# UNIVERSAL ENGINEERING PROGRAMMER - AN IN-HOUSE DEVELOPMENT TOOL FOR DEVELOPING AND TESTING IMPLANTABLE MEDICAL DEVICES IN ST. JUDE MEDICAL

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

Khoa Tat Do

March 2011

© 2011

Khoa Tat Do

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: UNIVERSAL ENGINEERING PROGRAMMER - AN IN-HOUSE DEVELOPMENT TOOL FOR DEVELOPING AND TESTING IMPLANTABLE MEDICAL DEVICES IN ST. JUDE MEDICAL

AUTHOR: Khoa Tat Do

DATE SUBMITTED: March 4th 2011

COMMITTEE CHAIR: Lily Laiho, Dr. and Professor

COMMITTEE MEMBER: Robert Crockett, Dr. and Professor

COMMITTEE MEMBER: Kristen O'Halloran Cardinal, Dr. and Professor

## ABSTRACT

UNIVERSAL ENGINEERING PROGRAMMER - AN IN-HOUSE DEVELOPMENT TOOL FOR DEVELOPING AND TESTING IMPLANTABLE MEDICAL DEVICES IN ST. JUDE MEDICAL

During development and testing of the functionality of the pacemaker and defibrillator device, engineers in the St. Jude Medical Cardiac Rhythm Management Division use an in-house development tool called Universal Engineering Programmer (UEP) to ensure the device functions as expected, before it can be used to test on an animal or a human during the implantation process. In addition, some applications of UEP are incorporated into the official releases of the device product. UEP has been developed and used by engineers across departments in the St. Jude Medical Cardiac Rhythm Management Division (CRMD). This thesis covers the flexible and reusable design and implementation of UEP features, to allow engineers to easily and effectively develop and test the devices.

Keywords: Universal Engineering Programmer, St. Jude Medical, Cardiac Rhythm Management Division

# ACKNOWLEDGMENTS

First, I would like to thank Cal Poly, College of Engineering, and Dr. Lily Laiho for her input on the technology content for this thesis. Also I would like to thank Dr. Kristen O'Halloran Cardinal and Dr. Robert Crockett for being committee members on my thesis. Finally, I would like to thank Rich Jew, Product Manager and Muthuvale Shanmugam, Project Manager of Tools Systems at St. Jude Medical for giving me the opportunity to develop the Universal Engineering Programmer (UEP) project and for supporting me as needed.

## TABLE OF CONTENTS

**Page**

ACRONYMS AND ABBREVIATIONS……………………………………..viii

LIST OF FIGURES………………………………………………………x

LIST OF TABLES……………………………………………………… xiii

I. INTRODUCTION………………………………………………..1

Anatomy of the heart……………………………………………..1

Cardiac Rhythm of the heart……………………………………...2

Cardiac Rhythm Management Devices of St. Jude Medical………. 5

The Device Development Cycle……………………….…………8

Existing Testing Tools and the issues………………………………9

Universal Engineering Programmer (UEP)…………………………11

II OBJECTIVES……………………………………………………… 12

III METHODS AND MATERIALS…………………………………...15

Architecture…………………………………………………………15

UEP Graphical User Interface (GUI)………………………………16

UEP Application Program Interface (API)………………………….34

UEP Common Object Model (COM)……………………………….38

UEP – Summary of Benefit…………………………………………41

Key Features………………………………………………...42

Functionalities……………………………………………….43

Benefits……………………………………………………...44

VI. UEP AND THE TEST FRAMEWORK……………………………50

Background……………………………………………………………50

The Motivation…………………………………………………...…52

Legacy approach……………………………………….…52

Vision of Future Testing………………………………….53

UEP Verification Test Framework……………………………….54

Test Framework Components…………………………………….55

Test Suite…………………………………………………55

The Configuration Manager……………………………...58

Managed UEP API Wrapper………………………………. 59

Configuration files……………………………………… 59

Summary of Benefits of UEP Test Framework……………………60

Future Plans…………………………………………………………61

V. NeXus – UEP NEXT GENERATION…………………………...62

VI. CONCLUSION AND FUTURE WORKS……………………......65

Conclusion……………………………………………………….....65

Future works………………………………………………………...67

REFERENCES…………………………………………………………...69

APPENDIX A - Sample of XML files...........................................71

APPENDIX B – Sample of UEP Test Scripts……….……………...76

APPENDIX C – Sample of Test Library Code……….……………79

APPENDIX D – Sample of Firmware Bench Testing code...………84

APPENDIX E – Sample of SMART code…………………………..85

## ACRONYMS AND ABBREVIATIONS

| | |
|---|---|
| CRMD | Cardiac Rhythm Management Division |
| UEP | Universal Engineering Programmer |
| RA | Right Atrium, one of the four main chambers of the heart |
| RV | Right Ventricle, one of the four main chambers of the heart |
| LA | Left Atrium, one of the four main chambers of the heart |
| LV | Left Ventricle, one of the four main chambers of the heart |
| VT | Ventricular Tachycardia |
| VF | Ventricular Fibrillation |
| ICD | Implantable Cardiac Defibrillator |
| V&V | Verification and Validation |
| EPWorkSpace | An old engineering tool |
| ATE | Automation Testing Equipment |
| DMA | Direct Memory Access |
| RTEGM | real time electrograms |
| STEGM | stored electrograms |
| COM | Common Object Model |
| GUI | Graphical User Interface |
| API | Application Programming Interface |
| RF | Radio Frequency |
| SMART | System for Making Automated and Random Test |
| EIIS | External Instrument Interface Specification |
| XML | eXtensible Markup Language. |

| | |
|---|---|
| DTM | Digital Telemetry Module |
| IDL | Interface Definition Language |
| Tcl, Perl,VBScript | Programming scripting languages |
| DCP | Device Clinical Parameters |
| RAM | Random Access Memory |
| ROM | Read only Memory |
| DOORS | Dynamic Object-Oriented Requirements System |
| DLL | Dynamic Link Library |

## LIST OF FIGURES

**Figure** **Page**


Figure 1: Anatomy of the heart and the blood circulation……………………………….1

Figure 2: The electrical system of the heart………………………………………......4

Figure 3: St. Jude Medical CRMD - Implantable Cardiac Defibrillator………….. .....5

Figure 4: St. Jude Medical CRMD - Pacemaker……………………………….…......5

Figure 5: St. Jude Medical CRMD – Merlin Patient Care System……………………7

Figure 6: Phases of device development – why another Programmer?........................8

Figure 7: EPWorkspace – An application to develop and test CRMD devices........…10

Figure 8: UEP Architecture - building blocks part of various systems……………….13

Figure 9: UEP Architecture – components…………………………………………...16

Figure 10: UEP GUI…………………………………………………………………..18

Figure 11: Close Physical Channel and Open Logical Channel GUI command……...20

Figure 12: Sequences to send a mailbox command to the device and response

data from the device……………………………………………………….22

Figure 13: Macro in UEP……………………………………………………………..23

Figure 14: Memory Watch feature in UEP GUI……………………………………...25

Figure 15: Direct Memory Access – Block Read feature in GUI…………………….26

Figure 16: Direct Memory Access – Block Write in GUI……………………………27

Figure 17: Direct Memory Access - General mailbox command feature in GUI…….28

Figure 18: Direct Memory Access - Unity XIMailbox command feature in GUI…....29

Figure 19: Interrogate command feature in GUI……………………………………...29

Figure 20: Set and Get Parameters values and Program the device in GUI…………..30

Figure 21: Device Clinical Parameter Viewer allows user to view and change the parameters……………………………………………………..30

Figure 22: Set up Real Time Electrograms GUI……………………………………...31

Figure 23: Start and Stop Real Time Electrograms GUI……………………………..31

Figure 24: Real Time EGM display in UEP GUI…………………………………….32

Figure 25: Stored EGM displayed in UEP…………………………………………...33

Figure 26: BlockRead – Device Access Memory feature supported by UEPAPI…...34

Figure 27: BlockReadLong – Device Access Memory feature supported by UEPAPI…………………………………………………………………...35

Figure 28: BlockWrite – Device Access Memory feature supported by UEPAPI…...35

Figure 29: Mailbox and MailboxXI – Device Access Memory feature supported by UEPAPI…………………………………………………………………..35

Figure 30: Interrogate and ProgramDevice – Interrogate and ProgramDevice supported by UEPAPI…………………………………………………….36

Figure 31: SetDCPValue – Interrogate and ProgramDevice supported by UEPAPI…………………………………………………….36

Figure 32: GetDCPValue – Interrogate and ProgramDevice supported by UEPAPI……………………………………………………..36

Figure 33: RTEGM and Stored EGM feature supported by UEPAPI……………….37

Figure 34: Device Access Memory feature supported by UEP COM interfaces…….39

Figure 35: Interrogate and Program Device feature supported by UEP COM Interfaces……………………………………………………………….39

Figure 36: RTEGM and Stored EGM feature supported by UEP COM interfaces….40

Figure 37: UEP as programmer for first ICD RAM device…………………………47

Figure 38: UEP as a programmer for Pig Ischemia study……………………………48

Figure 39: UEP as a programmer for Lead Impedance study………………………..48

Figure 40: UEP as a programmer for Human Morphology study……………………49

Figure 41: UEP is used to develop Holter Monitor…………………………………..49

Figure 42: Sample of a UEP test script………………………………………………51

Figure 43: UEP Test Framework Architecture………………………………………55

Figure 44: the Team System GUI editor……………………………………………..57

Figure 45: test suite is organized in the UEP Test Framework……………………....57

Figure 46: UEP Test Framework Configuration……………………………………..58

Figure 47: Test Data Configuration in UEP test Framework ………………………..60

Figure 48: UEP NeXus Graphical User Interface…………………………………....62

Figure 49: UEP NeXus a Widget Framework………………………………………..63

Figure 50: UEP participates in phases of device development in CRMD……………66

## LIST OF TABLES

**Table** **Page**

Table 1: UEP commands supported by the 3 major components: GUI, API and COM 19

Table 2: UEP components being used by groups across CRMD 44

# I. INTRODUCTION

## Anatomy of the Heart

The heart is a muscular organ located in the upper body chest area between the lungs. The main purpose of the heart is to pump blood around the body. The heart is divided into separate right and left sections by the septum. Each of the right and left sections is also divided into upper and lower compartments known as aria and ventricles respectively. There are four main chambers of the heart; they are: Right Atrium (RA), Right Ventricle (RV), Left Atrium (LA), and Left Ventricle (LV) as illustrated in Figure 1.

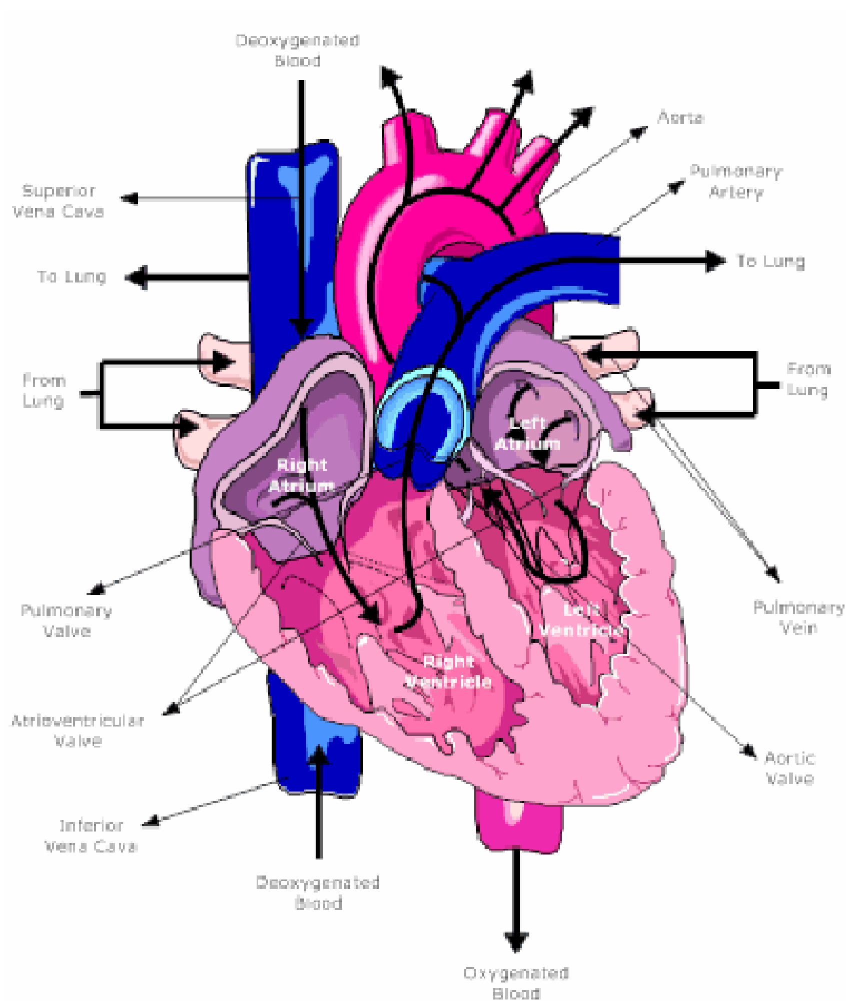



*Figure 1: Anatomy of the heart and the blood circulation*

The atrium is the receiving chamber and the ventricle is the ejecting chamber. The deoxygenated blood from the body is pumped to the right atrium and then to the right ventricle. This blood is then pumped to the lungs by the right ventricle. In the lungs, this deoxygenated blood becomes oxygenated, and then enters the left atrium of the heart. The oxygenated blood is then pumped to the rest of the body through the left ventricle. The functions of the heart can briefly be described as delivering the oxygen and nutrients such as glucose, electrolytes, etc. to the tissue. The heart also picks up wastes and carbon dioxide from the cells.

**Cardiac Rhythm of the heart**

The four chambers, RA, LA, RV and LV, consist of cells that are specialized in such a way that they can beat on their own. The Sino atria node or SA node is the natural pacemaker of the heart that is located in the RA and generates electrical impulses causing the cardiac muscle of the heart to contract at a paced interval called the heart beat. Typically, the frequency of this impulse is around 60 to 70 beats per minute at rest. Besides the main pacemaker, there are other pacemakers such as the Atria Ventricular node (AV node) which is also known as the secondary pacemaker. It has an intrinsic rate of 40 to 60 beats per minute. The bundle of His is known as the AV bundle or atrioventricular bundle, and it has an intrinsic rate of around 30 to 40 beats per minute.

The electric impulse generated from the SA node is normally conducted through the AV node because this provides the path of least resistance for the impulse to proceed to the ventricles. The conduction is then delayed in the AV node, which allows for maximal

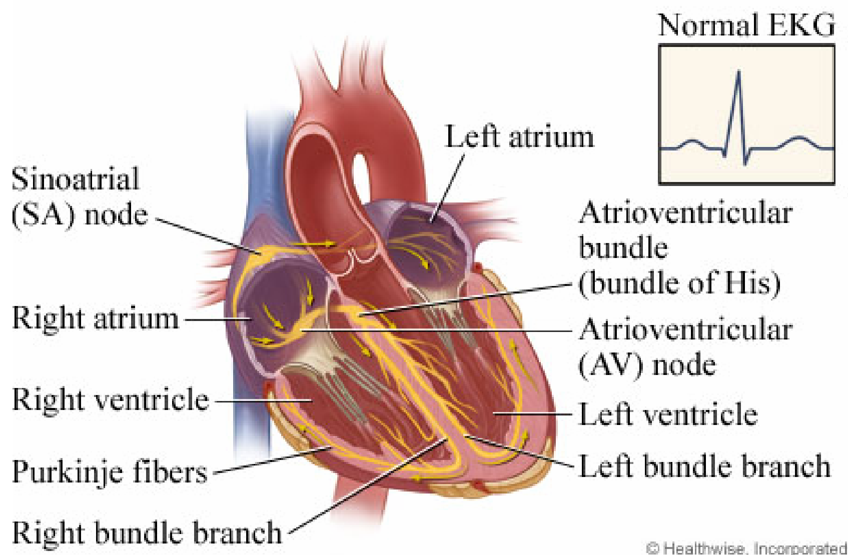
ventricular filling with blood. The bundle of His is located distal to the AV node within the ventricular septum and continues the electrical impulse from the AV node down to the ventricles. It then divides into the left and right bundle branches, which are responsible for conducting the impulse to their respective ventricles. The left and right bundle branches ultimately subdivide into a complex network of conducting fibers, named the Purkinje fibers. The conduction velocity in the Purkinje fiber is faster than any other conduction velocity system within the heart with a velocity, estimated to be in the range of 1 to 4 m/s. This allows for rapid endocardial ventricular activation.

Normally, for an adult at rest, the heart rate is between 60 to 100 beats per minute. However, the heart rate can also be irregular. The abnormally slow heart rate is called bradycardia and it is usually less than 60 beats per minute. Tachycardia is an abnormal rate that is faster than 100 beats per minute.

Tachycardia can be very dangerous. When the heart beats rapidly, the heart pumps less efficiently and provides less blood flow to the rest of the body including the heart itself. The increased heart rate also leads to increased work and oxygen demand for the heart, which could perhaps cause a heart attack. The two common potentially life-threatening tachycardias are ventricular tachycardia (VT) and ventricular fibrillation (VF). Ventricular tachycardia is a fast heart rhythm caused by electrical impulses originating in one of the ventricles. Ventricular tachycardia can decrease blood delivery by the heart and progress to a more serious heart rhythm called ventricular fibrillation. Ventricular fibrillation is fast and irregular rhythm, which causes the heart's beats to be so fast and

irregular that the heart stops pumping blood. Ventricular Fibrillation is a leading cause of sudden cardiac death.

Bradycardia means the heart beats slower than the normal heart rate of 60 to 100 beats caused by the heart’s electrical signals (Figure 2). The issue may be from the SA Node or the heart's natural pacemaker not working properly or from the electrical pathways of the heart being disrupted by damages from diseases. As a result, there will not be enough blood to meet the body's needs. There can be many symptoms associated with bradycadia such as shortness of breath, lightheadedness, dizzy, tiredness, chest pain, and fainting, and worst of all, it can be life threatening.




The heart's electrical signal travels through pathways (yellow). This stimulates your upper chambers (atria) and lower chambers (ventricles) to contract.

*Figure 2: The electrical system of the heart*

**Cardiac Rhythm Management Devices of St. Jude Medical**

St. Jude Medical Cardiac Rhythm Management Devices are divided into two lines of product, the Pacemaker and Implantable Cardiac Defibrillator (ICD).

*Figure 3: St. Jude Medical CRMD – Pacemaker*

*Figure 4: St. Jude Medical CRMD - Implantable Cardiac Defibrillator*

The Pacemaker (Figure 3) is a small implantable device used for treating bradycardia that sends electrical pulses to the heart whenever it senses that the heartbeats are too slow. The Pacemaker is used to maintain an adequate heart rate either because the heart's native pacemaker is not fast enough or there are issues that block the heart's electrical conduction system. On the other hand, the ICD (Figure 4) treats tachyarrhytmia. An ICD is a miniaturized computer that monitors the heart's rhythm for very fast and potentially dangerous rhythm disorders, and it delivers therapy when a dangerously fast heart rhythm is detected. As soon as an abnormal heart rhythm occurs, the ICD can send a shock to the heart muscle to defibrillate it or stop the cycle of rapid twitching.

The Pacemaker or ICD is implanted under the skin in the patient's upper left shoulder and is connected to the heart by leads or thin wires. The leads are inserted through a vein to the heart where one end is connected to the hearth muscle and the other end is connected to the Pacemaker or ICD. The leads sense the heart rhythm and transmit this information to the device, which adapts its response to the patient's needs.

The Pacemaker and ICD are used to maintain the patient's hearth rhythm. They are also stored information that can be used by the physicians to program the device. The device can help patient live a longer, more productive and healthier life.

**Merlin Programmer**

In addition to the two Cardiac rhythm management devices at St. Jude Medical, the Pacemaker and the ICD, another device called the Merlin programmer is included as part of the CRMD product line.

*Figure 5: St. Jude Medical CRMD – Merlin Patient Care System*

Merlin Patient Care System (Figure 5) is basically a computer that cardiac care clinicians use to retrieve and analyze data from implanted ICDs and pacemakers and make programmatic changes to them. Merlin Patient Care System is intuitive and easy-to-use system that can help clinicians retrieve the patient reports which are stored in the

implanted device very quickly and effectively. Clinicians can easily acquire real time sense and impedance measurements with the press of buttons. The Merlin Patient Care System can help clinicians tailor to patient's specific needs; and it makes so simple for clinicians to conduct patient follow-ups.

**The device development cycle**

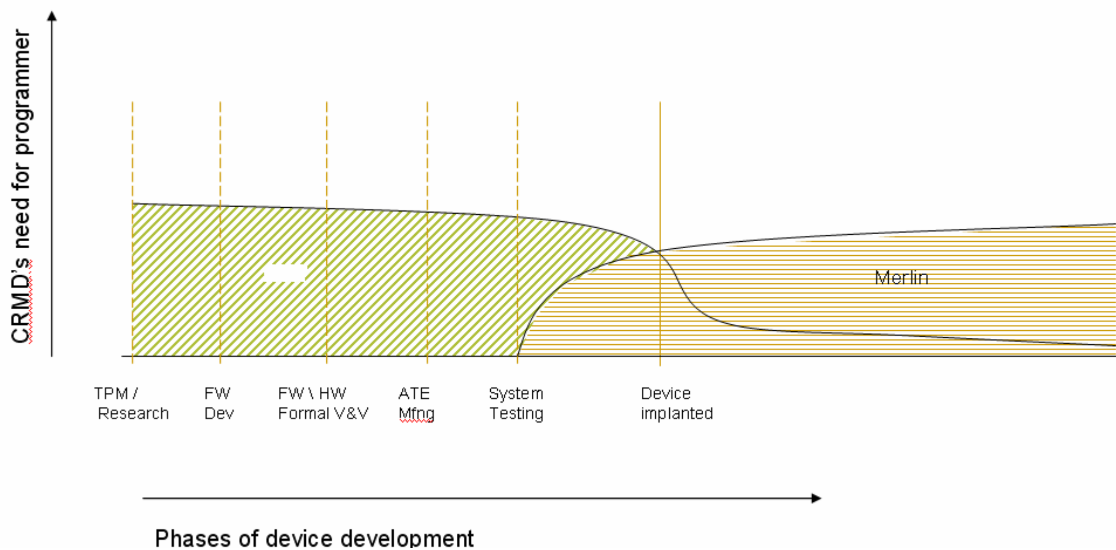



*Figure 6: Phases of device development – why another Programmer?*

As illustrated in Figure 6, there are many phases in the development of CRMD before the device can be implanted in humans. These phases are: Research, Hardware and Firmware development, Hardware and Firmware formal Verification and Validation (V&V), Automation Testing Equipment (ATE) Manufacturing, and system testing. It takes about two to three years to complete the development cycle for a device before

introducing it to the market. The development of Merlin programmer takes lots of effort and time, and indeed, it has not happened parallel as the development of the device. The Merlin programmer is a product that can be used when the design and implementation of the device's hardware and firmware are about completed. Thus, it is used in the last two phases which are system testing phase and device implant phase to ensure the functionalities of device as well as the Merlin programmer work as expected before introducing to the market.

