT

Thus far we have been presenting details of input devices, readers, and sensors. It should be pointed out that similar designs exist for defining output devices, writers, and actuators that for sake of brevity will not be discussed herein. However, there are a few exceptions: for example the output devices have mechanisms that can be used to disable and enable cyclic output. This is useful in a case like radio frequency generation, where the power output needs to be ramped up gradually during a short period of time. Another application of enabling/disabling cyclic output is triggering hardware shutdown mechanism in case of emergency conditions detected by software. In such case the cyclic output that engages a hardware watchdog timer is disabled to trip

the watchdog and cut off power to hardware components that otherwise may cause harm or injury.

PERIODIC INPUT/OUTPUT

IoCycle class implements periodic input and output mechanisms. Depending on the input sampling frequencies or period of output operations there may be several instances of *IoCycle* in a given medical device implementation, each configured with a different period. As demonstrated in Figure 15, an *IoCycle* keeps lists of references to *InputDevice*, *OutputDevice* and their corresponding *IoReader* and *IoWriter* objects. These references are assigned to the *IoCycle* by the *Configuration* subsystem when the system is powered on. An input/output device and its corresponding reader/writer may associate only with one *IoCycle* instance that determines their periodicity.

IoCycle has a deadline timer that its expiry time is set to the period of the *IoCycle*. The *startup* operation of *IoCycle* creates a pair of POSIX threads when the system is started up. The deadline timer runs in the context of one of these threads – the timer thread. Every time the timer times out, the instructions running in its context, signal the other thread, referred as the IO thread, to run through the list of referenced writers and readers and call their write and read operations. The reason for separating write/read operations in their own thread of control is to ensure that real-time requirements for completing IO are met and take emergency actions if they are slipped. After signaling the IO thread, the timer thread waits (blocks) until it receives completion from all writers/readers or until expiry of the wait duration. The wait duration acts as a deadline on readers/writers for completing their tasks. If the writers/readers do not complete their write/read

operations within this deadline, the timer thread is awakened to interrupt the IO thread, thereby canceling any pending IO and taking measures for missed IO cycle. In the implementation of the architecture for the two existing medical devices, the deadline is configurable number set to value slightly less than the period of the IO cycle. The allotted time does not include all of the cycles' period based on the assumption that there will be some time needed for the application code to process completion notifications.

As mentioned earlier, the architecture runs on real-time LINUX and uses POSIX thread primitives for scheduling and priority settings (29). The overall priority of *IoCycle* thread is determined during the design and development of the medical device software by using a technique such as Rate Monotonic Analysis (10), which assigns higher priorities to threads that require higher sampling rates. The priority of the IO thread is one less than the timer thread to allow the timer thread to react to a prolonged execution of the IO thread. The policy and priority of the timer thread. The thread policy used in the existing implementations is FIFO (First in First Out), with this policy, a thread runs until it blocks and preempts a lower priority thread when it is ready to run (29). The policy and priority of *IoCycle* threads take effect when the startup method of *IoCycle* is invoked.

Before initiating the read operation on *IoReaders*, the *IoCycle* sets the status of the associated *InputDevice* instances to in progress and the same is done with respect to *IoWriter* and *OutputDevice* objects associated with the cycle. Recall that when *IoReader/ IoWriter* complete their *read/write* operation, they update the status of their

corresponding *InputDevice*/ *OutputDevice* instances to reflect success/failure of the operation. In case of readers, their *InputDevice* is also updated with the value that was read from the hardware; however, this update is recorded in the *oldValueM* attribute of the *InputDevice* and is swapped with the value in *newValueM* field only if the entire read operations of the *IoCycle* are completed within the deadline. If the readers fail to complete within the deadline the old/new values are not swapped a statuses of all the associated *InputDevice* objects are set to fail. In this case the cycle completion notifications on the input/output devices still we go out but with a failure indication.

IoCycle class defines three signals and provides interfaces that allow the application code to subscribe to those signals. These signals are all published when the timer thread resumes its work due to successful completion of reader/writers or due to deadline expiry. Detail of these signals follows:

The *CycleEndSignal* is used to convey that cycle has finished. This signal carries a pre-allocated instance of *IoCycleNotificationCommand*, commands are introduced in *Controller* section. At this time it suffices to say that once this command is executed all the old/new values of the input devices are swapped and the call back methods of the *Sensor* objects associated with those input devices are called, thus allowing the application code to react to newly retrieved values. The intended use of the signal, as we shall see in more detail later, is to queue the command argument on the device queue so that it can be executed in the context of the software device's thread and not in the context of the *IoCycle* timer thread.

The *IoCycleTimeoutSignal* is generated at the end of the cycle if the input/output operations do not complete within the allotted deadline, usually this signal is used during testing to log warnings.

The *MissedCyclesLimitSignal* is generated if the number of consecutive missed deadlines exceeds a preset threshold. The threshold is defined per hazard requirements of the medical device and is set when the *IoCycle* instance is created. The response to the signal is out of the scope of the *IoCycle* and as we shall see is defined in the *Controller* class. The response is tailored based on the hazard analysis and safety requirements of the medical device. Typically the signal indicates a fault in the hardware or inability of the software to meet its real-time deadline requirements and is considered a safety risk, in one of the devices this amounts in disengaging the hardware watchdog thus cutting the power to the hardware and entering a failure mode where the software rejects all incoming commands from the operator but the shutdown command.

The *shutdown* operation of the *IoCycle* cancels the deadline timer and interrupts the IO thread thus cancels the pending input/output operation. This method is invoked when the system is shutdown.

There is a single *IoCycle* instance with a period of zero in the application process. The zero-period *IoCycle* groups the inputs/outputs that are aperiodic and are initiated only through immediate read/write operations described earlier.

BOOKKEEPING, STARTUP AND SHUTDOWN

The *IoService* is a singleton (6), in other words there is one instance of the *IoService* per application process. The instance provides interfaces that are used for bookkeeping, startup and shutdown of the input/output framework as a whole. The bookkeeping operations include adding instances of the *IoController*, *IoCycle* and all *IoDevice* to different lookup tables. *IoService* offers functions to look these instances up by their identifiers. The bookkeeping operations are primarily used by the *Configuration* subsystem to set up different IO objects needed for a given medical device application.

