

Creating software libraries to improve medical  
device testing of the Pacing System Analyzer  
(PSA) at 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  
Joel Canlas  
June 2011

© 2011  
Joel Canlas  
ALL RIGHTS RESERVED

## COMMITTEE MEMBERSHIP

Title: Creating software libraries to improve medical device testing of the Pacing System Analyzer (PSA) at St. Jude Medical

Author: Joel Canlas

Date Submitted: June 2011

Thesis Advisor: Lily Hsu Laiho, Ph.D.

Committee Chair: Lily Hsu Laiho, Ph.D – Biomedical Engineering

Committee Member: Kristen O'Halloran Cardinal, Ph.D – Biomedical Engineering

Committee Member: Robert Crockett, Ph. D – Biomedical Engineering

## ABSTRACT

Creating software libraries to improve medical device testing of the Pacing System Analyzer (PSA) at St. Jude Medical  
Joel Canlas

Software testing, specifically in the medical device field, has become increasingly complex over the last decade. Technological enhancements to simulate clinical scenarios and advancements in communicating to medical devices have created the need for better testing strategies and methodologies. Typical medical device companies have depended on manual testing processes to fulfill Food and Drug Administration (FDA) submission requirements specifically Class III devices which are life supporting, life sustaining devices. At St. Jude Medical, software testing of Class III devices such as implantable cardioverter-defibrillators (ICDs), pacemakers, and pacing analyzers are given top priority to ensure the highest quality in each product. High emphasis is made on improving software testing for ease of use and for catching more software errors in each device. A significant stride in testing has automated the process and has provided software verification teams with the tools they need to successfully test and deliver high quality products. By creating software libraries which interact with communication to the other interfaces needed to test medical devices, test engineers can focus on fully testing device requirements and will not be concerned with how each test will interact with the device or any other testing tools.

The main focus will be a specific St. Jude Medical device known as the Pacing System Analyzer (PSA). The PSA device will be used to demonstrate how verification engineers are able to benefit from software libraries and allow the testing process and test development to be fully automated.

New technologies and standards will be created to simulate clinical scenarios and to communicate to new devices. The goal is to use software engineering principles to create standard test libraries which sustain these changes while still allowing testers to focus on finding issues for each device.

Keywords: St Jude Medical, implantable cardioverter defibrillators, pacemakers, Pacing System Analyzer, software testing, software libraries, test methodologies, test technologies

## ACKNOWLEDGEMENTS

I would like to take this opportunity to give my sincere thanks and gratitude towards Professor Lily Laiho for her guidance, mentoring, and patience throughout the entire Biomedical Engineering Masters program at CalPoly (St Jude Medical Distance Learning).

## TABLE OF CONTENTS

LIST OF FIGURES .....	viii
BACKGROUND .....	1
Software Based Testing .....	1
St Jude Medical Devices: Pacemakers and ICDs .....	2
Medical Device Software Testing and the FDA .....	2
Requirements Driven Testing .....	3
St Jude Medical Devices and Internal Tools.....	4
Bridging the gap: Unified Testing Libraries.....	4
Pacing and Sensing Leads for ICDs and Pacemakers.....	5
PACING SYSTEM ANALYZER (PSA) .....	6
Pacing System Analyzer (PSA) Overview: .....	6
Data Transfer Protocol to Improve Speed .....	10
OBJECTIVE .....	11
Software Testing Process.....	11
Test Design .....	11
Test Cases and Test Scripts.....	12
Test Results.....	12
METHODS AND MATERIALS.....	14
Software Libraries.....	14
Updating Software Libraries.....	14
UTL Update Process .....	15
UTL Tools.....	15
Current UTL configuration for PSA device testing.....	17
SOFTWARE LIBRARY METHODS AND UPDATES FOR PSA .....	19
Example 1: Get Device Info .....	19
Example 2: Hardware Paced Pulse Information .....	22
Example 3: Initialize PStim .....	24
Example 4: Verify LED Status .....	27
Example 5: Dump Trace File.....	29
DISCUSSION AND FUTURE IMPROVEMENTS .....	31
DISCUSSION AND FUTURE IMPROVEMENTS .....	31
CONCLUDING REMARKS.....	32
ACRONYMS AND DEFINITIONS .....	33
APPENDIX A.....	34
REFERENCES .....	35

## LIST OF FIGURES

Figure 1. Merlin Programmer and PSA Device.....	9
Figure 2. Requirement Coverage in a test design document .....	11
Figure 3. Test Design Procedure.....	11
Figure 4. Example C++ code snippet for test cases .....	12
Figure 5. Example trace file output from test case .....	13
Figure 6. System Components of the UTL (all projects).....	17
Figure 7. System Components of the UTL (modified for PSA).....	18
Figure 8. GetDeviceInfo function.....	21
Figure 9. GetHWpacedPulseInfo function .....	23
Figure 10. InitPStim function .....	26
Figure 11. VerifyLEDStatus function.....	28
Figure 12. DumpTraceToFile function.....	30

## BACKGROUND

### Software Based Testing

Over the past decade software testing has become easier and harder than ever. Software testing has become more difficult due to the vast world of technology where new operating systems, programming languages, and the internet are constantly growing in complexity. In contrast, these advances have also helped to create automation of testing in order to streamline software testing processes. Clinical trials and studies have given clues to medical device researchers as to which algorithms can help patients and which therapies are most effective. Alongside these technologies the medical device field and medical device algorithms have increased in complexity. This increase has caused medical device software testing to become more difficult. In complex cases, software testers utilize device source code and work closely with software developers in order to fully understand each algorithm. This practice does not require any interpretation of software requirements and leads to designing test scenarios according to software implementation. Furthermore, there are instances where software requirements are interpreted differently by the developer and the tester causing untested software to be delivered to devices.

At St. Jude Medical, quality is always top priority. Requirements based testing has been at the core of the software testing department for many years. The software requirements, derived from system specifications, are contributed to by doctors, field clinicians, and marketing groups. System requirements are then derived from the system specifications and are used by testing groups as the basis of test design and test

implementation. Every software requirement is tested as stated by the standard operating procedures document submitted upon FDA approval.

The software testing group has adapted quickly to these changes in requirements and software constantly modifying test strategies to ensure quality products. However, testing methodologies need to be constantly questioned and examined to answer the question: are tests designed to find software defects or are tests designed to prove defects were not found? The main concern in software development testing is that software engineers have been focusing too much on how to test the system rather than what they should be testing. Plain and simple: Software testing is the process of executing a test script or program with the intent of finding errors. Improving on current test methodologies will help keep this statement valid for years to come.

#### St Jude Medical Devices: Pacemakers and ICDs

St Jude Medical's Cardiac Rhythm Management Division produces two main products: implantable pacemakers and implantable cardioverter defibrillators (ICD). A pacemaker is a device that paces a slow pacing heart using low amplitude electrical impulses. An ICD is a device that monitors a fast pacing heart and delivers high-voltage shocks to try and contain its pacing rate.

#### Medical Device Software Testing and the FDA

Medical device software testing is driven by a number of factors. First, there is an increased concern for safety in the medical device field. Every year less than 1% of all implanted devices are returned to the original manufacturer to investigate reasons for

failure. Failures may include high risk issues such as rapidly declining battery life but may also include low risk issues such as diagnostic recording. Nevertheless, the FDA heavily scrutinizes medical devices produced by a company especially those in development of a Class III life sustaining device such as a pacemaker or an ICD. Second, there is increased competition from other medical device companies. Improving software testing will catch more software defects and issues. With fewer software bugs in the field, our product will be more reliable and doctors and clinicians will choose our family of products over the competition.

The main concern regarding medical device software testing is that St. Jude Medical is not a typical software company. Many software companies are comprised of various testing departments, but none is subject to the scrutiny of strict regulatory agents such as the FDA. While most software companies can test products after launch, a medical device company must ensure a quality product before any device is assembled, shipped, and implanted into a patient.