Due to the fact that Merlin programmer is not available during the phases of research, hardware and firmware development, and device testing, CRMD engineers need to have a tool that can help in development, troubleshoot, and V&V. As a result, there is a need for a tool as an aid to develop and test the Implantable Medical Device (IMD)

**Existing Testing Tools and the issues**

Several tools existed in different departments that served very similar needs. One was the Dos Programmer which was written in the Dos operation system developed in 1998, and the other was EPWorkSpace (Figure 7), a Windows-based version developed in 2002. Both of these tools were used widely at that time by engineers from hardware, firmware, and validation and verification departments. These tools helped the engineers in such ways that they can do direct memory access read and write to the device. They can download the firmware to the device, retrieve all clinical parameters, and send telemetry commands to program clinical parameters. In general, these tools assisted the CRMD engineers during the device development cycle.

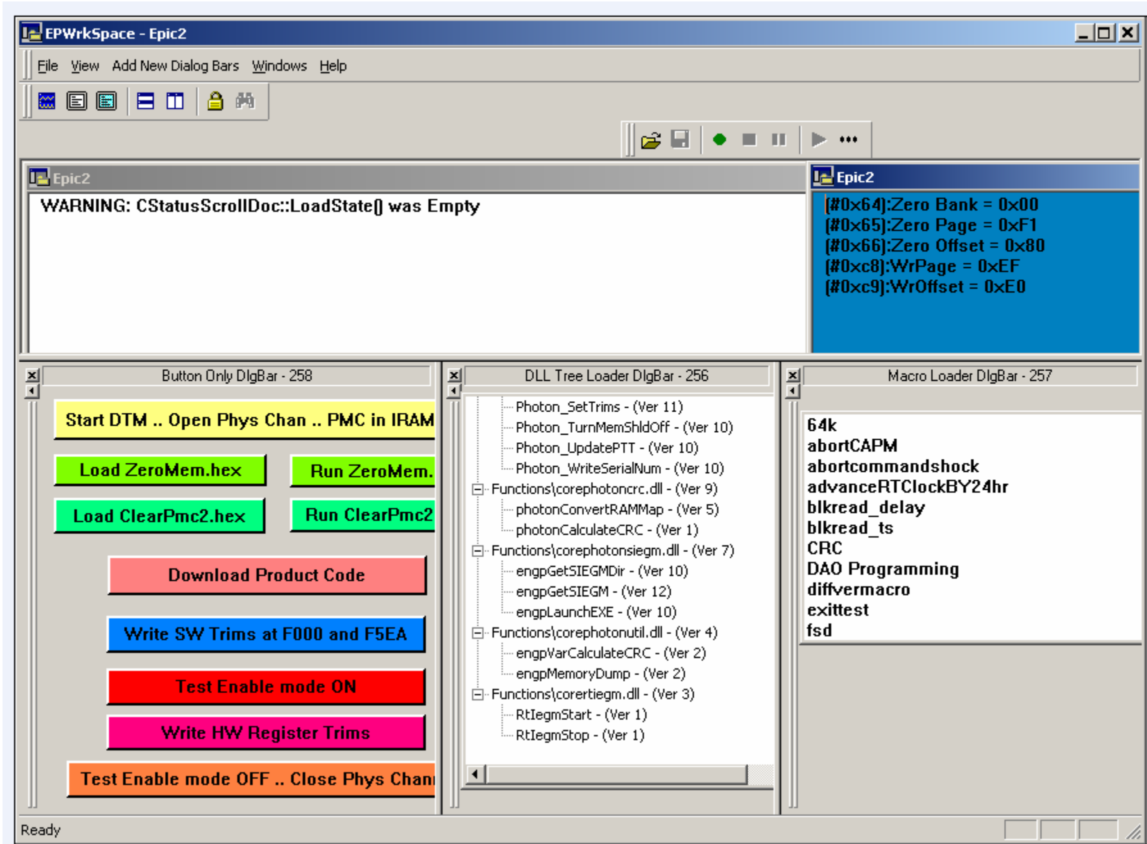



*Figure 7: EPWorkspace – An application to develop and test CRMD devices*

However, there were just so many issues that were faced by the engineers. A few of these problems are listed below:

- The ability to interrogate and program the device is missing.
- The use of device Direct Memory Access (DMA) is not sufficient to read and write data from and into the device.
- There are no macros to support repetitive tasks.
- There are no log files for debugging and troubleshooting.
- There is no capability to view the devices' variables in real time.
- There is no capability of viewing real time electrograms (RTEGM) and stored electrograms (SEGM).

Moreover, these tools were floating from one department to other departments, which ended up causing version mismatches. This resulted in engineers spending more time in troubleshooting the tools instead of focusing on troubleshooting the issues related to the device development.

In addition, the above tools lacked functionalities to support a complete testing strategy. It was very difficult to develop a full testing suite that met the company guidelines as well as the requirements. Also, the tools developed had to be customized for each department so they could not be shared easily. This issue seriously increased the cost of development and testing of the IMD. It also impacted the release time frame of particular devices to the market.

**Universal Engineering Programmer (UEP)**

UEP was created in early 2004 as a result of the various problems faced by departments not being able to share unified tools in the development and testing phases. It is a tool developed in C++ and Common Object Model (COM) that runs in Windows environment that combines all of the features that existed in previous tools. UEP utilizes a flexible architecture that makes it easy to develop features that meet the requirements of both development and testing.

# II OBJECTIVES

The UEP team introduced the first release of UEP in late 2004. Since then, UEP has successfully provided a tool to assist in the development and testing IMD across St. Jude Medical Cardiac Rhythm Management division. It meets the needs of firmware developers and testers for a means to perform all the telemetry, programming, and data access functions that will eventually be used by the Merlin Programmer but are not available during firmware development and testing phase. It also supports many test and diagnostic functions that are only used during firmware development and testing and will never be included in the Merlin Programmer. The functionality of the UEP is available through both an interactive Graphical User Interface (GUI) and through a software Application Program Interface (API), which is integrated into various automated firmware test tools. Besides, the UEP also incorporates telemetry software components that supports both inductive and Radio Frequency (RF) telemetry for all current devices. The objectives of UEP can be highlighted as below:

- UEP is a tool to aid in the development of Pacer, ICD & Unity IMDs.
- There are many testing systems exists in different phases of the device development that include the Verification and Validation Test Library system, the Automation Testing Equipment system, the custom Clinical Programmer, and the System for Making Automated and Random Test (SMART). These testing systems can easily integrate UEP as a component (Figure 8).

- UEP can be used across the CRMD departments such as Device Manufacturing, Hardware Development & Testing, Firmware Development & Testing, Clinical System Engineering, Research, etc.

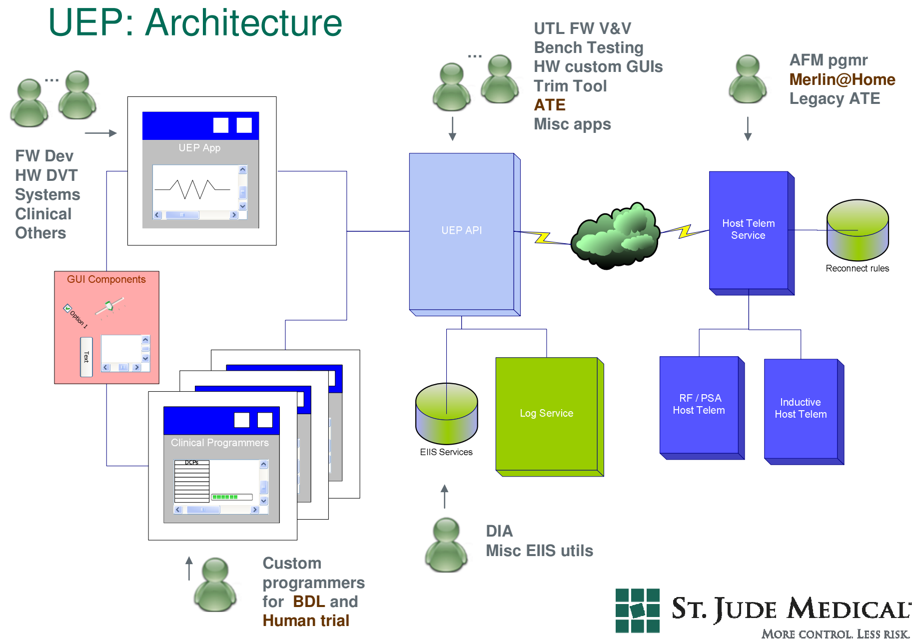



*Figure 8: UEP Architecture – can be integrated to various testing systems*

As illustrated in Figure 8, the UEPAPI is a set of APIs from UEP that can be integrated to the Custom programmers for Animal or Human Trial, firmware and hardware development system, testing library for firmware verification and validation, Device Image Analyzer (DIA) as well as the Merlin Programmer.

In order to meet the above objectives, UEP should be developed to have the capacity of performing the three main features below:

- **Direct Access Memory (DMA)**: Users should be able to use UEP to read and write data from or to memory locations. Users should be able to send the mailbox commands to the device for different purposes.
- **Interrogate and Programming**: Users should be able to interrogate all data from the devices. The data is stored in the device memory location in raw format and then decoded into the clinical data. Users should be able to set up specific parameters and program these parameters to the device.
- **Set up and display real time EGM and stored EGM**: Users can set up the configuration of RTEGM using UEP and display the real time EGM. Users can also store the RTEGM into different data format files for post processing.

# III METHODS AND MATERIALS

This chapter describes the architecture of UEP and the components that are deployed with UEP to meet the objectives as indicated above.

## Architecture

UEP consists of five major components. Figure 9 shows the architecture of UEP with components: GUI, UEPAPI, COM, EIISSERVICE and Telemetry.

- **GUI (Graphical User Interface)**: A friendly Graphical User Interface that perform all UEP user interface functionalities.
- **API (Application Program Interface)**: A set of APIs packed into a library that can easily be integrated into other applications to perform all UEP functionalities.
- **COM (Common Object Model)**: This is the main core of UEP. It is implemented in Common Object Model and C++ to perform all UEP functionalities. The API and GUI can access to all UEP functionalities via this COM component.
- **EIISService (External Instrument Interface Specification Service)**: This component is responsible for loading a specific device data from a set of xml files. It also performs the encoding and decoding device parameters from raw values to clinical values and vice versa based on the formula algorithm provided by the External Instrument Interface Specification (EIIS) eXtensible Markup Language (XML) files.

- **Telemetry**: This component is responsible for the low level interface that interacts with the DTM (Digital Telemetry Module) or RF (Radio Frequency) wand and then communicates with the device protocol to perform all DMA calls and telemetry commands.

UEP : Architecture

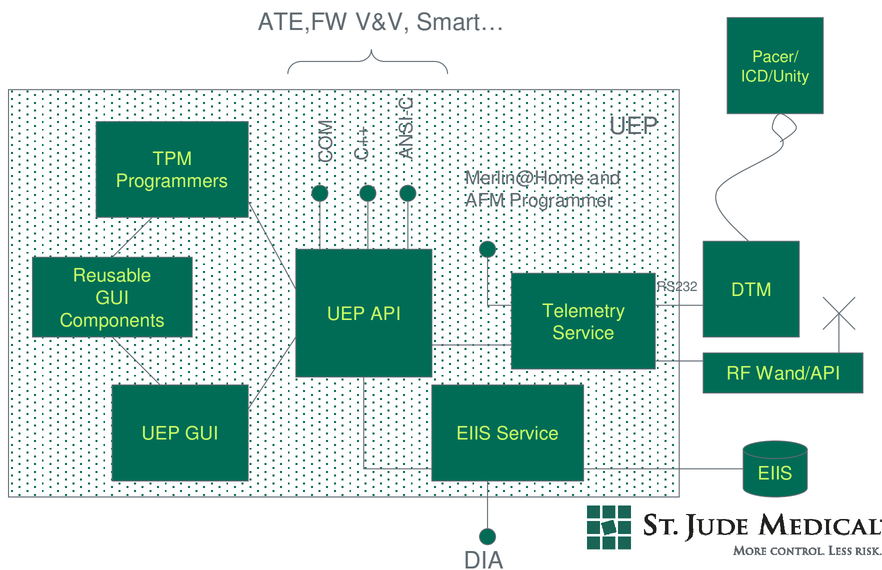



*Figure 9: UEP Architecture - components*

## UEP Graphical User Interface (GUI)

UEP GUI provides easy user configuration via a graphical user interface. The purpose of the UEP GUI is to enable software engineers to send commands to and receive and display data from any of the following kinds of cardiac devices:

- Pacemaker with 8K (baud) telemetry
- Unity device with 8K telemetry

- Unity device with 64K or radio frequency (RF) telemetry
- Implantable cardioverter defibrillator (ICD) with 8K and 64K telemetry

The main UEP GUI consists of the following principal sections as showed in Figure 10.

- The main menu
- Four toolbars
- The programmable parameter window (at the upper left directly beneath the toolbars section)
- The log window (at the upper right directly beneath the toolbars section)
- The real-time electrogram (RTEGM) window (at the lower left)
- The memory watch window (at the lower right)
- The status bar (at the bottom of the dialog)

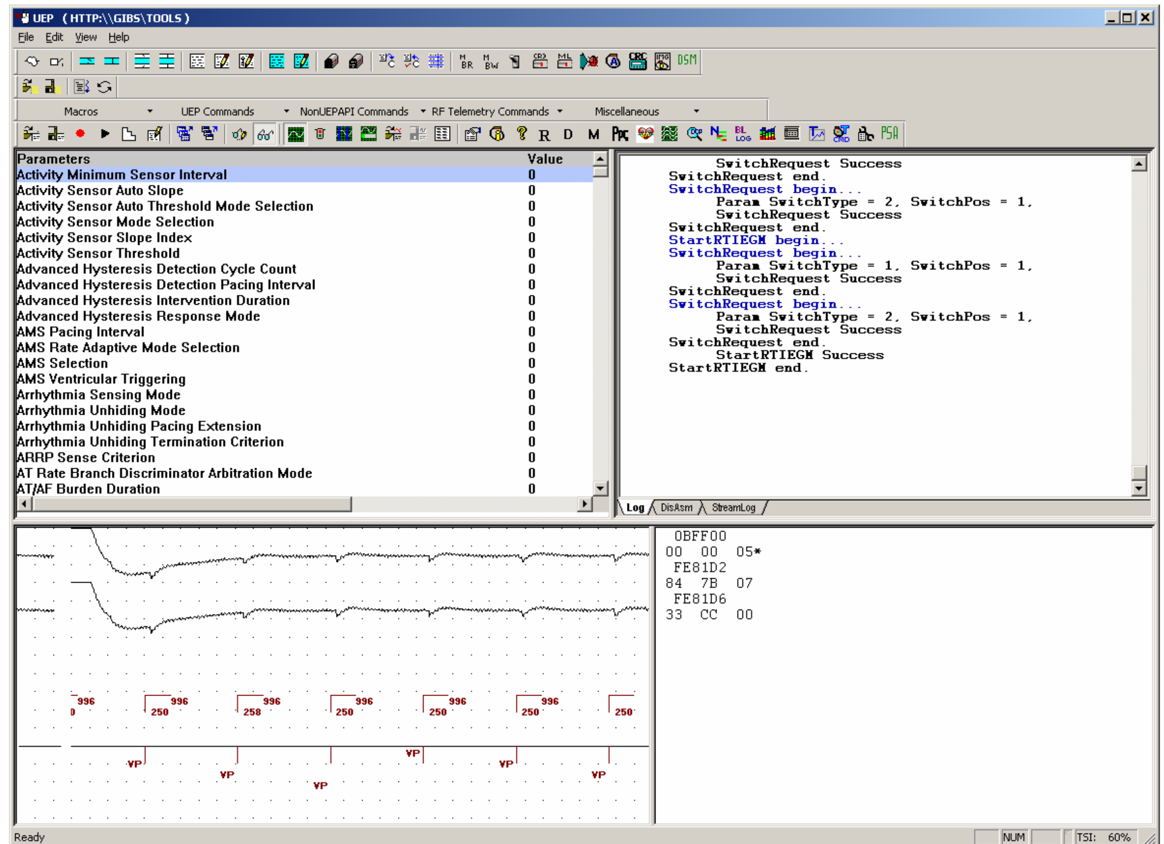



*Figure 10: UEP GUI*

## Toolbars

The four toolbars, located immediately below the main menu, contain buttons that enable the user to send commands to the cardiac device via the digital telemetry module (DTM) or RF wand and to exercise various other capabilities of the UEP, such as downloading device software, recording macros, and establishing memory watches. The user can interrogate the device, program it with the values on the screen or in a file, save a set of values in a file, and make a given set of parameters active or permanent.

Most of the UEP commands are supported by the GUI. The users can initiate and communicate with the device in terms of direct memory read/write, setting up real time EGM, interrogate, program, and send telemetry commands to the device. Table 1 below shows some typical commands that supported in GUI, API and COM

| UEP Commands Group | GUI | API | COM |
|---|---|---|---|
| SetTelemetry | Yes | Yes | Yes |
| InitDTM | Yes | Yes | Yes |
| Open Physical Channel | Yes | Yes | Yes |
| Close Physical Channel | Yes | Yes | Yes |
| Open Logical Channel | Yes | Yes | Yes |
| Close Logical Channel | Yes | Yes | Yes |
| Block Read/Write | Yes | Yes | Yes |
| Access Write | Yes | Yes | Yes |
| Bulk DMA Read | Yes | Yes | Yes |
| Bulk DMA Write | Yes | Yes | Yes |
| MailBox | Yes | Yes | Yes |
| MailBox Custom | Yes | No | No |
| Download Code | Yes | Yes | Yes |
| Verify Code | Yes | Yes | Yes |
| SetSymbolTable | Yes | Yes | Yes |
| Multiple Byte Read/Write | Yes | Yes | Yes |
| Switch Request | Yes | Yes | Yes |
| XIMailBox | Yes | Yes | Yes |
| Multi-Link MailBox Commands | Yes | Yes | Yes |
| CRC | Yes | Yes | No |
| Load, Save and Program Param | Yes | Yes | Yes |
| Stored EGM Display | Yes | Yes | No |
| Device Info | Yes | Yes | Yes |
| Macro Load,Save, Record, Play, Edit | Yes | No | No |
| Interrogate | Yes | Yes | Yes |
| Measure Data | Yes | No | No |
| Runaway Protection (RAP) Trim | Yes | No | No |
| Radio Frequency (RF) Telemetry Commands | Yes | Yes | Yes |
| Raw DTM command | Yes | No | No |
| ReadBankPages | Yes | No | No |
| Memory Watch | Yes | No | No |
| RTEGM Set up and Display | Yes | Yes | Yes |
| Stored EGM Set up | Yes | Yes | Yes |

*Table 1: UEP commands supported by GUI, API and COM*

Illustrated below are some scenarios for using the toolbar in the UEP GUI. For example, after initializing the device, the user can start the communication with the device by sending command open or close the physical channel of the device's hardware. To communicate with the device's firmware, the user needs to open or close the logical channel of the device. Figure 11 below shows the GUI for these commands.

Click the **ClosePhysicalChannel** icon on the first toolbar:

Click **OpenLogicalChannel** icon on the first toolbar.

*For Unity ICD devices, the UEP displays the CPOpenLogicalChannel dialog:*

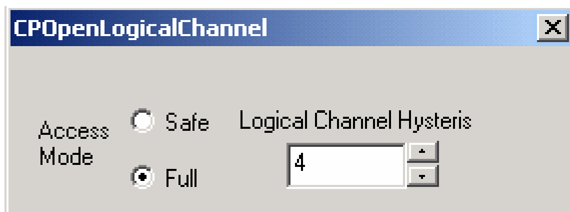
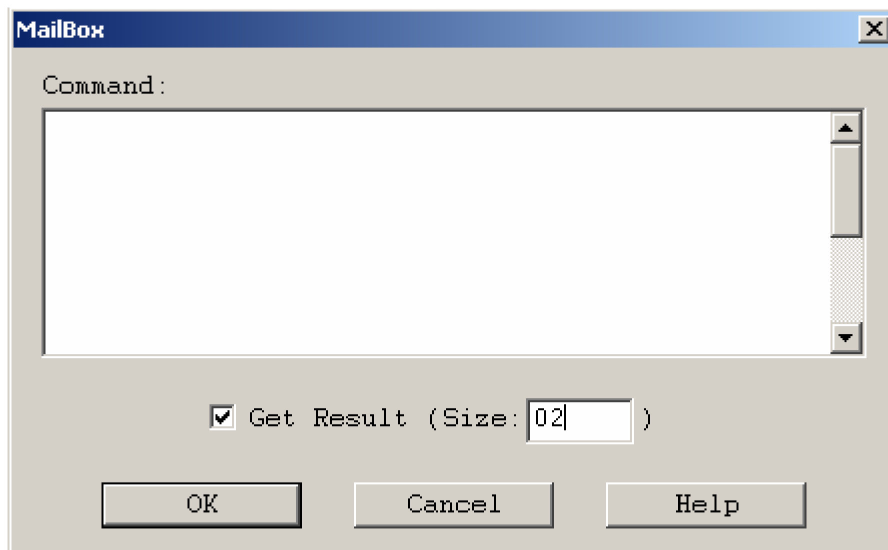



*Figure 11: Close Physical Channel and Open Logical Channel GUI command*

The user can send any mailbox command specified as a sequence of bytes to the device. Figure 12 shows the sequences to send a mailbox command to the device as well as the response data from the device.

Click the **MailBox** icon on the first toolbar.