The other two important functions of the *IoService* are the *startup* and *shutdown* operations. The *startup* operation iterates through all of the *IoController* objects and calls their *startup* operations which result in initialization of the hardware controllers and/or the readers/writers associated with the controllers. After starting up the controllers, the *IoCycle* instances are started up, which as we saw earlier results in performing the cyclic input/output operations and dispatching their results to the other components of the medical device application. The *shutdown* operation of the *IoService* first invokes the shutdown operation of the *IoCycle* resulting in cancellation of the pending input/output and of the timer of the cycle and then proceeds to calling the *shutdown* on all the underlying *IoController* objects to shutdown the hardware boards and clean up the system resources such as file descriptors used by readers/writers. The *startup* and *shutdown* operations are blocking calls, that is, the caller thread is blocked until the operation is completed. The operations return a Boolean result to indicate the success or failure of the operation.

SOFTWARE DEVICE FRAMEWORK

The *Devices* package is a component of the architecture that includes the application and operational logic for a given medical device controller. Except for a few core classes that define the usage guidelines, all other classes in this package are defined based on the requirements of a target medical device.

A software device, or for short device, is an abstraction that is typically used to control a hardware device. It defines operations necessary to startup, shutdown, and control the behavior of the hardware device by receiving sensory input from the hardware and actuating outputs to induce change in the state of the hardware. The objective of software device framework is to encourage an object-oriented design of the medical device software controller that parallels that of the hardware design. The software is constructed as a composite design pattern (6) which partitions the devices into tree structure, at its root is a device referred to as the equipment, a non-terminal node of the tree is an assembly or parent of other devices and the terminal nodes are devices that do not contain any other devices.

Commands are instructions that direct the operation of the medical device and are introduced in the *Controller* section. The equipment is the entry point of all commands, in other words, execution of command that is intended to control operation of the medical device is translated to a call into a method in the root device. Direct calls to other devices are not permitted. Calls are forwarded to appropriate methods of relevant child devices to fulfill a command. This is done because a parent device, ultimately the equipment, keeps track of conditions that may affect the behavior of its child devices

and depending on some conditions might refuse propagating a command down the chain. Commands propagate from equipment down to appropriate child devices. The success/failure status of executing a command is propagated from a leaf device up to its parent device and ultimately to the equipment where it is delivered to the Graphical User Interface (GUI) or a remote client through the *Controller* subsystem.

Figure 23 depicts the attributes and operations of the *Device* class, which is abstract base class of all software devices. The *startup* and *shutdown* abstract operations ought to be implemented by all the concrete subclasses. The purposes of these operations are explained in subsequent sections. These and other operations that may require interaction with the hardware are required by architecture to be implemented as request and reply mechanisms, that is, the caller thread will request the execution of the operation but will not be blocked for its completion and the result of the operation, at some point later, is communicated back as a response to the caller. The *Controller* section describes how the architecture enforces this mode of operation.

FIGURE 23: DEVICE ABSTRACT CLASS

The *Response* type is a signaling (publish/subscribe) mechanism that is used by the devices to communicate results of such operations to the parent devices or to any other subscriber. This merely is meant to communicate consequences of an operation within the context of devices thread and is not the full mechanism employed by the architecture for enforcing request/reply semantics; rather it is a component of this interaction. The operations *onStartupUpdate* and *onShutdownUpdate* allows the caller to register callbacks for receiving results of startup and shutdown operations. Instances of *DeviceChainError* class are used to communicate failures from a device to its parent

device and so on. This class has a mechanism that allows each device in the call hierarchy to add its explanation of failure to that propagated upwards by a child device.

Figure 24 depicts the abstract base class of all devices that have a child device. It defines operations that are required to be implemented by its concrete subclasses.

FIGURE 24: PARENT DEVICE ABSTRACT CLASS

The *addChild* operation is used by the Configuration subsystem to add child devices to their parent device during construction of the device hierarchy. The *ChildStartupUpdate* and *ChildShutdownUpdate* are callbacks for receiving the result of startup and shutdown operations from child devices.

The UML class diagram of Figure 25 depicts a simplified device hierarchy for a hypothetical radiofrequency generator.

FIGURE 25: A HYPOTHETICAL DEVICE HIERARCHY

In this hypothetical application *RfEquipment* is the only parent device which is comprised of two other child device, the *AblationCatheter* and *RfBoard* classes. *RfEquipment* receives the commands for starting and stopping the application and after required checks (e.g. is the equipment in an operational state, i.e. has it been started up without any failures) passes them the *RfBoard* which includes the actual implementation for delivering radio frequency energy. The *abtaionUpdate* is the

callback method that eventually receives the results of the ablation procedure. The *AblationCatheter* continuously monitors the temperature from five temperature sensors and reports the max of the valid temperature readings to *RfEquipment* through a signaling mechanism similar to that explained for startup and shutdown operations. The *temperatureUpdate* method of the *RfEquipment* receives the temperature readings and stops any ongoing operation if the temperature exceeds some programmable threshold value.

Communication across child devices is not permitted; the reason for this is to reduce tight-coupling and therefore complexity. As such a parent device forms a mediator design pattern (6) that encapsulates interactions among child devices and thereby prevents child devices to refer to one another, thus promoting loose coupling. The simplified UML collaboration diagram of Figure 26 illustrates an example of this mediation. Sequences numbers on the message links indicate the order of occurrence of messages in time. The *RfEquipment* object initiates ablation by issuing *startAblation* on *RfBoard* device, sometime later the *AblationCatheter* device sends a temperature update to its parent device *RfEquipment*, which decides that the temperature has exceeded some threshold and stops ablation by sending *stopAblation* message to the *RfBoard*.

FIGURE 26: COLLABORATION ACROSS DEVICES

CONSTRUCTION

The constructor of all devices accepts a name and a *Context* argument. The name is used to uniquely identify the device. The *Context* argument, which will be defined in the *Configuration* subsystem, carries the name of sensors and outputs actuators used by the device and contains named parameters that are defined in the configuration file for use of the device. Device parameters are discussed in *Configuration* section, an example of a named parameter for a device such as Equipment, is one named “*temprarture_threshold*” and may have been set to have a value of 100 [°C]. The constructors of the device initializes its member variables, opens any sensors or actuators that it must use and setup its callback functions for receiving notifications from sensors (actuators). Typically, after opening a sensor with a callback, the device

calls *block* operation on the sensor to not receive any signals that may change its states until the device has explicitly started up.

STARTUP

The implementation of *startup* operation for a device is meant to initialize the state of the device and child devices in order to ready the device for receiving and processing commands. For child devices this implies initializing any hardware that they control. Parent devices call *startup* on their child devices and await startup completion from all child devices prior to executing their own startup procedure. If any of the child devices reports failure of startup, the failure is propagated back to *Equipment*, which takes all the parent devices in the path, including, *Equipment*, into a fault state. The fault state prevents accepting any commands other than *shutdown*. During startup a device calls *unblock* on its sensors to receive input notifications from IO.