### Requirements Driven Testing

Software testing at St Jude Medical is based on software requirements that are derived from a list of system requirements which are derived from clinical studies, doctorate research, or marketing requests. The FDA does not require medical device companies to test every single requirement listed for each device, but St Jude Medical has a test scenario for each requirement to once again ensure and deliver high quality products. Although St Jude Medical adheres to requirements based testing, there are many different types of software testing. There are other software groups within the

software organization which perform bench tests (simple functional tests created and executed by developer), ad-hoc tests (non-requirements based testing by creating edge-case scenarios), and randomized testing (which randomly selects inputs to the device in order to test as many combinations of input data as possible).

### St Jude Medical Devices and Internal Tools

St Jude Medical devices include embedded software which runs a very simple instruction set on a small processor running a real time operating system. Operating systems on normal personal computers (Microsoft Windows) do not provide testers the speed of testing the fast capabilities of a medical device. This is the cornerstone for the need of internal tools developed to assist in testing. With the creation of Heart Simulators (HS), Simulation Test Tools (STT), Digital Interface Modules (DIM), and the Universal Engineering Programmer (UEP), test teams at St Jude can verify software requirements on devices running real time operating systems and not experience delays or synchronization issues while running test scripts (see Chapter 4 – Methods and Materials for more information)

### Bridging the gap: Unified Testing Libraries

Test engineers use requirements listed in the Systems Requirement Specifications (SRS) to create and design test cases. In previous years, engineers would need to send commands and inputs to the device manually and record output data using a logic analyzer. Manual verification of each requirement was performed by signing off on logic analyzer logs. Since then, the software test group has improved its processes by adopting

object oriented programming languages such as C++ and Java to create test scripts which run on Unified Test Systems (UTS). This system is comprised of an actual working breadboard which mimics the hardware on an actual implantable device and utilizes the tools and interfaces testers can use to interact with the device. More importantly, the logic analyzer has been replaced with a digital interface module for recording all device data. The data can be stored and merged with logged test results to create a testing system that is cutting edge using the latest technology that is available today.

The dilemma of how testers will interact with this intricate test system is still an ongoing process. Most test engineers in the verification and validation (V&V) group have biomedical or bioscience experience, not software programming expertise. A set of C++ testing libraries was created to help bridge the gap. The Unified Test Library (UTL) team was created to help create, maintain, and improve software testing amongst the V&V team members. The main goal of this team is to create a set of software libraries and toolsets to help test devices by focusing on “what to test” in contrast to “how to test”. The users can call library functions for specific tasks and the libraries will be responsible for communicating data between the internal tools and the device. Taking this concept even further, the automation of this entire process has increased productivity across all levels of development and is now being used by other software testing groups within St Jude Medical.

#### Pacing and Sensing Leads for ICDs and Pacemakers

Medical device testing has been held to the highest standards by the FDA. St. Jude Medical produces Class III devices such as pacemakers and ICDs to treat patients

with arrhythmias and other heart conditions. The ICDs and pacemakers are implantable and are connected to plastic coated metal wires known as leads. Two categories of leads exist: Sensing and Pacing. The sensing leads carry heart rate information signals back to the ICD or pacemaker. The pacing leads deliver small pulse waves to various heart chambers (pacemakers) and have the ability to deliver a high energy shock (ICD) up to 840 volts. The latest technological advances have allowed both the pacing and sensing lead to be combined and functions as a sensing and pacing/shocking composite lead.

A new product created by St Jude Medical called the Pacing System Analyzer (PSA) has been developed as a supplemental tool to analyze leads during a surgical procedure and is aimed at implant or replacements of cardiac leads and devices. This product will be the case study and main emphasis for improving medical device software testing and the UTL.

### PACING SYSTEM ANALYZER (PSA)

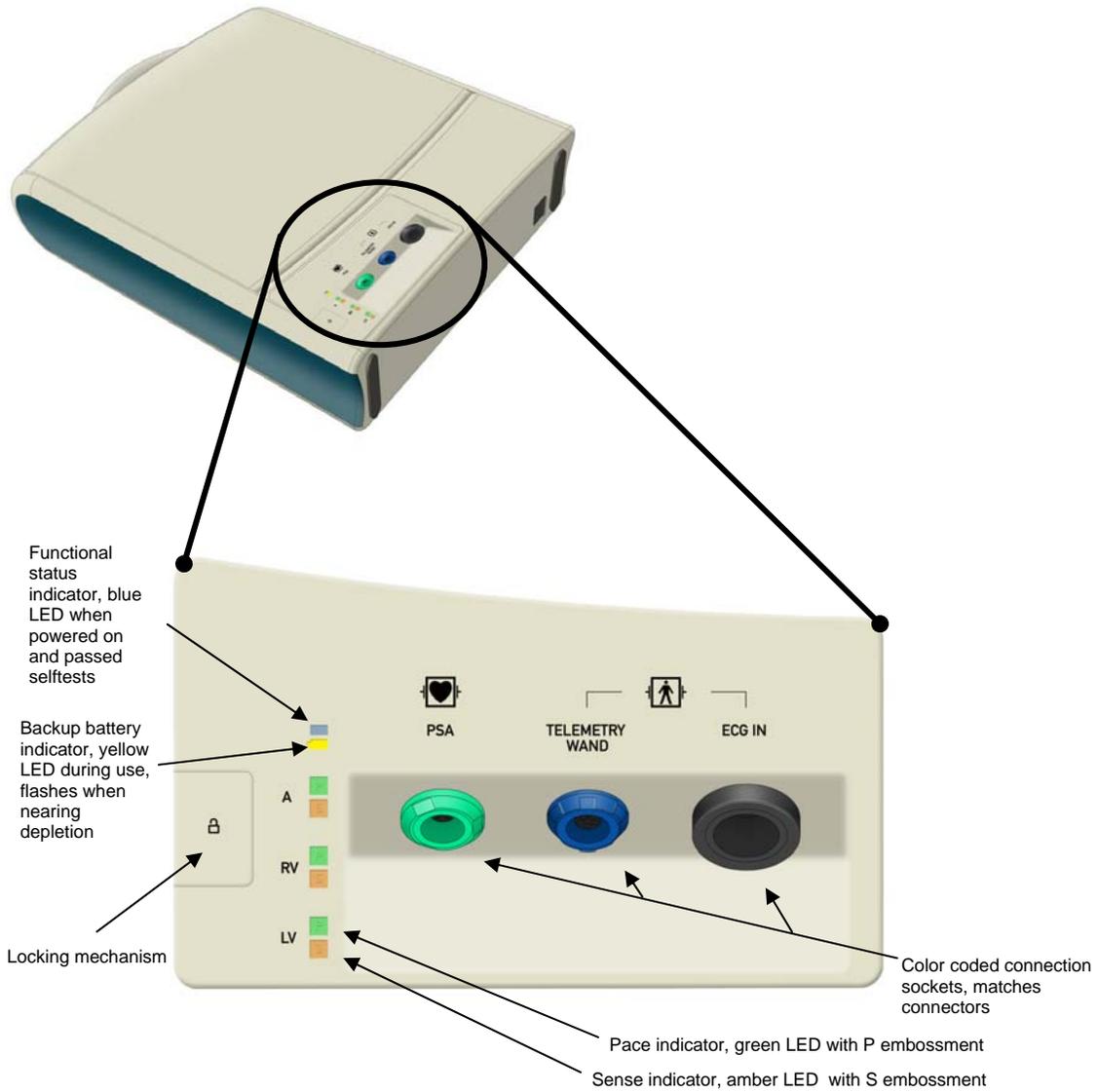
#### Pacing System Analyzer (PSA) Overview:

The main goal of the PSA device is to provide the clinicians enough data to ensure the pacemaker or ICD and the leads are functioning properly and positioned in a suitable cardiac location. First, the leads are inserted into the patient's heart through veins and are anchored to the heart chamber wall. Next, the leads are connected to the PSA device. The PSA device then performs a list of tasks including analyzing the amplitude of a cardiac signal (P and R waves), analyzing the impedance of the leads (ensuring the lead is not damaged and the impedance falls within a specified range), and analyzing the capture threshold data. After the leads have passed through a series of

tests, the doctors and clinicians deem it suitable as a successful lead implant. The leads are then securely connected to the pacemaker or ICD and tested once again through similar algorithms within the device. A key feature of the PSA device is that it will be integrated alongside the St. Jude Medical Merlin programmer device for programming pacemakers and ICDs.