*UEP then displays the MailBox dialog:*




*User then specifies a sequence of hex bytes that compose the mailbox command.*

*User can check the Get Result checkbox if user wants results shown in the log window. The size in bytes of the result must also be specified.*

*Specify the size in bytes of the result.*

*The maximum size of a mailbox command is 250 or 255 bytes for ICD 8K and ICD 64K device.*

*As an example, user send the Device Info command <00 C0 04 01 C5>.  The command response will be returned with the size of 0x2D.*

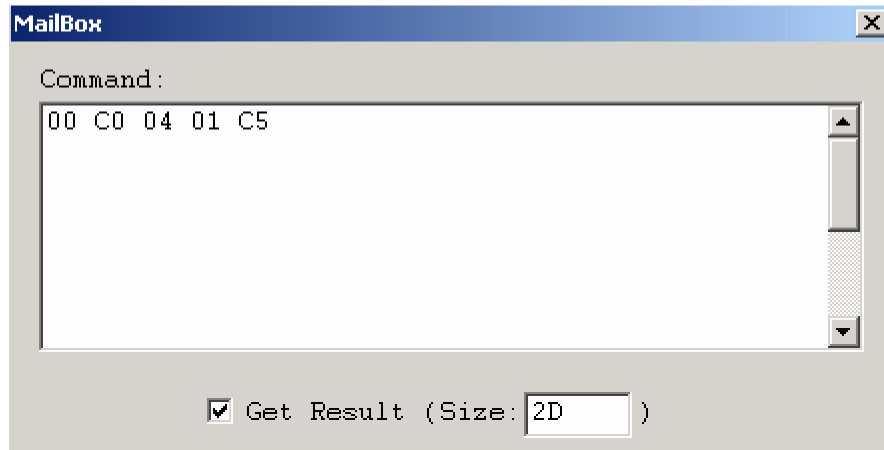


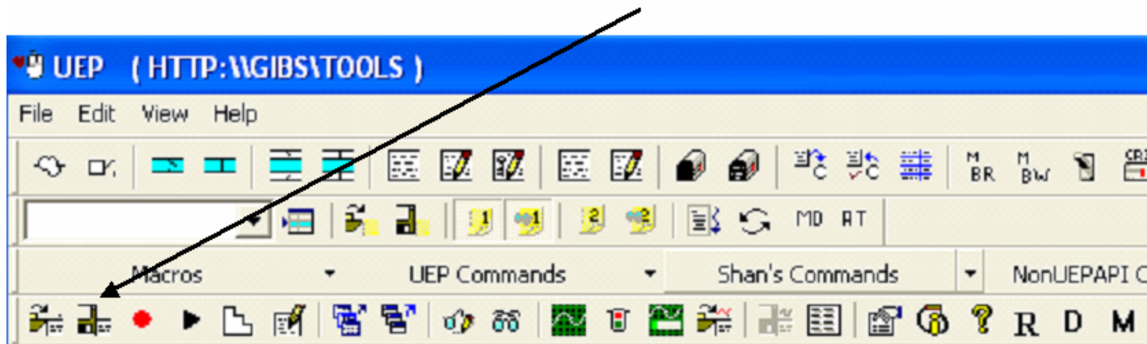

*Figure 12: Sequences to send a mailbox command to the device and response data from the device.*

The log shows the reply from the device in response to the DeviceInfo command

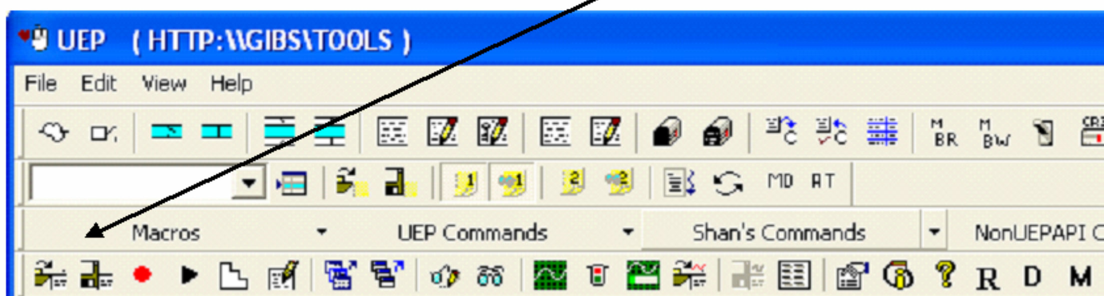
00   01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00  **00 00 C0 24 01 02 30 67 48 24 00 00 00 03 00 01**
10  **01 01 01 10 00 7F 01 00 7E FD 0B 01 B2 00 00 76**
20  **DC 09 CF 39 12 2F 0A 33 0A 0A 33 0A 0A**

Macro is a very good feature of UEP.  A macro is a sequence of UEP or non UEPAPI commands and associated parameter values.  The user can record, edit, execute (play), save, and read a macro from a file.  Macro can be used as a sequence of commands repeatedly executed for a specific purpose.  Figure 13 illustrates the use of macros in the UEP GUI

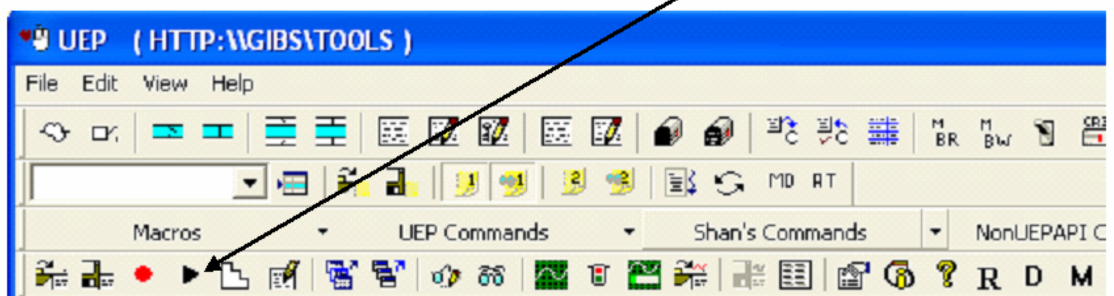
**Macro Load** icon on the fourth toolbar.




**Macro Save** icon on the fourth toolbar.




**Macro Record** icon on the fourth toolbar.

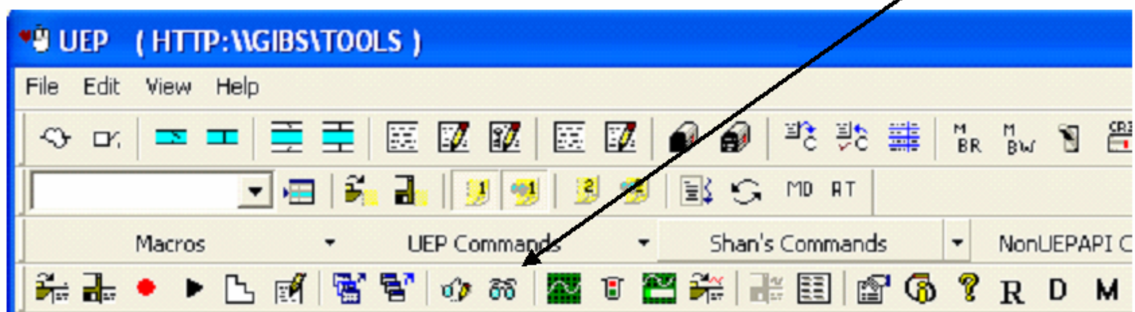



**Macro Play** icon on the fourth toolbar.

*Figure 13: Macros in UEP*

Memory watch is one of UEP's important features. The user can specify a list of location of blocks of memory whose contents is report periodically by UEP. Each memory block is specified in terms of a starting address and number of bytes. This feature allows users to track the data in specific blocks of memory in the device. Figure 14 illustrates the way a user can set the memory watch in the GUI.

**Set Memory Watch** button on the fourth toolbar.




*UEP GUI displays the Memory Watch Editor:*

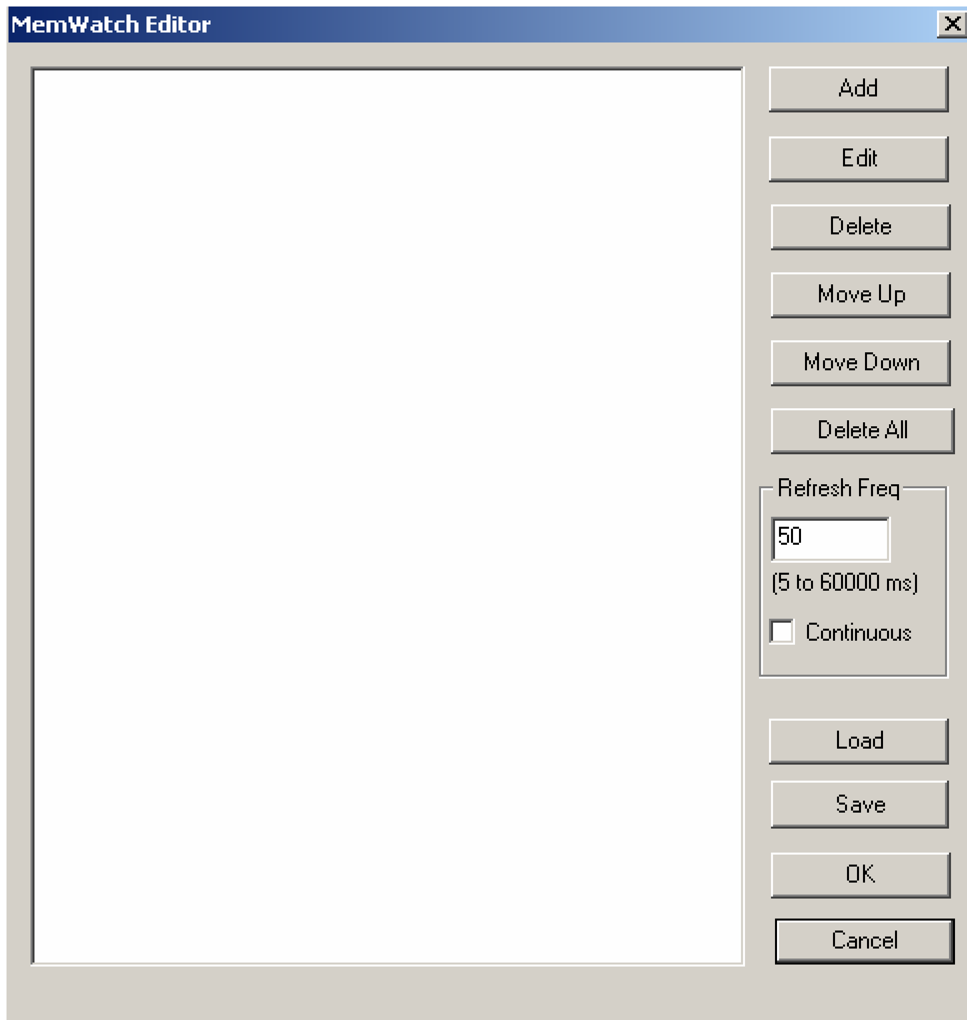



*Figure 14: Memory Watch feature in UEP GUI*

**The three main UEP features**

Previous sections are the illustrations of some of many features in GUI. This section shows the three main features in GUI that strongly support the UEP' objectives as described in chapter 3. In general, the functionality of the UEP is available through an interactive GUI, a software API, and COM component. Thus, users of different system platforms or programming scripts can always use these three components to develop testing systems or to integrate to their own testing system.

### Direct Memory Access

The user can use the UEP GUI to perform reading of the values of specific memory locations by block read command or writing a value to a specific location by block write. Users can send device commands via the general mailbox command or the Unity mailbox command.  Figure 15 illustrates the Block Read feature in GUI for direct memory access; the user can specify an address from memory location in the device together with the size of the return date.  UEP returns the values of this address to users.

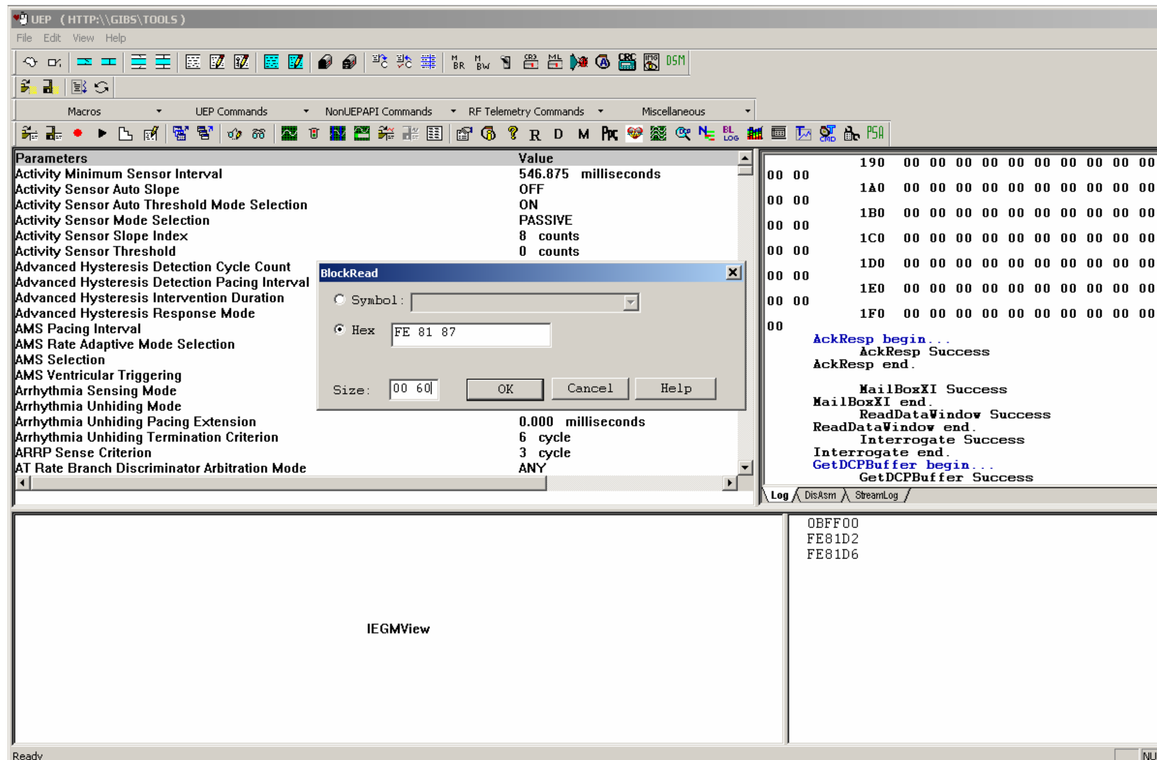



*Figure 15: Direct Memory Access – Block Read feature in GUI*

In a similar manner, Figure 16 illustrates the Block Write feature in GUI for direct memory access; the user can specify an address from memory location in the device together with the data they want to write to this address.

Figures 17 shows how a general mailbox command can be used in GUI. Mailbox commands are a set of commands to request the device for a specific purpose. An example of this is the command that requests the device to return the real time EGM of the patient. The response of that mailbox command will be returned to users for the status of the command whether it is passed or failed and the data associated with that response.

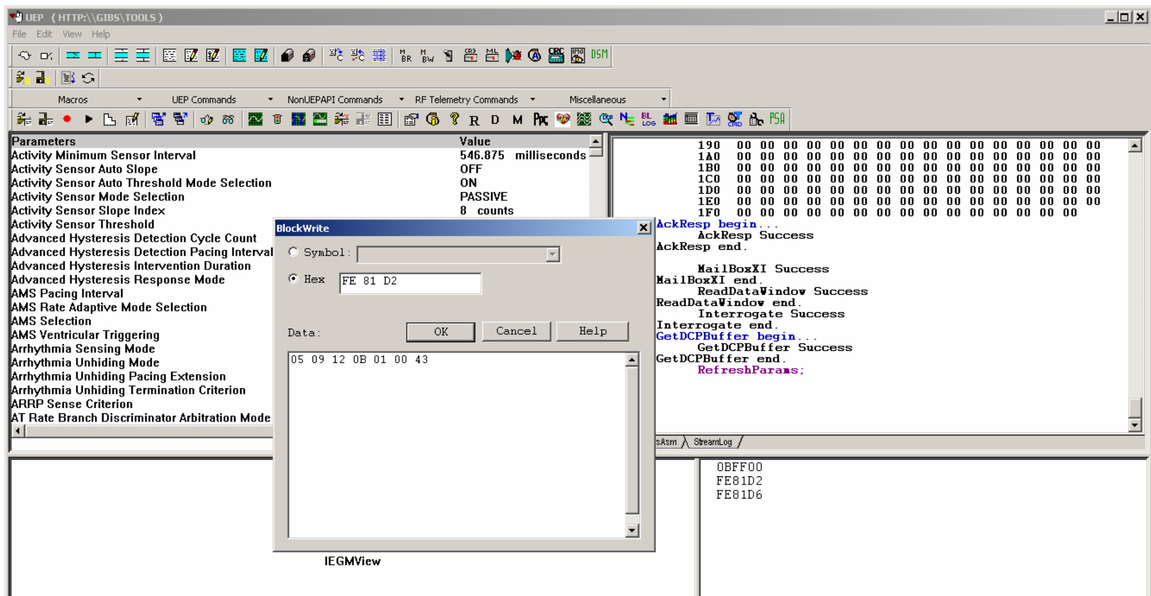



*Figure 16: Direct Memory Access – Block Write in GUI*

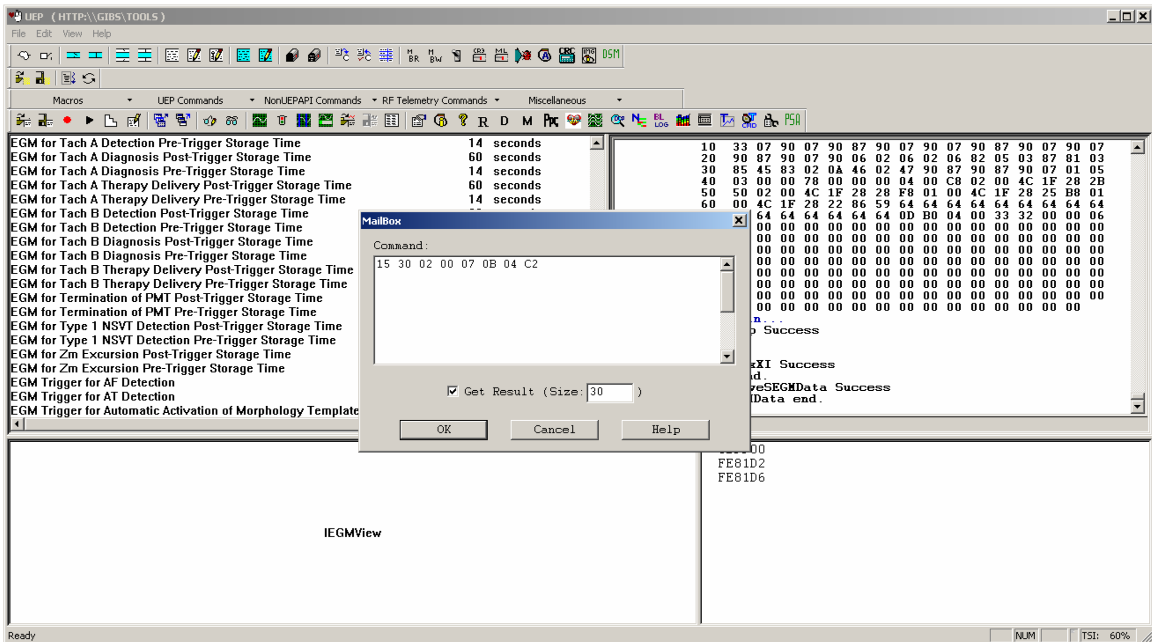


*Figure 17: Direct Memory Access - General mailbox command feature in GUI*

Figures 18 shows how a XI (External Interface) mailbox command can be used in GUI.

XI Mailbox command is similar to regular mailbox command but used for different firmware protocol. XI Mailbox commands are also a set of commands to request the device for a specific purpose. An example of this is the command that requests the device to return the device configuration. The response of XI mailbox command will be returned to users for the status of the command whether it is passed or failed and the data associated with that response.

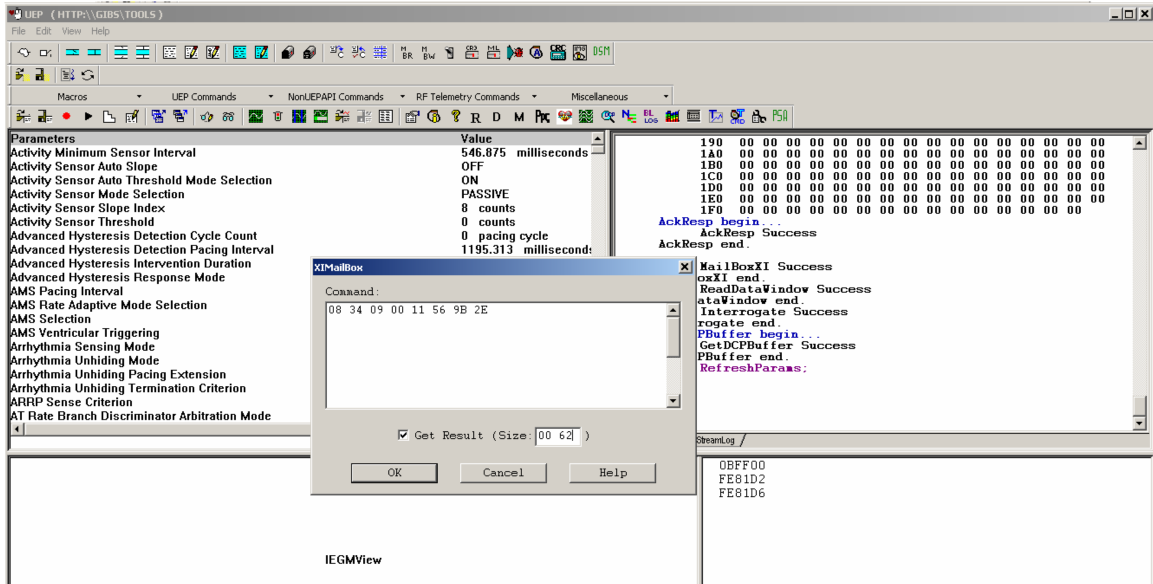



*Figure 18: Direct Memory Access - Unity XIMailbox command feature in GUI*

Figure 19 illustrates how the users use UEP GUI to interrogate the device. Interrogate is the technical term that is used for retrieval of all device parameters. The device after being downloaded the firmware, contains a set of nominal parameters. Interrogate command in GUI retrieves all device parameters from the device and display it for users to view. The users can change these parameters as illustrated in Figure 20.