SHUTDOWN

The implementation of shutdown operation for a device take the hardware controlled by the device to a safe state, blocks reception of input signals and takes the device into an offline state where it will not respond to any commands thereafter. As in *startup*, a device initiates *shutdown* on its child devices and awaits completion of the child device shutdowns before commencing its own shutdown procedure. However, report of failure from a child device, while reported upward, does not prevent a parent device from executing its shutdown procedure.

REQUEST AND REPLY SCHEME

All the commands including *startup* and *shutdown* have a request/ response scheme. All the commands execute in the context of a single thread, referred to as device thread. A command is placed on a message queue (26) , upon its receipt by the communication or GUI thread. Later, at some point, the command is picked up from the queue by the device thread, and is translated to a call to *Equipment* and eventually is propagated to calls to one or more leaf devices. The device uses IO facilities, notably actuator to output instructions to hardware, and sometime later, possibly due to change of state in the hardware the device receives notifications on status of the hardware by means of its sensor objects an through the IO. The device analyzes the sensory feedback and sends a success/failure response to its parent device. This allows different devices to be at different execution states yet all the commands to be executed in the context of a single thread of control – the device thread, thus eliminating the need to maintain complex mutual exclusions that otherwise would be required in presence of more than one thread.

DIAGNOSTICS

The Diagnostics package includes a number of monitoring components that are used to observe health of the medical device in parallel to the normal operation of the device and independent of its application logic. The UML class diagram of Figure 27 depicts the core classes in this package and their relations.

FIGURE 27: CORE DIAGNOSTICS CLASSES

Monitor abstract class, as the name implies, is a component designed to monitor a target or a number of targets for a number of conditions that their occurrence is detected and announced by the monitor. Targets are typically sensor values or error indications from hardware. *Monitor* defines interfaces that need to be fulfilled by subclasses for connecting/disconnecting from/to targets.

EmergencyMonitor is an abstract class that implements the state machine for detecting conditions that can be elevated to the status of an emergency. The function of *EmergencyMonitor* is detection of emergencies only, it does not respond to the emergency. The response to an emergency is determined by the safety requirement of a particular medical device and the response to emergencies raised by monitors is incorporated in the *Controller* module of the device. *EmergencyMonitor* is associated with an *EmergencyMap* that is typically built from XML specification in the configuration file. The map defines errors that are detected by the monitor. An error is identified by an error code that is identified by some subclass of *EmergencyMonitor*. An error within the map has a Boolean retry attribute, in this context, an error that its retry value is false is elevated to emergency as soon as the condition it represents is detected. An emergency monitor has retry count and retry duration attributes, in case of retry-error, as soon as the error condition is detected a timer is started, which its duration is set to retry-duration. If the number of non-retry errors equals the max-retry while the timer is running an emergency condition is raised, otherwise the timer is cancelled and the detection count is reset.

InputMonitor is an abstract subclass of *EmergencyMonitor* designed to monitor sensory inputs and can have parameters that are passed to it via a *Context* object during initialization. A *Context* object simply maintains a map of input/output device names and parameters applicable to the *InputMonitor*.

It should be noted that all the core monitor classes are abstract and cannot be instantiated unless extended by a concrete class for a specific purpose. For example, a

concrete subclass of *EmergencyMonitor* is one that monitors a CAN bus and listens to emergency messages that may be issued by CAN nodes on the bus (30). Another example will be in use of an external esophageal probe for temperature monitoring during atrial ablation to avoid injury to esophagus (31). In such case the external temperature probe may be connected to the radiofrequency generator and its input made available through IO sensors. The temperature then may be monitored by a specialized *InputMonitor* subclass that includes thermocouple/thermistor sensor(s) and a threshold temperature parameter. The monitor would raise an emergency condition when the temperature is above the specified threshold. The implementation of the temperature monitor will simply trigger *critical* signal, defined in *EmergencyMonitor*, when the temperature raises the set threshold. The reaction of the application, and not the monitor, will be to stop ablation when the emergency signal from the monitor is received.

THE EMERGENCY MONITOR STATE MACHINE

EmergencyMonitor class implements the state machine for detection emergencies for all its derived classes based on the following assumptions: specifics of monitoring a target is left to the derived classes, and derived classes call *critical* member function upon detection of their observed condition.

FIGURE 28: UML STATE DIAGRAM OF EMERGENCY MONITOR

The state machine of *EmergencyMonitor* implements the logic of identifying what constitutes an emergency or abandoning the history of signals that have not been elevated to an emergency condition. The runtime behavior of the *EmergencyMonitor* is illustrated in UML state chart of Figure 28.

All monitors are initially created as disabled, when the system start up is executed by the *Controller* module, monitors are enabled. This is depicted as the *enable* signal transitioning the monitor from *idle* to *monitoring* state, amounting to execution of the entry action in the *monitoring* state and thus connecting the monitor to the target(s) that it monitors. The abstract member function *connect*, defined in *Monitor*, which must be overloaded by concrete subclasses, establishes the necessary connections to targets.

The *critical* signal is raised by subclasses to indicate that a condition that may result in emergency has been detected. The *error* argument of the *critical* trigger contains data retrieved from the associated *ErrorMap*. Receipt of a *critical* signal by the state machine when in *monitoring* state causes transition to *evaluating* state, upon entry to this state the *retry* attribute of the *error* argument of the trigger is checked. If retry is allowed, a deadline timer is started, which purpose is to open a window of time for tracking other possible retry-errors of which their total count in that window of time may exceed a set threshold causing an emergency condition to be raised. After executing entry actions of in *evaluating* state, transition is made either to *emergency* or *counting* states. The transition is determined by the guard conditions depicted in the diagram. In short, errors that have no retry option will transition to *emergency* state. Entry action of *emergency* state cancels the timer and calls the *detected* function which raises the detection signal in the *Monitor* base class. At this point a software component, typically the *Controller*, which has subscribed to emergencies by calling *onDetection* signal of *Monitor*, will be able to receive the emergency signal and respond to it per requirements.

The transition from *evaluating* to *counting* state occurs if the error in critical signal can be retried. The monitor will remain in counting state until either deadline timer that was started in *evaluating* state expires or enough errors are counted before the timer expires. In case sufficient numbers of *critical* triggers are signaled while in *counting* state, the monitor will transition to *emergency* state.

