User Manual

DA1458x/DA1468x Production Line Tool Libraries

UM-B-040

Abstract

This document describes the DA1458x/DA1468x Production Line Tool source code libraries. The heart of the source code comes in the form of Windows Dynamic Link Libraries (DLLs). These DLLs give the necessary APIs for validating and programming DA14580/1/2/3, DA14585/6, DA14680/1/2/3 and DA15100/1 Bluetooth® low energy devices in the factory production line.

DA1458x/DA1468x Production Line Tool Libraries

Contents

Figures

Tables

1 Terms and Definitions

API	Application Programming Interface
BD	Bluetooth Device
.bin	Firmware files in binary format
BLE	Bluetooth low energy
CFG	Configuration
CLI	Command Line Interface
COM	Communication port
CPLD	Complex Programmable Logic Device
CSV	Comma Separated Values
DLL	Dynamic Link Library
DMA	Direct Memory Access
DMM	Digital Multi Meter
DTM	Direct Test Mode (as specified by the BLE Core standard)
DUT	Device Under Test
DVM	Digital Voltage Meter
EEPROM	Electrically Erasable Programmable Read-Only Memory
.exe	Executable file
FTDI	Future Technology Devices International Ltd.
GPIO	General Purpose Input-Output
GU	Golden Unit
GUI	Graphical User Interface
Hex	Firmware file in ASCII format
HW	hardware
IC	Integrated Circuit
IDE	Integrated Development Environment
I2C	Inter-Integrated Circuit
JTAG	Joint Test Action Group
OS	Operating System
OTP	One Time Programmable (memory)
PC	Personal Computer
PLTD	Production Line Tool DLL
RAM	Random Access Memory
RCX	Resistor Crystal Oscillator
RF	Radio Frequency
RX	Receive
SCPI	Standard Commands for Programmable Instruments
SoC	System on Chip
SDK	Software Development Kit
SPI	Serial Peripheral Interface
stdio	Standard Input Output
SW	Software
TX	Transmit
UART	Universal Asynchronous Receiver/Transmitter
UI	User Interface

USB	Universal Serial Bus
VISA	Virtual Instrument Software Architecture
VPP	Programming supply voltage (pin)
XML	Extensible Markup Language
XTAL	Crystal
XSD	XML Schema Definition

2 References

[1] UM-B-041, Production Line Tool Hardware and GUI, User manual, Dialog Semiconductor.

[2] UM-B-014, DA1458x Bluetooth Smart Development Kit – Expert, Dialog Semiconductor.

[3] UM-B-010, DA14580 Proximity application, User manual, Dialog Semiconductor.

[4] FT4232H – Hi-Speed Quad USB UART IC, FTDI Chip.

[5] FT232 – USB UART IC, FTDI Chip

[6] AN-B-020, DA14580 End product testing and programming guidelines, Application note, Dialog Semiconductor.

[7] Anritsu MT8852B, https://www.anritsu.com/en-US/test-measurement/products/mt8852b

[8] Keysight 34401A, http://www.keysight.com/en/pd-1000001295%3Aepsg%3Apro-pn-34401A/digital-multimeter-6-digit?cc=US&lc=eng

[9] Keithley 2000, http://www.tek.com/tektronix-and-keithley-digital-multimeter/keithley-2000-series-6%C2%BD-digit-multimeter-scanning

[10] Papouch TMU USB thermometer, https://www.papouch.com/en/shop/product/tmu-usb-thermometer/

[11] NI USB TC-01, http://sine.ni.com/nips/cds/view/p/lang/en/nid/208177

[12] Honeywell Xenon 1900, https://www.honeywellaidc.com/products/barcode-scanners/general-duty/xenon-1900g-1902g

[13] Zebra/Motorola LS2208, https://www.zebra.com/us/en/products/scanners/general-purpose-scanners/handheld/ls2208.html

[14] UM-B-056, DA1468x Software Developer's Guide, User manual, Dialog Semiconductor.

[15] Litepoint IQXel-M, http://www.litepoint.com/test-solutions-for-manufacturing/iqxel-m/

[16] NI USB-6009 DAQ, http://sine.ni.com/nips/cds/view/p/lang/en/nid/201987

[17] Keysight 34461A, http://www.keysight.com/en/pd-2270273-pn-34461A/digital-multimeter-6-digit-34401a-replacement-truevolt-dmm?cc=GR&lc=eng

3 Introduction

The DA1458x/DA1468x Production Line Tool is a tool able to perform functional validation and memory programming of Dialog Semiconductor Bluetooth® low energy devices, during factory production. It supports devices that use DA14580, DA14581, DA14582, DA14583, DA14585, DA14586, DA1468, DA14680, DA14682, DA14683, DA15100 and DA15101 SoCs (also referred to as 'DUT' or simply 'device' in this document). The Production Line Tool (PLT) requires dedicated hardware [1] to be operational.

This guide describes the software part of the DA1458x/DA1468x Production Line Tool included in the DA1458x_DA14658x_PLT_v4.2 released package. Figure 1 shows the various components of the Production Line Tool. The diagram is split into two parts: Production Line Tool Software (top, this document) and Production Line Tool Hardware (bottom).

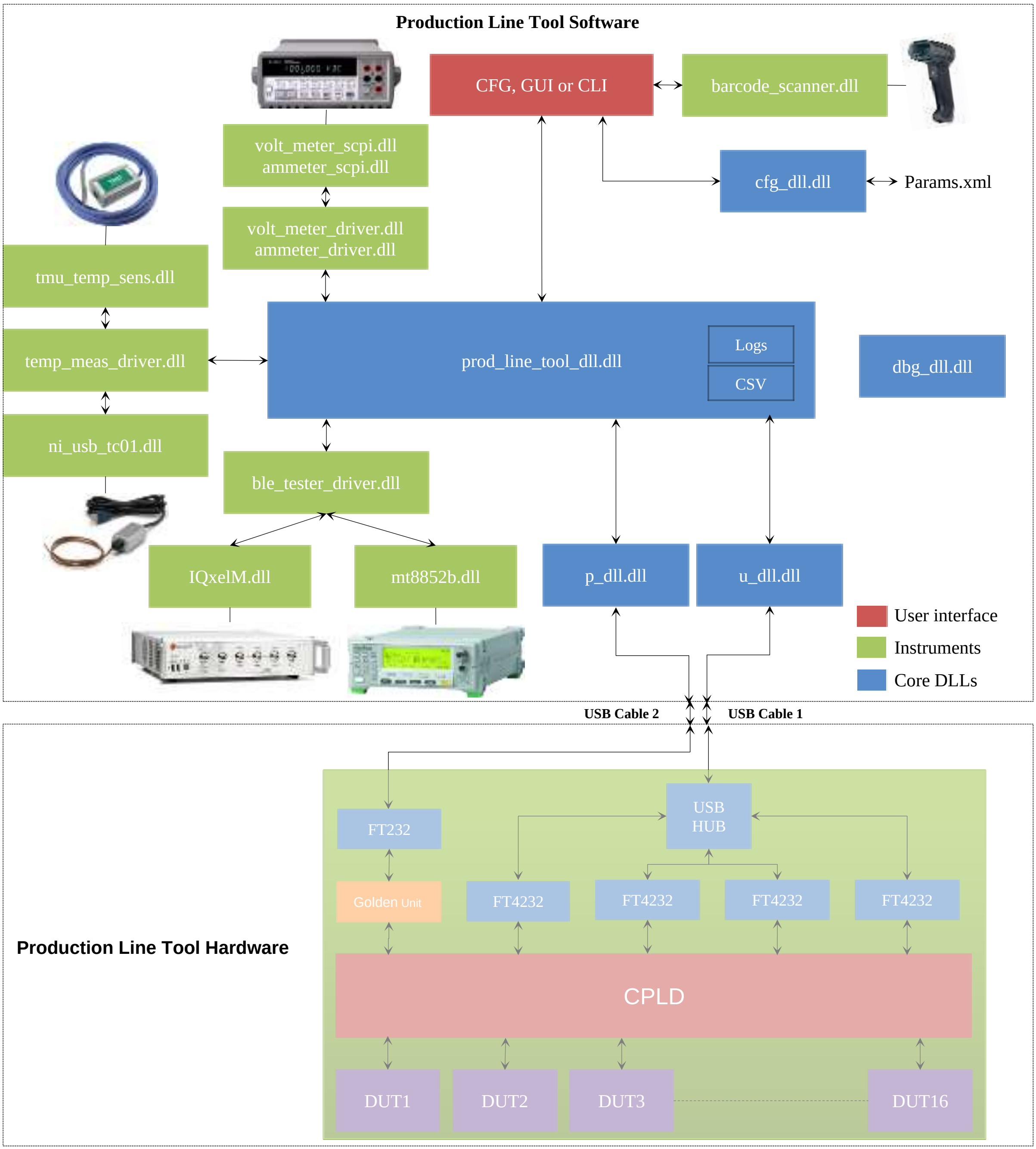


Figure 1: Production Line Tool Block Diagram

The DA1458x/DA1468x Production Line Tool runs on a Windows 7/8/8.1/10 PC. It communicates with the Production Line Tool hardware [1] through two USB cables. One of the USB cables is directly connected to the Golden Unit (GU) through an FT232 IC [5]. The Golden Unit is a factory validated and calibrated DA14580-QFN48 daughterboard device [2]. The other cable is connected, via a USB hub, to four USB-to-UART interface ICs [4]. The UART interfaces are connected, through the CPLD device, to the DUTs.

The Production Line Tool software consists of the software blocks described in Table 1.

Table 1: Production Line Tool Software Blocks

Library	Purpose	Description
prod_line_tool_dll.dll	Main test state machines	The `prod_line_tool_dll.dll` is the top level dynamic link library. It is the heart of the system. It implements basic state machines using any of the DLLs surround it to download the necessary firmware to the devices and run the desired tests. It keeps detailed logging per device tested, as well as a summary of the test results in a CSV formatted file.
u_dll.dll	Memory programming	The `u_dll.dll` is used to download firmware to the device's system RAM. It is also responsible for any other device's memory operation (write or erase SPI Flash, write EEPROM, write QSPI or OTP, etc.).
p_dll.dll	Functional/Peripheral tests	The `p_dll.dll` is used to set the DUTs in the various functional test modes, such as RF tests, XTAL trim calibration, sensor tests and other. The same DLL is used to issue specific commands to the Golden Unit (GU). With these specific commands the GU is able to control the CPLD on the Production Line Tool hardware [1]. Some of the CPLD commands include the control of the DUT power supplies, the UART connections, the XTAL trim calibration pulse generation, and other.
dbg_dll.dll	Debug prints	The `dbg_dll.dll` is a debug print DLL. It provides a simple API in order to print various levels of messages to a specific output (console or file). It is used by all the software blocks. Therefore, every software block has its own debug information configured and controlled separately. For example, the `p_dll.dll` debug print can be enabled and have debug prints send to the console. At the same time the `u_dll.dll` debug can be disabled and the `prod_line_tool_dll.dll` debug print can be enabled but show only the error prints to a file.
cfg_dll.dll	Configuration parameters	The `cfg_dll.dll` is the configuration parameter DLL. It provides a simple API to import, export and validate configuration settings from or to an XML file (`params.xml`). These parameters tell what tests will be performed (e.g. RF Test, XTAL trim test) and which memories will be burned (e.g. OTP, SPI or I2C EEPROM) They hold various other settings needed for a complete DUT validation. Every single parameter imported is validated using an XML schema file, provided together with the tool (`params.xsd`). The XML schema file (`params.xsd`) should not be modified in any way unless a new configuration parameter is introduced.
ammeter_driver.dll	Current measurements	The `ammeter_driver.dll` is a DLL that provides a generic interface to Digital Multi Meters (DMMs). It is used for measuring the device current. It loads all ammeter instrument DLLs found inside the `ammeter_instr_plugins` folder. Any DLL placed inside this folder, the driver will load it and use it if selected by the user. Ideally, any DMM could be loaded and used by the driver, as long as it follows the specific driver API.
ammeter_scpi.dll	Current measurements	This is an actual DMM current measurement DLL. On one end, it follows the `ammeter_driver.dll` API. On the other end it communicates to SCPI based instruments to configure them to the appropriate scaling and acquire current measurements. Keysight 34401A [8], Keysight 34461A [17] and Keithley 2000 [9] have been tested to be compatible with this DLL.
ni6009.dll	Current measurements	This is a DLL to interface to the NI-6009 USB DAQ [16]. It is compatible to the `ammeter_driver.dll` API and thus can be used to take current measurements. However, external shunt

Library	Purpose	Description
		resistor will be required. Due to the difficulty to change the current measurement scaling, it is only recommended if idle current measurements need to be taken and not sleep.
volt_meter_driver.dll	DA14680/1-00 ADC calibration	The `volt_meter_driver.dll` is a DLL that provides a generic API interface to Digital Voltage Meters (DVMs). The driver can load many different voltage meter instrument DLLs as long as they follow a specific API. It is used to calibrate the ADC gain offset in DA14681-00 (AD) silicon based devices. No other device needs this type of calibration as the calibration is performed during IC manufacturing.
volt_meter_scpi.dll	DA14680/1-00 ADC calibration	The `volt_meter_scpi.dll` is the actual interface to the voltage meter instrument. SCPI generic commands are used to setup the instrument and take measurements. This particular DLL has been tested with the Keysight 34401A [8] and Keithley 2000 [9] instruments. Users could implement their own voltage meter DLL and place it in a specific folder. The `volt_meter_driver.dll` will load it and use it if selected.
ble_tester_driver.dll	RF Direct Test Mode measurements	The `ble_tester_driver.dll` is a DLL that provides a generic API to BLE tester instruments. Various BLE testers could possibly be supported through this interface. Currently, the Litepoint IQXel-M [15] and the Anritsu MT8852B [7] are supported and have been tested to be compatible to this driver. However, one could implement any other instrument DLL to be interfaced to this driver. The driver will load this custom DLL, which could be used if selected in the CFG PLT or in the parameter XML file.
IQxelM.dll	RF Direct Test Mode measurements	The `IQxelM.dll` is a DLL that communicates with the Litepoint IQXel-M [15] BLE tester instrument. It is able to perform standard Direct Test Mode BLE tests.
mt8852b.dll	RF Direct Test Mode measurements	The `mt8852b.dll` is a DLL that communicates with the Anritsu MT8852B [7] BLE tester instrument. It is able to perform standard Direct Test Mode RF BLE tests.
temp_meas_driver.dll	Ambient temperature measurements	The `temp_meas_driver.dll` is a driver DLL that can load and use any temperature sensor DLL. The temperature measurements are not currently used in a particular test. The ambient temperature measurement is just saved in the log files. However, by minor software changes it could be used to calibrate sensors. Currently, the PLT supports two different temperature sensors. These described next. The PLT users can select any of these two DLLs to take temperature measurements or implement a new instrument DLL and place it in the appropriate folder. The PLT with the use of the driver will load it and use it, if it is selected by the user.
tmu_temp_sens.dll	Ambient temperature measurements	The `tmu_temp_sens.dll` is a DLL that communicates with the Papouch-TMU USB thermometer [10]. It reads the ambient temperature, which is then passed to the `prod_line_tool_dll.dll` through the `temp_meas_driver.dll`.
ni_usb_tc01.dll	Ambient temperature measurements	The `ni_usb_tc01.dll` is a DLL able to communicate with the NI USB TC01 thermocouple measurement device [11]. As with the `tmu_temp_sens.dll`, the DLL reads the ambient temperature, which is passed to the `prod_line_tool_dll.dll` through the `temp_meas_driver.dll`.
CFG	Setup the configuration parameters	The CFG is a graphical user interface (GUI) Windows application with its official build name to be **DA1458x_DA1468x_CFG_PLT.exe**. It mostly uses the `cfg_dll.dll` to load and save PLT configuration parameters

Library	Purpose	Description
		in the `params.xml` file. It also uses the `prod_line_tool_dll.dll` to automatically fill some configuration parameters, like the DUT and GU COM ports, or the available instrument DLL names.
GUI	The main test execution application	This is the test execution graphical user interface Windows application with its official name to be **DA1458x_DA1468x_GUI_PLT.exe**. It provides a user friendly interface to start device testing, check results and logging.
CLI	The main test execution CLI application	This is the test execution command line interface Windows application with its official name to be **DA1458x_DA1468x_CLI_PLT.exe**. As with the GUI, one can start device testing, check the results and perform various other actions, like disable/enable devices, set device BD addresses, etc.
barcode_scanner.dll	Scan BD addresses and data to be programmed to memory	The `barcode_scanner.dll` is used only by the GUI application. It is used to interface a barcode scanner in order to scan device BD addresses and data for memory programming, prior to each test. The DLL has been tested with the Honeywell Xenon 1900 and the Motorola LS2208 barcode scanners [12] [13]. However, it is expected that any other barcode scanner with a USB to serial interface to be compatible with this DLL.

4 Source Code

All the Production Line Tool source code is organized in a **Microsoft Visual Studio 2015 Express** solution. The source code and the Visual Studio solution files can be found under the folder `source\production_line_tool` in the `DA1458x_DA1468x_PLT_v_x.x` released software package. To open the Microsoft Visual Studio Express 2015 project, select the `production_line_tool.sln` file.