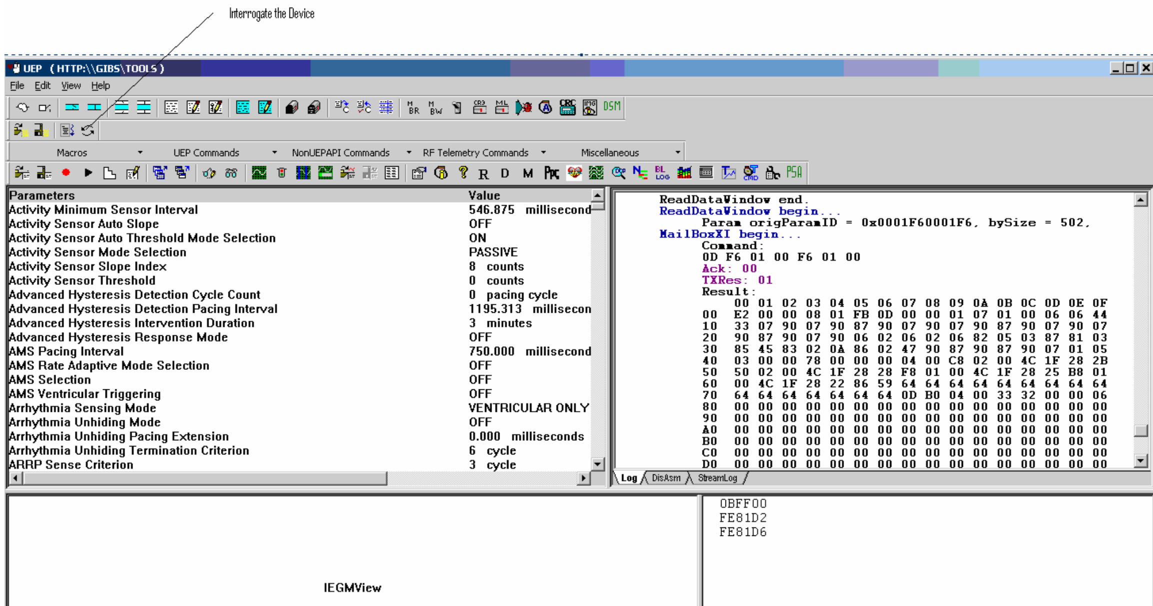



*Figure 19: Interrogate command feature in GUI*

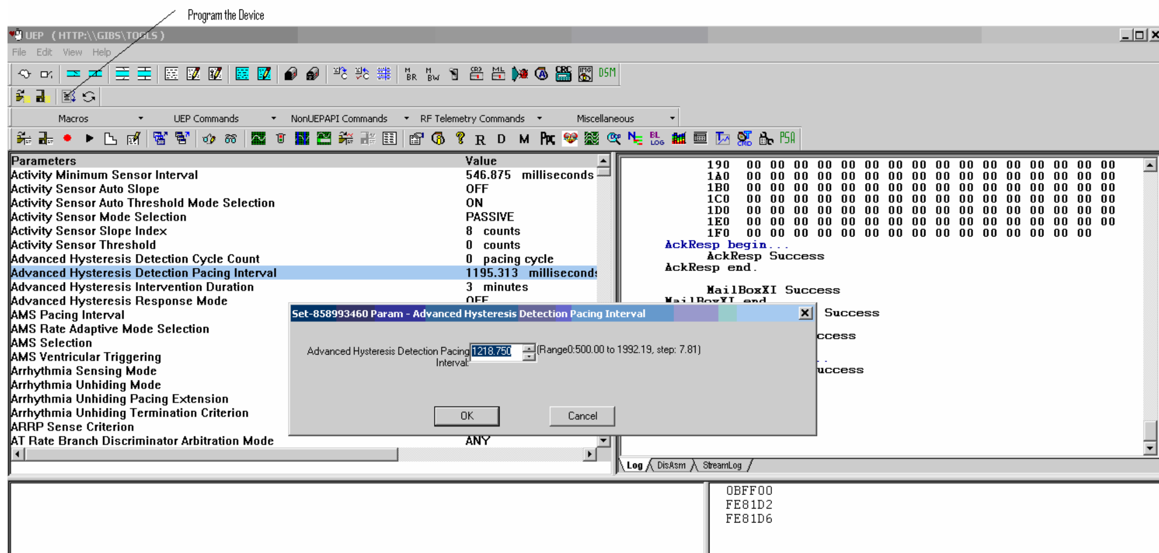



*Figure 20: Set and Get Parameters values and Program the device in GUI*

Figure 21 shows the UEP Device Clinical Parameters (DCP) and Device Variable (DV) viewers. DCP has the values in clinical term whereas the DV has the value in raw. The DV value resides in the device memory. The DCP and DV are related by the encode and decoding algorithm.

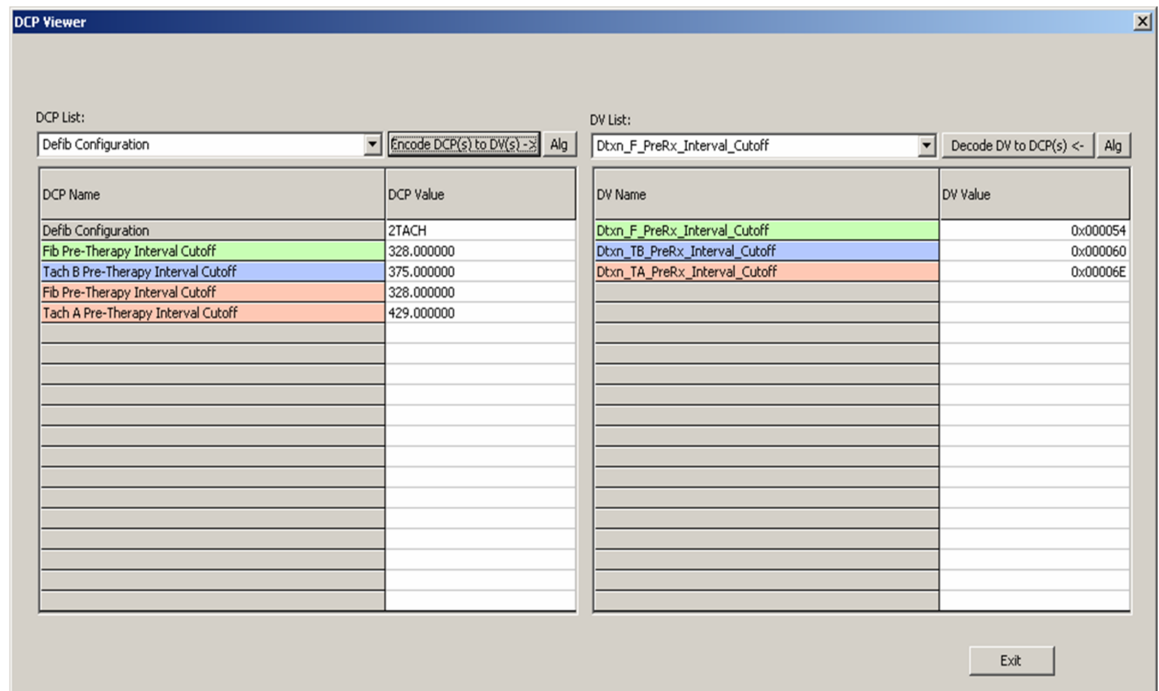



*Figure 21: Device Clinical Parameter Viewer allows user to view and change the parameters.*

### Real Time Electrograms (RTEGM)

On of the main feature that UEP support is Real Time Electrograms (RTEGM). Figure 22 illustrates the way to set up RTEGM in terms of number of channels, EGM sources selection and the choice of saving Stored EGM for post processing. RTEGM provides the user with a way to control the display of real-time electrograms (RTEGMs) and markers. Figure 23 shows how to start and stop RTEGM in GUI.

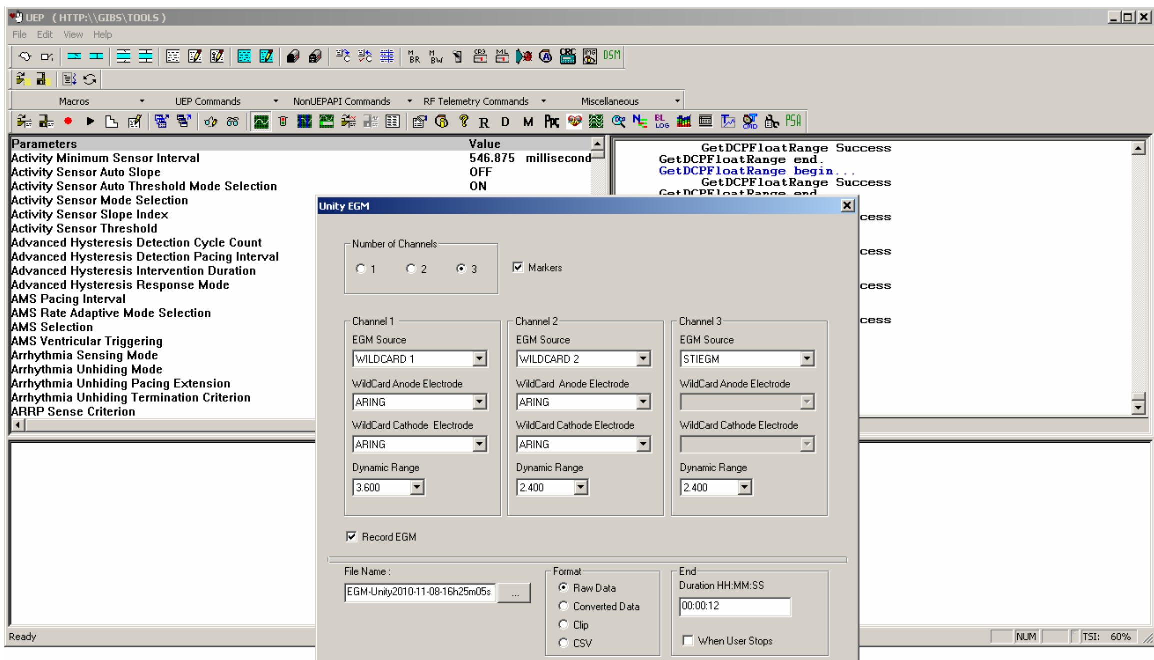



*Figure 22: Set up Real Time Electrograms GUI*

**RTEGM Start/Stop**

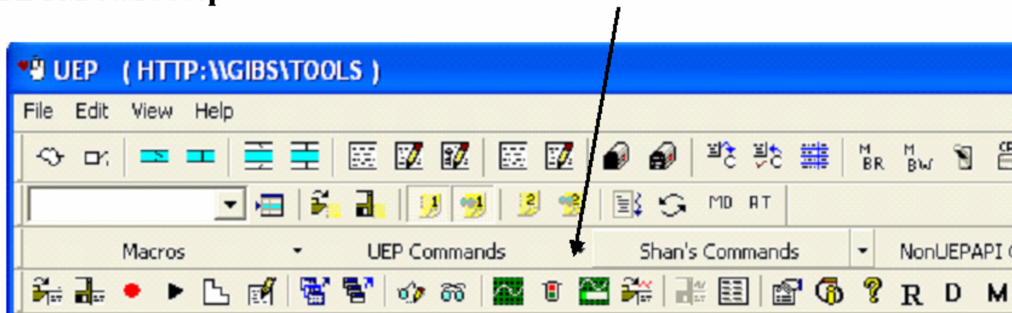



*Figure 23: Start and Stop Real Time Electrograms GUI*

User can set up the display of RTEGM by enabling the returning markers and setting up the number of channels, its associated channel sources, and dynamic ranges. User can also indicate whether RTIEGMs are to be recorded in a file. Figure 24 shows the RTEGM flowing from the devices to UEP which is displayed on the screen as wave forms with details marker and intervals. Figure 25 illustrates the Stored EGM after saved into the device's memory; it can be displayed in GUI for post procession.

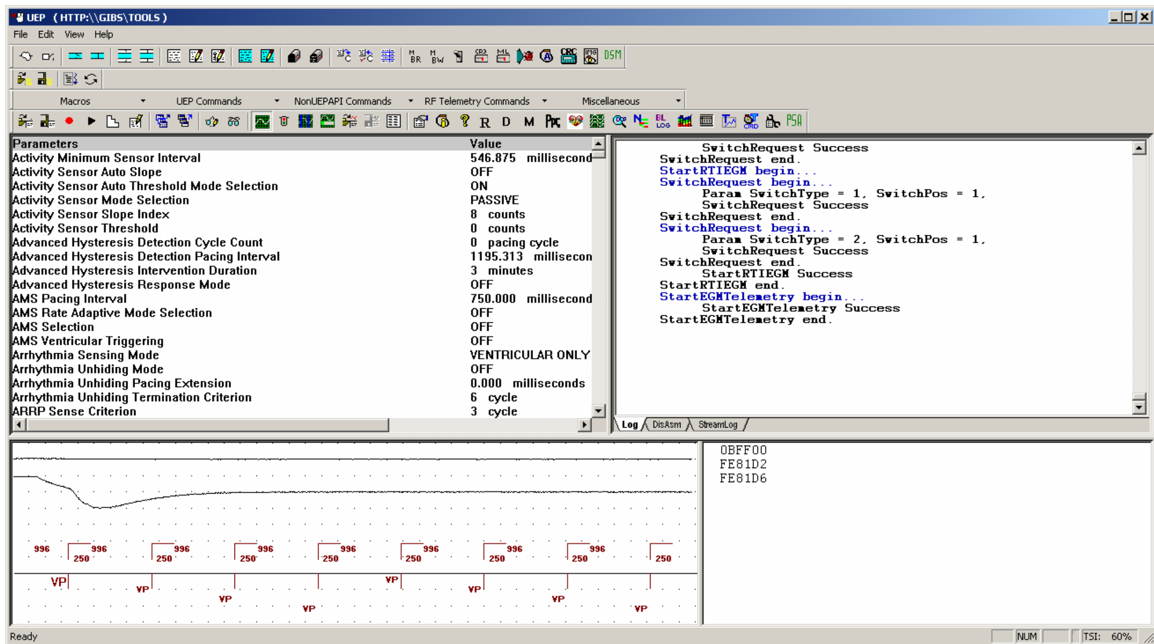



*Figure 24: Real Time EGM display in UEP GUI*

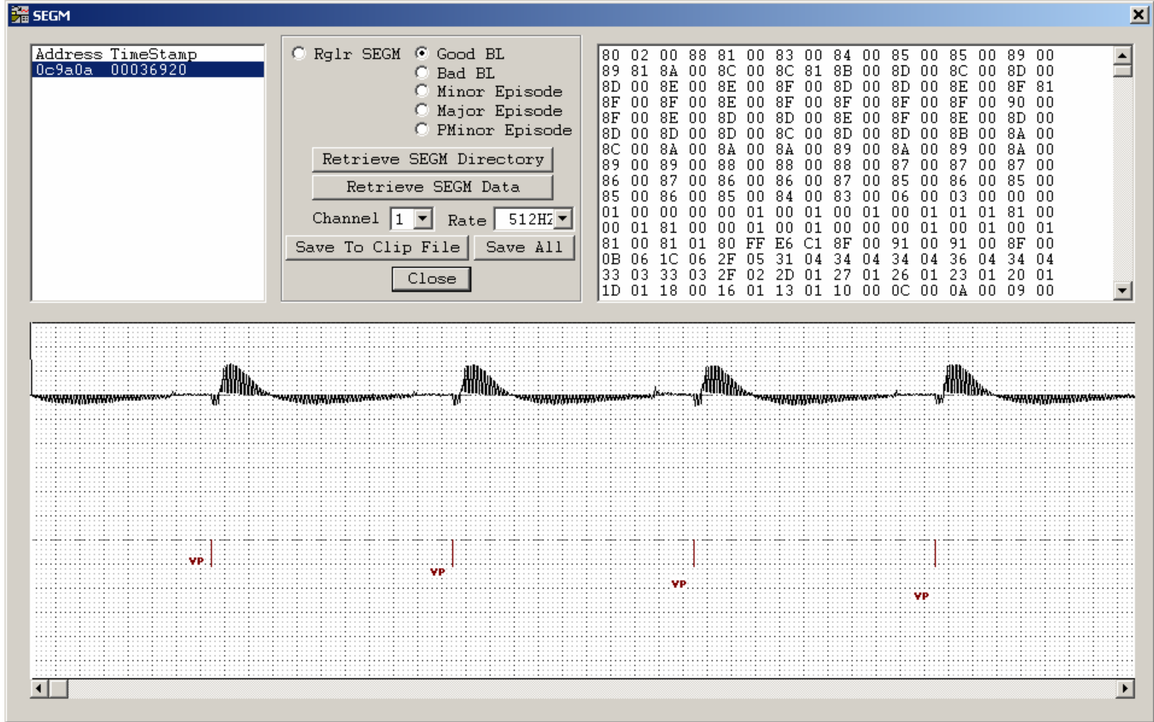



*Figure 25: Stored EGM displayed in UEP*

## UEP Application Program Interface (API)

UEPAPI is a set of APIs deployed together with the UEP GUI. The purpose of the API is to enable engineers to write applications that interface to the UEP. All UEP functionalities provided by the GUI are also provided by the API. The header file UEPAPI.h contains all export APIs, so the user just needs to include this header file and statically or dynamically load the library UEPAPI.dll to their applications.

The following sections illustrate how user can use the APIs to perform the 3 main features of UEP: Device Memory Access, Interrogate and Program Device, and Real Time EGM.

### Device Memory Access

BlockRead is an API that allows user to query the raw data from the device memory. There are 3 different flavors of BlockRead so that the passing arguments into the API can be a string (char * as first flavor), integer (int addr as the second flavor), or the specific bank, page, and offset (unsigned char byBank, unsigned char byPage, unsigned char byOffset as the third flavor). Figure 26 shows the BlockRead APIs to provide DMA support.

```
int BlockRead(char* addr, unsigned char bySize, unsigned char*
pbyResult);
int BlockRead(int addr, unsigned char bySize, unsigned char*
pbyResult);
int BlockRead(unsigned char byBank, unsigned char byPage, unsigned char
byOffset, unsigned char bySize, unsigned char* pbyResult);
```

*Figure 26: BlockRead - Device Access Memory feature supported by UEPAPI*

Since the BlockRead API only supports up to 256 bytes, Figure 27 below shows the BlockReadLong APIs can be used to read a memory address up to 4095 bytes and these also come with the same three flavors as above:

```
int BlockReadLong(char* addr, unsigned short bySize, unsigned char*
pbyResult);
int BlockReadLong(int addr, unsigned short bySize, unsigned char*
pbyResult);
int BlockReadLong(unsigned char byBank, unsigned char byPage, unsigned
char byOffset, unsigned short bySize, unsigned char* pbyResult);
```

*Figure 27: BlockReadLong - Device Access Memory feature supported by UEPAPI*

BlockWrite is an API that allows users to write raw data into the memory location. User however, can only write 256 bytes at a time to the device memory. Figure 28 below shows the flavors of BlockWrite APIs:

```
int BlockWrite(char* addr, unsigned char bySize, unsigned char* pData);
int BlockWrite(int addr, unsigned char bySize, unsigned char* pData);
int BlockWrite(unsigned char byBank, unsigned char byPage, unsigned
char byOffset, unsigned char bySize, unsigned char* pData);
```

*Figure 28: BlockWrite - Device Access Memory feature supported by UEPAPI*

Mailbox and MailboxXI APIs are the APIs that allow user to send the command to the device for a specific purpose. Figure 29 below shows the Mailbox and MailboxXI APIs

```
int MailBox(unsigned char byCmdDataSize, unsigned char* parrbyCmdData,
unsigned char byReturnData,unsigned char byReturnDataSize, unsigned
char* parrbyReturnDataBuf);

int MailBoxXI(unsigned char byCmdDataSize, unsigned char*
parrbyCmdData, unsigned char byGetResult, unsigned short
byResultSize,unsigned char* byAck, unsigned char* byTXStatus, unsigned
char* parrbyReturnDataBuf);
```

*Figure 29: Mailbox and MailboxXI - Device Access Memory feature supported by UEPAPI*

**Interrogate and Program Device**

The user can query all raw data in the device and then decode the raw data to the clinical values. Figure 30 shows the Interrogate and ProgramDevice API.

```
int Interrogate(EPParamSetType eParamSet = ACTIVE, EPGroupType grpType
= epClinical,const char* subGroupName = "");
int ProgramDevice(EPParamSetType eParamSet, int nModel, int nSerialNum,
int nSoftVer, int nDiagNum, unsigned char* nDiagId);
```

*Figure 30:Interrogate and ProgramDevice - Interrogate and ProgramDevice feature supported by UEPAPI*

Users can retrieve the value of a specific parameter. Figure 31 shows the GetDCPValue in different flavors based on the type of the value.

```
int GetDCPValueAsFloat(const char* pszDCPName,float* fFloatVal, char*
pszUnits);
int GetDCPValueAsInteger(const char* pszDCPName,int* nIntVal, char*
pszUnits);
int GetDCPValueAsLong(const char* DCPName, __int64* int64Val, char*
pszUnits);
int GetDCPValueAsString(const char* pszDCPName,char* pszEnumVal);
int GetDCPValueAsWString(const char* pszDCPName, wchar_t* pszEnumVal);
```

*Figure 31:SetDCPValue - Interrogate and ProgramDevice feature supported by UEPAPI*

Users can also set the value of a specific parameter before calling the ProgramDevice to permanently set the value into the device memory. Figure 32 shows the SetDCPValue in different flavors based on the type of the value.

```
int SetDCPValueAsWString(const char* pszDCPName, const wchar_t*
pszEnumVal);
int SetDCPValueAsFloat(const char* pszDCPName,float fFloatVal);
int SetDCPValueAsInteger(const char* pszDCPName,int nIntVal);
int SetDCPValueAsLong(const char* DCPName, __int64 int64Val);
int SetDCPValueAsString(const char* pszDCPName,char* pszEnumVal);
```

*Figure 32 GetDCPValue - Interrogate and ProgramDevice feature supported by UEPAPI*

**Real Time EGM (RTEGM)**

An implantable cardiac medical device has the capability of sensing cardiac events on a sensing lead positioned in relation with the patient's heart, and storing the electro gram (EGM) of the heart in device memory. UEP API allows the user to configure and start the sensing of RTEGM. It also allows the users to save RTEGM to the device storage and retrieve these stored EGM for post processing. Figure 33 shows the APIs which support RTEGM and Stored EGM feature.

```
int ConfigureUnityRTEGM(UnityEGMConfig ConfigRec);
where UnityEGMConfig is defined as:
struct UnityEGMConfig
{
   int numChannel;
   bool bMarker;
   struct
   {
      EPRTEGMSource chanSource;
      EPWildCardElectrode WildCardAnode;
      EPWildCardElectrode WildCardCathode;
      float DynamicRange;
   } ChanConfig[3];
};

Once done, user can call the API to start/stop the RTEGM
int StartRTIEGM(int nEGMOn, int nMarkerOn, int nSecBufSize);
int StopRTIEGM();

User can retrieve the stored EGM inside the device using the following
APIs

int RetrieveSEGMData(const char* szAddress, unsigned char* pbySEGMData,
int* nEGMSize);
int RetrieveSEGMDirectory(char** pszAddress, char** pszTimeStamp, int*
nSEGMs);
int RetrieveUnitySEGMDirectory(EPSEGMType segmType, char** pszAddress,
char** pszTimeStamp, int* nSEGMs);
```

*Figure 33 – RTEGM and Stored EGM feature supported by UEPAPI*

## UEP Common Object Model (COM)

COM is a binary-interface standard for software component. COM allows reuse of objects with no knowledge of their internal implementation. UEP COM is deployed together with the GUI and API. This is the main core of UEP and is implemented in C++ and IDL (Interface Definition Language). The UEP COM communicates directly to the lower level which in turn communicates with the device. The users of late binding programming scripts such as Tcl, Perl and VBScript only need to add the reference of UEP COM library to their scripts, and then invoke COM APIs exposed by the COM object to communicate with the device in the same way as using the GUI and API.