Note that after executing the entry action of *emergency* state, which was described earlier, the monitor unconditionally transitions to *standby* state. The reason for this transition is not to generate spurious emergency conditions thereafter. This is achieved by executing entry actions of *standby* state, which disconnect the monitor from its target, disables the monitor and cancels any ongoing timer. The timer of the monitor runs in its own thread of control, the policy and priority of the thread are determined by the *Configuration* subsystem and are set during startup of the monitor. The *run* method of the *EmergencyMonitor* (Figure 27) creates the timer thread when the monitor is enabled by the system controller during the startup procedure.

CONFIGURATION

The *Configuration* subsystem includes services that set up the runtime configuration of the medical device software an XML (17) file that includes the configuration specification. This includes constructing and establishing necessary relations for input/output devices, readers/writers, input/output controllers, IO cycles, software devices, and monitors for that medical device an passing parameters and sensor/actuator information to software devices and monitors.

While the number of software devices and the input/output components is fixed for a specific medical device the device parameters may vary for different sites or configurations, for example it may be required to set up a device parameter such as the one representing a temperature limit, to different values depending on discretion of the clinician, or depending on the regulation of the country for which the medical devices is deployed. Moreover, during the development phase, the hardware/software of the intended medical device frequently changes or components become available according to the development schedule. Without services of *Configuration* subsystem, the construction of components, their relations, and parameters would have be hardcoded, amounting to a system that is inflexible, costly, and error prone to change – the code needs to be modified, inspected, tested and released for every single change.

Configuration subsystem allows speedy reconfiguration of the software device or input/output components and change of the parameters by updating the XML configuration file. However, the cost of achieving this flexibility is to create factories that automate creation of software components. The design of *Configuration* subsystem contains a set of core classes and abstract factories (6) that facilitate building the above mentioned components from an XML file. The *Configuration* subsystem is specialized to meet the need of a particular medical device by extending those factory classes or core classes. Before, delving into the description of these classes we briefly describe the process of describing the configuration of a medical device.

DEFINING THE CONFIGURATION

The process of defining configuration of a medical device starts by specifying an XML Schema Definition (20) for the medical device. The schema defines the layout for content of the XML document and imposes constraints on the attributes used in the document. *XMLSpy* (18), an XML editing tool, and *Eclipse* (19) were used to define and update the schema. Then the XML configuration file was populated by either *XMLSpy* or *Eclipse* to include the specification of the controller at hand. After that the abstract factories in the *Configuration* package were extended to meet the need of the specific project. Implementation of factories and core Configuration classes utilize foundation classed of Xerces (16) class libraries and XML validation parser to retrieve information from the XML configuration file. In practice these steps are carried out not sequentially but in increments and iteratively and are verified through unit tests until satisfactory end results for that particular release of the medical device are achieved.

The overall configuration layout for a given system consists of the following sections: the name and version of the controller, global parameters that are accessible to different components through a global parameter map, specifications of IO cycles, specifications of input/output controllers, specification of software devices, specification of monitors, and log settings. Figure 29 depicts the XML schema that defines the layout of the configuration file for the example RF Generator.

FIGURE 29: HIGH LEVEL SCHEMA DEFINITION FOR RF GENERATOR

Each white rectangle is the name of an XML element. An element is a collection of attributes and contained elements that determine the structure of the element and ultimately the tree structure of the document. For the purpose of our discussion an attribute designates a name and value pair in an XML element. In this diagram the label *control_system_t* on top of the yellow rectangle defines the type of the root element called *control_system*, which is comprised of *name* and *version* attributes and a

sequence of other elements depicted in the dashed yellow rectangle. The symbol denotes a sequence element and implies that the elements to its right must appear in the containing element in the order (top to bottom) shown by the sequence. The symbol denotes a choice element and it implies that only one of the elements to its right may appear in the containing element. The numbers under a sequence or choice element specify number of times the element could appear (∞ implies unbounded number of times).

IO SPECIFICATION

The input/output specifications for a given system consist of *the io_cycle* elements and collective IO controllers, in Figure 29, these are elements ending with *_controller*.

Definition of a controller very closely reflects the underlying hardware and may have different specifications from one project to another. At minimum the information required by the XML specification should include enough details to build the input/output controller, its input/output devices and their corresponding readers/writers described in the Input/ Output Framework section. As an example consider the element *rf_spi_sontroller*, its type is repented by *rf_spi_controller_t* depicted in Figure 30.

FIGURE 30: SCHEMA DESCRIBING A SPI CONTROLLER

The *name* attribute is used for accessing a SPI controller through operating system file primitives, the *model* attribute is used by Configuration to locate the proper factory that can extract information from this an y element and build the controller and its ancillary components. The *frequency* and *order* attribute are used to initialize the hardware controller to run at a clock rate set by this attribute and specify byte ordering respectively. Eventually the sequence of choices tells that controller could have any number of *generic_reader*, *reader*, and *writer* elements, which in turn define the specifics of creating reader/writer objects pertinent to our SPI controller and known to the SPI controller factory. Figure 31 depicts schema definitions for a *generic_reader* element. This element describes a generic input device and its reader, such as one defined for temperature input in IO framework.

FIGURE 31: SPI GENERIC INPUT/READER

The *cycle_period* attribute specifies to which *IoCycle* the reader/input device belongs, i.e. specify the sampling requirements. The *hardware* attribute provides the factory to further narrow down the type of the input device, e.g. to create a thermistor or thermocouple object. The other attributes of this type are not discussed any further.

Figure 32 depicts excerpts from the XML configuration file of RF generator that define a SPI controller with 6MHz clock rate and specifications for a generic reader/input

device. Once the *Configuration* consults the above specs for the *generic_reader*, it would create objects that we described earlier in *Extending Generic Input: Temperature Sensor Example* section, with the temperature input named to *SPI_Temperature_1* and cycle period of 40ms (Line 2).

```

1. <rf_spi_controller name="/dev/spidev2.0" order="least_significant"
   frequency="6000000" model="rf_spi_controller" >

2. <generic_reader hardware="thermocouple" chip_select="GPIO_CS10_CS"
   word_size="8" word_count="2" cycle_period="40"
   input_name="SPI_Temperature_1" metrics_id=" SPI_Temperature_1-r">

   ...

3. <parameter name="celsius_factor" value="10"/>
4. <filters>
5.   <filter type="moving_average_filter" size="5"></filter>
6. </filters>
7. </generic_reader>
   ...
8. </rf_spi_controller>

```

FIGURE 32: XML SPECIFICATION FOR A SPI CONTROLLER AND A THEROCOUPPE

DEVICE SPECIFICATION

The Schema definition for *device* element is recursive in order to support the composite pattern used in design of the software devices. It includes any parameters input/output and nested devices that the software device may have as shown in schema depicted in Figure 33:

FIGURE 33: SCHEMA DEFINITION FOR SOFTWARE DEVICES

Consider the example of *Catheter* device with four thermocouples and one thermistor connection mentioned earlier in *Extending Generic Input: Temperature Sensor Example* section. The XML excerpts from the configuration file depicted in show specification of a *Catheter* device with the five IO points. Note that the sensor/actuators used by the device are specified by indirection. For example, Line 5 allows *Catheter* device to create a *TemperatureSensor* object, which was defined Figure 32 and refer to it as “T0”:

```

1. <device name="Catheter" model="Catheter">
   ...
2.   <parameter name="temperature_lower_limit" value="10"
       unit="celcius"/>
3.   <parameter name="temperature_upper_limit" value="100"
       unit="celcius"/>
4.   <io name="T0" ref="SPI_Temperature_1"/>
5.   <io name="T1" ref="SPI_Temperature_2"/>
6.   <io name="T2" ref="SPI_Temperature_3"/>
7.   <io name="T3" ref="SPI_Temperature_4"/>
8.   <io name="T4" ref="SPI_Thermister"/>
   ...
9. </device >

```

FIGURE 34 : XML SPECIFICATION FOR CATHETER DEVICE

Another benefit of using indirection for specifying IO is in cases that are multiple devices of the same type. For example if in our equipment we had two catheters, then the *Catheter* class could be instantiated twice, and each instance of it could still use a name such as “T0” while refer to a different (physical) input device.

Parameter Specification

The syntax of the *parameter* element is the same as that of the global parameters and is depicted in Figure 35. At minimum a parameter consists of a *name* and *value* pair.

There are other optional attributes and optional elements that further specify *parameter*.

These describe operations that *Configuration* subsystem can perform on the parameter.

For example in the RF generator a parameter may be modified during operation of the device. The *fixed* Boolean attribute should have a value of false to allow modifications to a parameter. Furthermore modifications to parameters may be saved back to the configuration file if the *persist* Boolean attribute has a value of true to allow modifications to persist beyond life of the running application.

FIGURE 35: SCHEMA DEFINITION FOR PARAMETERS

The details of other attributes and elements of the parameter type and that of the monitors and logging are not covered in this text. It suffices to say that specification of monitors, the same as devices, contains information needed to create a specific monitor type. The logging specification allows log level of different software components to be set to different level of logging for example setting the log level of a specific module to *info* allows collecting tracing details of operation of the module for diagnostics or test purposes.

DESIGN OF CONFIGURATION

The design of *Configuration* subsystem consists of a number of abstract factories (6) that allow creation of input/output controllers by *Configuration* code. Use of abstract factories allows the code that creates the controllers to just refer to the base abstraction of *IoControllerFactory* without having knowledge of the concrete factories that create various controllers. This allows adding new hardware by simply creating appropriate subclasses of *IoControllerFactory* and without modifying the code that traverses the XML file and creates the controllers and their contents. The same paradigm is used to create software devices and monitor objects. The abstract factories are then extended to create specific class instances and set up their relationships as needed in a particular medical device. The *Configurator* class uses the abstract factories and specifications included in an XML configuration file to create all input/output, software, devices, monitors, and all system parameters. The creation of the factories and scope of their life-time is controlled by the *Controller* subsystem.

THE CONFIGURATOR CLASS

The *parse* operation of the *Configurator* class uses *Xerces* validating parser (16) to parse the configuration file and build the Document Object Model (DOM), which is an in-memory representation of the configuration file. The *build* method of the *Configurator* uses the classes of the *Xerces* C++ library to navigate the DOM and construct input/output, software device, and monitor objects. The *parse* and *build* methods are called by the *Controller* module during system initialization.

FIGURE 36: THE CONFIGURATOR CLASS

In some of the medical devices, such as RF Generator, it is required to allow global or even software device parameters, such as temperature or ablation duration, to be altered by the operator and saved back in the configuration file for future uses. The *save* method of *Configurator* allows all the changes made to these parameters to be committed to the configuration file. The details of updating parameters is within the scope of parameters design and is not handed by *Configurator* class, these details are

not covered herein. Additionally changes made to XML document are limited to parameters with *preset* and *perist* properties shown in Figure 35.

The constructor of a factory automatically adds the factory to a relevant lookup tables in the *Configuration* subsystem. Then the *Configurator* looks these maps up to locate the proper factories and create software components specified in the XML configuration files.

Creation of IO objects

Recall from the example of Figure 29 that different IO controllers where described by different XML elements each having a unique tag names such as *rf_gpio_controller* or *rf_spi_controller*. The names parallel that of the concrete subclasses of *Io::IoController* class. The abstract *createIoController* method of the *IoControllerFactory* accepts a pointer to the element in the XML document that describes the specification of a particular input/output controller. A derived class of *IoControllerFactory* overloads the *createIoController* method to parse and build a specific controller known to that factory.

A number of concrete derived classes of *IoControllerFactory* that are used in RF Generator are depicted in class diagram of Figure 37. The constructor of *IoControllerFactory* adds the created factory instance to *IoControllerFactoryMap*. Each factory has the tag name for the XML element that has to process and a ranking number. The newly created factories are added to a map in a non-descending order determined by their ranking number. This is to facilitate the order of creation of the IO controllers because creation of one Io controller may depend on existence of another IO

Controller. Typically for specifying dependencies a topological sorting (32) of the controllers is required, but at this time a simple sorting of the controller factories has been sufficient. The factories are ordered by assigning a rank number to each of them, then the *add* method of the *IoControllerFactory* places the newly created factory on list that is kept sorted by rank numbers of factories.

FIGURE 37: *IOCONTROLLERFACTROY* AND EXAMPLES OF ITS CONCRETE CLASSES

The *createIoControllers* method of the *Configurator* traverses this sorted list of factories, retrieves the tag name of each factory, retrieves the XML element in the configuration document corresponding to the tag name, and passes the element to

createIoController method of the visited factory to create the IO controller in the correct order.

CREATION OF DEVICE OBJECTS

Construction of software devices is similar to that of IO controllers in use of the factories but the ordering required by controllers does not exist among devices. Instead, devices form a tree structure hierarchy that is dictated by the XML schema for devices as depicted in Figure 33. The *createDevice* method of *Configurator* is recursively applied to *device* elements in the document. This method uses the *model* attribute of a device specification, shown in the schema definition of Figure 33, to locate the factory for the that device and calls the *create* method of the factory, passing the device name and other device specifications such as its IO and parameters in an object of *Context* class, the overloaded *create* method in the located factory knows how to create the appropriate software device, passing the context to the device for its initialization. The UML class diagram of Figure 38 shows a number of device factories used in the RF Generator. The qualifier relation from *DeviceMap* singleton to *DeviceFactory*, implies that the *DeviceMap* can be used to look up a *DeviceFactory* by model name of the device.