Twenty different projects are included in the Visual Studio 2015 `production_line_tool.sln` solution. These are illustrated in Figure 2.

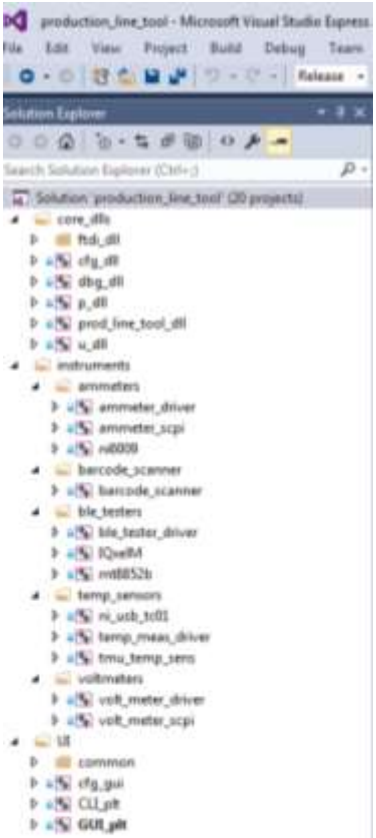


Figure 2: Production Line Tool Visual Studio 2015 Solution Explorer Projects

The projects are grouped into three main categories. These are illustrated in Table 2.

Table 2: Production Line Tool - Visual Studio Project Grouping

Project Group	Description
core_dlls	The `core_dlls` project folder contains the most important DLLs that are needed for almost any type of test or memory programming. Without these, the application will not be able to operate.
instruments	The `instruments` project folder contains DLLs responsible to interface external measurement instruments, like BLE testers, voltage meters, current meters, barcode scanners and temperature measurement sensors. Some of these projects are **optional** and can be **unloaded** prior of building the tool as they require external components (like NI VISA) to be installed to the PC.
UI	The `UI` project folder contains the three user interface projects. ● `cfg_gui`. Builds the **DA1458x_DA1468x_CFG_PLT.exe** application. ● `CLI_plt`. Builds the **DA1458x_DA1468x_CLI_PLT.exe** application.

Project Group	Description
	● GUI_plt. Builds the **DA1458x_DA1468x_GUI_PLT.exe** application.

The source code of each Visual Studio project can be found under the directories given in Table 3.

Table 3: Visual Studio Project Directories

Project	Directory
cfg_dll	source\production_line_tool\core_dlls\cfg_dll
dbg_dll	source\production_line_tool\core_dlls\dbg_dll
p_dll	source\production_line_tool\core_dlls\p_dll
prod_line_tool_dll	source\production_line_tool\core_dlls\prod_line_tool_dll
u_dll	source\production_line_tool\core_dlls\u_dll
barcode_scanner	source\production_line_tool\instruments\barcode_scanner
ble_tester_driver	source\production_line_tool\instruments\ble_testers\ble_tester_driver
IQxelM	source\production_line_tool\instruments\ble_testers\IQxelM
mt8852b	source\production_line_tool\instruments\ble_testers\mt8852b
ni_usb_tc01	source\production_line_tool\instruments\temp_sensors\ni_usb_tc01
temp_meas_driver	source\production_line_tool\instruments\temp_sensors\temp_meas_driver
tmu_temp_sens	source\production_line_tool\instruments\temp_sensors\tmu_temp_sens
ammeter_driver	source\production_line_tool\instruments\ammeters\ammeter_driver
ammeter_scpi	source\production_line_tool\instruments\ammeters\ammeter_scpi
ni6009	source\production_line_tool\instruments\ammeters\ni6009
volt_meter_driver	source\production_line_tool\instruments\voltmeter\volt_meter_driver
volt_meter_scpi	source\production_line_tool\instruments\voltmeter\volt_meter_scpi
cfg_gui	source\production_line_tool\UI\cfg_GUI
CLI_plt	source\production_line_tool\UI\CLI_plt
GUI_plt	source\production_line_tool\UI\GUI_plt

4.1 Prerequisites

Before building and running the code the items indicated in Table 4 should be installed into the PC. Some are optional and will only be required if particular tests are going to be performed.

Table 4: Production Line Tool Prerequisites

Item	Optional	Description
Visual Studio 2015 Express	No	The IDE used to program and debug the Production Line Tool.
MSXML6	No	Installed by default in PCs with Windows 7/8/8.1/10 OS.
.NET framework 4	No	Needed for the graphical user interface applications.
Latest FTDI drivers	No	Tested with FTDI v2.12.24 and v2.12.26 drivers
Honeywell Xenon 1900 drivers	Yes	Used if a barcode scanner is going to be used for scanning the device BD addresses and/or memory data.
Motorola LS2208 drivers	Yes	Used if a barcode scanner is going to be used for scanning the device BD addresses and/or memory data.
NI-VISA 15.5	Yes	Used for instrument control, like BLE tester and voltage meter.

Item	Optional	Description
NI-488.2 15.5	Yes	Used for instrument control through the GPIB interface.
NI_DAQmx	Yes	Used for instrument control, like temperature measurements using the NI USB TC01 sensor.

4.2 Building the Source Code for Windows 7/8/8.1/10

To build the solution (including all DLLs, the CFG, GUI and the CLI executables), one has to make sure that the active Visual Studio 2015 configuration is set to the Release Configuration.

Before building the code, a simple configuration is needed, as follows:

1. In the Visual Studio 2015 Solution Explorer window, right-click on Solution 'production_line_tool'.
2. Select Properties->Common Properties->Startup Project.
3. Open the Single start-up project dropdown menu.
4. Select either `cfg_gui`, `CLI_plt` or the `GUI_plt` and apply the change. The selection depends on which application is going to start in the Visual Studio debug operation, after key F5 is pressed.
5. Alternatively, one could select 'Multiple startup projects' to start all or some of the executables.
6. Exit the solution options.
7. In the top toolbar make sure the Release build configuration is selected as shown in Figure 3.

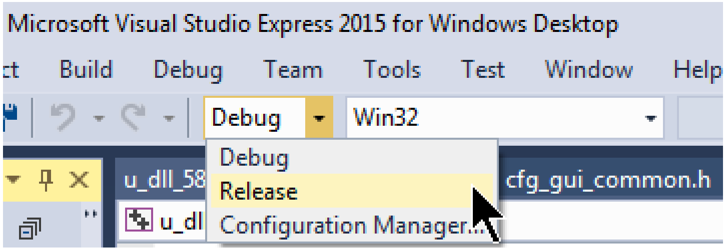


Figure 3: Visual Studio 2015 Release Configuration

8. If current measurements are not required to be performed, then the instrument DLL projects should better be unloaded as they will require NI-VISA to be built. Right click the `ammeter_scpi` and `ni6009` projects and select 'Unload Project', as shown in Figure 4.

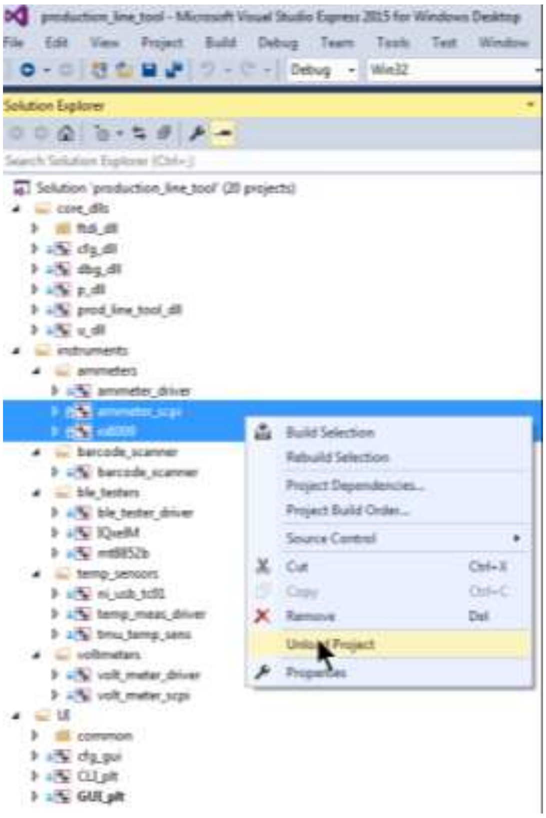


Figure 4: Unload Visual Studio Ammeter Projects

9. If the device to be tested is not DA14680-00 (AD), then the ADC gain calibration is not required. The `volt_meter_scpi` project used for this purpose should be unloaded, following the method described above.
10. If BLE tester measurements are not required, then the `IQxelM` and the `mt8852b` projects should also be unloaded. Otherwise they will require NI-VISA to be installed to be able to build them and of course user should have the appropriate BLE tester instrument.
11. The same process should be applied for the `ni_usb_tc01` and `tmu_temp_sens` projects. If no temperature measurement is required, then the instrument projects should be unloaded. Project `ni_usb_tc01` uses NI-VISA libraries and cannot be built unless the libraries are installed by the user. TMU thermometer [10] can be used instead, which uses a simple USB-to-serial interface.
12. If all of the mentioned instruments are not required and were unloaded, then the Visual Studio explorer `instruments` tab should look like the one in Figure 5.

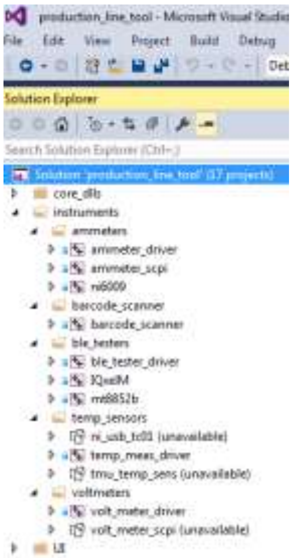


Figure 5: Visual Studio Explorer Tab with Projects Unloaded

13. Select 'Build->Build Solution' or press F7.

All DLLs and application executables will be created in `source\production_line_tool\Release` directory.

4.3 DA14580/1/2/3/5/6 Required Firmware

The DA1458x product family requires at least two pieces of firmware for the Production Line Tool to be operational. The `flash_programmer_XXX.bin` firmware is used by the `u_dll.dll` to be able to download the customer firmware to be written into the memories (OTP, SPI Flash or I2C EEPROM). The `flash_programmer_580.bin` firmware is used for DA14580, DA14581, DA14582 and DA14583 ICs, while the `flash_programmer_585.bin` firmware is used for the DA14585 and DA14586 ICs.

The `p_dll.dll` requires the `prod_test_580.bin` (for DA14580 or DA14583 devices), the `prod_test_581.bin` (for DA14581 devices), the `prod_test_582.bin` (for DA14582 devices) or the `prod_test_585.bin` firmware (for DA14585 and DA14586 devices) to perform the RF, XTAL trim calibration, the audio, the scan and other tests. Customers will also need to use their own firmware to be written in the SPI Flash, EEPROM or OTP memory.

The required firmware for the production tests used by the `p_dll.dll` and the required firmware for memory programming used by the `u_dll.dll` can be found under the directory `\source\production_line_tool\UI\common\binaries`.

To test the `u_dll.dll` memory programming features, a proximity reporter firmware could be written to the OTP, SPI Flash or the EEPROM. A proximity reporter firmware example (`prox_reporter_580.bin` and `prox_reporter_585.bin`) is included and can be found under the same directory.

Finally, the firmware for the Golden Unit (`prod_test_GU.bin)` is included under the directory `source\production_line_tool\UI\common\binaries\GU`. This image should be updated in the Golden Unit (GU) SPI Flash memory mounted in the Production Line Tool hardware [1].

Summarizing, the `source\production_line_tool\UI\common\binaries` directory should contain the firmware files for DA1458x devices as indicated in Table 5.

Table 5: DA1458x Required Firmware

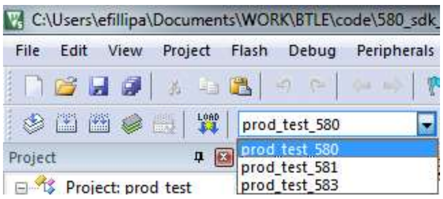
Firmware	Description
`prod_test_580.bin`	DA14580/583 firmware for RF, XTAL trim, sensor and scan test.
`prod_test_581.bin`	DA14581 firmware for RF test, XTAL trim, sensor and scan test.
`prod_test_582.bin`	DA14582 firmware for RF, XTAL trim, audio, sensor and scan test.
`prod_test_585.bin`	DA14585/586 firmware for RF, XTAL trim, audio, sensor and scan test.
`flash_programmer_580.bin`	Firmware used by DA14580/581/582/583 for memory programming.
`flash_programmer_585.bin`	Firmware used by DA14585/586 for memory programming.
`prox_reporter_580.bin`	Example binary of a proximity reporter to test memory programming of DA14580/581/582/583 devices.
`prox_reporter_585.bin`	Example binary of a proximity reporter to test memory programming of DA14585/586 devices.

When a Visual Studio 'Release' build is performed, the entire binaries folder is copied from the `source\production_line_tool\UI\common\binaries` directory to the `source\production_line_tool\Release\binaries` directory.

4.3.1 Building the DA14580/1/2/3/5/6 Firmware

If users need to add extra tests to expand the PLT functionality, then a specific build procedure should be followed for the firmware (`prod_test_580.bin`, `prod_test_581.bin`, `prod_test_582.bin`, `prod_test_585.bin`, `flash_programmer_580.bin` and `flash_programmer_585.bin`) to be used by the PLT software. The steps required to build the firmware are analyzed in Table 6, Table 7, Table 8 and Table 9.

Table 6: DA14580/1/2/3 – Steps to Build the Production Test Firmware

Step	Description
1	Download SDK 5.0.4 from the Dialog BLE customer portal (DA1458x_SDK_5.0.4).
2	Go to `source\production_line_tool\fw_files\DUT` PLT folder. Depending on the device IC used, open the DA14580_581_583 or the DA14582 folder.
3	Copy the files taken from these folders to the equivalent SDK files downloaded at step 1.
4	Go to SDK 5.0.4 `DA1458x_SDK\5.0.4\projects\target_apps\prod_test\prod_test\Keil_5` folder and open the `prod_test.uvprojx` project in Keil IDE.
5	Build the code by first selecting the appropriate target.  Users should select '`prod_test_580`' to build the production test firmware for DA14580, DA14582 and DA14583 devices. Users should select '`prod_test_581`' for DA14581 devices.
6	If code was built for DA14580 then copy `DA1458x_SDK\5.0.4\projects\target_apps\prod_test\prod_test\Keil_5\out_580\prod_test_580.bin` to `source\production_line_tool\UI\common\binaries\prod_test_580.bin`.

Step	Description
	If code was built for DA14581 then copy `DA1458x_SDK\5.0.4\projects\target_apps\prod_test\prod_test\Keil_5\out_581\prod_test_581.bin` to `source\production_line_tool\UI\common\binaries\prod_test_581.bin`. If code was built for DA14582 then copy `DA1458x_SDK\5.0.4\projects\target_apps\prod_test\prod_test\Keil_5\out_580\prod_test_580.bin` to `source\production_line_tool\UI\common\binaries\prod_test_582.bin`. If code was built for DA14583 then copy `DA1458x_SDK\5.0.4\projects\target_apps\prod_test\prod_test\Keil_5\out_580\prod_test_580.bin` to `source\production_line_tool\UI\common\binaries\prod_test_580.bin`.

Table 7: DA14585/6 – Steps to Build the Production Test Firmware

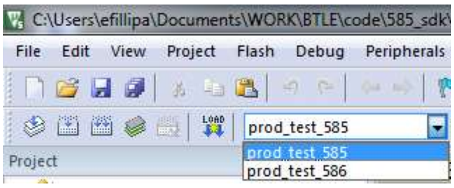
Step	Description
1	Download SDK 6.0.4.326 from the Dialog BLE customer portal (DA14585_SDK_6.0.4.326).
2	Go to `source\production_line_tool\fw_files\DUT\DA14585_586` PLT folder.
3	Copy the files taken from this folder to the equivalent SDK files downloaded at step 1.
4	Go to SDK 6.0.4.326 `DA14585_SDK_6.0.4.326_0\DA14585_SDK\6.0.4.326\projects\target_apps\prod_test\prod_test\Keil_5` folder and open the `prod_test.uvprojx` project in Keil IDE.
5	Build the code by selecting the '`prod_test_585`' target.  Users should select '`prod_test_585`' to build the production test firmware for DA14585 and DA14586 devices.
6	Copy `DA1458x_SDK\5.0.4\projects\target_apps\prod_test\prod_test\Keil_5\out_585\prod_test_585.bin` to `source\production_line_tool\UI\common\binaries\prod_test_585.bin`. This procedure is the same for DA14585 and DA14586 devices.