Figure 1 displays the St Jude Medical Merlin Programmer which is used by field clinicians and doctors for programming the various parameters of an ICD or pacemaker. The new PSA device has been added to the programmer in the lower left hand section of the device. The device is powered via Universal Serial Bus (USB) from the programmer. Some of the features include: PSA socket (where leads are inserted for analysis), Telemetry Wand socket (allows the PSA device to communicate with the ICD or pacemaker via inductive telemetry), and ECG In socket (monitors patient's heart activity and is displayed on the Merlin programmer screen). There are also various light emitting diodes (LEDs) on the left hand side of the device for quick and easy notifications. The blue LED is the power indicator and notifies the user that the PSA device is being powered and is ready for use. The yellow LED indicator notifies the user when PSA device has lost its power from the USB connection and is running on the backup battery (9V battery installed inside the PSA device). This yellow LED will flash when the 9V battery is nearing depletion. The physical locking mechanism on the left allows users to check the USB connection or replace the backup 9 Volt battery attached inside the PSA unit. The 6 LEDs to the left of the locking mechanism (green and amber) are pacing and sensing LEDs. Each LED will notify the user if a pace or sense event is detected and in which detection area. "A" denotes the right atrium chamber of the heart. "RV" denotes

the right ventricular chamber of the heart. “LV” denotes the left ventricular chamber of the heart.



**Figure 1. Merlin Programmer and PSA Device**

## Data Transfer Protocol to Improve Speed

There have been major improvements on the PSA device when compared to legacy St Jude medical devices. Performance of the device was of the highest concern among the developers. The PSA device utilizes the Universal Serial Bus (USB) protocol to transfer data between the Merlin programmer and the PSA device. This communication protocol allows the PSA device to administer tests and analyze lead data at a much faster rate. In turn, it will help St. Jude Medical field representatives, doctors, and clinicians get the data they need from each of the implantable leads before attaching to a pacemaker or ICD. The different pacing tests that are performed on the leads and devices are performed faster and with higher accuracy and precision.

## OBJECTIVE

### Software Testing Process

The main objective of software libraries is to assist verification engineers to be able to communicate, interact, and gather information from the device quickly and efficiently to allow the focus to be concentrated on test designs which find software issues. An example test case scenario will be outlined:

### Test Design

Test engineers will create new test cases by analyzing system requirements (Figure 2) and creating a test design document which will give a high-level overview of how the test will be executed and the procedure the test will follow (Figure 3). The engineers must consider the input parameters to the device and must also list out the expected results in the form of various test points from within the test case. The test engineer must ensure that each requirement is fully tested and that all positive and negative test scenarios are covered as well.

Req Tag	Feature	Type	Requirement Text
3426	Pace	Pos/Neg	When an Atrial Pace Pulse is requested the Main CPU software shall turn the Atrial Paced Activity LED ON for 100 ms.

**Figure 2. Requirement Coverage in a test design document**

<p style="text-align: center;"><b>Procedure</b></p> <pre>//Atrial only For 5 cycles   Wait for 1 A   Verify Atrial Paced Activity LED is ON for 100ms   Verify Primary and Secondary Ventricular Paced Activity LEDs are OFF</pre>
--

**Figure 3. Test Design Procedure**

## Test Cases and Test Scripts

The standard for testing is for test engineers to create a C++ (cpp) file which contains a sequential series of software library function calls. Each test case is broken up into different sections and must follow the test design procedure. A new specialized C++ programming library will be created for use by the V&V test engineers who will be designing and running tests for the new PSA device. The class will follow the structure and behavior of previous class libraries created for other projects. Test engineers will utilize UTL library calls to execute test scenarios. The function call “gPSA.VerifyLEDStatus(...)” in Figure 4 is an example of a new UTL library function call created for the PSA project being used in a test script. Also note that the logic created in the cpp file matches the sequential scenario listed in the test design. Each test case is created in this manner and library function calls are used when necessary to retrieve/send data to the device.

```
459     // Verify the test results
460
461     gLog.WriteMessage("Atrial pacing only");
462     //Verify Atrial LED for 5 cycles
463     for(int i =0; i < 5; i++)
464     {
465         gESS.WaitForEvent("A", 1, 2);
466         gPSA.VerifyLEDStatus(RA_PACE_LED, LED_ON);
467         gPSA.VerifyLEDStatus(RV_PACE_LED, LED_OFF);
468         gPSA.VerifyLEDStatus(LV_PACE_LED, LED_OFF);
469
470         //Verify LED still ON
471         Sleep(LEDWaitTime - PROCESSING_DELAY);
```

**Figure 4. Example C++ code snippet for test cases**

## Test Results

Each test script is compiled and is executed on a UTS cart station. After each test

script has finished, a trace file is generated which contains a list of messages indicating which test points have passed or failed. The trace file will also list any errors reported by the testing libraries, internal tools, or by the Windows operating system (Figure 5).

285846	Atrial pacing only
285846	Waiting for 1 A's to occur in 2.000 seconds:
286412	
286413	CEventSequence::WaitForEvent() detected 1 of 1 A pulses.
286421	LED Data: 82
286421	PASS #242: Verify LED Data: RA PACE LED is set to 1 Expected: Equal To 1, Actual: 1, Tol: 0, Units: n/a
286429	LED Data: 82
286429	PASS #243: Verify LED Data: RV PACE LED is set to 0 Expected: Equal To 0, Actual: 0, Tol: 0, Units: n/a
286434	LED Data: 82

**Figure 5. Example trace file output from test case**

## METHODS AND MATERIALS

### Software Libraries

Software libraries are a set of functions that are organized into functional subcategories. The purpose of each function is to provide the user a way to interface with a given device or tool. It is most useful in cases where the device can only return a large data set of information. The user is only interested in a certain portion of the data being returned and would have to constantly analyze only the subset of information that is needed. Software libraries are a way of reducing programming code redundancy to help promote shared usage by all test engineers.

### Updating Software Libraries

Due to the large number of software requirements for each device, it is impossible to create a separate set of requirements that will satisfy the needs of software testing. It is for this reason that there is a different approach when creating software libraries for medical device testing. New requirements and new features to the device are constantly being added. Over time the need arises for new functionality within the software libraries. The process for creating or updating a software library function is as follows:

- Software Work Request (SWR) driven: A test engineer finds an issue in their test and it is related to a coding error in the UTL or internal tool

- Enhancements and Continuous Improvement: UTL team members or test engineers suggest an enhancement or request a feature that make functions more efficient, easier to use, low impact to other tests, limit code redundancy

### UTL Update Process

After a request is made to the UTL team for updates, it is assigned to a UTL team member for investigation. The UTL engineer will determine the cost and impact of the work. Priority is given on a project basis, but is not limited to project deadlines. Changes to the UTL can occur after a project has finished if a new method or a more efficient method is accepted by the verification team. Each change or modification work is then tracked in an SWR database where all affected groups are informed of the changes. Simple tests are run to ensure its functionality and another UTL team member verifies that the code follows coding standards. Once the SWR has been verified, the function is now available for the rest of the team to use in their test scripts.

### UTL Tools

The main tool that the UTL team members use to create the verification libraries is Microsoft Visual Studio 2005. From within this software we can use the built in compilers, development environment, and debugging tools to successfully create and test new functionality. As new features and requests are developed, changes are made to specific dynamically linked libraries (DLLs) and managed via an internal database system to track updates and changes.