FIGURE 38: SOFTWARE DEVICE FACTORIES EXAMPLE

The *DeviceMap* singleton includes another look up table, depicted in UML class diagram of Figure 39 that contains the crated devices. This is used for locating devices by their names.

FIGURE 39: LOOK UP TABLE FOR SOFTWARE DEVICES

CONTROLLER

The *Controller* subsystem consolidates the operation of all other subsystems into a whole functioning medical device controller. The package implements the following functionalities:

- Accepting and executing commands from a client, the client may be a remote entity on a network or a local GUI resident in the same executable as the controller.
- Sending response status of commands and system status to clients.
- Synchronizing startup and shutdown operations across the input/output, software devices, and monitor subsystems
- Other functions specific to a given medical device controller, such as checking heartbeats from an external client and taking suitable safety reaction upon loss of heartbeat to prevent injury or damage

The *Controller* is comprised of a number of core objects, including an instance of *SystemController* class, a slew of *Command* objects (6), two message queues (33) for queuing incoming commands and outgoing responses. *SystemController* also includes implementation of response to emergencies, and recording of runtime stack back trace for analysis of crashes that may occur during development. Depending on the medical device, *Controller* may include other components, such as network communication components for sending and receiving messages to/from remote clients.

The class diagram of Figure 40 depicts the core classes of the *Controller* and their relations, descriptions are covered in the following sections.

FIGURE 40: CONTROLLER CLASSES

COMMAND OBJECTS

The command design pattern is used to incorporate requests in an object in order to allow queuing object and processing them by other threads (6). The *Command* abstract class implements this pattern, its subclasses implement semantics of carrying out

requests by overloading the *execute* method of the *Command* class. *Commands* objects are constructed from incoming request and are placed on the request queue for consumption of the device thread, which drives the operation of software devices. The device thread picks the object and calls the *execute* method of the command thus carrying out the intent of the request in its context. Figure 40 shows a number of *Command* subclasses that are used on both medical devices.

IoCycleNotificationCommand is used to communicate end of IO cycle and availability of new data to the application. Recall that *IoCycle* object publishes a signal when it completes IO for a cycle. *SystemController* class subscribes to the signals from all existing *IoCycle* objects and upon receiving a signal, creates and places an instance of the *IoCycleNotificationCommand* on the request queue. The consequence of executing the command, in the context of the device thread, is invocation of IO callback functions of software devices, and thus their reaction to IO events, in the context of device thread.

The *Startup* and *Shutdown* commands as their names imply are used to start up the *Controller* and shut it down (more on this in *SystemController* section).

EquipmentCommand is the abstract base class of commands that are to be forwarded to the root device, i.e. the equipment. Derived classes of *EquipmentCommand* implement the commands that control the operation of the equipment and are specific to a given medical device.

Recall that devices employ a request and response protocol, the result of execution of an *EquipmentCommand* is a success/failure response from the equipment that is

encapsulated in another type of the *Command* object and is placed in the response queue by the device thread. The consumer of the messages from this queue is the GUI thread or communication thread that dispatches the response to the external client.

MESSAGE QUEUE

The message queuing pattern allows threads to pass information to one another; it is simple and because the access is serialized through the message queue, mutual exclusion problems that arise with concurrent access of multiple threads are eliminated (26). *MessageQueue<T>* class is implemented as a circular buffer, where *T* stands for the type of objects placed in the buffer. The access to the buffer is limited to *put* and *get* methods. These two methods synchronize access to the buffer by using a mutex, which ensures mutual exclusion to buffer between separate threads that call *put* and *get* methods. A condition variable (29) is used to communicate information about the empty or full status of the ring buffer, between the threads that run through *put* and *get*. A thread that attempts to put an item on a full queue is blocked until such time that another thread gets an item from the queue and signals the blocked thread through the condition variables about the vacancy state. Likewise a thread that attempts to get an item from empty queue is blocked until another thread puts an item on the queue and signals the thread that has been blocked for availability of items for consumption. Two message queues are used in this architecture: request and response queues. The request queue is used to communicate incoming commands from the GUI (or remote client) and end of cycle notifications from *IoCycle* threads to the device thread. The response queue is used to communicate status of commands or other system status, typically from

device thread, to the remote client or the GUI. Both of the queues can be instantiated as a *MessageQueue* of pointers to *Command* objects. The *put* method places its command argument at the rear of the queue and the *get* method picks the command at the front of queue, effectively implementing a first-in-first-out (FIFO) strategy. The *put* method accepts a second position argument that by default is set to queue rear but can be explicitly set to point to front of the queue, to communicate emergency conditions that need to be handled with a higher priority.

MEMORY POOLS

The command objects, except for instances *IoCycleNotificationCommand* are dynamically created to include parameters of the incoming request. Using the standard C++ *new/delete* operators to allocate and release heap-based memory may eventually, at some instance of time, lead to heap memory fragmentation and the need to compact the heap by the underlying memory manager. This could potentially interfere with the predictability needed in a real-time system to meet the deadlines. To avoid this issue, the dynamic creation and release of command objects is done from pre-allocated memory pools. The *new/delete* operators are overloaded to get an instance of a specific derived class of *Command* from its memory pool and once done return the object to its pool. Maximum size of memory pools, in terms of number of objects, is defined as the size of the *MessageQueue* (default 7) for incoming objects. This may lead to unused memory which is an acceptable tradeoff in avoiding failed deadline risks.

SYSTEMCONTROLLER CLASS

SystemController class is responsible for configuring the medical device on power up, startup and shutdown of the medical device, processing of commands that direct the behavior of the device, and dispatching the outcome of the commands to their point of origin.

The *initialize* method of the controller calls on the *Configurator* to parse and set up the software components from the specifications included in the configuration file. The method also registers *onCycleEnd*, *onMonitorEmergency*, *onClientMissedHeartbeat*, and *onMissedCycleLimit* functions to intercept signals regarding end of IO cycle, emergencies from monitors, and missed heartbeat signals.

The *emergency* method may directly be called by the equipment in case the device hierarchy runs into an emergency condition when running a command.

The *run* method effectively creates the device thread that loops to get and execute commands from the command queue and exits after completion of the shutdown command. Recall that the thread blocks if the queue is empty.