Table 8: DA14580/1/2/3 – Steps to Build the Flash Programmer Firmware

Step	Description
1	Download SDK 5.0.4 from the Dialog BLE customer portal (DA1458x_SDK_5.0.4).
2	Go to `source\production_line_tool\fw_files\DUT` PLT folder. Depending on the device IC used, open the DA14580_581_583 or the DA14582 folder.
3	Copy the files taken from these folders to the equivalent SDK files downloaded at step 1.
4	Go to SDK 5.0.4 `DA1458x_SDK\5.0.4\DA1458x_SDK\5.0.4\utilities\flash_programmer` folder and open the `programmer.uvprojx` project in Keil IDE.
5	Build the code by selecting the '`programmer_uart`' target.

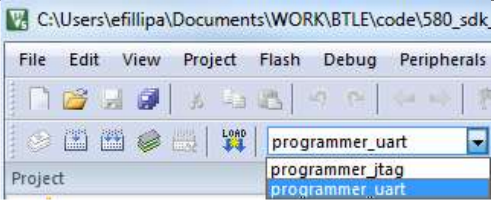
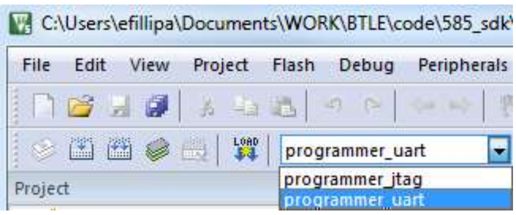
Step	Description
	
6	Copy DA1458x_SDK\5.0.4\utilities\flash_programmer\Out_uart\flash_programmer.bin to source\production_line_tool\UI\common\binaries\flash_programmer_580.bin. This procedure is the same for DA14580, DA14581, DA14582 and DA14583 devices.

Table 9: DA14585/6– Steps to Build the Flash Programmer Firmware

Step	Description
1	Download SDK 6.0.4.326 from the Dialog BLE customer portal (DA14585_SDK_6.0.4.326).
2	Go to source\production_line_tool\fw_files\DUT\DA14585_586 PLT folder.
3	Copy the files taken from this folder to the equivalent SDK files downloaded at step 1.
4	Go to SDK 6.0.4.326 DA14585_SDK_6.0.4.326_0\DA14585_SDK\6.0.4.326\utilities\flash_programmer folder and open the programmer.uvprojx project in Keil IDE.
5	Build by selecting the 'programmer_uart' target. 
6	Copy DA14585_SDK_6.0.4.326_0\DA14585_SDK\6.0.4.326\utilities\flash_programmer\Out_uart to source\production_line_tool\UI\common\binaries\flash_programmer_585.bin. This procedure is the same for DA14585 and DA14586 devices.

4.4 DA1468x Required Firmware

Similar to the DA1458x product family, the DA1468x and DA1510x chipsets also requires two pieces of firmware. One is the uartboot_XXX.bin, which is used by the u_dll.dll to burn the QSPI and the OTP memories. In addition, the p_dll.dll requires the prod_test_681_00.bin for DA14681-00 (AD) or DA14680-00 (AD) devices, the prod_test_681_01.bin for DA14681-01 (AE) or DA14680-01 (AE) devices and the prod_test_683_00.bin for DA14683-00 (BB), DA14682-00 (BB), DA15100-00 (BB) and DA15101-00 (BB). These are needed to perform XTAL trim calibration, RF tests and other tests.

The firmware files can be found under the source\production_line_tool\UI\common\binaries directory.

To test the u_dll.dll memory programming feature, a proximity reporter firmware could be written to the QSPI Flash memory. A proximity reporter firmware example (pxp_reporter_681_01.bin.cached, pxp_reporter_681_00.bin.cached and pxp_reporter_683_00.bin.cached) is included and can be found under source\production_line_tool\UI\common\binaries directory. The .cached format is created using the bin2image.exe application on a firmware binary. It is a special bootable format for DA1468x/DA1510x devices. The bin2image.exe application can be found under the source\production_line_tool\UI\common\binaries directory.

4.4.1 Building the DA1468x Production Test Firmware

If users need to add extra tests to expand the PLT functionality for DA1468x/DA1510x based devices, then the DA1468x/DA1510x production test firmware (`plt_fw.bin`) should be modified. Details on how to build a DA1468x/DA1510x proximity reporter application project are given in [14]. Similar process should be followed to build the `plt_fw` project.

The PLT v_4.2 uses the firmware from `DA1468x_DA15xxx_SDK_1.0.10.1072` DA1468x/DA1510x SDK. Prior of building the `plt_fw` project, the files inside `source\production_line_tool\fw_files\DUT\DA1468x-DA15xxx` should be copied to the equivalent `DA1468x_DA15xxx_SDK_1.0.10.1072` SDK files. These are some patches required specifically for the PLT.

The binary produced when building the `plt_fw` project is named `plt_fw.bin`. This should be renamed either to `prod_test_681_00.bin`, `prod_test_681_01.bin` or `prod_test_683_00` and be placed under the `source\production_line_tool\UI\common\binaries` directory.

4.4.2 Building the DA1468x Memory Programmer Firmware

The firmware responsible to perform the memory operations to DA1468x/DA1510x devices is called `uartboot.bin`. If users would like to add extra functionality, then the `uartboot` Eclipse project should be modified and build. Details on how to build a DA1468x proximity reporter application project in an Eclipse IDE environment are given in [14]. Similar process should be followed for the `uartboot.bin` project. DA14681-01, DA14680-01, DA14682-00, DA14683-00, DA15100-00 and DA15101-00 devices use the same `uartboot.bin` firmware as auto IC detection exists in the firmware start-up function.

Prior of building the `uartboot.bin`, the files inside `source\production_line_tool\fw_files\DUT\DA1468x-DA15xxx` should be copied to the equivalent `DA1468x_DA15xxx_SDK_1.0.10.1072` SDK files. These are some patches required specifically for the PLT. The output binary should be renamed to `uartboot_681_01.bin` and be placed under the `source\production_line_tool\UI\common\binaries` directory.

4.5 Running the Applications

The directory `source\production_line_tool\Release` contains all the necessary files, after a successful project build, for the application to run. Bear in mind that for the `Release` folder to be created, the project has to be built with a 'Release' Visual Studio configuration as mentioned in section 4.2.

Figure 6 shows the files created inside the `Release` directory. Table 10 gives a short description of the files and folders contained in that directory.

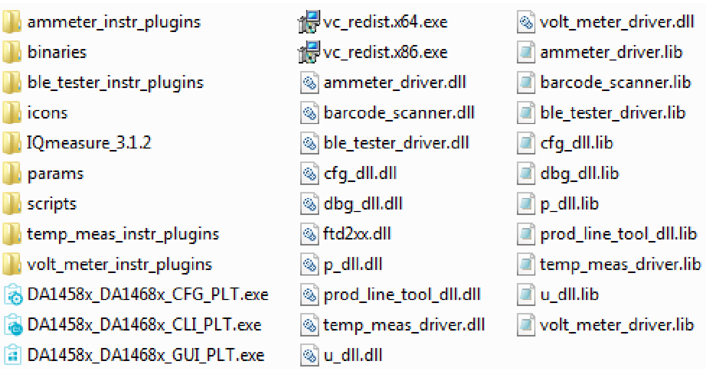


Figure 6: Production Line Tool Release Directory

Table 10: Description of Build Output Files

File or Folder	Description
ammeter_instr_plugins/	Contains the ammeter instrument plug-n DLLs. This folder is only created if any of the ammeter projects is included during build.
ammeter_instr_plugins/ammeter_scpi.dll	This is the DLL that performs current measurements, by interfacing to SCPI compatible DMMs.
ammeter_instr_plugins/ni6009.dll	This is the DLL for the NI USB-6009 DAQ [16] able to perform current measurements.
binaries/	Contains the necessary firmware binaries as described in section 4.3 and 4.4.
ble_tester_instr_plugins/	Contains the BLE tester instrument DLLs. This folder is only created if any of the BLE tester projects is included during build.
ble_tester_instr_plugins/mt8852b.dll	This is the DLL that performs BLE RF tests using the Anritsu MT88252B instrument [7].
ble_tester_instr_plugins/IQxelM.dll	This is the DLL that performs BLE RF tests using the Litepoint IQxelM instrument [15].
icons/	Contains pictures used by the PLT applications.
IQmeasure_3.1.2	Contains Litepoint IQxelM instrument [15] specific DLLs, provided as is by Litepoint.
params/	Contains the configuration `params.xml` file, the XML schema `params.xsd` and a BD address file sample, named `bd_address.ini`.
scripts/	Contains example scripts to be executed by the PLT before or after the tests. User can edit them or create new ones with different names. The new created scripts can be selected from the DA1458x_DA1468x_CFG_PLT.exe.
temp_meas_instr_plugins/	Contains the temperature measurement instrument DLLs.
temp_meas_instr_plugins/ni_usb_tc01.dll	The `ni_usb_tc01.dll` is the DLL used to interface a NI USB TC01 temperature sensor [11], for temperature measurements.
temp_meas_instr_plugins/ tmu_temp_sens.dll	The `tmu_temp_sens.dll` is the DLL used to interface a Papouch TMU sensor [10], for temperature measurements.
volt_meter_instr_plugins/	Contains the voltage meter instrument DLLs. These are used to calibrate the internal ADC for DA1468x-00 devices.
volt_meter_instr_plugins/volt_meter_scpi.dll	The `volt_meter_scpi.dll` is a DLL that implements basic control to a DVM instrument using SCPI commands. It uses NI-VISA libraries and GPIB interface.
DA1458x_DA1468x_CFG_PLT.exe	This is the configuration application. It is a graphical user interface application used to edit the configuration XML file, `params.xml`.
DA1458x_DA1468x_CLI_PLT.exe	This is the command line interface tool. It performs production tests and memory programming through a console.
DA1458x_DA1468x_GUI_PLT.exe	This is the graphical user interface tool. It performs production tests and memory programming through a graphical interface.
vc_redist.x64.exe/vc_redist.x86.exe	Visual Studio 2015 redistributable packages for 32 and 64-bit machines. These should be installed prior of PLT execution.

File or Folder	Description
ammeter_driver.dll/.lib	This is the DLL that loads and accesses the current measurement instrument DLLs from inside the ammeter_instr_plugins folder. It is the middle layer software between the PLT host application and the instrument measurement DLL plug in.
barcode_scanner.dll/.lib	This is the DLL that implements the control to a USB to serial barcode scanner instrument, for scanning device BD addresses.
ble_tester_driver.dll/.lib	This is the DLL that loads and accesses all BLE tester instrument DLLs from the `ble_tester_instr_plugins` folder
cfg_dll.dll/.lib	This is the configuration parameter handling DLL. It can validate, load and save parameters from a given XML file.
dbg_dll.dll/.lib	The `dbg_dll.dll` file is a DLL used to print debug messages to a file or to a debug console.
ftd2xx.dll	This is the FTDI DLL. It is used to hard reset the Golden Unit from the application when needed, via an FTDI GPIO pin.
p_dll.dll/.lib	This is the production test DLL that performs device functional tests.
prod_line_tool_dll.dll/.lib	This is the core DLL. The heart of the system that performs basic state machines for all tests and memory actions to be executed.
temp_meas_driver.dll/.lib	This is the temperature measurement driver DLL. It loads and accesses all temperature measurement DLLs from `temp_meas_instr_plugins` folder.
u_dll.dll/.lib	This is the DLL that performs memory actions.
volt_meter_driver.dll/.lib	This is the voltage meter driver DLL. It loads and accesses all voltage meter DLLs from folder `volt_meter_instr_plugins`.

4.5.1 DA1458x_DA1468x_CFG_PLT.exe

Double click the **DA1458x_DA1468x_CFG_PLT.exe** to run the configuration application. Most probably an error message will pop, as shown in Figure 7. This is to indicate that the Golden Unit COM port is either not valid or the GU USB cable from the PLT board is not connected to the PC.

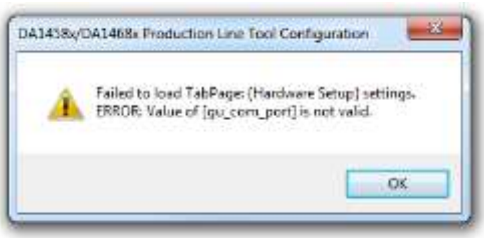


Figure 7: DA1458x_DA1468x_CFG_PLT.exe GU COM Port Error Message

Press ‘OK’ if the message in Figure 7 is shown. The configuration application initial screen will then be shown (Figure 8). Connect the two USB cables from the PLT board to the PC and after the PC has enumerated all 17 COM ports (16 for the devices and one for the GU) press ‘Auto’ under the ‘Golden Unit COM Port’. This action will automatically find the GU COM port. Press the ‘Save’ button.

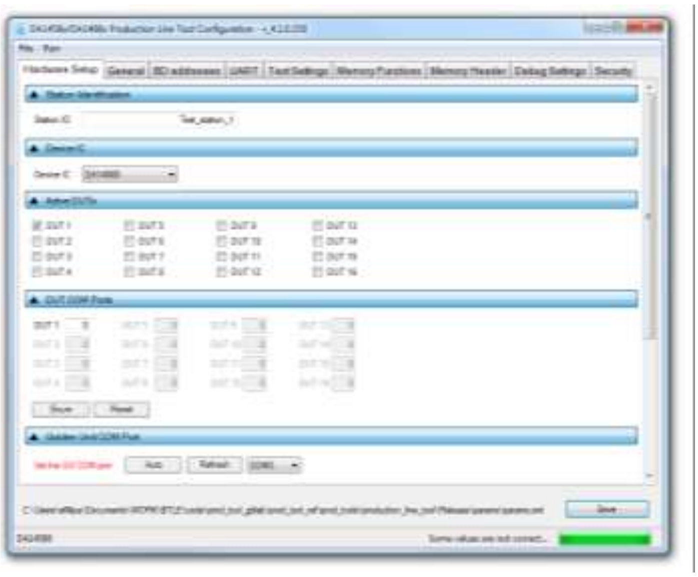


Figure 8: DA1458x_DA1468x_CFG_PLT.exe Initial Screen with GU COM Port Error

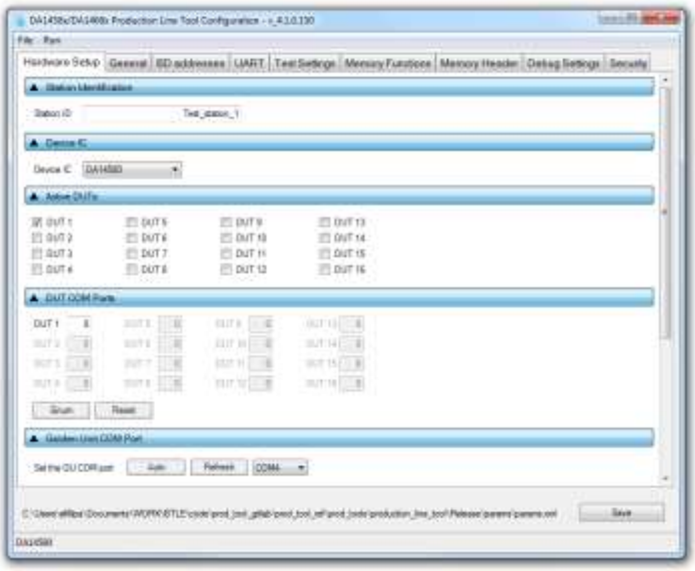


Figure 9: DA1458x_DA1468x_CFG_PLT.exe Initial Screen

4.5.2 DA1458x_DA1468x_GUI_PLT.exe

Double click the **DA1458x_DA1468x_GUI_PLT.exe** to run the GUI PLT application. The initial screen will appear as shown in Figure 10.

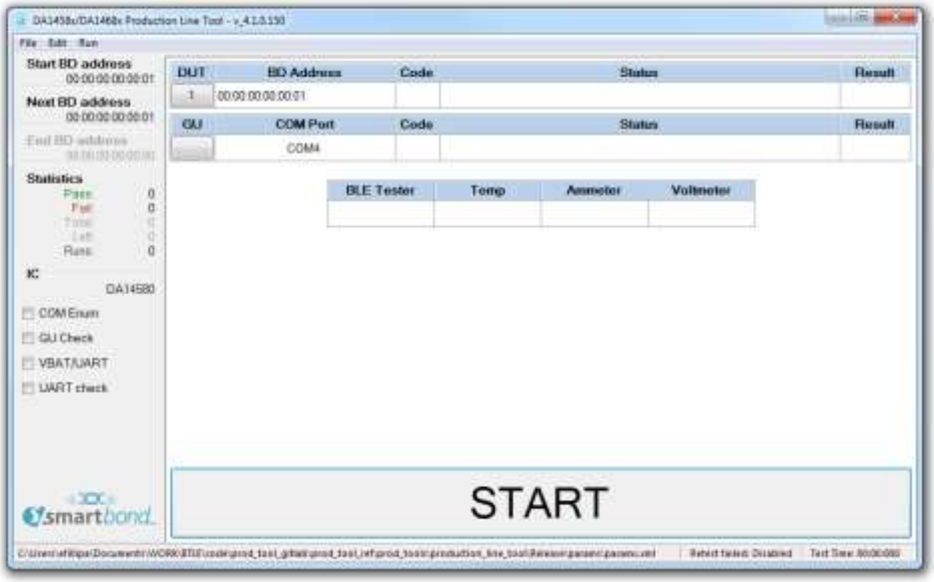


Figure 10: DA1458x_DA1468x_GUI_PLT.exe Initial Screen

4.5.3 DA1458x_DA1468x_CLI_PLT.exe

Double click the **DA1458x_DA1468x_CLI_PLT.exe** to run the CLI PLT application. The initial screen will then appear as shown in Figure 11.