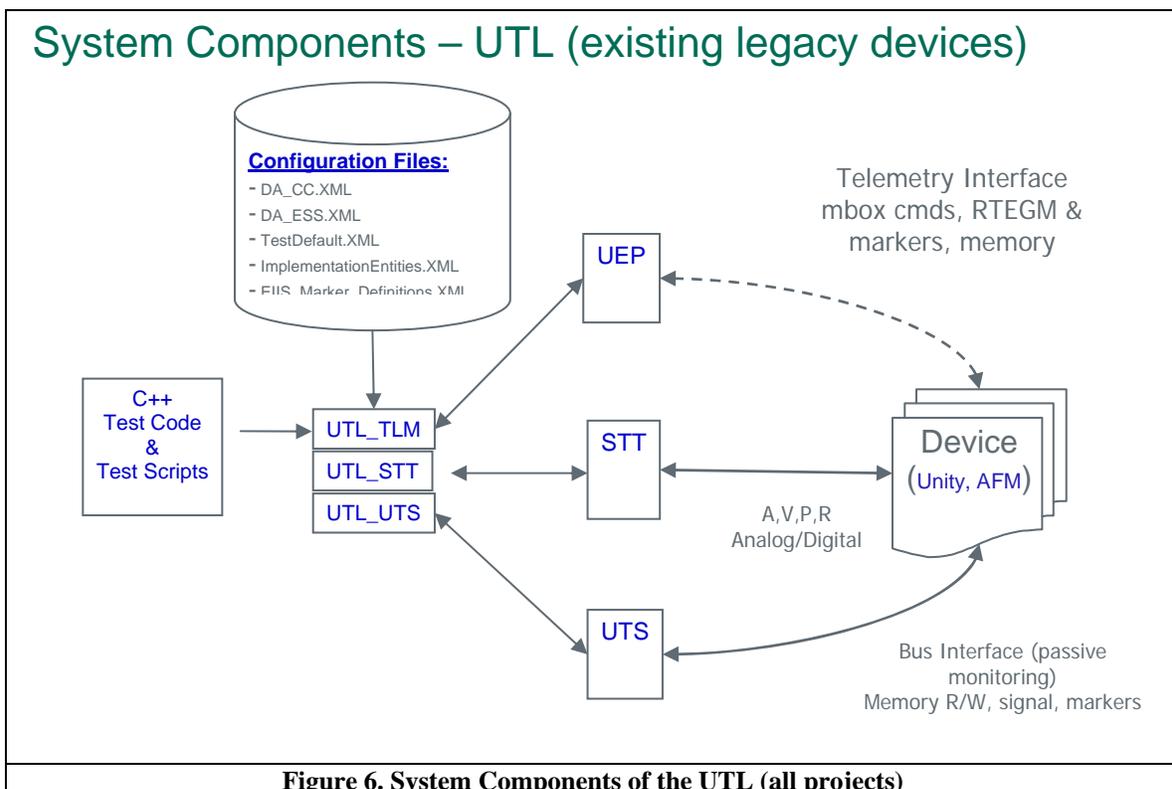
## UTL Functional Subcategories

The base UTL structure is divided into 3 main components: UTL\_TLM, UTL\_STT, and UTL\_UTS. Each component is specific to a tool that is used by the V&V teams to communicate and interact with each device. The UTL\_TLM project is a set of cpp files responsible for interacting with the Universal Engineering Programmer (UEP) which has API calls for communicating to a device via telemetry, RF, and USB communication. This is where a majority of the UTL code resides and is the cornerstone for all projects within the UTL. The UTL\_STT project is a set of cpp files responsible for communicating to a Simulation Test Tool (STT) which is a hardware and software component responsible for simulating heart rhythms which are sent to the device. The UTL\_UTS project is a set of cpp files responsible for recording any events and triggers that occur across the main bus of the Central Processing Unit (CPU) on the device and monitors data for each signal (similar to a logic analyzer). The main focus will be on the UTL\_TLM entity because the UEP is the main component which communicates to the PSA device via the RF protocol.

## Existing UTL configuration

Figure 6 shows the relationship between the testing code, software libraries, tools, and devices. Prior to PSA device testing, legacy devices would utilize the current structure of the UTL software libraries. The dotted line between the UEP and the device represents a non-wired communication protocol such as inductive telemetry or radio frequency (RF). The STT and UTS tools are hard wired to the device and allow digital/analog heart simulations to be sent to the device (STT) or device CPU activity to

be monitored through a standard bus interface (UTS). Each test script, along with configuration database files, becomes inputs to the UTL software libraries. When a test script is executed, the UTL will execute each function requested by the test script. Specific commands or parameter inputs can be sent to the device in the UTL\_TLM layer while requested events such as a paced event can be sent to the device in the UTL\_STT layer. The software libraries create an advantage for the end user by hiding the necessary programming code needed to interact between the tool and the device. A central software library allows for ease of code maintenance and, more importantly, leads to a shared standardized set of functions for all device interaction.

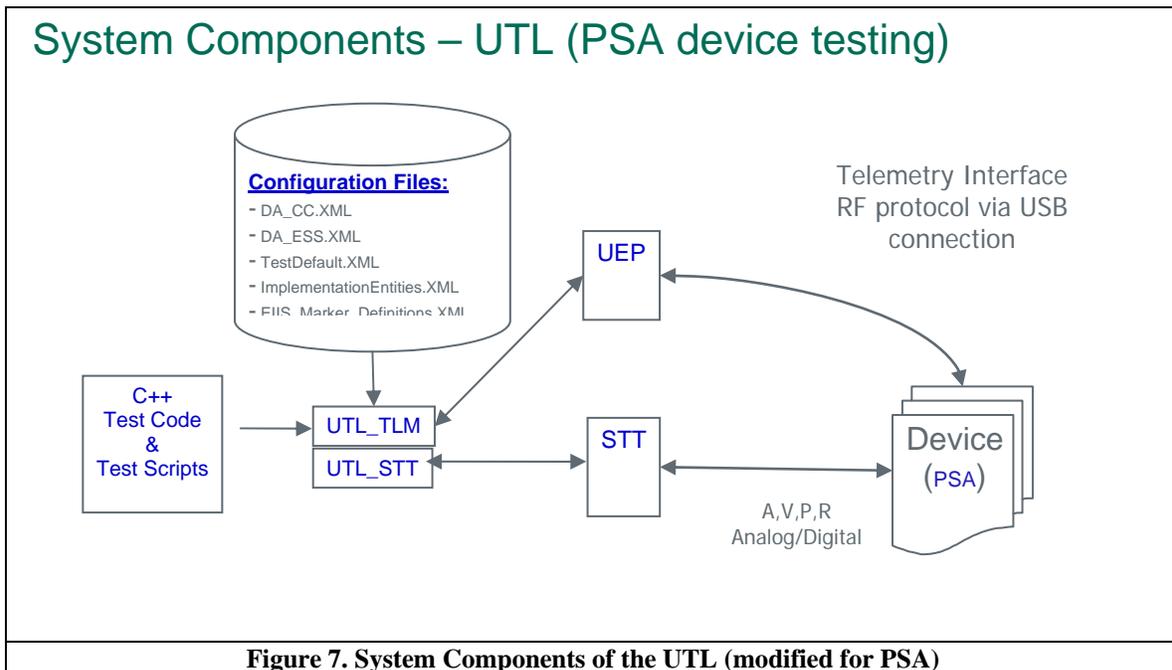


**Figure 6. System Components of the UTL (all projects)**

Current UTL configuration for PSA device testing

In contrast, Figure 7 shows how the UTL has been updated and modified for PSA device testing. The PSA device relies heavily on RF protocol to communicate to any

external device however it does not use RF frequencies and signals. The RF protocol is translated along USB hard-wired communication to the Merlin Programmer. The RF protocol, proxies, and APIs are used to communicate to the PSA device and it is the only method of command input to the device. The UTL\_TLM project has been updated with a PSA C++ class file and header file for all activities associated with the device. The STT communicates via the same method as legacy devices so no changes were needed to the UTL\_STT project. The PSA processor does not include an interface for a UTS or logic analyzer connection so this entity has been omitted and cannot be used for PSA device testing. Similar to the configuration in Figure 6, the software libraries shown in Figure 7 also create an advantage for the end user by hiding interaction between the tools and device.