The following sections illustrate UEP COM interfaces that can be used to perform the 3 main features of UEP as described in the GUI and API section.

### Device Memory Access

In a similar manner as the GUI and API, UEP COM interfaces allow user to DMA read, write, or sending Mailbox, MailboxXI to the device, by using BlockRead, BlockWrite, Mailbox, and MailboxXI respectively. Figure 34 illustrates the UEP COM interfaces defined in the Interface Definition Language (IDL) that support DMA feature.

```
HRESULT BlockReadLong([in] BSTR strSymbol, [in] unsigned char byBank,
[in] unsigned char byPage, [in] unsigned char byOffset, [in] unsigned
short bySize, [out] VARIANT* parrbyData, [out, retval] int* pnErrRet);

HRESULT BlockWrite([in] BSTR strSymbol, [in] unsigned char byBank, [in]
unsigned char byPage, [in] unsigned char byOffset, [in] VARIANT*
parrbyData, [out, retval] int* pnErrRet);
```

```
HRESULT MailBox([in] VARIANT *parrbyCmdData, [in] unsigned char
byGetResult, [in] unsigned char byResultSize, [out] VARIANT*
parrbyResultData, [out, retval] int* pnErrRet);


HRESULT MailBoxXI([in] VARIANT *parrbyCmdData, [in] unsigned char
byGetResult, [in] unsigned short byResultSize, [out] unsigned char*
byAck, [out] unsigned char* byTXRes, [out] VARIANT* parrbyResultData,
[out, retval] int* pnErrRet);
```

*Figure 34 Device Access Memory feature support by UEP COM interfaces*

**Interrogate and Program device**

COM Interfaces supports the Interrogate and Program the Device. Figure 35 shows the interfaces: Interrogate, ProgramDevice, SetDCPValue and GetDCPValue.

```
HRESULT Interrogate([in] EPParamSetType paramSet, [out]unsigned char*
Ack,[out]unsigned char* TxResp, [out] int* pnErrRet, [in,
defaultvalue(epClinical)]EPGroupType grpType,[in,
defaultvalue(NULL)]BSTR subGroup);

HRESULT ProgramDevice([in] EPParamSetType paramSet, [in] int model,
[in] int serialNum, [in] int softVer, [in] int diagNum, [in] unsigned
char* diagId, [out]unsigned char* Ack,[out]unsigned char*
TxResp,[out,retval] int* pnErrRet);

HRESULT SetDCPValueAsInteger([in]BSTR DCPName,[in]int intVal,[out]int*
pnErrRet);
HRESULT SetDCPValueAsString([in]BSTR DCPName,[in]BSTR EnumVal,[out]int*
pnErrRet);

HRESULT SetDCPValueAsFloat([in] BSTR DCPName, [in] float floatval,
[out] int* pnErrRet);

HRESULT SetDCPValueAsInt64([in]BSTR DCPName,[in]__int64
longVal,[out]int* pnErrRet);
      [id(108), helpstring("method GetDCPInt64Range")]

HRESULT GetDCPValueAsInt64([in]BSTR DCPName,[out]__int64*
longVal,[out]BSTR* strUnits,[out]int* pnErrRet);

HRESULT GetDCPValueAsInteger([in]BSTR DCPName,[out]int*
intVal,[out]BSTR* strUnits,[out,]int* pnErrRet);

HRESULT GetDCPValueAsFloat([in]BSTR DCPName,[out]float*
floatVal,[out]BSTR* strUnits,[out,]int* pnErrRet);

HRESULT GetDCPValueAsString([in]BSTR DCPName,[out]BSTR*
StringVal,[out]int* pnErrRet);
```

*Figure 35 Interrogate and Program Device feature support by UEP COM interfaces*

**Real Time EGM**

The users of UEP COM can invoke Real Time EGM and retrieve Stored EGM. Figure 36 shows the interfaces: StartRTEGM, StopRTEGM, RetrieveSEGM, RetrieveUnitySEGMDirectory and RetrieveSEGMData.

```
HRESULT StopRTIEGM([out, retval] int* pnErrRet);
      [id(97), helpstring("method StartRTIEGM")]

HRESULT StartRTIEGM([in] int nEGMOn, [in] int nMarkerOn, [in] int
nSecBufSize, [out, retval] int* pnErrRet);

HRESULT RetrieveUnitySEGMDirectory([in] EPSEGMType segmType, [out]
VARIANT *parrAddress, [out] VARIANT* pTimeStamp,[out, retval] int
*pnErrRet);

HRESULT RetrieveSEGM([out] VARIANT* varArray, [out, retval] int*
pnErrRet);

HRESULT RetrieveSEGMDirectory([out] VARIANT *parrAddress, [out]
VARIANT* pTimeStamp,[out, retval] int *pnErrRet);
      [id(88), helpstring("method RetrieveSEGMData")]

HRESULT RetrieveSEGMData([in]VARIANT* pvarSEGMStartAddress, [out]int*
nEGMChunks, [out]VARIANT* pvarSEGMSize, [out]VARIANT*
pvarSEGMLinkAddress, [out]VARIANT* pvarSEGMData, [out, retval]int*
pnErrRet);
```

*Figure 36 Real TimeEGM, Stored EGM feature support by UEP COM interfaces*

In addition to the three main components GUI, API and COM, the following two components together allow the UEP to achieve the objectives described in the objective chapter

## EIISService

The device parameters, the variables of firmware, and the registers of hardware are laid out together with the encoding/decoding formula in a format of XML files. Appendix A shows this lay out in some samples of the XML files. The purpose of EIISService is to load a set of xml files that belong to a particular firmware schema. Upon loading the parameters, the EIISService performs the encoding and decoding formula of these

parameters to convert from raw values to clinical values. UEP interacts with EIISService to retrieve information of the data in different data types and sends the data to EIISService to perform encoding and decoding formula algorithm.

## Telemetry

The role of UEP's telemetry component is to communicate with the device protocol for all device interaction purposes**.** The GUI and API communicate with COM, and the COM communicates with Telemetry to interact with the device. The telemetry component is responsible for the hand shake with the device hardware and firmware. Once the device is initiated, the telemetry component passes the commands which are invoked from the GUI, API or COM level to the device for processing. Upon completion of the command process, the telemetry returns the response to the GUI, API or COM component.

## UEP – Summary of Benefit

UEP is a Windows-based PC application which could be used as an Implantable Medical Device programmer primarily in the various stages of an IMD life cycle including development, testing, clinical trials and device implantation. UEP offers Language independent API with a Bridge interface for seamless integration with other systems. In other words, UEP has a very flexible architecture so that it is easy to split into lightweight programmer building blocks. The above sections described the three main features of UEP, the DMA, RTEGM, STEGM and Interrogate and Program Device, which are

supported by the three main components: GUI, API and COM. This section describes the summary of benefits that UEP provide in terms of key features and functionalities:

**Key Features**:

- Interact with IMD device both Inductive and RF. Support Core and Extended Telemetry services through Digital Telemetry Module (DTM) & RF.
- Device Clinical Parameters (DCP) Programming using encode and decode algorithm from the (External Instrument Interface Specification (EIIS).
- Device Clinical Parameters can also be interrogated from the device and displayed in a DCP viewer. These parameters can be saved to a file and can be loaded into the device as a nominal set of device parameters.
- Telemetry commands available for that device can be launched via UEP.
- Configure and display Real time IEGM and detail markers as well as interval that the cardiac events occurred. These Real time IEGM can be saved in the device memory as stored IEGM, and then can be retrieved for post processing purposes.
- Set up the monitoring of a particular memory address using Memory watch feature. The data from these memory locations can be dumped into a file for post processing purposes.
- Repetitive process can be done by using UEP Macros record and playback. The macro capability helps a lot in initializing device processes or performing multiple telemetry commands repetitively.

- Firmware can be downloaded to the device in different flavors such as Random Access Memory (RAM), Read only Memory (ROM), RAM ROM switching, as well as just the firmware code for testing.
- The complete setting of the device parameters can be achieved via the call to the Ship setting command.
- The complete structure of memory or device image can be saved and loaded.
- Display Diagnostic data (Trend, Merlin Enhance Diagnostic, Histogram, Alert, and Episode)

**Functionalities:**

- Used mainly during development and testing.
- Some applications are customized to use during trial procedure.
- Some applications are used for certain human trials and elaborate animal studies
- Support both inductive and RF devices
- Supports both current projects (Pacer, ICD, AFM, Unity) and research projects (Nautilus, PPG, T-Wave sensing).
- The functionality of the UEP is available through an interactive GUI, a software API, and COM component.
- Provide detailed logging
- Share CPU resources with other software
- Integration with various test environments (Library, SMART, UTS, ATE, Firmware Bench Testing, Clinical Testing Scripts, etc.)
- Effective system level troubleshooting support through the logging mechanism.

- Ability to handle multiple devices concurrently.
- Quick turnaround in development and testing.
- Tight deadlines and working with multi-site team

**Benefits:**

Table 2 shows the groups across CRMD has chosen UEP as a tool for the development and testing of IMDs. The table also shows UEP components that being used by the groups. From the table, UEP components have been used widely and UEP has become a significant tool for many groups in St. Jude Medical CRMD division to integrate with other system applications for development and testing the devices.

| Group | GUI | API | COM |
|---|---|---|---|
| Unity Software Library Testing Group | yes | yes | No |
| Unity Testing System Group | yes | Yes | No |
| Verification and Validation Group | yes | yes | No |
| System Integration Group | yes | No | Yes |
| Clinical Research Group | Yes | Yes | Yes |
| Hardware Development Group | Yes | yes | yes |
| Firmware Development Group | Yes | Yes | Yes |
| Clinical System Engineer Group | yes | No | no |
| Logistic Group | Yes | Yes | Yes |
| Quality Assurance Group | Yes | yes | No |
| SMART group | Yes | Yes | No |
| Failure Analysis Group | Yes | No | no |
| External Instrument Group | Yes | No | No |
| Automation Testing Equipment Group | Yes | Yes | no |

*Table 2: UEP components being used by groups across CRMD*

**Firmware**

Due to the very flexible architecture, UEP can easily implement a feature requested in a very quick turnaround time. UEP engineers work closely with EIIS & firmware engineers to ensure that Firmware engineers can use UEP as a development tool to complete tasks. Also, Firmware Engineers can use the Bench test application that integrates with UEPAPI to complete the unit testing for feature development. Most of the time, Firmware Engineers can use UEP as a troubleshooting tool to test the features before the firmware is released to other departments.

**Verification and Validation (V&V)**

Since the UEP shares Central Processing Unit (CPU) resources with various test environments and the API is very easy to integrate with many testing environments, the V&V group uses UEPAPI to integrate with the testing library system to develop a complete test suite to test all firmware releases for all IMDs. Due to the fact that UEP has very detailed logging in both high and low level of all transactions with the device, V&V is able to use UEP to perform system level troubleshooting.

**Hardware**

Hardware group uses UEP on the design of the device hardware. The hardware design involves an iteration process to come up with the right set of chips and memory to support a specific device. UEP is then used as a main aid to simulate and testing the hardware. Hardware can also customize UEP GUI for a particular hardware/firmware development. These custom GUIs can be used for some clinical testing on animals.

### Automation Testing Equipment (ATE)

ATE switched from its own in-house host telemetry to UEP's host telemetry on VATS systems for legacy device testing & manufacturing. ATE uses UEP as a testing aid for different devices being manufactured such as Pacer, ICD, and Unity inductive and RF devices. UEP provides the Ship Setting API that allows ATE to completely configure a device to the default settings in manufacturing. Also, with the ability to handle multiple devices concurrently, UEP speeds up the testing process which is very important to CRMD prior to releasing the devices to market.

### Host Telemetry for Merlin

The UEP also used by Merlin AFM and it is used by all flavors of Merlin@Home. Since UEP architecture has support for Linux, Embedded Linux, Windows, and WinCE, UEP is used for Merlin on both Windows and web research efforts.

### Clinical Trial

The Scientists in Research group use UEP to test many research features before introducing the features to Clinical System Engineering group and then turning the features to the product. The Research group and Clinical System engineering group has used UEP as the first RAM switch on implanted ICD. UEP applications are customized towards the trial procedures used for certain Clinical Human Trial Procedure and elaborate animal studies.

**UEP for Animal Studies and Clinical Human Trial Procedure**

The UEP has been customized to be applications for use in animal studies and Clinical Human trial procedure. This section describes several applications that customized UEP components have been placed into Programmer for clinical uses.

Figure 37 shows the UEP application as High Voltage Lead Impedance Check (HVLIC) which is the programmer to test the first IMD RAM (Random Access Memory) device for human trial. UEP is used to download the RAM into the device. The lead impedance is configured and then physician can perform different therapies to patients using UEP.

UEP - TPM/Research (HVLIC)

- Custom Clinical Programmer for First ICD RAM device Human trial
- Custom clinical programmer for High Voltage Lead Impedance Check Human trial

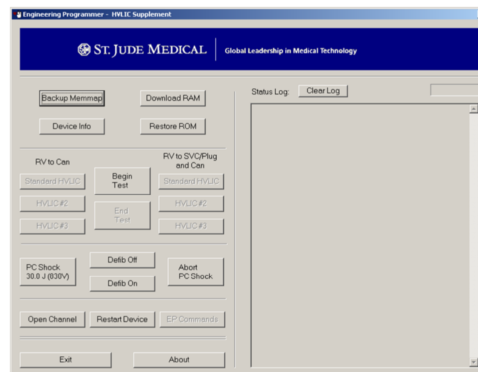



*Figure 37: UEP as programmer for first ICD RAM device*

Another animal study that uses UEP as a customer clinical programmer for pig study is called Ischemia. The sole purpose of Ischemia project is to track and trend ST Segment shifts, which is indicative of Ischemia or alternatively, a lack of oxygen in the system. The Ischemia feature can essentially predict an impending heart attack to pre-empt the

patient from getting an emergency shock and receive a more thorough treatment at the Hospital. Figure 30 shows UEP as a programmer for a study of Ischema on pig.

UEP - TPM/Research (Ischemia)

- Custom Clinical Programmer for a pig study at non-SJM facility.
- Screen designs reused by Merlin for Unity 1.3

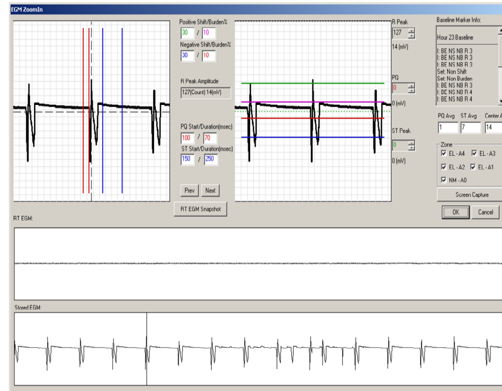


*Figure 38: UEP as a programmer for Pig Ischemia study*

The purpose of Nautilus study is for electronic repositioning of the lead impedance to avoid Phrenic Nerve Stimulation. Figure 39 shows UEP as a programmer for a study of lead impedance.

UEP based Clinical Programmer – V

Nautilus study (Tested on Merlin hardware)

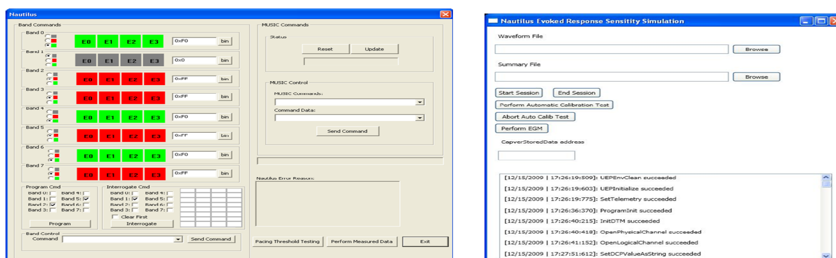


*Figure 39: UEP as a programmer for Lead Impedance study*

Morphology study on human uses UEP as a programmer that can collect the real time EGM on human and save it to the file for post processing. UEP is used inside the Merlin Programmer to program different device configuration and then to collect and display EGM. Figure 40 shows UEP inside Merlin Programmer for Human Morphology study.

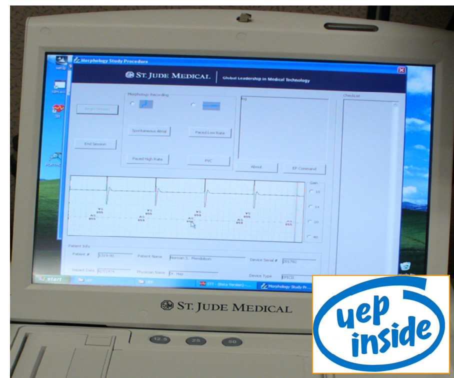



*Figure 40: UEP as a programmer for Human Morphology study*

UEP telemetry component is used to develop the Holter Monitor which is a portable device for continuously monitoring various electrical activity of heart. Figure 41 shows UEP host telemetry is used to develop the Holter Monitor.

*Figure 41: UEP is used to develop Holter Monitor*

# VI. UEP AND THE TEST FRAMEWORK

As UEP becomes an important tool in the development and testing of the IMD, the quality of UEP is a big concern. Every UEP release is integrated to the testing systems of different departments. If UEP release contains many issues, the impact across CRMD departments can be huge. Therefore, all UEP developers consider the quality of UEP release to be the top priority and so the whole team takes testing of UEP very seriously.

**Background**

Once a month, a UEP release is announced to users with new features or bug fixes. Prior to the release, UEP V&V engineers perform the testing activities. These tasks involve running a set of test scripts written in C++ that uses UEPAPI to validate the functionalities of UEP. Samples of test scripts are shown in Figure 42 below. More samples of test scripts are in Appendix B.

```
#include "UEPAPI.h"
#include "TestUtilities.h"
#include "Configuration.h"
using namespace std;
CUEPAPI uep;
int main()
{
	StartClock();
	EPParamSetType ParamSet = ACTIVE;

	Step(1);
	TesterAct("ENSURE THAT 4 ALTERED HEX FILES DESCRIBED IN STEP 6
	BELOW ARE IN DIRECTORY C:\\DEVICESOFTWARE");

	int dtmType = 0;

	getUnityModelROM(modelName, romName, projectName, &dtmType);
```

```
Attempt(uep, "UEPEnvClean", uep.UEPEnvClean());

Step(2);

int Error = 0;

Attempt(uep, "UEPInitialize", uep.UEPInitialize());

Attempt("Disable LogService", uep.EnableLogService(false));

Step(3);
Attempt(uep, "InitDTM", uep.InitDTM(byRequest, nComPort, deviceType));
WriteHeader(uep,"5", false);

Step(4);

Attempt(uep, "OpenPhysicalChannel", uep.OpenPhysicalChannel());

Step(5);

Error = uep.DownloadCode("non-existent file");

cout << "Attempt to download non-existent file.\n";

ReportIntegers(FileNotFound, Error);

Step(6);

Error = uep.UnityDownloadCode
("c:\\SWTools\\UEP\\FW\\FW_download_hv.hex",
 "non-existent file");

cout << "Attempt to download non-existent file.\n";

ReportIntegers(FileNotFound, Error);
.
.
```

*Figure 42: Sample of a UEP test script*

The above test script basically includes UEPAPI header file, and dynamically link with the UEPAPI library, so that all functionalities of UEP can be exposed to use. The test script initializes the device and then downloads the firmware code into the device.

## The Motivation

### Legacy Approach

The current UEP testing activity is manual and not data driven. Prior to testing UEP release, V&V engineers hard coded EIIS and other data for different versions of the firmware. As a result of different sets of data from one device to another device, a test script could be run in one release but not in another release. The test scripts were not modular and were also not backward compatible for devices. Hence, the whole test script suite was not easy to modify, maintain, and reuse to test multiple configurations of IMDs.

In addition, each test script includes several UEP APIs, which makes it very difficult to isolate the cause of failures. The test scripts were designed without a logging mechanism to trace the function calls and parameters. UEP V&V engineers had to manually copy the screen output to a file as a result for the test script.

The most important thing to note is that the functionality requirement coverage is not explicit in the script. Testers have to specifically code the input and output expected of the functionality in order to verify the requirement. This is time consuming and is very difficult for the requirement coverage verification. Due to the fact that this is completely manual activity, V&V engineers may not incorporate the new requirements into the scripts or may overlook the verification of some existing requirements.

Finally, each test script is built separately and executed by command line. V&V engineers can only run one script at a time, which is definitely inefficient. Moreover,

the process of building, running and documenting each test script is long, tedious, and error-prone. Typically it takes a week or sometimes two to completely test a release. Also, there are so many times where UEP developers are involved in the testing activities with V&V engineers and must work extra hours in order to meet the deadline of releasing UEP.

**Vision of Future Testing**

Since it is a part of many testing systems across CRMD departments, the release of UEP becomes a big concern in term of library dependencies. V&V engineers need to release UEP prior to the releases of other testing systems so that engineers of other testing systems have time to build, link, and test with the UEP libraries.

Another feature worth mentioning is when the devices are being manufactured, the product release process needs to have the ability to test multiple devices concurrently. This means that the formal release of UEP is required to test multiple devices. But the verification of multiple configurations usually takes a long time and the UEP team must be given a reasonable timeframe to test before releasing to manufacturing.

In order to achieve all of the above, the test scripts should be organized cleanly and easy to execute. Test scripts should be backward compatible with all firmware versions for multiple project configurations to avoid manual modification. Test scripts should be created to be data driven such that automation testing can be achieved with only a single mouse click.

On the other hand, the test scripts should be designed in such a way that the documents generated for test descriptions and requirements can be automated for each release. In addition, there must be a good automatically reporting mechanism for test result in XML format.