Figure 11: DA1458x_DA1468x_CLI_PLT.exe Initial Screen

A user guide document [1] is available to help users get familiar with all three applications.

5 CFG_DLL

The CFG_DLL dynamic link library provides the necessary API functionality to import, validate and export configuration parameters to or from an XML file. These parameters tell which tests will be performed, which memories will be written, which binary will be written into the memories, what the memory interface GPIOs will be and many other settings that the user needs to configure prior to each test. The `cfg_dll.dll` is only used by the User Interface software applications (CFG, GUI or CLI). The parameters loaded by the UI tools using the `cfg_dll.dll` are passed to the `prod_line_tool_dll.dll`. The `prod_line_tool_dll.dll` copies these parameters to its local data structures in an appropriate format to be passed to the rest of the DLLs or used internally for controlling the test state machines.

A default configuration parameter file, `params.xml`, is provided in the directory `source\production_line_tool\UI\common\params`. This file is automatically edited when a change and a save is made in the `DA1458x_DA1468x_CFG_PLT.exe` tool. Some parameters are also automatically modified by the `DA1458x_DA1468x_GUI_PLT.exe` and `DA1458x_DA1468x_CLI_PLT.exe`, like the statistics and the next device BD addresses. This file, although not suggested, can also be edited manually. One could then use the "`File -> Open XML file`" option in the CFG or GUI tool to import the new parameters. Similarly, the CLI command "`i params\params.xml`" will import a new configuration file to the CLI application.

The parameter validation is performed using the XML schema file, `params.xsd`. This file contains the parameter default values as well as the type and range of values that each parameter can accept.

Finally, the `cfg_dll.dll` provides an API function to import DUT BD addresses from a file. A sample file of device BD addresses, `bd_address.ini`, is provided and can be found under the `source\production_line_tool\UI\common\params` directory.

Note: The end of the bd_address.ini file must have an all-zeroes BD address.

5.1 CFG_DLL API Functions

The CFG_DLL API header file can be found in `source\production_line_tool\core_dlls\cfg_dll\cfg_dll.h`.

It has the following user accessible functions.

```
CFG_DLL_API int cfg_dbg_init(_dbg_params *dbg_params_t);
CFG_DLL_API int cfg_dbg_close(void);
CFG_DLL_API int cfg_init(char *file_path_t);
CFG_DLL_API int cfg_close(void);
CFG_DLL_API int cfg_get_value(char *param_name, uint8_t idx, char *param_value);
CFG_DLL_API int cfg_set_default_values(void);
CFG_DLL_API int cfg_get_info(char *param_name, char *info);
CFG_DLL_API int cfg_set_value(char *param_name, uint8_t idx, char *param_value);
CFG_DLL_API int cfg_check_value(char *param_name, uint8_t idx, char *param_value);
CFG_DLL_API int cfg_check_param_idx(char *param_name, uint8_t idx);
CFG_DLL_API int cfg_add_param_idx(char *param_name);
CFG_DLL_API int cfg_del_param_idx(char *param_name);
CFG_DLL_API int cfg_import_settings(_cfg_params *cfg_params_t, _cfg_errors
      *cfg_errors_t);
CFG_DLL_API char *cfg_get_param_name(int idx);
CFG_DLL_API int cfg_cross_check_settings(_cfg_params *cfg_params_t, _cfg_errors
      *cfg_errors_t);
CFG_DLL_API int cfg_export_settings(_cfg_params *cfg_params_t);
CFG_DLL_API int cfg_load_bd_addr(_cfg_params *cfg_params_t, uint8_t *next_bd_addr);
```

A short description of each API function follows below.

CFG_DLL_API int cfg_dbg_init(_dbg_params *dbg_params_t)

The `cfg_dbg_init` API function is used to initialize the debug print messages of the `cfg_dll.dll` library. It uses `dbg_dll.dll` for printing messages to either a debug console or a file. The function returns `CFG_SUCCESS` if the debug operation was correctly initialized or `CFG_ERROR` if an error occurred.

CFG_DLL_API int cfg_dbg_close(void)

The `cfg_dbg_close` API function is used to stop the debug print messages of the `cfg_dll.dll` and free all allocated resources previously acquired. The function returns `CFG_SUCCESS` if the debug was correctly closed or `CFG_ERROR` if an error was occurred.

CFG_DLL_API int cfg_init(char *file_path_t)

The `cfg_init` API function is used to initialize the `cfg_dll.dll` library. It loads the XML and XSD files into memory. It is crucial, for better system memory performance, to initialize the `cfg_dll.dll` once in an application lifetime. Otherwise, the MSXML's internal memory management caching will keep older files loaded into RAM for longer time than it is expected by the developer, even if these are released when the `cfg_close` API function is called. The function returns `CFG_SUCCESS` if the `cfg_dll.dll` was correctly initialized, `CFG_CANNOT_OPEN_FILE` if the file given at the `file_path_t` function argument cannot be opened or `CFG_ERROR` if another error has occurred.

CFG_DLL_API int cfg_close(void)

The `cfg_close` API function is used to close the `cfg_dll.dll` library and release all acquired resources, like COM objects and MSXML DOM XML and schema documents. The function returns `CFG_SUCCESS` if the `cfg_dll.dll` was correctly closed or `CFG_ERROR` if an error has occurred.

CFG_DLL_API int cfg_get_value(char *param_name, uint8_t idx, char *param_value)

The `cfg_get_value` API function returns the value in string format of a given parameter name with a specific index. The purpose of the index, `idx`, is to distinguish input parameter names that exist multiple times in the XML file. Such parameters exist for tests that can be executed multiple times. For example, the RF RX test can be performed up to 10 times in 10 different channels. In that case, the configuration parameter `rssi_freq` will exist 10 times in the file. It will be distinguished by the `item` property. The following is an example or three `rssi_freq` parameters.

```
<rssi_freq item="1">2410</rssi_freq>
<rssi_freq item="2">2420</rssi_freq>
<rssi_freq item="3">2430</rssi_freq>
```

To get the value of the `rssi_freq` for the second test one should call the API function like that:
`cfg_get_value("rssi_freq", 2, &param_value);`

Before returning the parameter value, the API function validates it with the definition given in the XML schema document, `params.xsd`. If the parameter value in the XML file is wrong, the default value will be returned in the `param_value` argument, taken from the `params.xsd` file. In that case the function will return `CFG_PARAM_ERROR`. If there is no error, `CFG_SUCCESS` will be returned and the `param_value` argument will have a valid parameter value. If another error occurs while parsing the value, `param_value` argument will be invalid and the function will return `CFG_ERROR`.

CFG_DLL_API int cfg_get_info(char *param_name, char *info)

The `cfg_get_info` API function returns the `x:info` attribute from the XML schema element in the `params.xsd` document. This attribute provides a short tooltip description for each different parameter that is loaded in the graphical user interface tool. An example XML schema element is shown below.

```
<xs:element name="dut_num_15"
       type="xs:boolean"
```



```
        x:use="required"
        x:default="false"
        x:info="Disable if testing at DUT position 15 is not required."/>
```

This is the element describing the behavior of the XML dut_num_15 parameter. The info attribute gives a short description of the dut_num_15 parameter.

The API function returns CFG_SUCCESS if it succeeded to get the information attribute from the XML schema document, in which case the info argument will have a valid value, or CFG_ERROR if an error was occurred. In that case the info argument will be invalid.

CFG_DLL_API int cfg_set_default_values(void)

The cfg_set_default_values API function sets all XML parameters to their default values. The default values are taken from the XML schema file (params.xsd). As an example consider the reset_duration XML value. The value in the XML may have changed to 100.

```
<reset_duration>100</reset_duration>
```

The default value of the reset_duration inside the params.xsd file is 50 as shown below.

```
<xs:element name="reset_duration"
            type="x:cfg_reset_time"
            x:use="required"
            x:default="50"
            x:info="Reset time duration in ms. 10ms to 1000ms is supported."/>
```

By calling the cfg_set_default_values function the reset_duration inside the params.xml file will get the default value of 50.

CFG_DLL_API int cfg_set_value(char *param_name, uint8_t idx, char *param_value)

The cfg_set_value API function sets the value of a given parameter name with a specific index. The function returns CFG_SUCCESS if the cfg_dll.dll succeeded to set the value or CFG_ERROR if an error was occurred.

CFG_DLL_API int cfg_check_value(char *param_name, uint8_t idx, char *param_value)

The cfg_check_value API function checks if the value in the param_value input function argument is valid for the parameter with name pointed by the param_name input function argument. The function checks the parameter definition in the XML schema document (params.xsd) to find whether the given value is valid. The schema document contains data types, boundaries and other restrictions for each particular parameter value so to protect for errors that could occur during parameter parsing. Device could eventually be destroyed if erroneous parameters are programmed in the One Time Programmable memory (OTP), so parameter validation is a key part of the configuration DLL. An example of such a value boundary is shown next.

```
<xs:element name="RF_path_loss_DUT_1"
        type="x:cfg_dut_path_losses"
        x:use="required"
        x:default="0"
        x:info="Set the RF path losses in dB between the device and the GU or the BLE
tester instrument."/>

<!--cfg_dut_path_losses-->
<xs:simpleType name="cfg_dut_path_losses">
        <xs:restriction base="xs:float">
                <xs:minInclusive value="0"/>
                <xs:maxInclusive value="40"/>
        </xs:restriction>
</xs:simpleType>
```


The above XML schema parts are definitions for the `RF_path_loss_DUT_1`, which keep the path losses value for DUT 1. The attribute `type` describes the type that the `RF_path_loss_DUT_1` value can take. This should be of type `cfg_dut_path_losses`. Looking at the definition of the `cfg_dut_path_losses` element we can see that this type is float with minimum value 0 and maximum 40. Checking the XML file for `RF_path_loss_DUT_1` the following exists.

```
<RF_path_loss_DUT_1>0.00</RF_path_loss_DUT_1>
```

If a negative value or a value larger than 40 is set, the validation of the value will fail. Similarly, if the `cfg_check_value` API function is called with `cfg_check_value("RF_path_loss_DUT_1", 0, "45")` the function will return `CFG_PARAM_ERROR`.

The API function will return `CFG_SUCCESS` if the value is a valid value that the `param_name` can take or `CFG_ERROR` on system failure.

CFG_DLL_API int cfg_check_param_idx(char *param_name, uint8_t idx)

The `cfg_check_param_idx` API function checks whether a parameter index for a specific parameter name exists. As noted in the `cfg_get_value` API function, some parameter names could exist more than one time in the XML file, if the test that they belong is to be executed more than one time. The return of the `cfg_check_param_idx` API function helps the CFG application to draw the exact number of tests.

For example consider an XML file that has the following entries. These are entries for two RSSI tests using the GU as transmitter.

```
<rssi_test_enable item="1">true</rssi_test_enable
<rssi_test_enable item="2">true</rssi_test_enable>
<rssi_freq item="1">2424</rssi_freq>
<rssi_freq item="2">2450</rssi_freq>
<rssi_limit item="1">-70.0</rssi_limit>
<rssi_limit item="2">-70.0</rssi_limit>
```

When the CFG application is started it will call the `cfg_check_param_idx` API function with the `idx` parameter starting from 0 up to `MAX_SUPPORTED_RF_TESTS`, until it returns `CFG_ERROR`. For any success loop it will draw a new test tab. For this particular XML example, two tabs of RF RX tests will be drawn as illustrated in Figure 12.

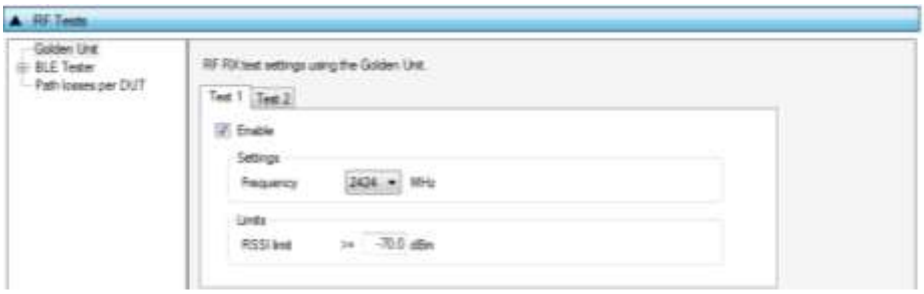


Figure 12: DA1458x_DA1468x_CFG_PLT.exe with Two GU RF Tests

CFG_DLL_API int cfg_add_param_idx(char *param_name)

The `cfg_add_param_idx` API function will add a new test item in the XML file with the given parameter name. If the function succeeds it will return `CFG_SUCCESS`, otherwise `CFG_ERROR`. As an example consider the following XML entries.

```
<rssi_limit item="1">-70.0</rssi_limit>
<rssi_limit item="2">-70.0</rssi_limit>
```

If the API function is called as `cfg_add_param_idx("rssi_limit");` then the XML file will be updated to the following.

```
<rssi_limit item="1">-70.0</rssi_limit>
<rssi_limit item="2">-70.0</rssi_limit>
<rssi_limit item="3">-70.0</rssi_limit>
```

The default value used for new added parameter items is taken from the XML schema document, `params.xsd`. For this example the default value is -70. Below is the schema part for the `rssi_limit`, where the default value can be seen.

```
<xs:sequence minOccurs="1"
             maxOccurs ="10">
    <xs:element name="rssi_limit"
                type="x:cfg_rssi_limit_i"
                x:use="required"
                x:default="-70"
                x:info="The RSSI limit for…"
</xs:sequence>
```

The API function will return `CFG_SUCCESS` if it succeeds to add a new parameter item or `CFG_ERROR` otherwise.

CFG_DLL_API int cfg_del_param_idx(char *param_name)

The `cfg_del_param_idx` API function will delete the last test item in the XML file for the given parameter name. If the function succeeds it will return `CFG_SUCCESS`, otherwise `CFG_ERROR`. As an example consider the following XML entries.

```
<rssi_limit item="1">-70.0</rssi_limit>
<rssi_limit item="2">-70.0</rssi_limit>
```

If the API function is called as `cfg_del_param_idx("rssi_limit");` then then XML file will be updated to the following.

```
<rssi_limit item="1">-70.0</rssi_limit>
```

The API function will return `CFG_SUCCESS` if it succeeds to delete a parameter item or `CFG_ERROR` otherwise.

CFG_DLL_API int cfg_import_settings(_cfg_params *cfg_params_t, _cfg_errors *cfg_errors_t)

The `cfg_import_settings` API function imports all configuration parameters from the XML to the `cfg_params_t` data structure. The `cfg_errors_t` argument is an array. Each array position keeps a Boolean value that corresponds to each of the imported parameters. The index in the array is taken from the `_cfg_param_idx` enumeration. If the Boolean value is true then an error exists for the particular parameter, meaning that the value in the XML for this parameter is invalid. The function will return the default value instead in the `cfg_params_t` argument. Therefore, if a parameter value in the XML file is not valid (the value is out of range, misspelled or empty) the default value from the `params.xsd` file will be returned for this parameter in the `cfg_params_t` data structure. At the same time, an error flag will be set and returned for the specific parameter in the `cfg_error_t` data structure. The value in the XML file will remain as it was before, erroneous, and can only be replaced by a `cfg_export_settings` API function call (see next) or if the user corrects the value by hand.

This function can set an error flag in the `cfg_errors_t` data structure for a particular parameter even if this parameter is not going to be used. For example consider the following XML entries.

```
<xtal_trim_enable>false</xtal_trim_enable>
<xtal_trim_gpio>P0_5</xtal_trim_gpio>
<xtal_trim_otp_burn>none</xtal_trim_otp_burn>
```

The `xtal_trim_otp_burn` value has an error since it should either be `false` or `true`, but is `none`. The `cfg_import_settings` API function will read it and set an error flag in the `XTAL_TRIM_BURN_OTP` index of the `cfg_errors_t` array. The value in the `cfg_params_t` array for the `xtal_trim_otp_burn` parameter will be the default one taken from the schema file. Therefore, even if the XTAL trim test is disabled the function will set an error on the `xtal_trim_otp_burn`. However, the parameter will only be used if the XTAL trim test is enabled. To eliminate such kind of errors, the API function `cfg_cross_check_settings` can be used, which is explained next.

CFG_DLL_API int cfg_cross_check_settings(_cfg_params *cfg_params_t, _cfg_errors *cfg_errors_t)

The `cfg_cross_check_settings` API function is used to check whether error flags enabled in the `cfg_errors_t` array are actual valid errors, as they may belong to tests and memory operations that are disabled. Whether tests or memory operations are disabled is checked from the input `cfg_params_t` data structure. If the function finds at least one valid error flag contained in the `cfg_errors_t` array it keeps the flag set and returns `CFG_ERROR`. For any error that is not valid the error flag is cleared. If no valid errors exist the function will return `CFG_SUCCESS`.