## SOFTWARE LIBRARY METHODS AND UPDATES FOR PSA

In order to create appropriate APIs and functions for the test development team a fine understanding of what needs to be tested is important. This section will go in-depth through 5 new functions added to the UTL software libraries specifically for the PSA project. The functions discussed include GetHWPacedPulseInfo, VerifyLEDStatus, GetDeviceInfo, InitPStim, and DumpTraceToFile.

### New PSA functions added in the UTL

This section will discuss 5 new methods added specifically for the testing needs of the PSA project and how they have increased support for the verification team. (See Appendix A for a list of all the new functions that have been added to the UTL for the PSA project. Functions highlighted in **BOLD** have been discussed in detail).

### Example 1: Get Device Info

Figure 8 shows a simple example of a new method added to the UTL software libraries for the PSA project. The Get Device Info function is utilized by the UTL and test engineers to grab important setup information from the device. This data includes PSA model number, serial number, hardware version, software version, and schema version (external instrument specification). Each initialization of a test script will automatically send the Get Device Info command and store all the current information. After sending the correct telemetry command, the UEP will return with the data encoded in 11 bytes which need to be parsed correctly. It is important to know that the data being

returned is in big-endian format and the bytes will need to be swapped. The serial number is a clear example of this being 4 bytes long and byte order has been reversed.

This data can be useful when testing new software versions including updates, patches, and code stitching. The previous data can be used to compare that the version numbers have changed and updated accordingly. The function will also save all the current data into a deviceData object for any additional testing. This data is common across all tests and can be used for multiple verification points.

```
1.  /// Get PSA device info using Device Status Mailbox command
2.  /// \param deviceData  device info object that will hold all the
3.  ///                    data returned by the PSA device info command
4.  /// \retval PASS      DeviceInfo command completed successfully
5.  /// \retval FAIL     DeviceInfo command failed
6.  int CPSA::GetDeviceInfo(DEVICE_INFO_DATA& deviceData)
7.  {
8.      COutput::Instance().LogCmdMessage("CPSA::GetDeviceInfo()",PUBLIC_FUNC);
9.      int result = FAIL;
10.     const int DeviceInfoCmdSize = 1;
11.     unsigned char DeviceInfoCmd[DeviceInfoCmdSize];
12.     DeviceInfoCmd[0] = PSA_DEVICE_INFO;
13.     unsigned char returnData[RET_DATA_MAX];
14.     int returnSize = 0;
15.     result = DeviceStatusMailbox(&DeviceInfoCmd[0], DeviceInfoCmdSize, returnData, &returnSize);
16.     if(PASS == result)
17.     {
18.         deviceData.cmdId = returnData[0];
19.         deviceData.modelNo = (((unsigned short)returnData[0x02])<<8) + returnData[0x01];
20.         deviceData.serialNo = (((unsigned int)returnData[0x06])<<24) + (((unsigned int)returnData[0x05])<<16) + (((unsigned int)returnData[0x04])<<8) + returnData[0x03];
21.         deviceData.hardwareVersion = returnData[0x07];
22.         deviceData.majorSWVer = returnData[0x08];
23.         deviceData.minorSWVer = returnData[0x09];
24.         deviceData.buildSWVer = returnData[0x0a];
25.         sprintf(deviceData.softwareVersion, "0x%02X%02X%02X", deviceData.majorSWVer, deviceData.minorSWVer, deviceData.buildSWVer);
26.         deviceData.schemaVersion = returnData[0x0f];
27.
28.         // Log device info
29.         COutput::Instance().WriteMessage("Retrieve PSA Device Info successful!\n");
30.         COutput::Instance().WriteMessage("The PSA Device Information:");
31.
32.         sprintf(m_msg, " Device Info cmd ID = %02X", deviceData.cmdId);
33.         COutput::Instance().WriteMessage(m_msg);
34.
35.         sprintf(m_msg, " Model Number = %d", deviceData.modelNo);
```

```

36.     COutput::Instance().WriteMessage(m_msg);
37.
38.     sprintf(m_msg, " Serial Number = %d", deviceData.serialNo);
39.     COutput::Instance().WriteMessage(m_msg);
40.
41.     sprintf(m_msg, " Hardware Version = %Xh", deviceData.hardwareVersion);
42.
43.     COutput::Instance().WriteMessage(m_msg);
44.     sprintf(m_msg, " Software Version = %s", deviceData.softwareVersion);
45.     COutput::Instance().WriteMessage(m_msg);
46.
47.     sprintf(m_msg, " Schema Version = %02Xh \n", deviceData.schemaVersion);
48.
49.     COutput::Instance().WriteMessage(m_msg);
50.
51.     return PASS;
52. }
53. else
54. {
55.     sprintf(m_msg, "Failed to Retrieve PSA Device Info");
56.     COutput::Instance().WriteSystemError(m_msg);
57. }
58. return FAIL;
59. }

```

**Figure 8. GetDeviceInfo function**

## Example 2: Hardware Paced Pulse Information

Figure 9 shows an example of a new function called `GetHWpacedPulseInfo` that was added to the `UTL_TLM` to allow testers to access the pacing pulse information from the device. To properly access the pacing information, a Test Remote Function call is needed to be sent to the device to query the hardware registers and return the corresponding values. This function replaces internal logic about the behavior of the UEP data protocol and saves about 50+ lines of code in the test script. The function is also responsible for converting parameters for the Test Remote Function (from integer to ASCII per UEPAPI) in order to correctly send the Test Remote Function call. Once the function is called, it will return the user an object `OutputInfo` with the corresponding pacing information from the specified pacing chamber. The function will return with failure or success state that can be analyzed by the tester for further test script action. This function is important as the pacing values are constantly verified across all requirement testing.

```
1.  /// Get the hardware register values for Pace Pulse Output
2.  /// \param chamber - Requested chamber for Pace Pulse Info
3.  /// \param OutputInfo - Pace Pulse Amplitude and Width
4.  /// \retval PASS - successful read the HW register
5.  /// \retval FAIL - failed to read the HW register
6.  /// NOTE: This function may return a negative integer value if the FE link is //
   / down
7.  int CPSA::GetHWpacedPulseInfo(ePaceChamber chamber, PACE_PULSE_INFO& OutputInfo)
8.  {
9.      Coutput::Instance().LogCmdMessage("CPSA::GetHWpacedPulseInfo()",PUBLIC_FUNC)
   ;
10.     int result = FAIL;
11.     int paceAmp = 0;
12.     int paceWidth = 0;
13.     char buffer[MAX_BUFFER];
14.     int status = 0;
15.
16.     switch (chamber)
17.     {
18.         case PACE_RA:
19.             _itoa(PACE_AMP_RA, buffer, 16);
```

```

20.         result = Test_Remote_Function_Call(READ_FEPLD_REGISTER, buffer, pace
    Amp);
21.         _itoa(PACE_PW_RA, buffer, 16);
22.         result |= Test_Remote_Function_Call(READ_FEPLD_REGISTER, buffer, pac
    eWidth);
23.         break;
24.     case PACE_RV:
25.         _itoa(PACE_AMP_RV, buffer, 16);
26.         result = Test_Remote_Function_Call(READ_FEPLD_REGISTER, buffer, pace
    Amp);
27.         _itoa(PACE_PW_RV, buffer, 16);
28.         result |= Test_Remote_Function_Call(READ_FEPLD_REGISTER, buffer, pac
    eWidth);
29.         break;
30.     case PACE_LV:
31.         _itoa(PACE_AMP_LV, buffer, 16);
32.         result = Test_Remote_Function_Call(READ_FEPLD_REGISTER, buffer, pace
    Amp);
33.         _itoa(PACE_PW_LV, buffer, 16);
34.         result |= Test_Remote_Function_Call(READ_FEPLD_REGISTER, buffer, pac
    eWidth);
35.         break;
36.     default:
37.         COutput::Instance().WriteSystemError("Invalid pacing chamber");
38.         return FAIL;
39.     }
40.
41.     if(PASS == result)
42.     {
43.         paceAmp -= PACE_OFFSET;
44.         OutputInfo.amp = ((float)paceAmp * PACE_AMP_RES);
45.         OutputInfo.width = ((float)paceWidth * PACE_WIDTH_RES);
46.         sprintf(m_msg, "The %s Pace Pulse Amplitude is %f and width is %f", CHAM
    BER_STR[chamber], OutputInfo.amp, OutputInfo.width);
47.         COutput::Instance().WriteMessage(m_msg);
48.         return PASS;
49.     }
50.     else
51.     {
52.         sprintf(m_msg, "Unable to GetHWpacedPulseInfo");
53.         COutput::Instance().WriteSystemError(m_msg);
54.         return FAIL;
55.     }
56. }