Lastly, the testing should be performed in both GUI based and Command line option.

UEP V&V engineers should definitely seek a solution as described above that not only enhances the quality of UEP but also decrease the turnaround time for every UEP release.

**UEP Verification Test Framework**

UEP developers and V&V engineers have jointly developed the UEP Verification Test Framework to achieve the vision of future testing as described above.

The architecture of UEP Test Framework is shown in Figure 43. The UEP Verification Test Framework is based on Visual Studio (VS) 2008 Unit Testing Framework with .NET Framework 3.5. It uses unmanaged UEP API's through Managed wrappers. The design of the framework allows it to be data driven with a set of generic test cases that can be run on multiple configurations. In addition, the library of VS Unit framework provides an automation process that not only generates the document requirements for the test cases, but also the XML report of results for the whole test suite.

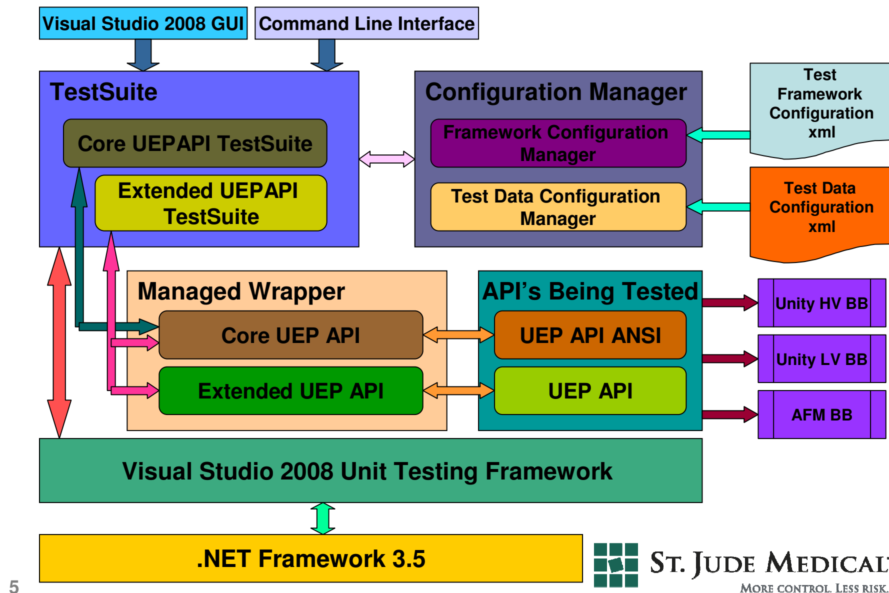



*Figure 43: UEP Test Framework Architecture*

### Test Framework Components

The test framework components consist of the test suite, the configuration manager, the managed UEP API wrapper and the configuration files.

**Test Suite** is a set of different test cases that are designed generically so that one can pick and choose them for different configurations or requirements. It covers all the testing for Core UEPAPI and Extended Core UEP API. Most of the individual tests verify only a single function. Other scripts test a pair of related or opposing functions like Open and Close Logical Channel, Program and Interrogate Device, or Set and Verify Ship Settings. The Team System automatically runs an initialization function at the

beginning of each test and a cleanup function at the end. When a test list is run, the Team System generates a file that gives the name and results (such as pass or fail) of each test.

Regarding the requirement verification, the section of a script that verifies requirement for example with ID = 2335 in DOORS is bracketed with comments:

*//Begin verification of ID 2335*

...

*//End verification of ID 2335*

This would make it possible in the future to automatically generate a table of verified IDs that could be compared against a file of requirements generated from the SRS in Dynamic Object-Oriented Requirements System (DOORS).

The Team System provides a GUI editor for organizing lists of tests (Ordered Tests) that it runs automatically in a specified sequence. All UEP test cases are assembled into a small number of lists of tests that can be run either manually or automatically with a single setup. Examples include manual inductive test, automated inductive test, manual RF test, or automated RF test.

Figure 44 shows the Team System GUI editor and Figure 45 shows how the test suite organized in the UEP Test Framework.

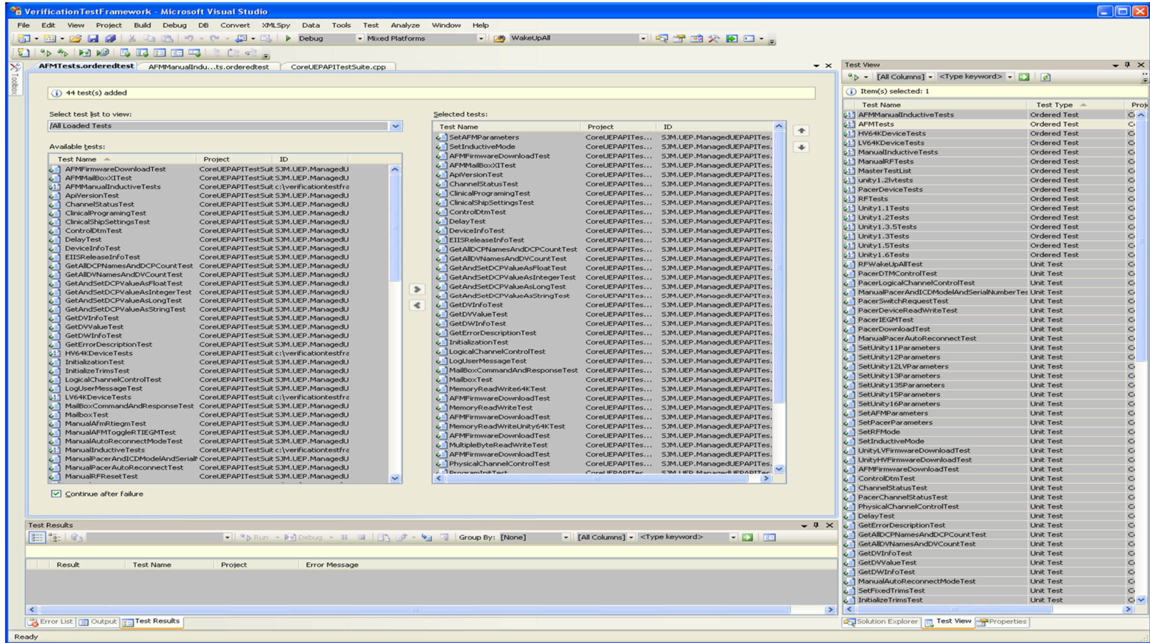



*Figure 44: the Team System GUI editor*

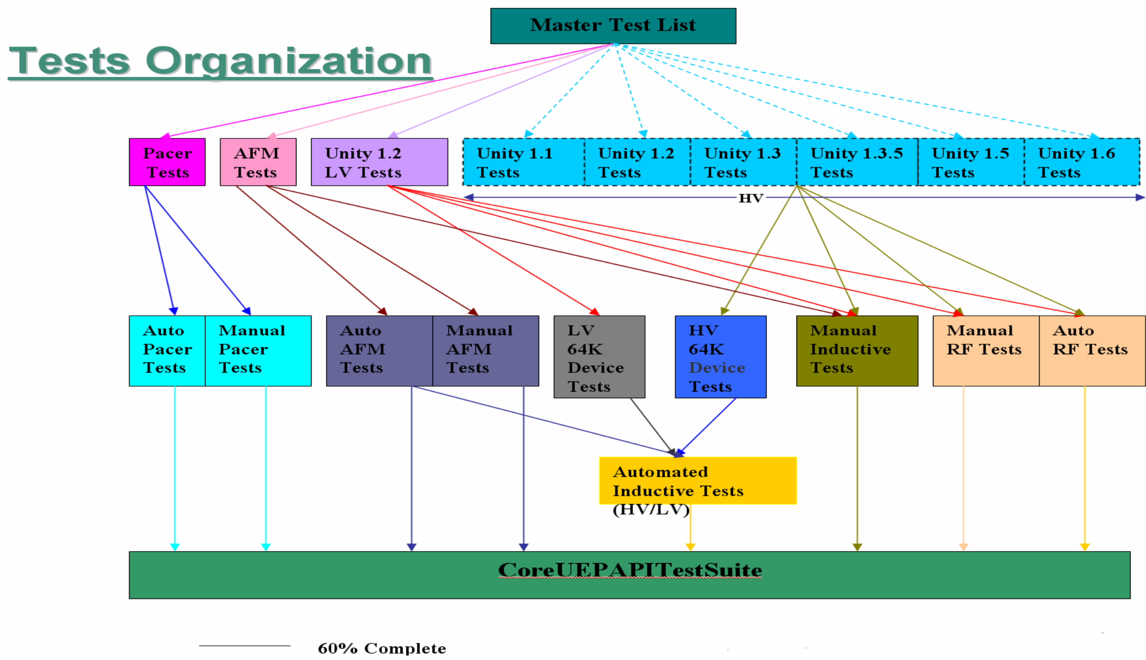



*Figure 45: test suite is organized in the UEP Test Framework.*

**The Configuration Manager** is responsible for loading the test framework configuration and test data configuration. Multiple configurations can be easily handled by having the parser extracts specific project-related data, and then saving it into a data structure that can be used by the framework. Each script has a summary with an XML tag that enables it to be extracted into a table and inserted into a test plan. Figure 46 shows the test framework configuration xml files.

## Framework Configuration


```xml
<VerificationTestFrameworkConfiguration xsi:noNamespaceSchemaLocation="VerificationTestFrameworkConfiguration.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <DefaultProject>Unity1.2LV</DefaultProject>
        <!-- Options are epCP64K, epCP8K, epICD64K, epICD8K, epPacer8K -->
        <DefaultDeviceType>epCP64K</DefaultDeviceType>
        <!-- Options are true or false -->
        <IsInductiveMode>true</IsInductiveMode>
        <UEPAPIVersion>4.84</UEPAPIVersion>
        <DTMConfiguration>
                <Request>0x50</Request>
                <Port>3</Port>
                <DTMVersion>3.5</DTMVersion>
                <DSPVersion>0.9</DSPVersion>
        </DTMConfiguration>
        <RFConfiguration>
                <RFWandAppVersion>0140</RFWandAppVersion>
                <RFWandProxyVersion>1. 0. 0. 5</RFWandProxyVersion>
                <RFWandAndIWUAppLocation>C:\Firmware\RF\app_iwu.out</RFWandAndIWUAppLocation>
        </RFConfiguration>
        <ProjectList>
            <Project name="Unity1.2LV">
                <IsBreadBoard>false</IsBreadBoard>
                <BreadBoardSerialNumber>0xC10000BD</BreadBoardSerialNumber>
                <FirmwarePath>C:\Firmware\Unity1.2LV\F0B.0E.76</FirmwarePath>
                <WandSwitchConfiguration>
                    <DTMConnector>1</DTMConnector>
                    <AWSPort>1</AWSPort>
                </WandSwitchConfiguration>
                <EIISConfiguration>
                    <FirmwareVersion>0x0B0E</FirmwareVersion>
                    <DeviceModel>LV</DeviceModel>
                    <ProjectHeader>Unity</ProjectHeader>
                    <ProjectVersion>U1.2</ProjectVersion>
                    <EIISVersion>EIIS2009.01.23</EIISVersion>
                </EIISConfiguration>
            </Project>
            <Project name="AFM">
              ..................
            </Project>
            .
            .
            .
        </ProjectList>
</VerificationTestFrameworkConfiguration>
```


*Figure 46: UEP Test Framework Configuration*

**Managed UEP API Wrapper** is a set of API written in managed code API that wraps all the unmanaged code of UEPAPI (for C++ language) and UEPAPI ANSI (for C language)

**Configuration files** consist of two XML files: VerificationTestFrameworkConfiguration.xml and EIISDataConfiguration.xml. The data driven objective is achieved in a way that these two files contain all the data that can vary with UEP release, firmware, or EIIS version. These files are updated manually, but the test cases do not need to be changed. All the control structures to select devices, firmware, or EIIS versions are avoided and, if needed, are relegated to auxiliary functions. In addition, the test framework provides individual testers with a simple interface to obtain data from the XML files

For example data about a DCP can be obtained as follows:

```
DCP^ dcp = testDataConfigManager->GetDCPData(index);
```

Class DCP contains such information as dcpName, dcpValue, dataType, units, step, offset, dvValue, blockOffset

Figure 47shows how the test data configuration is organized in the EIISDataConfiguration.xml file to allow data driven testing.

## Test Data Configuration

```
<EllSDataConfiguration xsi:noNamespaceSchemaLocation="EllSDataConfiguration.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <RevisionVersion>1.0.0</RevisionVersion>
        <TestCases>
                <TestCase name="SampleDataTest">
                        <EllSReleaseInfo idx="Idx1" id="1" name="Unity" code="0x4C000000" docNumber="40001869" date="2009-03-11  14:41:40" timestamp="2009-03-11
        02:25 PM" transformVersion="2.0" reportVersion="4.5.3" releaseVersion="2009.03.11">
                                <SupportedProjectVersions>
                                        <ProjectVersion>U1.2LV</ProjectVersion>
                                </SupportedProjectVersions>
                        </EllSReleaseInfo>
                        <DCPCount idx="Clinical_Count" count="291">
                                <SupportedProjectVersions>
                                        <ProjectVersion>U1.2LV</ProjectVersion>
                                </SupportedProjectVersions>
                        </DCPCount>
                        <DCPList>
                                <DCP idx="CLN_FIRST" name="AMS Pacing Interval" value="" units="text" returnCode="" step="" offset="" dvValue="" blockOffset="">
                                                <SupportedProjectVersions>
                                                        <ProjectVersion>U1.2LV</ProjectVersion>
                                                </SupportedProjectVersions>
                                </DCP>
                        </DCPList>
                        <DVCount idx="Clinical_Count" count="190">
                                <SupportedProjectVersions>
                                                <ProjectVersion>U1.2LV</ProjectVersion>
                                </SupportedProjectVersions>
                        </DVCount>
                        <DVList>
                                <DV idx="CLN_FIRST" name="AFD_AFibDet" value="" offset="" returnCode="" size="" blockOffset="" dwID="" isIndexed=""
        memoryBlockName=""  dataBlockName=""  memoryBlockSize="" maxEntries="" blockEntrySize="" diagGroups="" />
                                                <SupportedProjectVersions>
                                                        <ProjectVersion>U1.2LV</ProjectVersion>
                                                </SupportedProjectVersions>
                        </DVList>
                        <MemoryLocationList>
                                <MemoryLocation idx="Idx1" symbol="" address="0xFE0060" bank="" page="" offset="" size="23" returnCode="0"/>
                        </MemoryLocationList>
                </TestCase>
                <TestCase name="ProgramDeviceTest">
                        ....
                </TestCase>
                .
                .
                .
        </TestCases>
```

*Figure 47: Test Data Configuration in UEP test Framework*

**Summary of Benefits of UEP Test Framework**

Firstly, with the UEP Test Framework, the execution of one function or two opposite functions in one test makes it easier to isolate the cause of a failure. Secondly, updating of test code for a new release or configuration is reduced by putting release-specific or configuration-specific data in XML files. Thirdly, when a test list is run, the Team System generates a file that gives the name and result (pass or fail) of each test. Thus, manual copying of output files and pass/fail results of individual tests is eliminated. This feature allows the automation of reporting mechanism as well. Fourthly, a summary of a test is adjacent to the test which makes it easier to keep current and can be easily extracted for the best plan. Lastly, Testers and their reviewers can easily see the

requirement that a given section of a test is intended to verify, and whether a requirement allocated to a given test has been entirely overlooked.

**Future Plans**

Even though the development of UEP Test Framework is not complete, several UEP releases have been tested using the UEP Test Framework. The result seems to be very encouraging in terms of the quick turnaround time and the quality of the release. However, there is still a lot of work that needs to be done. Currently, there is about 80% coverage of the UEPAPI in the Test Framework. The remaining APIs should be included to get a complete test suite.

The automation of loading EIIS data from XML files should be based on version, project and schema of the firmware. In this way, the Test Framework can effectively support multiple configurations. Also, the automation of documentation generation should be changed in order to consolidate different pieces of documentation together into one process.

Visual Studio 2010 Unit Testing Framework offers a wide range of libraries that could help the development of UEP Test Framework to be more flexible and effective. Therefore, one goal of UEP V&V engineers is to port the current Test Framework in Visual Studio 2008 to Visual Studio 2010 Unit Testing Framework.

# V. NeXus – UEP NEXT GENERATION

This chapter briefly describes NeXus which is the name of the next generation of UEP. It is the new GUI that is being developed based on C# and Window Presentation Foundation (WPF). Figure 48 shows the Graphical Interface of the NeXus. The interface has most of the features offered in the old UEP GUI such as DCP Interrogate and Program, Real time Electrogram set up and display, log view, commands, but it completely managed code.

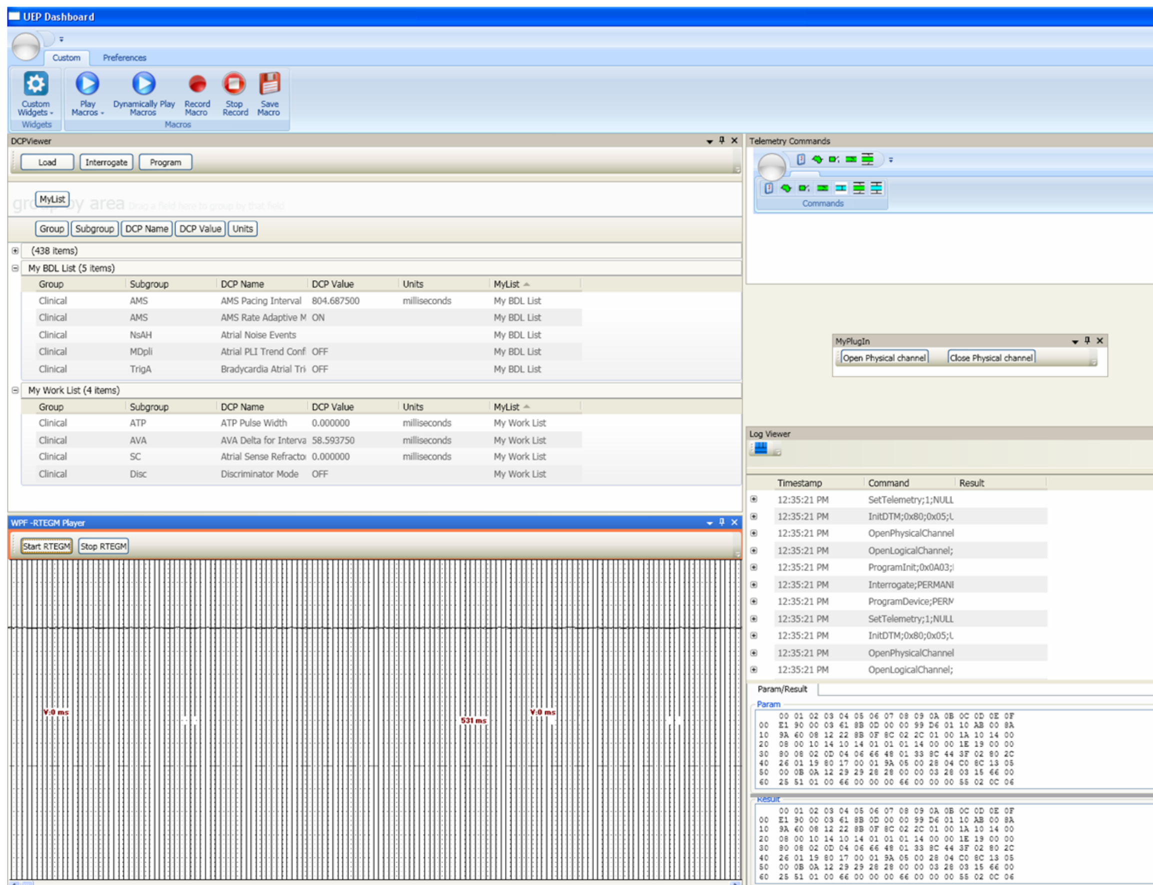
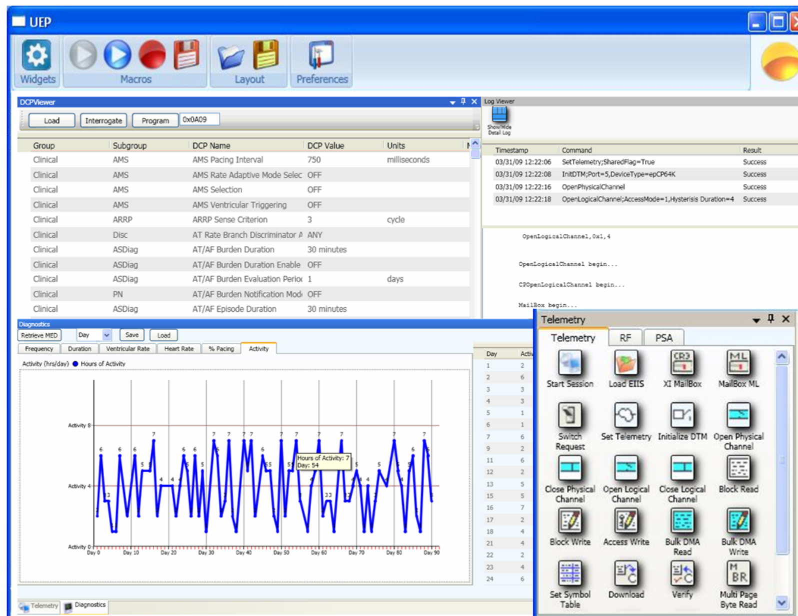



*Figure 48: UEP NeXus Graphical User Interface*

NeXus is indeed a widget framework that allows widget plug-in. Widget is basically a Dynamic Link Library (DLL) based on a set of libraries from a widget framework.

Engineers from other departments can develop widgets that can be loaded into UEP NeXus. These widgets can access UEP callable libraries to do some specific tasks. Figure 49 shows the telemetry widget and the Real time EGM widget that are loaded into UEP NeXus.

UEP: Widget Framework




. Widget Plugins

. Faster development

. Web-enabled

*Figure 49: UEP NeXus a Widget Framework*

While the old UEP GUI still has many users, the NeXus is being developing in parallel. All features of the old GUI will be ported to NeXus. Eventually, the old GUI will be phased out.

NeXus has so many benefits such as it can be a development framework for widget plug-in where users can create their own widgets and that can be loaded to the NeXus. This will help to eliminate the dependency and release schedule from one tool to another.

In addition, NeXus is developed in such a way that it is backward compatible to the legacy system.  Moreover, new features can be developed easily since it supports a lot of libraries.