CFG_DLL_API int cfg_export_settings(_cfg_params *cfg_params_t)

The `cfg_export_settings` API function is used to save all the configuration parameters in the XML file. The parameters to be saved are taken from the `cfg_params_t` function argument. No validation is performed in the parameters prior to be saved. The function, prior of saving the parameters, transforms each `cfg_params_t` data structure member from any type to string, in order to be saved in the XML file. If the save succeeds the function returns `CFG_SUCCESS`, otherwise it returns `CFG_ERROR`.

CFG_DLL_API char *cfg_get_param_name(int idx);

The `cfg_get_param_name` API function returns the parameter name for a given index. The index is taken from the `_cfg_param_idx` enumeration found in the API header file, `cfg_dll.h`.

CFG_DLL_API int cfg_load_bd_addr(_cfg_params *cfg_params_t, uint8_t *next_bd_addr)

The `cfg_load_bd_addr` API function, loads DUT BD addresses from a predefined file. The predefined file path is stored in `cfg_params_t->plt_ui_params.bd_addr.file_path` variable. If the file does not exist the function will return `CFG_ERROR`. The function searches the given file to find the initial BD address to start reading from. It will not start reading BD addresses from the beginning of the file but from the BD address pointed by the `cfg_params_t->plt_ui_params.bd_addr.next` variable.
If the next BD address does not exist in the file, the function will return an error. If it exists it will copy it to the first active DUT and the following ones to the rest of the active DUTs.

The end of the bd_address.ini file must have an all zero BD address.

As an example consider the following `bd_address.ini` file contents:

```
00:00:00:01:10:30
00:00:00:01:10:40
00:00:00:01:10:50
00:00:00:01:10:60
00:00:00:01:10:61
00:00:00:01:10:62
00:00:00:01:10:63
00:00:00:01:10:64
00:00:00:00:00:00
```

Consider the input function parameters, given in the first function argument (`cfg_params_t`) to have the following values:

```
cfg_params_t->plt_ui_params.bd_addr.next = 00:00:00:01:10:40;
```

```
cfg_params_t->pltd_device_params[0].is_active = true;
cfg_params_t->pltd_device_params[1].is_active = true;
cfg_params_t->pltd_device_params[2].is_active = true;
cfg_params_t->pltd_device_params[3].is_active = true;
cfg_params_t->pltd_device_params[4].is_active = true;
```



```
cfg_params_t->pltd_device_params[5].is_active = false;
…
cfg_params_t->pltd_device_params[15].is_active = false;
```

Only DUTs 0 to 4 are active. The next BD address is set to 00:00:00:01:10:40. If the `cfg_load_bd_addr` function is called then it will return the following results.

```
cfg_params_t->pltd_device_params[0].bd_addr = 00:00:00:01:10:40;
cfg_params_t->pltd_device_params[1].bd_addr = 00:00:00:01:10:50;
cfg_params_t->pltd_device_params[2].bd_addr = 00:00:00:01:10:60;
cfg_params_t->pltd_device_params[3].bd_addr = 00:00:00:01:10:61;
cfg_params_t->pltd_device_params[4].bd_addr = 00:00:00:01:10:62;
cfg_params_t->pltd_device_params[5].bd_addr = 00:00:00:00:00:00;
…
cfg_params_t->pltd_device_params[7].bd_addr = 00:00:00:00:00:00;
```

The DUT BD addresses will be copied to the first function argument (`cfg_params_t`). The second function argument of the `cfg_load_bd_addr` function will contain the next BD address to be used at the next PLT test run. This will be address 00:00:00:01:10:63. When the current test is finished, the UI application will save the returned `next_bd_addr` argument to the `cfg_params_t->plt_ui_params.bd_addr.next`. It can then be retrieved from the file in the next PLT test run.

The `CFG_DLL_API` is defined as follows.

```
#define CFG_DLL_EXPORTS
#ifdef CFG_DLL_EXPORTS
#define CFG_DLL_API __declspec(dllexport)
#else
#define CFG_DLL_API __declspec(dllimport)
#endif
```

The `declspec(dllexport)` keyword automatically places the exported names in the `.lib` file during compilation. The `.lib` file can be used when a static DLL link is required. The `declspec(dllimport)` keyword is used in DLL header files in order to import DLL public data and objects.

5.2 CFG_DLL API Details

The CFG_DLL API data structures, enumerations, return codes and other details can be found in the API header file `source\production_line_tool\core_dlls\cfg_dll\cfg_dll.h` or in the HTML based help pages loaded after pressing the `source\production_line_tool\help\help.html` link.

6 DBG_DLL

The DBG_DLL dynamic link library provides the necessary API functionality to print debug information in different outputs and for different message levels. All software blocks can access the `dbg_dll.dll`. By using a specific handle for each software block, every message can be separated and handled differently.

Note that printing debug information may introduce system delay and thus some tests may fail due to time out expirations. We suggest having debug information disabled in all software blocks and only partially enable when there is a real need for it. From PLT v4.0 and onwards, this system delay has been almost eliminated as debug print messages are printed from a lower priority queue. It is safer, but it is still suggested to have the debug prints disabled.

6.1 DBG_DLL API Functions

The DBG_DLL API header file can be found in `source\production_line_tool\core_dlls\dbg_dll\dbg_dll.h`.

It has the following user accessible functions.

```
DBG_DLL_API int dbg_init(void **dbg_session, _dbg_params *dbg_params_t);
DBG_DLL_API int dbg_close(void *dbg_session);
DBG_DLL_API void dbg_print(void *dbg_session, DBG_LEVEL dbg_level, char *dbg_sw, char
*func, int line, char *fmt, ...);
```

A short description of each API function follows.

DBG_DLL_API int dbg_init(void **dbg_session, _dbg_params *dbg_params_t)

The dbg_init API function initializes the debug session for a specific software block. It returns a handle in the first function argument specific to the software block. If the debug print output is set to DBG_TO_STDIO then a console will open to print messages. The dbg_dll.dll will keep the debug console open if at least one of the active debug sessions has its output set to DBG_TO_STDIO. If the function succeeds, it returns DBG_SUCCESS and the dbg_session handle is valid. If it fails due to invalid input parameters that could exist in the dbg_params_t data structure, it returns DBG_WRONG_PARAMS. In any other failure it returns DBG_ERROR.

DBG_DLL_API int dbg_close(void *dbg_session)

The dbg_close API function closes the specific debug session and frees all allocated resources for the particular session pointed by the dbg_session input pointer parameter. The dbg_dll.dll will keep the debug console open if at least one of the active debug sessions has its output set to DBG_TO_STDIO. If the close was successful the function returns DBG_SUCCESS otherwise it returns DBG_ERROR.

DBG_DLL_API void dbg_print(void *dbg_session, DBG_LEVEL dbg_level, char *dbg_sw, char *func, int line, char *fmt, ...)

The dbg_print API function prints the debug information for the specific session pointed by the dbg_session input pointer parameter. The rest of the input parameters provide information for a more detailed print format.

The DBG_DLL_API is defined as follows.

```
#define DBG_DLL_EXPORTS
#ifdef DBG_DLL_EXPORTS
#define DBG_DLL_API __declspec(dllexport)
#else
#define DBG_DLL_API __declspec(dllimport)
#endif
```

The declspec(dllexport) keyword automatically places the exported names to the .lib file during compilation. The .lib file can be used when a static DLL link is required. The declspec(dllimport) keyword is used in header files that use the DLL in order to import DLL public data and objects.

6.1.1 DBG_DLL Function Input Parameters

The dbg_init API function takes a data structure as argument, which contains the debug settings. The dbg_print API function has some fixed arguments but also variable length, which are required for different kind of prints.

The following sections describe the function parameters in detail.

6.1.1.1 Function dbg_init Input Arguments

The dbg_init function takes a pointer as argument to the following data structure.

```
typedef struct __dbg_params
{
    bool        dbg_enable;
    int         dbg_out;
```



```
    int         dbg_level;
    char        dbg_file_path[FILE_PATH_SIZE];
    _dbg_clbk   dbg_clbk;
}_dbg_params;
```

Additionally, there are two enumerations that specify the selection of the debug output and the selection of the debug level. These two enumerations are the following:

```
typedef enum _DBG_OUTPUT
{
    DBG_TO_STDIO          = 0x1, /*!< Send debug to stdio output. */
    DBG_TO_FILE           = 0x2, /*!< Send debug info to a file. */
    DBG_TO_CLBK           = 0x4, /*!< Use a callback function. */
    DBG_OUTPUT_INVALID    = 0x8  /*!< Invalid debug output. */
}DBG_OUTPUT;

typedef enum _DBG_LEVEL
{
    DBG_LVL_ERR            = 0x1, /*!< Error debug level. */
    DBG_LVL_INFO           = 0x2, /*!< Information debug level. */
    DBG_LVL_DEBUG          = 0x4, /*!< Level for debug prints. */
    DBG_LVL_INVALID        = 0x8  /*!< Invalid debug level. */
}DBG_LEVEL;
```

```
bool dbg_enable;
```

The dbg_enable parameter is used to enable or disable the debug prints.

```
int dbg_out;
```

The dbg_out parameter value tells where the debug messages will be printed at. It can be any combination of the DBG_OUTPUT enumeration given above. That means, for example, that debug messages can be sent to either a file or stdio or even to both of them.

```
int dbg_level;
```

The dbg_level parameter value tells which debug level will be allowed to be printed. It can be any combination of the DBG_LEVEL enumeration given above.

```
char dbg_file_path[FILE_PATH_SIZE];
```

The dbg_file_path parameter specifies the path where the debug output file will be stored. It is used only when DBG_TO_FILE is set in the dbg_out value.

The FILE_PATH_SIZE is defined as:

```
#define FILE_PATH_SIZE              256
```

```
_dbg_clbk dbg_clbk;
```

The dbg_clbk parameter is a callback function registration that will return the debug message on a string to be used by the calling process. The callback function has the following type:

```
typedef void (__stdcall *_dbg_clbk)(char *dbg_str);
```

```
void **dbg_session
```

The dbg_session parameter is an output parameter returned by the dbg_init function. This parameter should be stored by the block that called the dbg_init function, and used whenever the dbg_print or dbg_close function is to be called.

6.1.1.2 Function dbg_close Input Arguments

Function dbg_close takes only one argument, the handle to the debug session.

`void *dbg_session`

The `dbg_session` parameter is a handle to the debug session, acquired when the `dbg_init` API function was called.

6.1.1.3 Function dbg_print Input Arguments

`void *dbg_session`

The `dbg_session` parameter is a handle to the debug session, acquired when the `dbg_init` API function was called.

`DBG_LEVEL dbg_level`

The `dbg_level` parameter specifies the debug print level of the particular print. If the level of this particular print was not enabled in the `dbg_init` function, then this print will not be printed.

`char *dbg_sw`

The `dbg_sw` parameter is a string that contains the software block name. It will be printed in front of the actual debug message so users can distinguish prints by software blocks.

`char *func`

The `func` parameter is a string that contains the name of the function that this print came from. It will be printed in front of the actual debug message so users can easily point in the code where this message came from.

`int line`

The `line` parameter value specifies the line number from the file that this print came from. It will be printed in front of the actual debug message so user can easily point in the code where this message came from.

`char *fmt, ...`

The `fmt` parameter contains the variable length print arguments. It is the actual debug print message.

6.2 DBG_DLL API Details

The DBG_DLL API data structures, enumerations and return codes and other details can be found in the API header file `source\production_line_tool\core_dlls\dbg_dll\dbg_dll.h`, or in the HTML based help pages loaded after pressing the `source\production_line_tool\help\help.html` link.

7 U_DLL

The U_DLL dynamic link library provides the necessary API functionality to download any firmware to the DUT's system RAM. For example, it is used to download the production test firmware (`prod_test_580.bin`, `prod_test_581.bin`, `prod_test_582.bin`, or `prod_test_681_01.bin`) to the device. Having downloaded this particular firmware, the user could then use the P_DLL functions to send commands to the devices (via the UART), set it to continuous packet transmit, continuous packet receive mode or perform other test operations like XTAL trim.

The `u_dll.dll` can also be used to perform operations to externally attached memories (SPI Flash, I2C EEPROM or QSPI) or the internal OTP memory. First, the `flash_programmer.bin` (DA1458x) or the `uartboot.bin` (DA1468x) firmware has to be downloaded to the DUT's system RAM. Then, the `u_dll.dll` can erase or write any data to any of the supported memories.

File `source\production_line_tool\core_dlls\u_dll\u_dll.h` contains all the necessary API information. It can be included as is in any user project.

7.1 U_DLL API Functions

U_DLL has the following user accessible functions:

```
U_DLL_API int udll_init(void);
U_DLL_API int udll_dbg_init(_dbg_params *dbg_params_t);
U_DLL_API int udll_dbg_close(void);
U_DLL_API int udll_set_prog_params(_udll_params *udll_params_t);
U_DLL_API int udll_set_device_params(_udll_device_params *udll_device_params);
U_DLL_API int udll_start_prog(void);
U_DLL_API int udll_close(void);
```

A short description of each API function follows.

U_DLL_API int udll_init(void);

The `udll_init` API function initializes the `u_dll.dll` library. It should be called before any other operation with the `u_dll.dll` library. It returns `UDLL_SUCCESS`.

U_DLL_API int udll_dbg_init(_dbg_params *dbg_params_t);

The `udll_dbg_init` API function initializes the `u_dll.dll` debug print session. The `u_dll.dll` library has a dynamic link to the `dbg_dll.dll` such that the `dbg_dll.dll` debug API can be used. It returns `UDLL_SUCCESS` if the initialization was successfully performed or `UDLL_INTERNAL_ERROR` otherwise.

U_DLL_API int udll_dbg_close(void);

The `udll_dbg_close` API function closes the `u_dll.dll` debug session. It should be called before the `u_dll.dll` library is unloaded, otherwise the debug resources will not be freed and memory leaks will exist. It returns `UDLL_SUCCESS` if the close was successfully performed or `UDLL_INTERNAL_ERROR` otherwise.

U_DLL_API int udll_set_prog_params(_udll_params *udll_params_t);

With the `udll_set_prog_params` API function the user can set the appropriate `u_dll.dll` programming parameters. The function parameters specify which firmware the `u_dll.dll` will download, whether it will erase or write any memory and what data to write to that memory. It returns `UDLL_SUCCESS` if the operation was successful. It returns `UDLL_PROG_PARAMS_ERROR` if any of the input parameter is invalid or `UDLL_INTERNAL_ERROR` otherwise. For example if the input parameter `udll_580_params_t->baud_rate` has an invalid baud rate setting, other than 9600, 57600, 115200 or 1000000 then the API function will return `UDLL_PROG_PARAMS_ERROR`. If input `udll_params_t` parameter is NULL, then the API function will return UDLL_INTERNAL_ERROR. The HTML help pages loaded after pressing the source\production_line_tool\help\help.html link contain more details on the API function input parameters.

U_DLL_API int udll_set_device_params(_udll_device_params *udll_device_params);

With the udll_set_device_params API function users can set the device parameters. Up to 16 devices are supported. The host application should use the following pre-processor definition for the maximum allowable devices:

```
#define MAX_UDLL_DEVICES             16
```

Device parameters are parameters that are specific to each device to be tested. Among others, the device parameters include the COM port, the baud rate, the Bluetooth Device (BD) address and a user callback function that will be called every time a process finishes for each device. The HTML help pages loaded after pressing the source\production_line_tool\help\help.html link contain more details on the API function input parameters. The function returns UDLL_SUCCESS if the set of the parameter was successfully performed. It returns UDLL_PROG_PARAMS_ERROR if any of the input parameter is invalid or UDLL_INTERNAL_ERROR otherwise.

U_DLL_API int udll_start_prog(void);

The udll_start_prog API function performs a specific DUT memory action. The action to perform was set when the udll_set_prog_params function was called. For example, if the fw_load action was configured when udll_set_prog_params function was called, the udll_start_prog will read the _udll_580_fw_load parameters. It will then get the firmware from the host PC, pointed by the fw_load.fw_path parameter (e.g. flash_programmer.bin, uartboot_bin, prod_test_580.bin, etc.) and download it into the system RAM of each DUT in parallel. The callback function user_callback_udll (see section 7.1.1.2) is set in udll_set_device_params function. It will be called for each device to report its status. Status code UDLL_FW_DOWNLOAD_SUCCESS (see section 7.2) indicates a successful completion of the firmware download. Other codes denote the successful or erroneous completion of a memory action (SPI read, OTP write, QSPI erase, etc.).

U_DLL_API int udll_close(void);

The udll_close API function should be called after the udll_star_prog function has finished. It will release the COM ports and free any resources acquired by the u_dll.dll operation. It returns UDLL_SUCCESS.