```

**Figure 9. GetHWpacedPulseInfo function**

### Example 3: Initialize PStim

Figure 10 shows the most in-depth and complex function that is available for the PSA test engineers. The PStim function sends a command to the PSA device to send pacing stimuli through the leads and into the patient. This is used by the physician to check the validity and integrity of the leads implanted in the patient. Based on the different parameters given for the function (pacing chamber, pace amplitude, pace width, and pacing intervals) the function will appropriately create a specific command to send to the device. The function also provides bounds checking by qualifying the arguments passed to determine if the values are in range according to the EIIS (External Instrument Interface Specifications). For example, if a given amplitude is given outside the range of the programmable parameter range the function will return FAIL and will not send the PStim command to the device. If valid amplitude is given, the function will translate that value to step sizes in bytes in order to send the correct format of data for the command. Once all the parameters conditions are met, a byte arrays are constructed with the corresponding values and the complete command is sent to the device.

```
1.  /// Init PStim - The Initiate PStim command for PSA
2.  /// \param deliveryChamber -
    AI Delivery Chamber: PSA_PSTIM_RV OR PSA_PSTIM_ATRIUM
3.  /// \param stimuliAmp -
    Pulse Amplitude is the amplitude of the delivered stimuli
4.  /// \param stimuliWidth - Pulse Width is the width of the delivered stimuli
5.  /// \param S1S1Interval -
    specifies the interval between the delivery of S1 stimuli
6.  /// \param vSuppPacingInt -
    sepcified pacing interval; A value of '0.0' would indicated Ventricular Support
    Pacing is disabled
7.  /// \retval PASS    Successfully Initiated PStim
8.  /// \retval FAIL    Failed to Initiate PStim
9.  int CPSA::InitPStim(ePStimChamber deliveryChamber, float stimuliAmp, float stimu
    liWidth, float S1S1Interval, float vSuppPacingInt)
10. {
11.     COutput::Instance().LogCmdMessage("CPSA::InitPStim()",PUBLIC_FUNC);
12.     int result = FAIL;
13.     char msg[MESSAGE_LEN];
```

```

14.
15.     const int MAX_PSTIM_CMD_SIZE = 8;
16.     const int PSTIM_PULSE_AMP_OFFSET = 0x1C;
17.     const float PSTIM_AMP_WIDTH_RES = 0.05f;
18.     unsigned char initPstimCmd[MAX_PSTIM_CMD_SIZE];
19.     initPstimCmd[0] = PSA_INIT_PSTIM;
20.     unsigned char returnData[RET_DATA_MAX];
21.     int returnSize = 0;
22.     float fMin = 0.0, fMax = 0.0;
23.
24.     // PStim Stimuli Amplitude
25.     const string sPstimPaceStimAmp = "Programmed Stimulation Primary Pace Stimul
i Amplitude";
26.     float stepPstimAmpInterval = 0.0;
27.     if(PASS != EIISParser::Instance().GetDCPMinMaxRes(sPstimPaceStimAmp, fMin, f
Max, stepPstimAmpInterval))
28.     {
29.         sprintf(m_msg, "Failed to translate %s(%f) since range info couldn't be
retrieved. Init PStim command cannot be sent.", sPstimPaceStimAmp.c_str(), stimu
liAmp);
30.         COutput::Instance().WriteTestError(m_msg);
31.         return FAIL;
32.     }
33.     if((stimuliAmp < fMin) || (stimuliAmp > fMax) )
34.     {
35.         sprintf(m_msg, "%s out of range: (%f) Init PStim command cannot be sent.
", sPstimPaceStimAmp.c_str(), stimuliAmp);
36.         COutput::Instance().WriteTestError(m_msg);
37.         return FAIL;
38.     }
39.     BYTE PulseAmplitude = (BYTE) round(float(stimuliAmp/PSTIM_AMP_WIDTH_RES));
40.
41.     // PStim Stimuli Width
42.     const string sPstimPaceStimWidth = "Programmed Stimulation Primary Pace Stim
uli Width";
43.     float stepPstimWidthInterval = 0.0;
44.     if(PASS != EIISParser::Instance().GetDCPMinMaxRes(sPstimPaceStimWidth, fMin,
fMax, stepPstimWidthInterval))
45.     {
46.         sprintf(m_msg, "Failed to translate %s(%f) since range info couldn't be
retrieved. Init PStim command cannot be sent.", sPstimPaceStimWidth.c_str(), sti
muliWidth);
47.         COutput::Instance().WriteTestError(m_msg);
48.         return FAIL;
49.     }
50.     if((stimuliWidth < fMin) || (stimuliWidth > fMax) )
51.     {
52.         sprintf(m_msg, "%s out of range: (%f) Init PStim command cannot be sent.
", sPstimPaceStimWidth.c_str(), stimuliWidth);
53.         COutput::Instance().WriteTestError(m_msg);
54.         return FAIL;
55.     }
56.     BYTE PulseWidth = (BYTE) round(float(stimuliWidth/PSTIM_AMP_WIDTH_RES));
57.
58.     // PStim S1S1 Interval
59.     const string sStimS1S1Int = "Programmed Stimulation S1S1 Interval";
60.     float stepS1S1Interval = 0.0;
61.     if(PASS != EIISParser::Instance().GetDCPMinMaxRes(sStimS1S1Int, fMin, fMax,
stepS1S1Interval))
62.     {