# VI. CONCLUSION AND FUTURE WORKS

## Conclusion

When I joined the company in early 2004, Muthuvale Shanmugam, a software engineer had completed the backbone of UEP. I was assigned to work with him to complete the UEP, so that it could be released internally to other departments. The first release of UEP in late 2004 was a big success. Many users from different groups across CRMD had chosen UEP as a development and testing tool aid. We had numerous suggestions, feedback, and new feature requests from all over CRMD. The UEP team has grown to six developers and two V&V engineers in the US and four developers in Mumbai, India.

Figure 50 shows many phases of device development in CRMD such as Research, Hardware and Firmware development, Hardware and Firmware V&V formal testing, ATE manufacturing, system testing, and device implanted. UEP takes an important part as a main tool to assist many groups across the CRMD in their development and testing the devices. In addition, UEP has been chosen to develop some applications running in UEP backbone in System Testing phase and Device Implantation phase as mentioned in chapter 3.

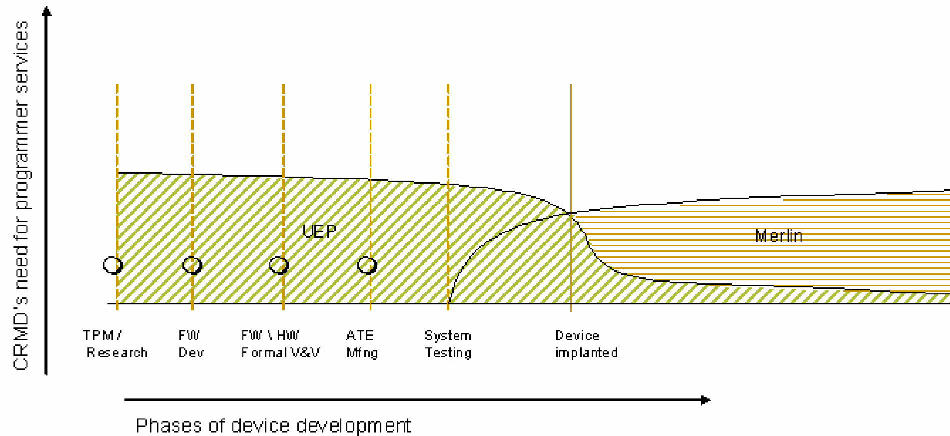



*Figure 50 – UEP participates in phases of device development in CRMD*

In conclusion, as mentioned in chapter 1, together with the CRMD devices, the Merlin Programmer is the main product. It is basically a software application that runs on a computer allowing cardiac care clinicians to retrieve and analyze data from implanted ICDs and pacemakers and make programmatic changes to them.

From Figure 50, we can see that the Merlin Programmer is not available during many early phases of the device development due to the fact that the device hardware and firmware design and implementation must be complete first so that Merlin Programmer can use the device to test its own functionalities. Besides, Merlin Programmer is an end user product that takes a long time to complete its development. As a matter of fact, programming features such as Device Clinical Parameters Interrogate and Program, Real

time Electro gram, Device Initialization, Telemetry commands, Device Memory Access etc. could take Merlin years to develop and test before releasing to market.

Engineers across CRMD recognize UEP as a tool aid with a very flexible architecture that could help develop the programming features mentioned above within several months. The quick turnaround time in development of new features has contributed to the device development in such a way that the users can always use UEP to troubleshoot and resolve issue and test the features quickly.

Some testing features such as Device Clinical Parameter info viewer, Telemetry probe, Editable EIIS, Diagnostics viewing, archival of large IEGM files, and deliberate out of boundary parameter entries are available only in UEP. These features have contributed tremendously to troubleshooting and post processing activities.

All of the UEP programming and testing features mentioned above are available to users in the form of callable libraries. These UEP callable libraries are integrated with many testing system including UTS, SMART and V&V Test libraries. Truly, UEP today becomes an important component in the device development phases of CRMD.

**Future works**

While the UEP has served as a main tool to the development and testing of CRMD devices, there has been an increasing need for new features and feedback and suggestions still need to be incorporated into UEP. These many enhancements are highlighted below:

When adding new features, UEP should always maintain backwards compatibility with legacy systems.

New features to be added include Trigger Based Logging, Dynamic Telemetry Command, User Defined Download code to the device, Device Simulation, Display and program Diagnostic Data in an interactive mode and post processing and analysis, and Rule based telemetry recovery mechanism to fix the channel lost.

Performance must be improved in features such as Device Initialization, Interrogate Diagnostic data and programming and Retrieving Stored IEGM.  In this way, other applications that depend on UEP will not worry about the performance impact.

Being used widely across the CRMD in development and testing the device, the UEP team needs to focus on the improving of trouble shooting aids in all UEP components so that engineers would easily narrow down the root causes of many system issues.  This is a huge time saving not only benefit to the development and testing team but also benefit to the company in term of scheduling the introduction of device to market.

Finally, the most important thing that the UEP team needs to focus on is the quality of every UEP release.  The team needs to extensively apply coding standards when writing the code and enforce the code review process.  Moreover, the memory management of UEP should be regularly checked against Bound Checker to avoid memory corruption.

## REFERENCES


- Unity External Instrument Interface, St. Jude Medical Cardiac Rhythm Management Division (Team Center Document Number 40001869)
- DTM2 64K Telemetry Host to Slave Software Interface Specification, St. Jude Medical Cardiac Rhythm Management Division (Team Center Document Number 60003719)
- Universal Engineering Programmer Application Program Interface SRS, St. Jude Medical Cardiac Rhythm Management Division (Team Center Document Number 60019011)
- Universal Engineering Programmer Graphical User
- Interface Software Requirements Specification, St. Jude Medical Cardiac Rhythm Management Division (Team Center Document Number 60021419)
- Tool Description Document for Trim Utility Widget v1.01, St. Jude Medical Cardiac Rhythm Management Division (Team Center Document Number 40007634)
- RF Base Station SW Requirements Specification - RF Wand, St. Jude Medical Cardiac Rhythm Management Division (Team Center Document Number 60010473)
- Firmware Verification and Validation Requirement, St. Jude Medical Cardiac Rhythm Management Division
- System for Making Automated and Random Test Requirement, St. Jude Medical Cardiac Rhythm Management Division

- Firmware Bench Testing Software Requirement, St. Jude Medical Cardiac Rhythm Management Division
- St. Jude Medical Website <http://www.sjm.com>


## APPENDIX A Sample of XML files

EIIS XML files contain the lay out of Device Clinical Parameters and Device Variables.

**EIIS_DCP_Definitions.xml: Device Clinical Parameters**

```
<Device_Clinical_Parameters>
  <Export Timestamp="2010-11-17  11:02 AM" ReportVer="4.15.1" RelVer="2010.11.17"/>
  <Enhance Date="2010-11-17  11:20:41" TransformVer="2.1"/>
  <Family ID="1" Name="Unity" Code="0x4C000000" DocNum="40001869"/>
  <DCPTypes>
    <DCPType Id="Alert" Description="Clinical Alert"/>
    <DCPType Id="ATE Trims" Description="ATE Only Trims - Non Imaged"/>
    <DCPType Id="Clinical" Description="Clinical Programmable Parameters"/>
           .
           .
           .
    </DCPTypes>
  <DCPFeatures>
    <DCPFeature Id="aATP" Description="Atrial ATP"/>
    <DCPFeature Id="ACT" Description="Automatic Calibration Test"/>
    <DCPFeature Id="ActD" Description="Activity Detection"/>
    <DCPFeature Id="ADgns" Description="Atrial Rhythm Diagnosis"/>
    <DCPFeature Id="ADtxn" Description="Atrial Rhythm Detection"/>
    <DCPFeature Id="AEDiag" Description="Atrial Episode Diagnostics"/>
    <DCPFeature Id="AERm" Description="AutoCapture Evoked Response
    .
    .
    .
 </DCPFeatures>
 <DCPReviewStatuses>
    <Status Id="NEW"/>
    <Status Id="MODIFIED"/>
    <Status Id="REVIEWED"/>
    <Status Id="DEPRECATED"/>
  </DCPReviewStatuses>
  <DCPDefinitions>
    <DCPDef Id="11164" ParamId="3565" DisplayName="ACCEL ADJUST ENB"
           WorkingName="ACCELADJUSTENB"
           SyncMode="NONE"
           Type="HW Trims"
           Feature="None"
           ReviewStatus="NEW"
           Is_Indexed="No"
           CodeGen="Yes"
           Max_Entries=""
           ModifiedDate="2010-02-05  02:04 PM">
      <RangeInfo ValType="Int" DefaultUnits="counts">
        <NumRange Start="0" End="255" Step="1" Units="counts"/>
      </RangeInfo>
      <Decode>
        <DVSources>
          <DVRef Name="ACCEL_ADJUST_ENB" Index="none">
            <BitFields/>
          </DVRef>
        </DVSources>
```

```
        <Algorithm>// mask out the upper 2 bytes to isolate
// the actual trim value

value = 0x0000FF &amp; [ACCEL_ADJUST_ENB].i;</Algorithm>
      </Decode>
      <Reset Type="Fixed" Value="" Units="">
        <DVSources/>
        <Algorithm/>
      </Reset>
      <ShipSetting Type="Fixed" Value="0" Units="counts">
        <DVSources/>
        <ConditionDVs/>
        <ConditionConstants/>
        <Algorithm>value = 0;</Algorithm>
      </ShipSetting>
      <Notes/>
      <Description>Hercules Accelerometer System Test Enable Mode (Required for writing
Accel_Gain_trim and Accel_Offset_Trim Registers)

This trim is used for Kynar sensor only.</Description>
    </DCPDef>
    <DCPDef Id="7234" ParamId="6320" DisplayName="Accel Divide Soft Trim"
         WorkingName="AccelDivideSoftTrim"
         SyncMode="NONE"
         Type="SW Trims"
         Feature="RAd"
         ReviewStatus="NEW"
         Is_Indexed="No"
         CodeGen="Yes"
         Max_Entries=""
         ModifiedDate="2008-08-30  08:27 PM">
      <RangeInfo ValType="Int" DefaultUnits="counts">
        <NumRange Start="0" End="255" Step="1" Units="counts"/>
      </RangeInfo>
      <Decode>
        <DVSources>
          <DVRef Name="Accel_Divide_Soft_Trim" Index="none">
            <BitFields/>
          </DVRef>
        </DVSources>
        <Algorithm>value = [Accel_Divide_Soft_Trim].i;</Algorithm>
      </Decode>
      <Reset Type="Fixed" Value="" Units="">
        <DVSources/>
        <Algorithm/>
      </Reset>
      <ShipSetting Type="Fixed" Value="100" Units="counts">
        <DVSources/>
        <ConditionDVs/>
        <ConditionConstants/>
        <Algorithm>value = 100;</Algorithm>
      </ShipSetting>
      <Notes/>
      <Description>A trim for MEMS sensor.</Description>
    </DCPDef>
    <DCPDef Id="11167" ParamId="3912" DisplayName="Accel Gain Soft Trim"
```

```
            WorkingName="AccelGainSoftTrim"
            SyncMode="NONE"
            Type="SW Trims"
            Feature="RAd"
            ReviewStatus="NEW"
            Is_Indexed="No"
            CodeGen="Yes"
            Max_Entries=""
            ModifiedDate="2010-02-05  02:11 PM">
        <RangeInfo ValType="Float" DefaultUnits="n/a">
          <NumRange Start="0.5" End="4.5" Step="0.25" Units="n/a"/>
        </RangeInfo>
        <Decode>
          <DVSources>
            <DVRef Name="Accel_Gain_Soft_Trim" Index="none">
              <BitFields/>
            </DVRef>
          </DVSources>
          <Algorithm>value = ([Accel_Gain_Soft_Trim].i * 0.25) + 0.5;</Algorithm>
        </Decode>
        <Reset Type="Fixed" Value="" Units="">
          <DVSources/>
          <Algorithm/>
        </Reset>
        <ShipSetting Type="Fixed" Value="0.5" Units="n/a">
          <DVSources/>
          <ConditionDVs/>
          <ConditionConstants/>
          <Algorithm>value = 0.5;</Algorithm>
        </ShipSetting>
        <Notes/>
        <Description>Acceleration gain soft trim - used for in Brady algorithms to facilitate the rate response
algorithm.
the Range in the EIIS DCP is different from the SRS range. the encode formula takes this into account. the
encode value in counts is compared with the table in the SRS (req# 1717) to get the Sensor Gain which is in
human readable form of 0.5 to 4.25

This trim is for Kynar sensor only.</Description>
      </DCPDef
   </DCPDefinitions>
</Device_Clinical_Parameters>
```


## EIIS_DV_Definitions.xml – Device Variables

```
<?xml version="1.0" encoding="UTF-8"?>
<Device_Variables>
  <Export Timestamp="2010-11-17  11:10 AM" ReportVer="4.5.2" RelVer="2010.11.17"/>
  <Enhance Date="2010-11-17  11:20:48" TransformVer="2.0"/>
  <Family ID="1" Name="Unity" Code="0x4C000000" DocNum="40001869"/>
  <DV_Type>
    <Type Id="Alert" Description="Clinical Alert"/>
    <Type Id="ATE Trims" Description="ATE Only Trims"/>
    <Type Id="CRC" Description="CRC"/>
    <Type Id="Device ID" Description="Device ID"/>
          .
          .
          .
    <Type Id="Working" Description="Working Variable"/>
  </DV_Type>
  <DV_Category>
    <Category Id="Architecture"/>
    <Category Id="Brady Summary Diagnostics"/>
    <Category Id="Bradycardia Parameter Set"/>
    <Category Id="CRC"/>
    <Category Id="Device ID"/>
    <Category Id="Electrogram Directory"/>
          .
          .
          .
</DV_Category>
  <DVReviewStatuses>
    <Status Id="NEW"/>
    <Status Id="MODIFIED"/>
    <Status Id="REVIEWED"/>
    <Status Id="DEPRECATED"/>
  </DVReviewStatuses>
  <DV_Definitions>
    <DVDef Id="4576" DisplayName="ACA_LV_AC_THistogram_Bin"
          Type="Diagnostic"
         Category="General Diagnostics"
         Feature="BiVDiag"
         ValueType="Unsigned_Int"
         FW_EnumSet=""
         Endian="LITTLE"
         Is_Indexed="Yes"
         CodeGen="Yes"
         ModifiedDate="2010-09-07  04:48 PM"
         ReviewStatus="NEW">
      <BitFields/>
      <DV_Locations>
        <DataWindow_Ref FW_Ver="0x080B" Schema="0x2B" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000"  DW_Size="0x000003"/>
        <DataWindow_Ref FW_Ver="0x090B" Schema="0x2B" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000" DW_Size="0x000003"/>
        <DataWindow_Ref FW_Ver="0x0A0B" Schema="0x5B" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000" DW_Size="0x000003"/>
        <DataWindow_Ref FW_Ver="0x0B0E" Schema="0x3E" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000" DW_Size="0x000003"/>
```

```
          <DataWindow_Ref FW_Ver="0x0C0D" Schema="0x6D" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000" DW_Size="0x000003"/>
          <DataWindow_Ref FW_Ver="0x0C0E" Schema="0x6D" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000" DW_Size="0x000003"/>
          <DataWindow_Ref FW_Ver="0x0E0A" Schema="0xEA" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000" DW_Size="0x000003"/>
          <DataWindow_Ref FW_Ver="0x0E0B" Schema="0xEA" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000"  DW_Size="0x000003"/>
          <DataWindow_Ref FW_Ver="0x1008" Schema="0xA8" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000"  DW_Size="0x000003"/>
        </DV_Locations>
        <Description>Left Ventricular Auto Capture Threshold Histogram Bin</Description>
      </DVDef>
      <DVDef Id="4576_7322" DisplayName="ACA_LV_AC_THistogram_Bin" Type="Diagnostic"
            Category="General Diagnostics"
            Feature="BiVDiag"
            ValueType="Unsigned_Int"
            FW_EnumSet=""
            Endian="LITTLE"
            Is_Indexed="Yes"
            CodeGen="Yes"
            ModifiedDate="2009-05-07  08:00 AM"
            ReviewStatus="NEW">
        <BitFields/>
        <Encode>
          <DCPSources>
            <DCPRef DisplayName="Left Ventricular Auto Capture Threshold Histogram Bin"
                  WorkingName="LeftVAutoCaptThreshHstgBin"
                  Index="current"/>
          </DCPSources>
          <Algorithm>// Encoding rules for internal testing purposes only
value = [LeftVAutoCaptThreshHstgBin]#.i;</Algorithm>
        </Encode>
        <DV_Locations>
          <DataWindow_Ref FW_Ver="0x080B" Schema="0x2B" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000"  DW_Size="0x000003"/>
          <DataWindow_Ref FW_Ver="0x090B" Schema="0x2B" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000"  DW_Size="0x000003"/>
          <DataWindow_Ref FW_Ver="0x0A0B" Schema="0x5B" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000"  DW_Size="0x000003"/>
          <DataWindow_Ref FW_Ver="0x0B0E" Schema="0x3E" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000" DW_Size="0x000003"/>
          <DataWindow_Ref FW_Ver="0x0C0D" Schema="0x6D" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000"  DW_Size="0x000003"/>
          <DataWindow_Ref FW_Ver="0x0C0E" Schema="0x6D" Block_Id="0x020D01" DW_Id="0x02"
Offset="0x000000" DW_Size="0x000003"/>
        </DV_Locations>
        <Description>Left Ventricular Auto Capture Threshold Histogram Bin</Description>
      </DVDef>
.
.
.
</DV_Definitions>
</Device_Variables>
```


## APPENDIX B – Sample of UEP Test Scripts

```
#include "C:\Program Files\SJMTools\UEP\HeaderFiles\UEPAPI.h"
#include "C:\Program Files\SJMTools\UEP\HeaderFiles\DefEnvData.h"
#include "P:\TestUtilities\TestUtilities.h"
#include "P:\TestUtilities\xmlParser.h"
#include <stdio.h>
#include <string>
#include "math.h"
using namespace std;

int ConvertBinaryToInteger(string str);
int _httoi(const TCHAR *value);


int main()
{
   CUEPAPI uep;
   Attempt(uep, "UEPEnvClean", uep.UEPEnvClean());

   Step(0);
   bool Downloading = false;
   if (PerformOptionalAction("Download"))
   {
      TesterAct("CYCLE POWER");
      Downloading = true;
   }
   Step(1);
   char nameOfDll[20] = "DEFEnv.dll";
   char server[20] = "Server";
   char log[20] = "STC";
   bool local = true;
   DefEnvData stEnv;
   strcpy(stEnv.dllName, nameOfDll);
   strcpy(stEnv.strServer, server);
   strcpy(stEnv.strLogfile, log);
   stEnv.localFlag = local;
   Attempt("UEPInitialize", uep.UEPInitialize(&stEnv));

   Step(2);
   Attempt(uep, "InitDTM", uep.InitDTM(byRequest, nComPort, deviceType));

   Step(3);
   Attempt(uep, "OpenPhysicalChannel", uep.OpenPhysicalChannel());
   if (Downloading)
   {
      cout << "Starting download . . . " << endl;
      Attempt(uep, "UnityDownloadCode", uep.UnityDownloadCode("c:\\Program
            Files\\SJMTools\\UEP\\FW\\RAM_release_application.hex"));
      Sleep(5*Seconds);
   }

   Step(4);
   Attempt(uep, "OpenLogicalChannel",uep.OpenLogicalChannel());

   Step(5);
   Attempt(uep, "ProgramInit", uep.ProgramInit(romName, modelName,
      projectName));
   WriteHeader(uep, "13");

   Step(6);
   EPParamSetType ParamSet = PERMANENT;
   Attempt(uep,"Interrogate", uep.Interrogate(ParamSet));
   Attempt(uep, "ProgramDeviceToShipSettings",
```

```
          uep.ProgramDeviceToShipSettings(ParamSet));

    Step(7);
    int numMisMatch = 0;
    const int MaxMisMatches = 1000;
    int Addresses[MaxMisMatches] = {0};
    byte ShipSettingValues[MaxMisMatches] = {0};
    byte CurrentValues[MaxMisMatches] = {0};
    byte Masks[MaxMisMatches] = {0};
    Attempt("VerifyHWRegisters",
    uep.VerifyHWRegisters(&numMisMatch, Addresses, ShipSettingValues,
         CurrentValues, Masks));

    Step(8);
    cout << "MISMATCH LIST" << hex << endl;
    for (int j = 0; j < numMisMatch; j++)
        cout << "Address = " << Addresses[j] << "\tCurrentValue = " <<
        (int)CurrentValues[j] << "\tMask = " << (int)Masks[j] << "\tShipSetting =
        " << (int)ShipSettingValues[j] << endl;

        Step(9);
        XMLNode MainNode = XMLNode::openFileHelper("C:\\Program
Files\\SJMTools\\UEP\\EIIS_HWR_Definitions.xml","HARDWAREREGISTERDEFINITIONS");
        XMLNode HwChipsetNode = MainNode.getChildNode("HwChipset");
        int numberChips = HwChipsetNode.nChildNode("HwChip");
        XMLNode HwChipNode, HwRegBlockNode, HwRegisterNode;
        int numberHwRegBlocks, numberHwRegisters = 0;
        int Address;
        const int ShipSettingSize = 100;
        char ShipSettingChars[ShipSettingSize] = "";
        byte ShipSetting;
        const int MaskSize = 10;
        char MaskChars[MaskSize] = "";
        string ShipSettingString, MaskString;
        byte Mask;
        byte CurrentValue = 0;
        bool Found = false;
        cout << hex;
        int NumberMismatches = 0;
        for (int i = 0; i < numberChips; i++)
        {
            HwChipNode = HwChipsetNode.getChildNode("HwChip", i);
            numberHwRegBlocks = HwChipNode.nChildNode("HwRegBlock");
            for (int j = 0; j < numberHwRegBlocks; j++)
            {
                HwRegBlockNode = HwChipNode.getChildNode("HwRegBlock", j);
                numberHwRegisters = HwRegBlockNode.nChildNode("HwRegister");
                for (int k = 0; k < numberHwRegisters; k++)
                {
                  HwRegisterNode = HwRegBlockNode.getChildNode("HwRegister", k);
                  strcpy(ShipSettingChars,
                 HwRegisterNode.getAttribute("ShipSetting"));
                 if (strcmp(ShipSettingChars, "N/A") == 0 ||
                    ShipSettingChars[0] == 'M' || // Match EEPROM
                    ShipSettingChars[0] == 'T'  ||   // TBD or TRIM
                    ShipSettingChars[2] == 's')    // 1's complement of ...
                    continue;
                    cout << endl;
                    Address = _httoi(HwRegisterNode.getAttribute("Address"));
                    basic_string <char> ShipSettingString (ShipSettingChars, 9);
                    ShipSetting = ConvertBinaryToInteger(ShipSettingString);
                    strcpy(MaskChars, HwRegisterNode.getAttribute("Mask"));
                    basic_string <char> MaskString (MaskChars, MaskSize - 1);
```

```
                        Mask = ConvertBinaryToInteger(MaskString);
                        Attempt("BlockRead", uep.BlockRead(Address, 1, &CurrentValue),
                            false);
                        if ((CurrentValue & Mask) == ShipSetting)
                        {
                            Found = false;
                            for (int m = 0; m < numMisMatch; m++)
                            if (Addresses[m] == Address)
                            {
                                    Found = true;
                                    break;
                            }
                            Report(!Found, "Address with correct value not found in
                                    mismatch list?");
                            cout << "Address = " << Address << "\tCurrentValue = " <<
                                    (int)CurrentValue << "\tMask = " << (int)Mask <<
                                    "\tShipSetting = " << (int)ShipSetting << endl;
                        }
                        else
                        {
                            NumberMismatches++;
                            Found = false;
                            for (int m = 0; m < numMisMatch; m++)
                                    if (Addresses[m] == Address)
                                    {
                                            Found = true;
                                            break;
                                    }
                            Report(Found, "Address with incorrect value found in
                                    mismatch list?");
                            Report(ShipSetting, ShipSettingValues[m]);
                            Report(CurrentValue, CurrentValues[m]);
                            Report(Mask, Masks[m]);
                            cout << "Address = " << Address << "\tCurrentValue = " <<
                                    (int)CurrentValue << "\tMask = " << (int)Mask <<
                                    "\tShipSetting = " << (int)ShipSetting << endl;
                         }
                    }
            }
    }
```