The U_DLL_API is defined as follows.

```
#define U_DLL_EXPORTS
#ifdef U_DLL_EXPORTS
#define U_DLL_API __declspec(dllexport)
#else
#define U_DLL_API __declspec(dllimport)
#endif
```

The declspec(dllexport) keyword automatically places the exported names to the .lib file during compilation. The .lib file can be used when a static DLL link is required. The declspec(dllimport) keyword is used in header files that the DLL in order to import DLL public data and objects.

7.1.1 U_DLL Function Input Arguments

Two U_DLL API functions take pointers to data structures as arguments. Function udll_set_prog_params and udll_set_device_params provide the necessary configuration setup for the u_dll.dll to operate. When the configuration has been successfully executed, the necessary operations will be performed by calling the udll_start_prog function.

The following sections describe the function parameters in detail.

7.1.1.1 Function udll_set_prog_params Input Arguments

The `udll_set_prog_params` function takes a pointer to the following union data structure, as argument:

```
typedef union __udll_params
{
    _u_dut_ic                   dut_ic;
    _udll_580_params            params_580;
    _udll_680_params            params_680;
}_udll_params;

typedef enum __u_dut_ic
{
    U_DUT_IC_DA14580                    = 0,
    U_DUT_IC_DA14581                    = 1,
    U_DUT_IC_DA14582                    = 2,
    U_DUT_IC_DA14583                    = 3,
    U_DUT_IC_DA14585                    = 4,
    U_DUT_IC_DA14586                    = 5,
    U_DUT_IC_DA14681_00                 = 6,
    U_DUT_IC_DA14681_01                 = 7,
    U_DUT_IC_DA14683_00                 = 8,
    U_DUT_IC_DA15101_00                 = 9,
    U_DUT_IC_INVALID                    = 10
}_u_dut_ic;
```

The above enumeration indicates the Dialog BLE chipset used. According to this value the `u_dll.dll` software either reads the `_udll_580_params` or the `_udll_680_params` data structures described next.

```
typedef struct __udll_580_params
{
    _u_dut_ic                   dut_ic;
    uint32_t                    baud_rate;
    _udll_580_mem_params        mem;
}_udll_580_params;
```

```
uint32_t baud_rate;
```

The `baud_rate` parameter value indicates the DA1458x UART baud rate during firmware download and memory programming. It supports 9600, 19200, 57600, 115200 and 1Mb baud rates.

```
_udll_580_mem_params mem;
```

The `mem` parameter is a union that stores the memory action parameters that the `u_dll.dll` will perform. One memory action can be performed at any given time by the `u_dll.dll`. The union contents are described next.

```
typedef union __udll_580_mem_params
{
    _UDLL_ACTIONS               action;
    _udll_580_fw_load           fw_load;
    _udll_580_fw_ver            fw_ver_get;
    _udll_580_otp_img           otp_img;
    _udll_580_otp_hdr           otp_hdr;
    _udll_580_otp_bda           otp_bda;
    _udll_580_spi_img           spi_img;
    _udll_580_spi_erase         spi_erase;
    _udll_580_spi_check_empty   spi_check_empty;
    _udll_580_eeprom_img        eeprom_img;
    _udll_mem_data              mem_data;
```

```
    _udll_mem_read              mem_read;
}_udll_580_mem_params;
```



```
typedef enum __UDLL_ACTIONS
{
    FW_LOAD                             = 0,
    FW_VERSION_GET,
    OTP_IMG_WRITE,
    OTP_HDR_WRITE,
    OTP_BDA_WRITE,
    OTP_XTAL_WRITE,
    OTP_BDA_READ,
    OTP_ADC_CALIB_WRITE,
    SPI_IMG_WRITE,
    SPI_ERASE,
    SPI_CHECK_EMPTY,
    EEPROM_IMG_WRITE,
    QSPI_IMG_WRITE,
    QSPI_ERASE,
    QSPI_CHECK_EMPTY,
    QSPI_BDA_WRITE,
    QSPI_XTAL_TRIM_WRITE,
    QSPI_BDA_READ,
    QSPI_ADC_CALIB_WRITE,
    MEM_DATA_WRITE,
    MEM_READ,
    RAM_FW_DOWNLOAD,
    INVALID_UDLL_ACTION
}_UDLL_ACTIONS;
```

These are the current operations the `u_dll.dll` supports. For each one of the operations a different data structure exists. Some of these actions are only supported by the DA1458x chipset and some by the DA1468x chipset. The DA1458x chipset does not support the QSPI actions, while the DA1468x chipset does not support the SPI and I2C/EEPROM actions.

Further comments on the memory operation data structures can be found in the actual `u_dll.dll` API header file found under `source\production_line_tool\core_dlls\u_dll\u_dll.h` or in the HTML pages opened by pressing the `source\production_line_tool\help\help.html` link.

7.1.1.2 Function udll_set_device_params Input Arguments

The `udll_set_device_params` function takes a pointer as argument to the following data structure.

```
typedef struct __udll_device_params
{
    bool                        is_active;
    _U_DUT_NUM                  dut_num;
    uint32_t                    com_port_boot;
    uint32_t                    com_port_prog;
    uint8_t                     bd_addr[BD_ADDR_SIZE];
    _OTP_customer_field         OTP_customer_field;
    uint8_t                     xtal_trim_val[XTAL_TRIM_SIZE];
    int16_t                     adc_calib_val;
    uint8_t                     mem_data[MAX_MEM_DATA_SIZE];
    _user_callback_udll         user_callback_udll;
}_udll_device_params;
```

```
bool is_active;
```

The `is_active` parameter enables or disables the `u_dll.dll` operations for the specific DUT.

```
_U_DUT_NUM dut_num;
```


The dut_num parameter indicates the DUT number that corresponds to the PLT hardware DUT connector.

```
uint32_t com_port_boot;
```

The com_port_boot parameter specifies the actual device Windows COM port.

```
uint32_t com_port_prog;
```

The com_port_prog parameter is used only by DA1458x chipsets. It specifies the Windows COM port of the DUT that will be used during memory programming. Two COM ports can be used with different sets of UART pins. The reason for this is to be able to use different pins between booting and SPI Flash or EEPROM memory programming.

During device boot the UART pins can only be among the sets shown in Table 11. When a memory is present at those pins, the programming of that memory may not be possible and a different set of UART pins may be required. This new set of UART pins will also have a different Windows COM port number, specified by the com_port_prog parameter.

Table 11: DA1458x UART Pins Selection

UART TX Pin	UART RX Pin
P0_0	P0_1
P0_2	P0_3
P0_4	P0_5
P0_6	P0_7

```
uint8_t bd_addr[BD_ADDR_SIZE];
```

The bd_addr parameter specifies the BD address to be written in the DUT's memory. For DA1458x chipsets the BD address is written into the OTP header memory space. It will only be written when the OTP_BDA_WRITE action is used and the appropriate parameters in the _udll_580_otp_bda data structure are filled in. For DA1468x chipsets the BD address can be written either to the QSPI or OTP memory spaces. For the OTP memory space the OTP_BDA_WRITE action should be used and the appropriate parameters in the _udll_680_otp_bda data structure should be filled in. For the QSPI memory, action QSPI_BDA_WRITE should be used and _udll_680_qspi_bda data structure should be filled in.

The BD_ADDR_SIZE is defined as:

```
#define BD_ADDR_SIZE        4
```

```
_OTP_customer_field OTP_customer_field[OTP_CUSTOMER_FIELD_SIZE];
```

The OTP_customer_field parameter is only used by DA1458x devices. It holds the data to be written in the DUT OTP customer header field. It was moved from the udll_prog_params data structure in order to support different OTP customer field per DUT.

```
typedef struct __OTP_customer_field
{
    uint8_t         data[OTP_585_CUSTOMER_FIELD_SIZE];
    uint16_t        size;
}_OTP_customer_field;
```

The OTP_585_CUSTOMER_FIELD_SIZE is defined as:

```
#define OTP_585_CUSTOMER_FIELD_SIZE      144
```

```
uint8_t xtal_trim_val[XTAL_TRIM_SIZE];
```

The xtal_trim_val parameter holds the XTAL trim value to be programmed to the DUTs. For DA1458x devices, the XTAL trim value can be written by the prod_test_580.bin during the automatic XTAL trim operation. In that case this variable will not be used. If the automatic XTAL trim operation is not used and users want to have the same XTAL trim value for all DUTs, then this variable needs to be filled in. In that case the XTAL trim value will be written when the OTP header is going to be burned, using the `OTP_HDR_WRITE` action.

For DA1468x devices the XTAL trim value is always written using the `u_dll.dll` and the `uartboot.bin` firmware, irrespectively if the value was edited manually or calculated using the automatic process.

```
int16_t adc_calib_val;
```

The `adc_calib_val` parameter is only used by DA14681-00 (AD) silicon based devices. It keeps the ADC gain calibration value for each device, calculated during the ADC gain calibration test process.

```
uint8_t mem_data[MAX_MEM_DATA_SIZE];
```

This array holds generic data to be written to a specific memory according to the memory action selected inside the `_udll_mem_data` data structure.

```
_user_callback_udll user_callback_udll;
```

The `user_callback_udll` parameter is a pointer to a user space application function that will be called whenever the `u_dll.dll` wants to update the DUT status. The function type is as follows:

```
typedef void(*_user_callback_udll)(int g_com_port_number, int status);
```

The `g_com_port_number` value indicates the COM port of the DUT for which the callback is made. The `status` value indicates whether a `u_dll.dll` operation was successful or failed (see section 7.2).

7.2 U_DLL Status Codes

The following list shows the U_DLL status codes, which are returned directly by the `u_dll.dll` API functions or added in the `status` parameter of the `user_callback_udll` function.

```
typedef enum __UDLL_RETURN_CODES
{
    UDLL_SUCCESS = 0,
    UDLL_ACTION_RESPONSE_ERROR,
    UDLL_UART_RX_TIMEOUT_ERROR,
    UDLL_NO_CRC_MATCH_ERROR,
    UDLL_PROG_PARAMS_ERROR,
    UDLL_DEVICE_PARAMS_ERROR,
    UDLL_UART_WRITE_ERROR,
    UDLL_UART_READ_ERROR,
    UDLL_INTERNAL_ERROR,
    UDLL_COM_PORT_INIT_ERROR,
    UDLL_COM_PORT_ERROR,
    UDLL_CANNOT_ALLOCATE_MEMORY,
    UDLL_READ_FILE_SIZE_ERROR,
    UDLL_CANNOT_OPEN_FW_FILE,
    UDLL_CANNOT_OPEN_IMAGE_FILE,
    UDLL_UART_PINS_PATCH_ERROR,
    UDLL_INVALID_DBG_PARAMS,
    UDLL_DBG_DLL_ERROR,
    UDLL_FW_DOWNLOAD_START,
    UDLL_FW_DOWNLOAD_SUCCESS,
    UDLL_FW_VERSION_GET_START,
    UDLL_FW_VERSION_GET_SUCCESS,
    UDLL_SPI_ERASE_START,
    UDLL_SPI_ERASE_SUCCESS,
    UDLL_SPI_CHECK_EMPTY_START,
    UDLL_SPI_CHECK_EMPTY_SUCCESS,
```

```
    UDLL_SPI_WRITE_START,
    UDLL_SPI_WRITE_SUCCESS,
    UDLL_SPI_WRITE_ERROR,
    UDLL_EEPROM_WRITE_START,
    UDLL_EEPROM_WRITE_SUCCESS,
    UDLL_EEPROM_WRITE_ERROR,
    UDLL_OTP_WRITE_START,
    UDLL_OTP_WRITE_SUCCESS,
    UDLL_OTP_WRITE_ERROR,
    UDLL_OTP_HEAD_WRITE_START,
    UDLL_OTP_HEAD_WRITE_SUCCESS,
    UDLL_OTP_HEAD_WRITE_ERROR,
    UDLL_OTP_BD_ADDR_WRITE_START,
    UDLL_OTP_BD_ADDR_WRITE_SUCCESS,
    UDLL_OTP_BD_ADDR_WRITE_ERROR,
    UDLL_OTP_BD_ADDR_READ_START,
    UDLL_OTP_BD_ADDR_READ_SUCCESS,
    UDLL_OTP_BD_ADDR_CMP_SUCCESS,
    UDLL_OTP_BD_ADDR_CMP_ERROR,
    UDLL_OTP_XTAL_TRIM_WRITE_START,
    UDLL_OTP_XTAL_TRIM_WRITE_SUCCESS,
    UDLL_OTP_XTAL_TRIM_WRITE_ERROR,
    UDLL_OTP_ADC_CALIB_WRITE_START,
    UDLL_OTP_ADC_CALIB_WRITE_SUCCESS,
    UDLL_OTP_ADC_CALIB_WRITE_ERROR,
    UDLL_OTP_CHECK_EMPTY_START,
    UDLL_OTP_CHECK_EMPTY_SUCCESS,
    UDLL_OTP_CHECK_SAME_DATA_SUCCESS,
    UDLL_OTP_CHECK_EMPTY_ERROR,
    UDLL_QSPI_WRITE_START,
    UDLL_QSPI_WRITE_SUCCESS,
    UDLL_QSPI_WRITE_ERROR,
    UDLL_QSPI_ERASE_START,
    UDLL_QSPI_ERASE_SUCCESS,
    UDLL_QSPI_CHECK_EMPTY_START,
    UDLL_QSPI_CHECK_EMPTY_SUCCESS,
    UDLL_QSPI_CHECK_EMPTY_ERROR,
    UDLL_QSPI_BD_ADDR_WRITE_START,
    UDLL_QSPI_BD_ADDR_WRITE_SUCCESS,
    UDLL_QSPI_BD_ADDR_WRITE_ERROR,
    UDLL_QSPI_BD_ADDR_READ_START,
    UDLL_QSPI_BD_ADDR_READ_SUCCESS,
    UDLL_QSPI_BD_ADDR_CMP_SUCCESS,
    UDLL_QSPI_BD_ADDR_CMP_ERROR,
    UDLL_QSPI_XTAL_TRIM_WRITE_START,
    UDLL_QSPI_XTAL_TRIM_WRITE_SUCCESS,
    UDLL_QSPI_XTAL_TRIM_WRITE_ERROR,
    UDLL_QSPI_ADC_CALIB_WRITE_START,
    UDLL_QSPI_ADC_CALIB_WRITE_SUCCESS,
    UDLL_QSPI_ADC_CALIB_WRITE_ERROR,
    UDLL_MEM_DATA_WRITE_START,
    UDLL_MEM_DATA_WRITE_SUCCESS,
    UDLL_MEM_DATA_WRITE_ERROR,
    UDLL_MEM_READ_START,
    UDLL_MEM_READ_SUCCESS,
    UDLL_RAM_FW_DOWNLOAD_START,
    UDLL_RAM_FW_DOWNLOAD_SUCCESS,
    UDLL_RAM_FW_DOWNLOAD_ERROR,
}_UDLL_RETURN_CODES;
```

7.3 U_DLL API Details

The U_DLL API function arguments, data structures, enumerations, return codes and other details can be found in the API header file `source\production_line_tool\core_dlls\u_dll\u_dll.h`, or in the HTML based help pages loaded after pressing the `source\production_line_tool\help\help.html` link.

7.4 U_DLL Operation Example

Below, a step-by-step example is given to illustrate how a user can set up and operate the U_DLL.

1. **U_DLL initialization**

   Function call: `U_DLL_API void udll_init(void);`

2. **U_DLL programming parameter setup**

   Function call: `U_DLL_API int udll_set_prog_params(_udll_params *udll_params_t);`

   ```
   In: typedef struct _udll_params
   Out: U_DLL status codes
   ```

   The user should fill the `udll_params` data structure with the appropriate parameters. When RF tests and XTAL trimming are required the `FW_LOAD` action should be used and depending of the device chipset either the `_udll_580_fw_load` or the `_udll_680_fw_load` data structures should be filled.