```

```

63.     sprintf(m_msg, "Failed to translate %s(%f) since range info couldn't be
        retrieved. Init PStim command cannot be sent.", sStimS1S1Int.c_str(), S1S1Interval);
64.     COutput::Instance().WriteTestError(m_msg);
65.     return FAIL;
66. }
67. if((S1S1Interval < fMin) || (S1S1Interval > fMax) )
68. {
69.     sprintf(m_msg, "%s out of range: (%f) Init PStim command cannot be sent.
        ", sStimS1S1Int.c_str(), S1S1Interval);
70.     COutput::Instance().WriteTestError(m_msg);
71.     return FAIL;
72. }
73. unsigned int StimS1S1Interval = (unsigned int) round(float(S1S1Interval/step
        S1S1Interval));
74.
75. // PStim Ventricular Support Pacing Interval
76. const string sVSuppPaceInt = "Programmed Stimulation Ventricular Support Pac
        ing Interval";
77. float stepSuppPaceInterval = 0.0;
78. if(PASS != EIISParser::Instance().GetDCPMinMaxRes(sVSuppPaceInt, fMin, fMax,
        stepSuppPaceInterval))
79. {
80.     sprintf(m_msg, "Failed to translate %s(%f) since range info couldn't be
        retrieved. Init PStim command cannot be sent.", sVSuppPaceInt.c_str(), vSuppPaci
        ngInt);
81.     COutput::Instance().WriteTestError(m_msg);
82.     return FAIL;
83. }
84.
85. unsigned int VentSuppPaceInterval = (unsigned int) round(float(vSuppPacingIn
        t/stepSuppPaceInterval));
86.
87. initPStimCmd[1] = (BYTE)deliveryChamber; // Delivery Chamber
88. initPStimCmd[2] = PulseAmplitude + PSTIM_PULSE_AMP_OFFSET; // Pulse
        Amplitude
89. initPStimCmd[3] = PulseWidth; // Pulse Width
90. initPStimCmd[4] = StimS1S1Interval; // S1S1 Interval (little end
        ian)
91. initPStimCmd[5] = (StimS1S1Interval>>8);
92. initPStimCmd[6] = VentSuppPaceInterval; // Ventricular Support Pacin
        g Interval (little endian)
93. initPStimCmd[7] = (VentSuppPaceInterval>>8);
94.
95. result = DeviceStatusMailbox(&initPStimCmd[0], MAX_PSTIM_CMD_SIZE, returnDat
        a, &returnSize);
96. if(PASS == result)
97. {
98.     sprintf(msg, "Successfully sent Init PStim");
99.     COutput::Instance().WriteMessage(msg);
100.    return PASS;
101. }
102. else
103. {
104.     sprintf(msg, "Failed to send Init PStim");
105.     COutput::Instance().WriteSystemError(msg);
106. }
107. return FAIL;
108. }

```

**Figure 10. InitPStim function**

#### Example 4: Verify LED Status

Figure 11 shows an example of a function which does multi-bit parsing for PSA test engineers. The VerifyLEDStatus function is used when there are multiple LEDs that indicate pacing, sensing, backup battery power and status. This function is used by the testers to determine at a certain test point which of the LEDs is currently ON and which is OFF. There are 8 different LEDs on the PSA device and a matching enumeration structure has been created for each. The user can pass in which LED to verify and which state is expected. The function will send a command to the PSA device to poll the LED, parse out the corresponding bit from the returned data, and print a verification statement to the trace file with a passing or failing result.

```
1.  /// Verify LED Status
2.  /// This method will be used to verify the LED status
3.  /// \param name - name of LED to verify (i.e. POWER_LED, LO_BATT_LED..)
4.  /// \param state - expected state of the specified LED (i.e. LED_ON/LED_OFF)
5.  void CPSA::VerifyLEDStatus(eLEDName name, eLEDState expState)
6.  {
7.      COutput::Instance().LogCmdMessage("CPSA::VerifyLEDStatus()",PUBLIC_FUNC);
8.      int result = FAIL;
9.      BYTE LEDData = 0;
10.     BYTE tempLEDData = 0;
11.     result = ReadLEDRegister(LEDData);
12.
13.     if(result != PASS)
14.     {
15.         COutput::Instance().WriteSystemError("Unable to Read LED data!");
16.         return;
17.     }
18.
19.     switch(name)
20.     {
21.         case POWER_LED:
22.             tempLEDData = (LEDData & POWER_LED_MASK) >> 7;
23.             break;
24.         case LV_SENS_LED:
25.             tempLEDData = (LEDData & LV_SENS_LED_MASK) >> 6;
26.             break;
27.         case LV_PACE_LED:
28.             tempLEDData = (LEDData & LV_PACE_LED_MASK) >> 5;
29.             break;
30.         case RV_SENS_LED:
31.             tempLEDData = (LEDData & RV_SENS_LED_MASK) >> 4;
32.             break;
```

```

33.     case RV_PACE_LED:
34.         tempLEDData = (LEDData & RV_PACE_LED_MASK) >> 3;
35.         break;
36.     case RA_SENS_LED:
37.         tempLEDData = (LEDData & RA_SENS_LED_MASK) >> 2;
38.         break;
39.     case RA_PACE_LED:
40.         tempLEDData = (LEDData & RA_PACE_LED_MASK) >> 1;
41.         break;
42.     case LO_BATT_LED:
43.         tempLEDData = (LEDData & LO_BATT_LED_MASK);
44.         break;
45.     default:
46.         COutput::Instance().WriteSystemError("LED name not supported.");
47.         return;
48.     }
49.
50.     sprintf(m_msg, "Verify LED Data: %s is set to %i", PSA_LED_STR[name], expState);
51.     COutput::Instance().VerifyTestResult<int>(m_msg, (int)expState, (int)tempLEDData, ZERO_TOL, "n/a", EQUAL);
52.     return;
53. }
54.

```

**Figure 11. VerifyLEDStatus function**

## Example 5: Dump Trace File

Figure 12 shows the DumpTraceToFile function which logs and displays any trace dump messages from the PSA device. This function was added as an additional debugging method. The firmware that is running the PSA device has added code logic to output messages to a trace buffer internally in the system. This buffer will internally hold a list of any errors that have occurred in the device. At any point the user can use this API function call to dump or empty the message buffer to a verification trace file. Test designers can utilize this function to dump any errors to the trace if any previous PSA function call has failed.

The PSA dump trace in the device is a buffer in which each entry is only 1024 bytes (or characters) long. Therefore, any error message that is generated by the PSA device that is placed in the buffer that is larger than 1024 characters will need to be broken up into several buffer entries. The PSA device will return the buffer size and the function will create a new local buffer with the size of the error messages. To ensure each dump trace is saved correctly, the function will use the test logging name path (which is unique to each test run) as the name of the file of the dump trace.

```
1.  /// This function will get the current Trace Dump Message and put its contents i
   n a text file
2.  /// \retval PASS      successfully retrieved Trace Dump Message -
   output: *.PSAMemTrace file
3.  /// \retval FAIL     failed to retrieve Trace Dump Message
4.  int CPSA::DumpTraceToFile()
5.  {
6.  COutput::Instance().LogCmdMessage("CPSA::DumpTraceToFile()",PUBLIC_FUNC);
7.
8.      const int MESSAGE_SIZE = 1024;
9.      char tempMessage[MESSAGE_SIZE];
10.     int bufferSize = MESSAGE_SIZE;
11.     int result = FAIL;
12.     int len = 0;
13.     ofstream dumpFile;
14.     string temp = COutput::Instance().GetTestLogNamePlusPath();
15.     temp.append(".PSAMemTrace");
```

```

16.
17.     dumpFile.open(temp.c_str());
18.     if(dumpFile.rdstate() == ios::failbit)
19.     {
20.         sprintf(m_msg, "GetTraceDumpMessage not sucessful");
21.         COutput::Instance().WriteSystemError(m_msg);
22.         dumpFile.close();
23.         return FAIL;
24.     }
25.
26.     result = CUEP::Instance().GetTraceDump(tempMessage, &bufferSize);
27.
28.     if(PASS == result)
29.     {
30.         dumpFile << tempMessage;
31.         dumpFile.flush();
32.         dumpFile.close();
33.         sprintf(m_msg, "GetTraceDumpMessage successful: %s", temp.c_str());
34.         COutput::Instance().WriteMessage(m_msg);
35.         return PASS;
36.     }
37.     else if(result == UEP_TRACE_DUMP_RESULT) // need a larger buffer for trace d
ump
38.     {
39.
40.         char *largeMessage = new char[bufferSize];
41.         result = CUEP::Instance().GetTraceDump(largeMessage, &bufferSize);
42.         dumpFile << largeMessage;
43.         dumpFile.flush();
44.         delete[] largeMessage;
45.
46.         if(result == PASS)
47.         {
48.             sprintf(m_msg, "GetTraceDumpMessage successful: %s", temp.c_str());
49.
50.             COutput::Instance().WriteMessage(m_msg);
51.             dumpFile.flush();
52.             dumpFile.close();
53.             return PASS;
54.         }
55.
56.         sprintf(m_msg, "GetTraceDumpMessage not sucessful");
57.         COutput::Instance().WriteSystemError(m_msg);
58.         log_psa_uep_error(result);
59.         dumpFile.flush();
60.         dumpFile.close();
61.         return FAIL;
62.     }

```

**Figure 12. DumpTraceToFile function**

## DISCUSSION AND FUTURE IMPROVEMENTS

The functions created for the PSA project are just a few improvements and tools that the verification test team utilizes. Being the first project that has utilized a USB protocol, we can use the newly created PSA functions for future projects that will also communicate via the USB protocol. These updates allow the UTL to support any legacy testing for the PSA project and can support future projects that spawn from this PSA platform. If St Jude Medical decides to create similar devices that are based on this USB protocol, the software testing group already has the tools and methods in place and would not add any delay in project scheduling. This idea of shared or reusable code across projects has been helpful to firmware testing for a number of years and has been proven to cut costs, decrease project scope creep, and allow test engineers to help out development teams in all aspects of the development lifecycle.