The startup and shutdown commands call *startup* and *shutdown* methods of the *SystemController*, which in turn handle starting up and shutting down of IO service, devices and monitors. The order of startup is IO, devices, and then monitors because IO should be running before devices can accept commands that may actuate output. The order of shutdown is monitors, devices, and then IO; otherwise devices may actuate output on an IO that already has been stopped thereby causing untoward results. The implementation of *onMonitorEmergency*, *onClientMissedHeartbeat*, and

onMissedCyceLimit is system specific. In AF1 medical device, the reaction is to disable the cyclic output that keeps hardware watchdog engaged, so as to cut off the power to components that may risk safety, and then put an emergency machine off (EMO) command on the command queue to take equipment device to failure state which rejects all incoming commands but shutdown.

RESULTS

DEVELOPMENT TIMES

The design of the software architecture to implement the embedded and real-time controller for medical device AF1 began in July of 2008 and the first internal release for *in vitro* lab testing was delivered in January 2009 by a team consisting of four software engineers of which one was primarily focused on the architecture and overall design and the other three primarily focused on the implementation of application logic within the device framework. The first non-GLP acute animal study using this architecture was conducted in March 2009. This release included the *IO* framework, IO extensions including a major one whose design has not elaborated herein, logging, communication and integration with remote client, and roughly 70% of the application logic.

Configuration and *Metrics* components were added later and bulk of the subsequent modifications thereafter were in software devices due to refined product requirements and new algorithms from the research group. Prior to development of *Configuration*, changes required to system parameters required a new release, an effort that would take from an hour to one-half day depending on the level of testing; whereas, with availability of *Configuration*, the research, development, and verification staff have

been able to vary system parameters and log levels in matter of minutes in trying out new features. In addition, the software team has been able to vary the thread policies and priorities to collect metrics runs for performance evaluation without costly software builds. Both software and verification teams have been able to modify log levels to produce runtime information need for tracing or testing of the software. This product is in last stages of development and testing.

The development of Intuity Radio Frequency Generator began around January 2010 with two developers using the same architecture. A prototype was demonstrated in May 2010 at Heart Rhythms Society (HRS) conference in Denver. Now with five developers on board, the software development of this product is near completion of its functional requirements. Based on our experience, the turnaround times for software have been relatively short, with fewer software engineers, for products with similar complexities.

SOFTWARE PROBLEM REPORTS

There have been several software verification and validation testing and some ten animal studies to evaluate system performance hardware, mechanical, algorithm, and software performance *in vivo* for AF1. Figure 41 shows the distribution of Software Work Requests (SWR) by category for medical device AFJ through June of 2010. A SWR is indicative of a software bug or any changes needed after formal reviews have been conducted. The *GUI* category refers to the remote client software and its GUI, *Application* refers to the software components within the controller that implement the application logic, this would include mostly software devices that handle functionality for that medical device, *Architecture* refers to the architecture components that have

been discussed in this document, including IO extensions for that medical device, *CLNI* (close, no issue) refers to the collective SWRs that were filed but upon further investigation were found to be non-issue. With a total number of 224 SWRs, including CLNI and change requests for the algorithms, based on our experience, this is fairly low number for a project of this complexity.

FIGURE 41: DISTRIBUTION OF SOFTWARE WORK REQUEST

PERFORMANCE RESULTS

It was mentioned earlier in the IO framework that the architecture has provisions for signaling missed IO cycle, and IO cycle limits. The former is an indication that the readers and writers of a given IoCycle object were unable to complete their read/write operations within the deadline; the latter indicates that N IO cycles, where N is determined of safety requirements of a given system, have occurred in a sequence. Typically N is set to 1 or 2 at most. In medical device AFJ, the first signal is logged as a warning and the second signal is generated for N=2 for a 50ms IoCycle that includes

all the readers. The assumption is that two missed cycles may be an indication of hardware failure and should transition the system to a state that will not execute any further commands. Additionally, device AFJ has a hardware watchdog timer that is engaged every 90ms (with an actual hardware deadline of 100ms). The hardware watchdog is safety provision to disable the system if software is not functional to be on time to engage the watchdog. There have been challenges in meeting these deadlines, in fact during the testing missed IO cycles had become a norm rather than an exception. As a result *Metrics* facilities were added to the architecture in order to instrument key components and collect runtime stats. These include reader, writer, all IO callback processing, command execution, and communication times. The times collected are average and max durations for each item. There are scripts devised to test system under heavy load to create worst-case scenarios, to quote (26): “in hard real-time and safety critical systems, worst-case delivery time is the most important concern.” As a result the times are considered in meeting the deadlines. One of the obvious known reasons for missed cycles was lack of real-time thread policy and thread priority settings. The implementation of software used *boost* for its threading needs, *while* boost has an excellent array of software development tools, the threading facilities are designed to accommodate a slew of platforms and do not address real-time threading aspects at this time. Using *boost* and POSIX thread primitives, a thread library was created that allowed modifying thread policy and priority after creation of a thread and before letting the newly created thread begin its intended task. Another refinement was creation of the response queue for transmitting responses to the remote client in a thread with lower priority of the two IO cycles and device threads. These steps resolved the

watchdog timer deadlines but missed IO on the 50ms cycle were still observed. After consulting with MontaVista, the developer of the real-time operating system used in device AFJ, the resolution was to migrate from MontaVista Linux 5 Pro to MontaVista Linux 6. The migration has resolved the issue of two consecutive missed IO cycle limits.

It should be noted that the collection of metrics adds some overhead to the execution time. The overhead consists of getting the system time at the beginning and end of the measured items to compute durations and update number of times that the item has been called. This overhead is not present when metrics collection is not required because the C/C++ macros that instrument the code for metrics are conditionally compiled and are eliminated for production code.

CONCLUSION AND FUTURE WORK

The results obtained from implementation of the architecture and software development for two new generation of medical devices based on the explored architecture have proven to be successful in terms of timely software development turnaround time, low number of software bugs in the architecture and the application, and in terms of meeting real-time deadlines for product AF1 and AF2. Validation tests, animal studies and performance tests in terms of meeting real-time requirements have been successful. AF1 is undergoing further tests to evaluate performance of the new algorithms. While these algorithms are not part of the architecture and are meant to enhance the intended objective of the medical device, their implementation relies on the existing architecture.

AF2 is in the last stages of the software development and is developed on a hardware platform entirely different than AF1. The ability to reuse the architecture and develop new IO extensions in a relatively short time with merely two developers has demonstrated viability of the architecture for reuse and its flexibility to change. Both of the products are scheduled to undergo a change in their platform hardware for final production, using foundation classes and the capabilities discussed earlier make the software changes fairly straightforward and short.