   Let us consider that a DA14580 device is going to be tested. A sample function is given next to show how the `udll_set_prog_params` function can be initialized with the appropriate parameters in order to download the `prod_test_580.bin` firmware.

```
int example_udll_prog_params(void)
{
    int ret = UDLL_SUCCESS;
    _udll_params udll_params;

    memset(&udll_params, 0, sizeof(_udll_params));

    /*  Set the appropriate parameters for the prod_test_580.bin firmware download
        to a DA14580 device with UART baud rate at 115200 and UART GPIO pins
        TX=P0_4, RX=P0_5.
    */
    udll_params.dut_ic = U_DUT_IC_580;
    udll_params.params_580.baud_rate = 115200;
    udll_params.params_580.dut_ic = U_DUT_IC_580;
    udll_params.params_580.mem.action = FW_LOAD;
    udll_params.params_580.mem.fw_load.action = FW_LOAD;
    udll_params.params_580.mem.fw_load.en = true;
    strcpy(udll_params.params_580.mem.fw_load.fw_path, "prod_test_580.bin");
    udll_params.params_580.mem.fw_load.uart_boot_pins = P04_P05;
    udll_params.params_580.mem.fw_load.uart_change_pins = false;
    udll_params.params_580.mem.fw_load.uart_pins.uart_port_tx = 0;
    udll_params.params_580.mem.fw_load.uart_pins.uart_pin_tx = 4;
    udll_params.params_580.mem.fw_load.uart_pins.uart_port_rx = 0;
    udll_params.params_580.mem.fw_load.uart_pins.uart_pin_rx = 5;

    ret = udll_set_prog_params(&udll_params);
    if (ret != UDLL_SUCCESS) {
        printf("Error in udll_set_prog_params with return code [%d].\n", ret);
        return -1;
    }
    return 0;
```



```
}
```

3. **U_DLL device parameter setup**

Function call: `U_DLL_API int udll_set_device_params(_udll_device_params *udll_device_params);`

```
In: typedef struct _udll_device_params
Out: U_DLL status codes
```

The next step is to set the appropriate device parameters in the U_DLL by using the `udll_set_device_params` function. Let's consider that we have 3 devices active connected in the PLT DUT connectors 1, 2 and 16. We will need to provide the device parameters for these three devices. Next, an example code is given.

```
int example_udll_device_params(void)
{
    int ret = UDLL_SUCCESS;
    int i = 0;
    _udll_device_params udll_device_params;

    memset(&udll_device_params, 0, sizeof(_udll_device_params));

     /* Set the device parameters for all 16 DUTs.
        Activate only DUTs 1, 2 and 16.
        The rest of the DUTs should be set to false.
        The DUTs COM ports and BD addresses are stored in the com_port
        and bd_addr tables respectively.
        Also, the device_callback_udll function is used as a callback function
        to get the DUT results during operations.
        Variables com_port_boot and com_port_prog have the same COM port as
        we will not change ports during memory programming. */

    for (i=0; i<MAX_UDLL_DEVICES; i++)
    {
        memset(&udll_device_params, 0, sizeof(_udll_device_params));

        if ((i == 0) || (i == 1) || (i == 15))
            udll_device_params.is_active = true;
        else
            udll_device_params.is_active = false;

        udll_device_params.dut_num = (_U_DUT_NUM) i+1;
        udll_device_params.com_port_boot = com_port[i];
        udll_device_params.com_port_prog = com_port[i];
        memcpy(udll_device_params.bd_addr, bd_addr[i], BD_ADDR_SIZE);
        udll_device_params.user_callback_udll = device_callback_udll;
        ret = udll_set_device_params(&udll_device_params);
        if (ret != UDLL_SUCCESS) {
            printf("Error in udll_set_device_params with ret code [%d].\n", ret);
            return -1;
        }
    }
    return 0;
}
```

4. **U_DLL start programming**

Function call: `U_DLL_API int udll_start_prog(void);`

Calling this function will start the U_DLL operation. The U_DLL will perform the following steps.

a. Download the firmware indicated by `the udll_params.params_580.mem.fw_load.fw_path` parameter to the active DUTs system RAM.

b. Return a status code per DUT to indicate whether the firmware download was successful. The status code is returned through the device callback function. The callback function pointer was initialized as described in section 7.1.1.2.

c. Return a status code per DUT to indicate whether the operation was successful or not. The status code is returned through the device callback function. The callback function pointer was initialized as described in section 7.1.1.2.

5. U_DLL close

Function call: `U_DLL_API int udll_close(void);`

```
In: void
Out: U_DLL status codes
```

Calling this function will release the COM ports and the U_DLL resources.

8 P_DLL

The P_DLL dynamic link library provides the necessary API functionality to perform basic device tests, such as RF tests, crystal (XTAL) frequency trimming, audio tests, GPIO tests, sensor tests, or BLE scan tests. The p_dll.dll can set the DUTs in the following RF test modes: continuous packet TX, continuous RX or normal BLE central scanner. In continuous RX operation the DUT can report packet reception statistics and RSSI values.

For the p_dll.dll to be operational the production test firmware (prod_test_580.bin, prod_test_581.bin, prod_test_582.bin, prod_test_585.bin, prod_test_681_00.bin, prod_test_681_01.bin or prod_test_683_00.bin) must be downloaded to the DUT's system RAM. This can be done using the u_dll.dll commands described in section 7.

File source\production_line_tool\core_dlls\p_dll\p_dll.h contains all necessary API information. It can be included as is in any user project.

8.1 P_DLL API Functions

P_DLL has the following user accessible functions:

```
P_DLL_API int pdll_init(void);
P_DLL_API int pdll_dbg_init(_dbg_params *dbg_params_t);
P_DLL_API int pdll_dbg_close(void);
P_DLL_API int pdll_set_device_params(_pdll_device *pdll_device_t);
P_DLL_API int pdll_perform_test(_pdll_test_id test_id);
```

A short description of each API function follows.

P_DLL_API void pdll_init(void)

The pdll_init API function initializes the p_dll.dll. It should be called before any other operation with the p_dll.dll library. It always returns PDLL_NO_ERROR.

P_DLL_API int pdll_dbg_init(_dbg_params *dbg_params_t);

The pdll_dbg_init API function initializes the p_dll.dll debug print session. The p_dll.dll has a dynamic link to the dbg_dll.dll such that the debug API is available to be used. The function will return PDLL_NO_ERROR if no errors were reported, PDLL_DBG_DLL_ERROR if an error occurred during the dbg_dll.dll dynamic linking or PDLL_INVALID_DBG_PARAMS if the dbg_params_t contains invalid parameters.

P_DLL_API int pdll_dbg_close(void);

The pdll_dbg_close API function closes the p_dll.dll debug session. It should be called before the p_dll.dll library is unloaded, otherwise the debug resources will not be freed and memory leaks will exist. The function will return PDLL_NO_ERROR if no errors were reported or PDLL_DBG_DLL_ERROR if an error occurred during dbg_dll.dll unloading.

P_DLL_API int pdll_set_device_params(_pdll_device *pdll_device_t)

With the pdll_set_device_params API function users can set a single device's parameters. Up to 16 devices are supported. Therefore, the host application should call this function for as many active devices exist in order to set the parameters for all the devices to be tested.

The device parameters include the COM port, the UART baud rate, the frequency of the RF tests, the type of the RF test (continuous packet TX, continuous RX or even scan tests), the XTAL trim input reference pulse GPIO, a user callback function that will be called every time a process finishes and other values necessary to perform specific tests. Please check section 8.1.1 for a full description of the input parameters. The function returns PDLL_NO_ERROR is no errors occurred, PDLL_PARAMS_ERROR if a pdll_device_t input parameter is not valid or PDLL_CANNOT_ALLOCATE_MEMORY if no memory can be allocated to store the new device parameters.

P_DLL_API int pdll_perform_test(_pdll_test_id test_id)

By calling the pdll_perform_test API function all the devices that have been set up (when the pdll_set_device_params function was called) will start operating with the test specified from the enumeration parameter pdll_test_id. It sends test progress results to the upper layer software using callbacks. The callback function is given in pdll_device_t->user_callback_pdll when the pdll_set_device_params function is called.

The P_DLL_API is defined as follows.

```
#define P_DLL_EXPORTS
#ifdef P_DLL_EXPORTS
#define P_DLL_API __declspec(dllexport)
#else
#define P_DLL_API __declspec(dllimport)
#endif
```

The declspec(dllexport) keyword automatically places the exported names to the .lib file during compilation. The .lib file can be used when a static DLL link is required. The declspec(dllimport) keyword is used in header files that the DLL in order to import DLL public data and objects.

8.1.1 P_DLL Function Input Arguments

The P_DLL function parameters include a data structure and an enumeration. The data structure specifies the device test parameters and the enumeration the test to be performed.

In the next sections the function parameters are described in detail.

8.1.1.1 Function pdll_set_device_params Input Arguments

The pdll_set_device_params function takes a pointer as argument to the following data structure:

```
typedef struct __pdll_device
{
    bool                       is_active;
    _p_dut_ic                  dut_ic;
    uint32_t                   com_port_boot;
    uint32_t                   com_port_prog;
    _uart_pins                 uart_pins;
    uint32_t                   baud_rate;
    _user_callback_pdll        user_callback_pdll;
    _pdll_test_data            test;
}_pdll_device;
```

```
bool is_active;
```

The is_active parameter enables or disables the device under test.

```
_p_dut_ic dut_ic;
```

The dut_ic parameter contains the type of the device to be tested. The following enumeration shows the valid options for this parameter.

```
typedef enum __p_dut_ic
{
    P_DUT_IC_DA14580            = 0,
    P_DUT_IC_DA14581            = 1,
    P_DUT_IC_DA14582            = 2,
    P_DUT_IC_DA14583            = 3,
    P_DUT_IC_DA14585            = 4,
```

```
    P_DUT_IC_DA14586          = 5,
    P_DUT_IC_DA14681_00       = 6,
    P_DUT_IC_DA14681_01       = 7,
    P_DUT_IC_DA14683_00       = 8,
    P_DUT_IC_DA15101_00       = 9,
    P_DUT_IC_INVALID          = 10
}_p_dut_ic;
```

```
uint32_t com_port_boot;
```

The com_port_boot parameter specifies the actual device Windows COM port.

```
uint32_t com_port_prog;
```

The com_port_prog parameter specifies the device Windows COM port that will be used during tests. P_DLL has the option to use two different sets of UART pins. It can only be used in DA1458x devices.

During booting the UART pins can only be among the ones shown in Table 11 for DA1458x devices. However, these pins may also be connected to other peripherals that need to interact with during tests. For example, the audio codec integrated in DA14582 is using the GPIO P0_5 as 16MHz input clock source. So, P0_5 GPIO cannot be used if we need to test the audio. Therefore, the p_dll.dll software will issue a command to the production test firmware to change the UART GPIO pins to another pair. The second pair of DUT GPIOs should be connected to the next PLT hardware DUT connection (e.g. if first DUT UART is connected to PLT DUT 11, the second should be connected to PLT DUT connection 12), which eventually has a different com port number. So, the new set of UART pins will also have a different Windows COM port number as well, specified by the com_port_prog parameter.

```
_uart_pins uart_pins;
```

The uart_pins data structure holds the second UART GPIO pins as described above. The data structure is the following:

```
typedef struct __uart_pins
{
        uint8_t uart_port_tx;
        uint8_t uart_pin_tx;
        uint8_t uart_port_rx;
        uint8_t uart_pin_rx;
}_uart_pins
```

```
uint32_t baud_rate;
```

The baud_rate parameter specifies the baud rate for the UART communication between the p_dll.dll and the DUT. Currently, this parameter only supports a value of 115200 and users should use the following definition when setting the p_dll.dll baud rate.

```
#define PDLL_UART_BAUD_RATE          115200
```

```
_user_callback_pdll user_callback_pdll;
```

The user_callback_pdll parameter is a pointer to a user application function that will be called whenever the p_dll.dll wants to update the DUT status.

The function type and data structure are defined as follows:

```
typedef void (*_user_callback_pdll) (int com_port_number, int status, _rx_stats
*rx_stats_t);
```

- The com_port_number holds the COM port of the device. It is used as an index to indicate for which device the callback belongs.

- The status value indicates whether a p_dll.dll operation was successful or failed. The status value can be one among _PDLL_RETURN_CODES.
- The rx_stats_t value is a pointer to a structure that contains the test results. Table 12 gives a more detailed explanation of the rx_stats data structure.


```
typedef struct __rx_stats
{
    uint32_t        pkts_crc_ok;
    uint32_t        pkts_sync_err;
    uint32_t        pkts_crc_err;
    uint16_t        xtal_trim_val;
    uint8_t         custom_data;
    float           rssi;
    uint8_t         periph_bd_addr[MAX_DEVS_TO_SCAN][BD_ADDR_SIZE];
   _pdll_fw_versions pdll_fw_versions;
}_rx_stats;
```

Table 12: rx_stats Callback Parameters

Parameter	Description
`uint32_t status;`	Contains the `p_dll.dll` return status codes.

`uint32_t pkts_crc_ok;`
`uint32_t pkts_sync_err;`
`uint32_t pkts_crc_err;`
	Contains the received packet statistics when the device operates in the `start_pkt_rx_stats` mode.

`uint16_t xtal_trim_val;`	Returns the XTAL trimming value calculated during the `xtal_trim` test operation.
`uint8_t custom_data;`	Returns the `custom_data` during the `custom_test` operation.
`float rssi;`	Returns the average RSSI found when the device operates in the `start_pkt_rx_stats` mode.
`uint8_t periph_bd_addr[MAX_DEVS_TO_SCAN][BD_ADDR_SIZE];`	Contains the BD addresses found during the `start_scan` test. The `p_dll.dll` will only return BD addresses that were initially passed to the `_pdll_device` structure.
`_pdll_fw_versions pdll_fw_versions;`	The `p_dll.dll` and production test firmware versions for the Golden Unit and the devices under test.

`_pdll_test_data test;`

The `test` parameter is a union of data structures. Each data structure holds the settings of each test. The format of the union is shown next.

```
typedef union __pdll_test_data
{
    _pdll_test_id               id;
    _pdll_rf_test               rf;
    _pdll_audio_test            audio;
    _pdll_audio_tone            audio_tone;
    _pdll_custom_test           custom;
    _pdll_rdtester              rdtester;
    _pdll_scan_test             scan;
    _pdll_xtal_test             xtal_trim;
    _pdll_gpio_toggle           gpio_toggle;
    _pdll_adc_read              adc_read;
    _pdll_sensor_test           sensor;
    _pdll_otp_xtal_trim_read    otp_xtal_trim_read;
    _pdll_sleep_test            sleep;
    _pdll_uart_loop_test        uart_loop;
}_pdll_test_data;
```

More details on the test data structures can be found in the `p_dll.dll` API include file found under `production_line_tool\core_dlls\p_dll\p_dll.h`. Here a brief description will be given.

```
_pdll_test_id id;
```

The `id` parameter is an enumeration indicating the `p_dll.dll` operation to be performed. The following shows the current supported operations.

```
typedef enum __pdll_test_id
{
    dut_com_init                = 0,
    cont_pkt_tx                 = 1,
    pkt_tx                      = 2,
    stop_pkt_tx                 = 3,
    start_pkt_rx_stats          = 4,
    stop_pkt_rx_stats           = 5,
    custom_test                 = 6,
    xtal_trim                   = 7,
    xtal_trim_val_read          = 8,
    scan_test                   = 9,
    rdtester_init               = 10,
    rdtester_uart_connect       = 11,
    rdtester_uart_loop          = 12,
    rdtester_vbat_uart_ctrl     = 13,
    rdtester_vpp_ctrl           = 14,
    rdtester_rst_pulse          = 15,
    rdtester_xtal_pulse_uart    = 16,
    rdtester_xtal_pulse         = 17,
    rdtester_pulse_width        = 18,
    uart_resync                 = 19,
    audio_test                  = 20,
    audio_tone                  = 21,
    gpio_toggle                 = 22,
    adc_read                    = 23,
    sensor_test                 = 24,
    otp_xtal_trim_read          = 25,
    rdtester_vbat_ctrl          = 26,
    sleep                       = 27,
    uart_loop                   = 28,
    INVALID_TEST                = 29
}_pdll_test_id;
```

Table 13: P_DLL Supported Operations

Command	Description
dut_com_init	Initializes the DUT COM ports.
cont_pkt_tx	Direct Test Mode (DTM) continuous ‘packet transmit’ command as specified by the BLE Core standard.
pkt_tx	Dialog custom ‘packet transmit’ command. Transmits a specific number of packets.
stop_pkt_tx	Direct Test Mode (DTM) command to stop the continuous packet transmission, as specified by the BLE Core standard.
start_pkt_rx_stats	Dialog custom packet reception command. Receives packets from a specific BLE channel and returns extended statistics that include the RSSI level of the received signal.
stop_pkt_rx_stats	Dialog custom command to stop the custom packet reception operation (start_pkt_rx_stats).
custom_test	A custom test command where users can add their own tests.
xtal_trim	Automatic XTAL trim operation.

Command	Description
xtal_trim_val_read	Returns the value found from the automatic XTAL trim operation.
scan_test	BLE scan test. Usually performed by the GU to scan for the DUTs.
rdtester_init	PLT hardware CPLD initialization.
rdtester_uart_connect	PLT hardware, CPLD. Enable the UART connection between the PLT DUT connector and the FTDI. The UART lines go through the CPLD so they can be disconnected to avoid current leakages thus having correct power on reset operation.
rdtester_uart_loop	PLT hardware, CPLD. Enable the UART loopback mode in the CPLD. The PLT software sends a word to a specific DUT UART and the CPLD echoes that word back to the PLT application. Using this procedure, the PLT application can identify which Windows COM port has its DUT.
rdtester_vbat_ctrl	PLT - CPLD. Enable/disable the DUT VBAT at DUT connector pin 1.
rdtester_vpp_ctrl	PLT - CPLD. Enable/disable VPP at DUT connector pin 8 for OTP burn.
rdtester_rst_pulse	PLT hardware, CPLD. DUT reset pulse at PLT DUT connector pin 10.
rdtester_xtal_pulse_uart	PLT hardware, CPLD. Sends a reference pulse in the DUT UART RX pin 9 for XTAL trim calibration.
rdtester_xtal_pulse	PLT hardware, CPLD. Sends a reference pulse in the PLT DUT connector pin 2 for XTAL trim calibration.
rdtester_pulse_width	PLT hardware, CPLD. Sends a command to the CPLD to set the size of the XTAL trim reference pulse.
uart_resync	Sends a UART resync command to the DUT for UART resynchronization. If the XTAL trim pulse is given in the DUT UART RX pin, the DUT UART loses synchronization and returns frame errors. This P_DLL command will send characters 'RW!' to the DUT, which will resync the UART controller.
audio_test	Only available in DA14582 devices. It places the DUT in audio test mode. The DUT will send captured audio data to the PLT software through UART.
audio_tone	Play an audio tone. This command is send to the Golden Unit to start playing a 4KHz audio tone. The DUTs previously configured for audio_test, will start listen to that tone.
gpio_toggle	This test toggles a specific DUT GPIO.
adc_read	DA14681-00 ADC read samples, used in ADC gain calibration procedure.
sensor_test	DA14580, DA14581, DA14582 and DA14583 peripheral sensor testing. It can test sensors by reading their ID. It can also test interrupt and data ready (DRDY) GPIOs.
otp_xtal_trim_read	Reads the OTP XTAL trim value in order not to overwrite it in case of device retesting.
sleep	Sets the device into Deep or Extended sleep. Used for current measurements.
uart_loop	Sets the device into UART loop mode. Whatever data it receives the device will send them back to the user. This test is used for characterizing the PLT to DUT physical connections.