Beyond firmware testing, the UTL team's main goal is to have this shared resource used across all sectors of the company. The development team has recently adopted the UTL libraries and packages for creating unit and bench tests (where previously firmware tests were created and maintained by a separate tool). A lot of the teams across different sites communicate and interact with the same tools (STT, UEP, etc). Having a generic unified interface for all these tools may help other teams in the way they think about requirements, designs, implementation, and more importantly, testing. The success that the UTL has brought to the Firmware Verification team as a resource is something that should be spread across the company to give St Jude added value against market competitors.

## CONCLUDING REMARKS

The UTL team will continue to maintain and deliver improvements to existing UTL functionality. Because of its power and simplicity of use, the UTL will be used for many projects to come. Continuous improvement for the UTL is important because software testing is an invaluable step in software development. Software testing is the most important aspect of the development lifecycle to help keep maintenance costs at a minimum. Although most companies view software testing as a burden, the software organizations at a medical device company views testing as an essential ingredient for quality. Improvements and enhancements to the UTL have helped St Jude Medical streamline the testing process, shorten project deadlines and milestones, and minimize costs for projects in the upcoming years.

## ACRONYMS AND DEFINITIONS

UEP – Universal Engineering Programmer

V&V – Verification and Validation

UTL – Unified Test Library (Team)

API – Application Programming Interface

PSA – Pacing System Analyzer

USB – Universal Serial Bus

LED – Light Emitting Diode

RF – Radio Frequency

XML – Extensible Markup Language

TLM – Telemetry Module

STT – Simulation Test Tool

UTS – Unified Test System

EIIS – External Instrument Interface Specification

ASCII - American Standard Code for Information Interchange

## APPENDIX A

Description of API Call	Function Name
Initialize PSA device and unlock Test protocol	Init( )
LaunchApp	LaunchApp(AppName applicationName)
Launch BootLoader Application	LaunchBootloader();
Open PSA Communication channel	Open(bool ignoreUEPError = FALSE);
Close PSA Communication channel	Close();
Get Production Parameters	GetProductionParams(PRODUCTION_PARAMS& production_params);
<b>Get the hardware register values for Atrial/Primary Ventricular/Secondary Ventricular Pace Pulse Output</b>	<b>GetHWPacedPulseInfo(ePaceChamber chamber, PACE_PULSE_INFO&amp; OutputInfo);</b>
Get the impedance values from hardware	GetHWImpedanceValue(eRegProtocol impedanceRegister, int& impedanceVal);
Return the current Trace Debug Level	GetTraceDebugLevel(int& debugInfo);
Return the current Trace Info Level	GetTraceInfoLevel(int& traceInfo);
<b>Dump Trace to File</b>	<b>DumpTraceToFile();</b>
Decode a given trace level	GetPSASubsystemTraceLevel(PSASubSystem nPSASubSystem, int nTraceLevel, bool bEnable);
Check if CPSA has been initialized	IsInitialized(void);
Override IEGM command	OverrideIEGM(int channel, const short* EGMDData, int numberOfSamples);
Reset/Connect the PSA	ResetSlaveDevice(int timeOut = 20);
Reboot PSA Device	Reboot();
Exit PSA and release resources	ShutDown();
Unlock Test Protocol	UnlockTestProtocol();
Lock Test Protocol	LockTestProtocol();
Get the boot status from the device	GetBootStatus(bool ignoreSWERR = FALSE);
Device Status Mailbox Command	DeviceStatusMailbox(const unsigned char *cmdData, const int cmdLen, unsigned char *resultData, int *resultLen, bool ignoreUEPError = FALSE);
Get Wand Status	GetWandStatus();
Test Remote Function	Test_Remote_Function_Call(const char* cmd, const char* args, int& response);
Read LED register data (test remote function call)	ReadLEDRegister(BYTE &regData);
Force SWERR (test remote function call)	ForceSWERR();
Get Bit Status (clear SWERR)	GetBitStatus(EPBITStatus &status, bool ignoreUEPError = FALSE);
Toggle USB Power	ToggleUSBPower(eUSB_POWER powerUSB);
Toggle Battery Power	ToggleBattPower(eBATT_POWER powerBatt);
<b>Verify LED Status</b>	<b>VerifyLEDStatus(eLEDName name, eLEDState expState);</b>
Flash Slave Device Code or Download PSA FW	DownloadFW(char* strFileName);
Initialize Slave Device	InitSlaveDevice(EPDeviceType deviceType = epPSA, EPTelemType telemType = epPSATIm);
Launch Bootable File	LaunchBootableFileByStartAddress(int &status, int address = 0);
Program Param File	ProgramParamFile(EPPParamFile* paramFile, int &status, int checkSum = -1);
Get Param File	GetParamFile(int segID, EPPParamFile &paramFile);
InitRTEGM for streaming of markers	InitRTEGM();
<b>Get device info using Device Status Mailbox command</b>	<b>GetDeviceInfo(DEVICE_INFO_DATA&amp; deviceData);</b>
Get PSA Status	GetPSAStatus(unsigned char* response, int retSize);
Send Read Parameter Command	PSAReadParameter(unsigned char* paramSet);
Send Program Parameter Set	ProgramParamSet(unsigned char* paramSet);
<b>Init PStim</b>	<b>InitPStim(ePStimChamber deliveryChamber, float stimuliAmp, float stimuliWidth, float S1S1Interval, float vSuppPacingInt);</b>
Terminate PStim	TerminatePStim();

## REFERENCES

1. Kuhn, Richard D. and Reilly, Michael J. “An Investigation of the Applicability of Design of Experiments to Software Testing” National Institute of Standards and Technology. Gaithersburg, MD 20899. 2003. IEEE Computer Society.
2. Lindkvist, Leif. “Nemo PSA device” Clinical Systems Engineering Biweekly training slides. February 2009. St. Jude Medical.
3. Koenig, Steven C. et al. “Integrated Data Acquisition System for Medical Device Testing and Physiology Research in Compliance with Good Laboratory Practices” April 2003. Jewish Hospital Cardiothoracic Surgical Research Institute. University of Louisville.
4. Lindkvist, Leif. “Nemo PSA Introduction” Clinical Systems Engineering Powerpoint Presentation. March 2008.
5. Zhang, Jiajie et al. “Using usability heuristics to evaluate patient safety of medical devices” Journal of Biomedical Informatics 36(2003) 23 – 30. 2003 Elsevier Inc.
6. Kaner, Cem et al. “Testing Computer Software” 2<sup>nd</sup> edition. Wiley Computer Publishing. John Wiley and Sons Inc. 1999.
7. Bach, James et al. “Lessons Learned in Software Testing – A context driven approach” Wiley Computer Publishing. John Wiley and Sons Inc. 2002.
8. Myers, Glenford J. “The Art of Software Testing” 2<sup>nd</sup> edition. Wiley Computer Publishing. John Wiley and Sons Inc. 2004.
9. Patton, Ron. “Software Testing – Second Edition” Sams Publishing. 2006.