Design of the architecture will evolve to meet the unexpected needs, for example, the *Controller* has been revised to incorporate more than one outgoing message queue in order to accommodate several destinations with varying data transmission frequency and priorities. The IO cycle is in the process of being decomposed into recurring and

non-recurring cycles, with recurring cycle operating as before and the non-recurring to be used in cases that the IO is not driven by a recurring timer but is triggered on demand.

In conclusion we believe that sound application of object-oriented design techniques together with established design patterns, use of object-oriented programming language (C++), and building with foundation classes such as *STL* and *boost* had led to a short development time with robust software architecture. However, based on prior experience and observations, we feel it is imperative to quote Booch as a caveat to this approach: “Object-oriented analysis and design may be the only method we have today that can be employed to attack the complexity inherent in very large systems. In all fairness, however, the use of object-oriented development may be ill-advised for some domains, not for any technical reasons, but for nontechnical ones, such as the absence of suitably trained staff or good development environment.”

REFERENCES

1. *A Survey of Software Engineering Techniques in Medical Device*. **Feldmann, Raimund L., et al.** s.l. : IEEE Computer Society, 2007. 2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability. pp. 46-54.
2. *A general model of software architecture design derived from five industrial approaches*. **Hofmeister, Christine, et al.** The Journal of Systems and Software, s.l. : Elsevier, 2007, Vol. 80.
3. **IEEE**. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Standards Association*. [Online] 1990.
http://standards.ieee.org/reading/ieee/std_public/description/se/610.12-1990_desc.html.
IEEE Std 610.12-1990 .
4. **Dijkstra, Edsger W.** Programming Considered as a Human Activity. [book auth.] Edward Nash Yourdon. *Calssics in software engineering*. New York : Yourdon Press Compusting Series, 1979.
5. **Booch, Grady**. *Object-oriented Analysis and Design With Applications, Sec.* Santa Clara, CA : The Benjamin/Cummings Publishing Comapny, 1994.
6. **Gamma, Erich, et al.** *Design Patterns: Elements of Reusable Object-Oriented Software*. s.l. : Addison-Wesley, 1994.
7. **Stroustrup, Bjarne**. *The C++ Programming Language, Special Edition*. s.l. : Addison-Wesley, 2000.
8. **Henzinger, Thomas A. and Sifakis, Joseph**. The Embedded Systems Design Challenge. *FM 2006: Formal Methods*. s.l. : Springer Berlin / Heidelberg, 2006, Vol. 4085.
9. **Laplanche, Philip**. *Real-time Systems Design and Analysis, Third edition*. s.l. : IEEE Press/Wiley-Interscience, 2004.
10. **Dibble, Peter C**. *Real-time Platform Java Programming 2nd Edition*. s.l. : Dibble, 2008.
11. **OMG**. UML Resource Page. *OMG*. [Online] 2.0, Object Management Group, 6 2, 2010. [Cited: 6 25, 2010.] <http://www.uml.org/>.
12. **American Heart Association**. Radiofrequency Ablation. *American Heart Association*. [Online] 2010. [Cited: 7 16, 2010.]
<http://www.americanheart.org/presenter.jhtml?identifier=4682>.
13. **Cesario, David, et al.** Radio-Frequency Ablation. [book auth.] Andrea Natalle and Jose Jalife. *Atrial Fibrillation from Bench to Bedside*. s.l. : Human Press, 2008.
14. **STL**. [Online] 2009.
15. **boost**. [Online] 2009.
16. **Apache XML Project**. Xerces-C+ XML Parser. [Online] August 21, 2010 .
<http://xerces.apache.org/xerces-c/>.
17. **W3C**. Extensible Markup Language (XML) 1.0 (Fifth Edition). [Online] November 26, 2008. <http://www.w3.org/TR/REC-xml/>.
18. **Altova**. XMLSpy Profssional Edition . [Online] 2010.
http://www.altova.com/download/xmlspy/xml_editor_professional.html.
19. **Eclipse Foundation**. Eclipse Platform. *Eclipse*. [Online] 2010.
<http://www.eclipse.org/platform>.

20. **W3C.** XML Schema. W3C. [Online] October 28 , 2004.
<http://www.w3.org/standards/xml/schema>.
21. **GNU Foundation.** GCC 4.2 Release Series. *GCC, the GNU Compiler Collection* . [Online] 4.2.0, GNU Foundation, May 13, 2007. [Cited: June 29, 2010.] <http://gcc.gnu.org/gcc-4.2/>.
22. **Red Hat.** Red Hat Enterprise Linux 5. *redhat*. [Online] RedHat. [Cited: June 29, 2010.] <https://www.redhat.com/rhel/>.
23. Real-Time Linux. *MontaVista*. [Online] MontaVista. [Cited: June 24, 2010.] http://www.mvista.com/real_time_linux.php.
24. **Pfeiffer, Olaf, Ayre, Andrew and Keydel, Christian.** *Embedded Networking with CAN and CANopen*. s.l. : Copperhill Media Corporation, 2008.
25. **Stroustrup, Bjarne.** *The C++ Programming Language: Special Edition*. New Jersey : Addison-Wesley, 2000.
26. **Douglas, Bruce.** *Real-time Design Patterns: Robust Scalable Architecture for Real-time Systems*. s.l. : Addison-Wesley, 2003.
27. *A New and Useful Template Technique: Traits*. **Myers, Nathan**. s.l. : C++ Report, June 1995, C++ Report, pp. 32-35.
28. **Heath, Steve.** *Embedded Systems Design, Second Edition*. s.l. : Newness, An Imprint of Elsevier Science, 2003.
29. **Butenhof, David R.** *Programming with POSIX (R) Threads*. s.l. : Addison-Wesley, 1997.
30. **Pfeiffer, Olaf, Ayre, Andrew and Keydel, Christian.** *Emergencies*. s.l. : Copperhill Technologies Corporation, 2008. pp. 87-88.
31. *Esophageal injury and temperature monitoring during atrial fibrillation*. **Singhe et al.** 2008, Circulation: Arrhythmia and Electrophysiology.
32. **Rosen, Keneth. H.** *Discrete Mathematics and Its Applications 4th ed.* s.l. : McGraw-Hill, 1999.
33. **Douglass, Bruce Powel.** *Real-Time Design Patterns: Roubust Scalable Architecture for Real-Time Systems*. s.l. : Addison-Wesley, 2003.