## APPENDIX C – Sample of Test Library Code that integrates with UEPAPI

```
#include "UTL_TLM.h"
#include "CUEP.h"

// static members
unsigned char CMailBox::m_errorcode = ERROR_CODE_NOT_VALID;
CMDBUF CMailBox::CmdBuffer;
CMDBUFITER CMailBox::m_CmdBufIter;
std::map<unsigned char, string> CMailBox::m_failureReasonTbl;
bool CMailBox::m_skipNextError;

void s_Command::operator=(s_Command& rhs)
{
	int i=0;

	this->cmdID     = (unsigned char) rhs.cmdID;
	this->cmdSize   = rhs.cmdSize;
	this->retSize   = rhs.retSize;
	for (i=0; i<MAX_OPERANDS; i++)
	{
		this->operands[i] = rhs.operands[i];
	}
	for (i=0; i<MAX_RETURN_DATA_SIZE; i++)
	{
		this->retData[i] = rhs.retData[i];
	}
}


CMailBox::CMailBox()
{
	m_errorcode = ERROR_CODE_NOT_VALID;
	m_defaultMBoxTimeOut = 0;
	m_skipNextError = false;
}

CMailBox::~CMailBox()
{
	//ClearLinks();
}

///	Initialize CMailBox object
/// \retval		PASS - command completed successfully
/// \retval		FAIL - command failed
int CMailBox::Init()
{
	COutput::Instance().LogCmdMessage("CMailBox::Init()",PUBLIC_FUNC);

	int timeOut;
	GetTimeOutResponseTime(timeOut);
	sprintf(m_msg, "UEP mailbox timed out duration for performed type commands: %d seconds", timeOut);
	COutput::Instance().WriteMessage(m_msg);

	m_failureReasonTbl[TLM_ARGUMENTS_OUT_OF_RANGE] =
				"TLM_ARGUMENTS_OUT_OF_RANGE";
	m_failureReasonTbl[TLM_ACCESS_MODE_VIOLATION] = "TLM_ACCESS_MODE_VIOLATION";
	m_failureReasonTbl[TLM_SHIELDING_VIOLATION] = "TLM_SHIELDING_VIOLATION";
	m_failureReasonTbl[TLM_COMMANDED_SHOCK_ONGOING] =
			"TLM_COMMANDED_SHOCK_ONGOING";
	m_failureReasonTbl[TLM_PTT_ONGOING] = "TLM_PTT_ONGOING";
	m_failureReasonTbl[TLM_POTENTIAL_ATRIAL_EPISODE_ONGOING] =
			"TLM_POTENTIAL_ATRIAL_EPISODE_ONGOING";
```

```
        m_failureReasonTbl[TLM_POTENTIAL_VENTRICULAR_EPISODE_ONGOING] =
                "TLM_POTENTIAL_VENTRICULAR_EPISODE_ONGOING";
        m_failureReasonTbl[TLM_TARGET_CHAMBER_NOT_SUPPORTED] =
                "TLM_TARGET_CHAMBER_NOT_SUPPORTED";
        m_failureReasonTbl[TLM_RVAC_THRESHOLD_SEARCH_ONGOING] =
                "TLM_RVAC_THRESHOLD_SEARCH_ONGOING";
        m_failureReasonTbl[TLM_PTT_CHAMBER_NOT_RV] = "TLM_PTT_CHAMBER_NOT_RV";
        m_failureReasonTbl[TLM_RVAC_MODE_OFF_OR_CALIBRATE] =
                "TLM_RVAC_MODE_OFF_OR_CALIBRATE";
        m_failureReasonTbl[TLM_COMMAND_UNKNOWN] = "TLM_COMMAND_UNKNOWN";
        m_failureReasonTbl[TLM_HFEX_PROGRAMMED_OFF] = "TLM_HFEX_PROGRAMMED_OFF";
        m_failureReasonTbl[TLM_LOGICAL_CHANNEL_CLOSED] =
                "TLM_LOGICAL_CHANNEL_CLOSED";
        m_failureReasonTbl[TLM_AI_ONGOING] = "TLM_AI_ONGOING";
        m_failureReasonTbl[TLM_ATRIAL_HV_NOISE_ONGOING] =
                "TLM_ATRIAL_HV_NOISE_ONGOING";
        m_failureReasonTbl[TLM_VENTRICULAR_HV_NOISE_ONGOING] =
                "TLM_VENTRICULAR_HV_NOISE_ONGOING";
        m_failureReasonTbl[TLM_VENTRICULAR_LV_NOISE_ONGOING] =
        m_failureReasonTbl[UEP_FAILED] = "UEP_FAILED";
        m_failureReasonTbl[ERROR_CODE_NOT_VALID] = "ERROR_CODE_NOT_VALID";

        return PASS;
}

int CMailBox::SendCmd(s_Command &cmd, bool ignoreError)
{
        int result;
        BYTE ackData[1];
        BYTE TxStatus[1];
        char ackTime[50];
        bool bSkipMailBoxError = false;

        COutput::Instance().LogCmdMessage("CMailBox::SendCmd(s_Command &cmd)",PUBLIC_FUNC);

        CRF::Instance().HandleAsynchRFErrorIfAny();

        if ((true == ignoreError) || (true == m_skipNextError))
        {
                bSkipMailBoxError = true;
                m_skipNextError = false;        // clear the member flag
        }

        if(false EQ CRF::Instance().IsRFActive())
        {
                if (CSystem::Instance().m_UEPConnected == NO ||
                        CDevice::Instance().GetPhysicalChannelStatus() == CLOSED)
                {
                        COutput::Instance().WriteMessage("UEP or channel is OFF, couldn't perform
                                CMailBox::SendCmd()");
                        m_errorcode = UEP_FAILED;
                        return FAIL;
                }
        }

        // check return data buffer size (response length should at least be 3)
        if (cmd.retSize == 1 || cmd.retSize == 2)
        {
                sprintf(m_msg, "WARNING: return size = %d", cmd.retSize);
                COutput::Instance().WriteWarning(m_msg);
        }
```

```
// put the 1/2 delay here since most tests, like Mcmt.exe, are keyed of the sensed P or R and these tests
// expect the command to be sent after the sensed P or R.
if ((true EQ CRF::Instance().IsRFActive()) AND (true EQ CRF::Instance().IsInductiveEmulationActive()))
{
        Wait(50, "Wait 50 msec for before sending mbox command in case this command is keyed of the P
                and R");
}

// send command, don't wait for response if return data is not requested
if (cmd.retSize == 0)
{
   if (CmdBuffer.size() < MAX_LINK_COUNT)
   {
     result = CUEP::Instance().SendMailBoxXICommand(cmd.cmdSize, (BYTE *) &cmd, -1, ackTime);

     if (result != PASS)
     {
        m_errorcode = UEP_FAILED;
        sprintf(m_msg, "Failed to send mailbox command, ID=%02Xh, AckTime=%02Xh", cmd.cmdID,
                ackTime);
        PrintCmd(cmd);
        // don't need to check for error or command response bytes per user's request
        if(true == bSkipMailBoxError)
        {
                COutput::Instance().WriteMessage(m_msg);
                return PASS;
        }
        else
        {
                COutput::Instance().WriteSystemError(m_msg);
                CSystem::Instance().HandleUEPError(result);
                return FAIL;
        }
     }
     else
     {
        s_Command* cmdPtr = new s_Command [1];
        if (cmdPtr == NULL)
        {
           if (false EQ CSystem::Instance().m_ExceptionThrown)
           {
                CSystem::Instance().m_ExceptionThrown = true;
                throw "Failed to allocate memory in CMailBox::SendCmd()";
           }
        }
        (*cmdPtr) = cmd;
        CmdBuffer.insert(pair<BYTE,s_Command*> (cmd.cmdID, cmdPtr));
        CRF::Instance().EmulateDTMPerformance(RF_MBOX_COMP);
        return PASS;
     }
   }
  else
  {
      COutput::Instance().WriteSystemError("Cannot Send Command - All mailbox buffers in the device
        are full");
      return FAIL;
   }
}
// send command, wait for response
else
{
    result = CUEP::Instance().MailBoxXI((BYTE) cmd.cmdSize, (BYTE *) &cmd, 1, cmd.retSize +
```

```
                    TLM_RESPONSE_BYTE, ackData, TxStatus, cmd.retData);
   // copy returned data, remove first 6-byte DTM or UEP header
  for(int i = 0;i<cmd.retSize;i++)
 {
        cmd.retData[i] = cmd.retData[i+TLM_RESPONSE_BYTE];
 }

 // print error if mailbox command not sent successfully
 if (result != PASS)
{
        m_errorcode = UEP_FAILED;
        sprintf(m_msg, "Failed to send mailbox command, ID=%02Xh, AckByte=%02Xh,
                    FailReason=%02Xh", cmd.cmdID, ackData[0], cmd.retData[2]);
                    // don't need to check for error or command response bytes per user's request
                    if(true == bSkipMailBoxError)
                    {
                            COutput::Instance().WriteMessage(m_msg);
                            return PASS;
                    }
                    else
                    {
                            COutput::Instance().WriteSystemError(m_msg);
                            CSystem::Instance().HandleUEPError(result);
                            return FAIL;
                    }
        }
        else
        {
                    // is command being acknowledged
                    if(ackData[0] != TLM_ACKNOWLEDGED)
                    {
                        if(true == bSkipMailBoxError)
                        {
                            COutput::Instance().WriteMessage("Telemetry command not acknowledged");
                        }
                        else
                        {
                          COutput::Instance().WriteSystemError("Telemetry command not acknowledged");
                          return FAIL;
                         }
                     }
                     // check if the device returned an error
                     if(cmd.retData[1] != TLM_SUCCESS)
                    {
                        m_errorcode = cmd.retData[2];
                        // log mail box failure reason returned by firmware
                        if (m_failureReasonTbl.find(m_errorcode) NE m_failureReasonTbl.end())
                        {
                            sprintf(m_msg, "Mailbox failure reason: %s",
                                        m_failureReasonTbl[m_errorcode].c_str());
                            COutput::Instance().WriteHighlightedMessage(m_msg);
                         }
                         else
                         {
                            sprintf(m_msg, "Mailbox failure code \'%Xh\' isn't in the library list",
                                        m_errorcode);
                            COutput::Instance().WriteHighlightedMessage(m_msg);
                          }
                     }
                    else
                    {
                            m_errorcode = ERROR_CODE_NOT_VALID;
```

```
                    }
                    // log the content of mbox command if encountering an error or command line debug
                            //option >= 1
                    if ((cmd.retData[1] != TLM_SUCCESS) OR (COutput::Instance().GetDebugLevel() >=
                            TEST_DEBUG))
                    {
                            PrintCmd(cmd);
                    }
                    CRF::Instance().EmulateDTMPerformance(RF_MBOX_COMP);
                    return PASS;
            }
    }
}
```

## APPENDIX D – Sample of Firmware Bench Test code that integrates with UEPAPI

```
#include "StdAfx.h"
#include "BenchTestEnviron.h"
#include "Programming.h"

namespace bench_test {

 Programming::Programming(ParameterSet baseSet, ParameterSet targetSet)
    : baseSet(baseSet), targetSet(targetSet)
{
         if (baseSet != RESET_SET && baseSet != ACTIVE_SET)
        {
                throw std::runtime_error("Invalid baseSet initialization for Programming");
        }
        if (targetSet == RESET_SET)
        {
                throw std::runtime_error("Invalid targetSet initialization for Programming");
        }
 }

 Programming::~Programming()
 {
        for (std::vector<Programmable*>::iterator ppParam = parameters.begin();
                ppParam != parameters.end(); ppParam++)
        {
                delete (*ppParam);
        }
 }

 void Programming::Execute(void)
 {
         CUEPAPI& UEP = BenchTestEnviron::Instance()->GetUEPAPI();
        EPParamSetType paramSet = (this->baseSet == RESET_SET)? RESET: ACTIVE;

        RUN_UEP(UEP.Interrogate(paramSet));

        for (std::vector<Programmable* >::iterator iter = parameters.begin(); iter != parameters.end(); iter++)
        {
                (*iter)->Program();
        }

        Commit();
 }

 void Programming::Commit(void)
 {
        BenchTestEnviron * env = BenchTestEnviron::Instance();

        RUN_UEP(env->GetUEPAPI().ProgramDevice(
                (EPParamSetType) targetSet,
                env->GetModelNo(),
                env->GetSerialNo(),
                env->GetSWVersionNo(),
                0,
                0));
 }
} // namespace bench_test
```

## APPENDIX E – Sample of SMART code that integrates with UEPAPI

```
#include "StdAfx.h"
#include ".\telemetryclinicianoperations.h"
#include ".\telemetrydeviceoperations.h"
#using <mscorlib.dll>
#include <string.h>
#include <stdio.h>

static CUEPAPI uep;

//////////////////////////////////////////////////////////////////////
// Constructor
//////////////////////////////////////////////////////////////////////


SJM::Smart::TelemetryOperations::Clinician::Clinician()
{
		#ifdef _DEBUG
				Console::WriteLine("Clinician() --- Enter Clinician Constructor");
		#endif
		SJM::Smart::UTS::TestController::m_lTelemResponse = -1;
}
SJM::Smart::TelemetryOperations::Clinician::~Clinician()
{
		#ifdef _DEBUG
				Console::WriteLine("Clinician() --- Enter Clinician Destructor");
		#endif
		SJM::Smart::UTS::TestController::m_lTelemResponse = -1;
}

//////////////////////////////////////////////////////////////////////
// Clinician Name Space Functions
//////////////////////////////////////////////////////////////////////
void SJM::Smart::TelemetryOperations::Clinician::Interrogate(System::String* ParameterSetSelection)
{
		char chParameterSetSelection[256];

		EPParamSetType eParamSet;
		try
		{
				#ifdef _DEBUG
						Console::WriteLine("Clinician() --- Enter Interrogate");
				#endif

				 strcpy(chParameterSetSelection,
						 (char*)(void*)Marshal::StringToHGlobalAnsi(ParameterSetSelection));

				if(_stricmp(chParameterSetSelection,"active") == 0)
				{
						eParamSet = ACTIVE;
				}
				else if(_stricmp(chParameterSetSelection,"temporary") == 0)
				{
						eParamSet = TEMPORARY;
				}
				else if(_stricmp(chParameterSetSelection,"permanent") == 0)
				{
						eParamSet = PERMANENT;
				}
				else if(_stricmp(chParameterSetSelection,"evvi") == 0)
				{
						eParamSet = ACCELZ_SET0; // EVVI
```

```
            }
            else if(_stricmp(chParameterSetSelection,"reset") == 0)
            {
                        eParamSet = RESET;
            }
            else
            {
                        // Default PERMANENT
                        eParamSet = PERMANENT;
            }
            bool UEPNonBlocking = false;
            if(SJM::Smart::UTS::TestController::ISUEPBlockingModeEnabled() == false)
            {
                        //switch to blocking mode if non-blocking is enabled
                        //wait till the command is in progress and then disable UEP multithreading
                        UEPNonBlocking = true;
                        int retStatus  = -1;
                        try
                        {
                            Console::WriteLine("Interrogate() --- BEFORE StopUEPMultithreading");
                            retStatus = SJM::Smart::UTS::TestController::StopUEPMultithreading(false, false);
                            Console::WriteLine("Interrogate() --- AFTER StopUEPMultithreading");
                            Console::WriteLine(retStatus);
                        }
                        catch(System::Exception *e)
                        {
                                    // Thread Abort Exception - Please ignore it.
                                    Console::WriteLine("Interrogate() --- MAY BE Thread Abort Exception ");
                                    Console::WriteLine(e->Message);
                                    Console::WriteLine(e->StackTrace);
                        }
            }
            ::Sleep(2000);
            Console::WriteLine("Interrogate() --- BEFORE BLOCKING CALL");
            SJM::Smart::UTS::TestController::m_lTelemResponse = uep.Interrogate(eParamSet);
            if (SJM::Smart::UTS::TestController::m_lTelemResponse != success)
            {
                         char msg[1024];
                sprintf(msg, "\tFAILED --- Interrogate, Error Code = %d",
                        SJM::Smart::UTS::TestController::m_lTelemResponse);
                TCLog(msg);
            }
            Console::WriteLine("Interrogate() --- AFTER BLOCKING CALL");
            Console::WriteLine(SJM::Smart::UTS::TestController::m_lTelemResponse);
            if(UEPNonBlocking == true)
            {
                        int retStatus = SJM::Smart::UTS::TestController::StartUEPMultithreading(false);
            }
}
catch(System::Exception *e)
{
            Console::WriteLine("Interrogate() --- Exception");
            Console::WriteLine(e->Message);
            Console::WriteLine(e->StackTrace);
            char msg[1024];
            sprintf(msg, "\tException occurred in Clinician::Interrogate: %s", e->ToString());
            TCLog(msg);
            SJM::Smart::UTS::TestController::m_lTelemResponse = -1;
}
__finally
{
```

```
			#ifdef _DEBUG
				Console::WriteLine("Clinician() --- Exit Interrogate");
			#endif
		}
}


void SJM::Smart::TelemetryOperations::Clinician::GetDiagnosticData(System::String* ParameterSetSelection ,
System::String* SubGroupName)
{
		char chParameterSetSelection[256];
		char chSubGroupSelection[256];
		char* subGroup = NULL;

		EPParamSetType eParamSet;
		try
		{
			#ifdef _DEBUG
				Console::WriteLine("Clinician() --- Enter GetDiagnosticData");
			#endif

			if (SubGroupName == NULL)
			{
	SubGroupName = "ALL";
			}


		strcpy(chParameterSetSelection,(char*)(void*)Marshal::StringToHGlobalAnsi(ParameterSetSelection));
			strcpy(chSubGroupSelection,(char*)(void*)Marshal::StringToHGlobalAnsi(SubGroupName));

			if(_stricmp(chParameterSetSelection,"active") == 0)
			{
				eParamSet = ACTIVE;
			}
			else if(_stricmp(chParameterSetSelection,"temporary") == 0)
			{
				eParamSet = TEMPORARY;
			}
			else if(_stricmp(chParameterSetSelection,"permanent") == 0)
			{
				eParamSet = PERMANENT;
			}
			else if(_stricmp(chParameterSetSelection,"evvi") == 0)
			{
				eParamSet = ACCELZ_SET0; // EVVI
			}
			else if(_stricmp(chParameterSetSelection,"reset") == 0)
			{
				eParamSet = RESET;
			}
			else
			{
				// Default PERMANENT
				eParamSet = PERMANENT;
			}
			if (_stricmp(chSubGroupSelection,"ALL") != 0)
			{
			   subGroup = chSubGroupSelection;
			}

			bool UEPNonBlocking = false;
```

```
			if(SJM::Smart::UTS::TestController::ISUEPBlockingModeEnabled() == false)
			{
				//switch to blocking mode if non-blocking is enabled
				//wait till the command is in progress and then disable UEP multithreading
				UEPNonBlocking = true;
				int retStatus  = -1;
				try
				{
					Console::WriteLine("GetDiagnosticData() --- BEFORE StopUEPMultithreading");
					retStatus = SJM::Smart::UTS::TestController::StopUEPMultithreading(false, false);
					Console::WriteLine("GetDiagnosticData() --- AFTER StopUEPMultithreading");
					Console::WriteLine(retStatus);
				}
				catch(System::Exception *e)
				{
					// Thread Abort Exception - Please ignore it.
					Console::WriteLine("GetDiagnosticData() --- MAY BE Thread Abort Exception ");
					Console::WriteLine(e->Message);
					Console::WriteLine(e->StackTrace);
				}
			}
			::Sleep(2000);
			Console::WriteLine("GetDiagnosticData() --- BEFORE BLOCKING CALL");
			SJM::Smart::UTS::TestController::m_lTelemResponse =
			uep.Interrogate(eParamSet,epDiagnostics,subGroup); // Entire Diagnostics set
			if (SJM::Smart::UTS::TestController::m_lTelemResponse != success)
			{
			    char msg[1024];
			    sprintf(msg, "\tGetDiagnosticData FAILED --- Interrogate, Error Code = %d",
			    SJM::Smart::UTS::TestController::m_lTelemResponse);
			    TCLog(msg);
			}
			Console::WriteLine("GetDiagnosticData() --- AFTER BLOCKING CALL");
			Console::WriteLine(SJM::Smart::UTS::TestController::m_lTelemResponse);
			if(UEPNonBlocking == true)
			{
				int retStatus = SJM::Smart::UTS::TestController::StartUEPMultithreading(false);
			}
		}
		catch(System::Exception *e)
		{
			Console::WriteLine("GetDiagnosticData() --- Exception");
			Console::WriteLine(e->Message);
			Console::WriteLine(e->StackTrace);
			char msg[1024];
			sprintf(msg, "\tException occurred in Clinician::GetDiagnosticData: %s", e->ToString());
			TCLog(msg);
			SJM::Smart::UTS::TestController::m_lTelemResponse = -1;
		}
		__finally
		{
			#ifdef _DEBUG
				Console::WriteLine("Clinician() --- Exit GetDiagnosticData");
			#endif
		}
	}
```