`_pdll_rf_test rf;`

The `rf` data structure holds the parameters for the supported RF tests. The `start_pkt_rx_stats` and `pkt_tx p_dll.dll` library actions are using this data structure.

`_pdll_audio_test audio;`

The `audio` data structure holds the parameters for the audio test supported only by the DA14582 devices.

```
_pdll_audio_tone audio_tone;
```

The `audio_tone` data structure holds the parameters for generating an audio tone. It is used by the Golden Unit to generate a 4 kHz tone for the DA14582 audio test.

```
_pdll_custom_test custom;
```

The `custom` data structure holds the data of a generic custom test. The custom test sends a single byte to the DUT running the production test firmware. The firmware gets this byte and sends it back to the PLT host application. PLT produces a success if the data byte received is the same as the one it sent. Customers can add their own test in the production test firmware and trigger it using the `custom_test` operation.

```
_pdll_rdtester rdtester;
```

The `rdtester` data structure holds information for the Golden Unit CPLD control.

```
_pdll_scan_test scan;
```

The `scan` data structure holds scan test device parameters. The scan test is supported in both DA1458x and DA1468x devices. For the test to succeed the BD addresses should have been burned in the appropriate memory location such that the devices to start advertise with the BD address given by the PLT.

```
_pdll_xtal_test xtal_trim;
```

The `xtal_trim` data structure holds the parameters for the XTAL calibration procedure.

```
_pdll_gpio_toggle gpio_toggle;
```

The `gpio_toggle` data structure holds the parameters for the GPIO/LED test procedure.

```
_pdll_adc_read adc_read;
```

The `adc_read` data structure holds the parameters for the DA14681-00 ADC gain calibration procedure.

```
_pdll_sensor_test sensor;
```

The `sensor` data structure holds the parameters required for DA1458x peripheral sensor testing.

```
_pdll_otp_xtal_trim_read otp_xtal_trim_read;
```

The `otp_xtal_trim_read` data structure holds the parameters required to read the XTAL trim value from the OTP memory.

```
_pdll_sleep_test sleep;
```

The `sleep` data structure holds the parameters required for setting the device into sleep mode, used for current measurements.

8.1.1.2 Function pdll_perform_test Input Arguments

The function `pdll_perform_test` takes as argument the enumeration described in Table 13.

As already described, this enumeration indicates the actual test to be performed. Example: when calling the function `pdll_perform_test(cont_pkt_tx)`, all the devices with `pdll_device_params.pdll_prod_test = cont_pkt_tx` will start transmitting.

8.2 P_DLL Status Codes

The following list shows the `p_dll.dll` status codes, which are returned directly by the DLL API functions or added in the status parameter of the `user_callback_pdll` function.

```
typedef enum __PDLL_RETURN_CODES
{
    PDLL_NO_ERROR =                         0,
    PDLL_PARAMS_ERROR,
```

```
PDLL_RX_TIMEOUT,
PDLL_TX_TIMEOUT,
PDLL_UNEXPECTED_EVENT,
PDLL_CANNOT_ALLOCATE_MEMORY,
PDLL_INTERNAL_ERROR,
PDLL_THREAD_CREATION_ERROR,
PDLL_DBG_DLL_ERROR,
PDLL_INVALID_DBG_PARAMS,
PDLL_COM_PORT_START,
PDLL_COM_PORT_OK,
PDLL_COM_PORT_FAILED,
PDLL_FW_VERSION_GET_START,
PDLL_FW_VERSION_GET_OK,
PDLL_RDTESTER_INIT_START,
PDLL_RDTESTER_INIT_OK,
PDLL_RDTESTER_UART_CONNECT_START,
PDLL_RDTESTER_UART_CONNECT_OK,
PDLL_RDTESTER_UART_LOOPBACK_START,
PDLL_RDTESTER_UART_LOOPBACK_OK,
PDLL_RDTESTER_VBAT_CNTRL_START,
PDLL_RDTESTER_VBAT_CNTRL_OK,
PDLL_RDTESTER_VPP_CNTRL_START,
PDLL_RDTESTER_VPP_CNTRL_OK,
PDLL_RDTESTER_RST_PULSE_START,
PDLL_RDTESTER_RST_PULSE_OK,
PDLL_RDTESTER_UART_PULSE_START,
PDLL_RDTESTER_UART_PULSE_OK,
PDLL_RDTESTER_XTAL_PULSE_START,
PDLL_RDTESTER_XTAL_PULSE_OK,
PDLL_RDTESTER_PULSE_WIDTH_START,
PDLL_RDTESTER_PULSE_WIDTH_OK,
PDLL_RDTESTER_INVALID_COMMAND,
PDLL_XTAL_TRIM_START,
PDLL_XTAL_TRIM_OK,
PDLL_XTAL_TRIM_OUT_OF_RANGE,
PDLL_XTAL_TRIM_FREQ_CAL_NOT_CONNECTED,
PDLL_XTAL_TRIM_OTP_WRITE_FAILED,
PDLL_XTAL_TRIM_READ_START,
PDLL_XTAL_TRIM_READ_OK,
PDLL_UART_RESYNC_START,
PDLL_UART_RESYNC_OK,
PDLL_UART_RESYNC_FAILED,
PDLL_CONT_PKT_TX_START,
PDLL_CONT_PKT_TX_STARTED_OK,
PDLL_HCI_TEST_STOP_START,
PDLL_HCI_TEST_STOPPED_OK,
PDLL_PKT_TX_START,
PDLL_PKT_TX_STARTED_OK,
PDLL_PKT_TX_ENDED_OK,
PDLL_PKT_RX_STATS_START,
PDLL_PKT_RX_STATS_STARTED_OK,
PDLL_PKT_RX_STATS_STOP_START,
PDLL_PKT_RX_STATS_STOPPED_OK,
PDLL_PKT_RX_START,
PDLL_PKT_RX_STARTED_OK,
PDLL_CUSTOM_ACTION_START,
PDLL_CUSTOM_ACTION_OK,
PDLL_BLE_SCAN_START,
PDLL_BLE_SCAN_OK,
PDLL_AUDIO_TONE_START,
```



```
    PDLL_AUDIO_TONE_STARTED_OK,
    PDLL_AUDIO_TONE_STOP,
    PDLL_AUDIO_TONE_STOPPED_OK,
    PDLL_AUDIO_TEST_START,
    PDLL_AUDIO_TEST_ALREADY_ACTIVE,
    PDLL_AUDIO_TEST_STARTED_OK,
    PDLL_AUDIO_TEST_STOP,
    PDLL_AUDIO_TEST_STOPPED_OK,
    PDLL_AUDIO_TEST_PASSED,
    PDLL_AUDIO_TEST_FAILED,
    PDLL_AUDIO_TEST_INVALID_COMMAND,
    PDLL_GPIO_TOGGLE_START,
    PDLL_GPIO_TOGGLE_FINISHED_OK,
    PDLL_ADC_READ_START,
    PDLL_ADC_READ_OK,
    PDLL_SENSOR_TEST_START,
    PDLL_SENSOR_TEST_OK,
    PDLL_OTP_XTAL_TRIM_READ_START,
    PDLL_OTP_XTAL_TRIM_READ_OK,
    PDLL_SLEEP_START,
    PDLL_SLEEP_OK,
    PDLL_UART_LOOP_START,
    PDLL_UART_LOOP_FAILED,
    PDLL_UART_LOOP_OK,
}_PDLL_RETURN_CODES;

#define PDLL_HCI_STANDARD_ERROR_CODE_BASE        1000
```

The PDLL_HCI_STANDARD_ERROR_CODE_BASE return code is used when the production test firmware returns a standard HCI error code message. The P_DLL will add the standard HCI error code returned by the firmware to the PDLL_HCI_STANDARD_ERROR_CODE_BASE. The final return code value will be between 1000 and 1063.

8.3 P_DLL API Details

P_DLL API function arguments, data structures, enumerations, return codes and other details can be found in the API header file source\production_line_tool\core_dlls\p_dll\p_dll.h or in the HTML based help pages loaded after pressing the source\production_line_tool\help\help.html link.

8.4 P_DLL Operation Example

Below, a step-by-step example is given to briefly illustrate how users could set up and operate the P_DLL. It is assumed that the prod_test_580.bin firmware has already been downloaded to the DUT's system RAM using the U_DLL procedure.

8.4.1 Simple RX-TX Operation Example

1. **P_DLL initialization**

   Function call: `P_DLL_API int pdll_init(void);`

2. **P_DLL device parameter setup**

   Function call: `P_DLL_API int pdll_set_device_params(_pdll_device *pdll_device_t);`

   ```
   In: typedef struct __pdll_device
   Out: P_DLL status codes
   ```

   This function should be called once for every device to be tested. To set up a simple RX-TX test using two DUTs, this function should be called twice: once for each device. The first device should be set to cont_pkt_tx and the second to start_pkt_rx_stats.

3. **P_DLL start testing**

   Function call: `P_DLL_API int pdll_perform_test(_pdll_prod_tests pdll_prod_test);`

   ```
   In: typedef enum __pdll_prod_tests
   Out: P_DLL status codes
   ```

   Calling this function will instruct all DUTs with the specified `pdll_prod_test` to start the test. The P_DLL will then call the `user_callback_pdll` function to report the status of each device.

   For a simple RX-TX test this function should be called twice. Once to start the DUTs that have `pdll_device_params.test.id = cont_pkt_tx` and a second time for the DUTs that have `pdll_device_params.test.id = start_pkt_rx_stats;`

   These two calls will be as follows:

   ```
   pdll_perform_test(start_pkt_rx_stats);
   pdll_perform_test(cont_pkt_tx);
   ```

4. **P_DLL stop testing**

   Function call: `P_DLL_API int pdll_perform_test(_pdll_test_id test_id);`

   ```
   In: typedef enum __pdll_prod_tests
   Out: P_DLL status codes
   ```

   Calling this function will instruct each DUT with the specified `pdll_prod_test` to stop the test. The P_DLL will then call the `user_callback_pdll` function to report the status of each device. When the device operates in `start_pkt_rx_stats` mode, the user callback function will return with the RX statistics structure filled. The two function calls to stop a simple RX-TX test are:

   ```
   pdll_perform_test(stop_pkt_tx);
   pdll_perform_test(stop_pkt_rx_stats);
   ```

   Note: Some tests terminate without requiring an extra command to be received. For example, the `pkt_tx` test will send a specific number of packets and then terminate. In that case, the user does not have to send any special command to stop the test.

5. **P_DLL re-initialization**

   Function call: `P_DLL_API int pdll_init(void);`

   The `pdll_init` function should be called again at the end of the tests to release the COM ports and all other P_DLL resources.

8.4.2 Scan Operation Example

The following procedure explains how to perform a scan test on a single device. This test is actually meant to be performed by a Golden Unit device in order to scan for advertising peripheral DUTs.

1. **P_DLL initialization**

   Function call: `P_DLL_API int pdll_init(void);`

2. **P_DLL device parameter setup**

   Function call: `P_DLL_API int pdll_set_device_params(_pdll_device *pdll_device_t);`

   ```
   In: typedef struct __pdll_device
   Out: P_DLL status codes.
   ```

   This function should be called once for the GU device. Two important parameters should be set in the `pdll_device_params` structure: `pdll_prod_test` and `periph_bd_addr`.

   A code snippet could be as follows:

   ```
   _pdll_device pdll_device;
   pdll_device_params.is_active = 1;
   pdll_device_params.dut_ic = P_DUT_IC_DA14580;
   pdll_device_params.com_port_boot = 10; // The GU com port.
   pdll_device_params.com_port_prog = 10; // The GU com port.
   pdll_device_params.baud_rate = 115200;
   pdll_device_params.user_callback_pdll = user_callback_pdll;
   pdll_device_params.test.id = start_scan;
   // fill in the bd addresses to be found from a local array.
   ```



```
for (i=0; i<MAX_DEVS_TO_SCAN; i++) {
       memcpy(pdll_device_params.test.periph_bd_addr[i], bd_addr[i], BD_ADDR_SIZE);
}
pdll_set_device_params(&pdll_device_params);
```

3. **P_DLL start test**

   Function call: `P_DLL_API int pdll_perform_test(_pdll_test_id test_id);`

   `In: typedef enum __pdll_test_id`
   `Out: P_DLL status codes`

   Calling this function will instruct each DUT with the specified `pdll_prod_test` to start the test. From step 2 onwards, only the GU should be set to perform the scan test. The P_DLL will then call the `user_callback_pdll` to report the status of the BD addresses that were found.

   The following function call will start the scanning test:

   `pdll_perform_test(start_scan);`

   No stop command is required. The `user_callback_pdll` callback function will be called as soon as the BD addresses are found. In case not all addresses are returned to the result parameter (`rx_stats.periph_bd_addr`) of the callback function, steps 1 to 4 can be repeated.

4. **P_DLL re-initialization**

   Function call: `P_DLL_API int pdll_init(void);`

   The `pdll_init` function should be called again at the end of the tests to release the COM ports and all other P_DLL resources.

9 PROD_LINE_TOOL_DLL

The prod_line_tool_dll.dll, hereafter called PLTD, is a top level DLL that uses most of the other DLLs to run all appropriate device tests. It actually combines various state machine actions for each test to be performed under a simple API. Additionally, it is responsible for creating logs for every device under test as well as the golden unit.

File source\production_line_tool\core_dlls\prod_line_tool_dll\prod_line_tool_dll.h contains all necessary API information. It can be included as is in any user project.

9.1 PROD_LINE_TOOL_DLL API Functions

PLTD has the following user accessible functions:

```
PLTD_API int pltd_init(int gu_com);
PLTD_API void pltd_close(void);
PLTD_API int pltd_set_device_params(_pltd_device_params *pltd_device_params_t);
PLTD_API int pltd_set_general_params(_pltd_general_params *pltd_general_params_t);
PLTD_API int pltd_start(void);
PLTD_API int pltd_com_port_enum(uint32_t *com_port_dut);
PLTD_API int pltd_GU_com_find(int *gu_com_port);
PLTD_API int pltd_GU_check_LED(void);
PLTD_API int pltd_dbg_init(_pltd_dbg_params *pltd_dbg_params_t);
PLTD_API char *pltd_get_volt_meter_instr_names(char *prev_name);
PLTD_API char *pltd_get_ble_tester_instr_names(char *prev_name);
PLTD_API char *pltd_get_ammeter_instr_names(char *prev_name);
PLTD_API char *pltd_get_temp_meas_instr_names(char *prev_name);
PLTD_API int pltd_vbat_uart_set(bool start, uint16_t duts);
PLTD_API int pltd_uart_coms_test(_pltd_uart_test *uart_test);
PLTD_API _pltd_versions *pltd_get_versions(void);
```

A short description of each API function follows.

PLTD_API int pltd_init(int gu_com)

The pltd_init API function initializes the PLTD library. It should be called before any other operation with the PLTD library. This function returns PLTD_ERROR if a failure occurs or PLTD_SUCCESS otherwise.

PLTD_API void pltd_close(void)

The pltd_close API function should be called after the pltd_start function has returned. It will close the u_dll.dll and p_dll.dll libraries; close all open handles and free any acquired resources. This API function returns PLTD_ERROR if a failure occurs or PLTD_SUCCESS otherwise.

PLTD_API int pltd_set_device_params(_pltd_device_params *pltd_device_params_t)

With the pltd_set_device_params API function users can set the parameters for all DUTs. Up to 16 devices are supported. The host application should use the following pre-processor definition for the maximum allowable devices:

```
#define PDLT_MAX_DEVICES            16
```

The pltd_device_params_t parameters are specific for each DUT. This function should be called once for each of the 16 devices, even if the device is disabled. The parameters are explained in detail in section 9.1.1. This API function returns PLTD_WRONG_DEV_PARAMS when an input parameter is invalid. It returns PLTD_ERROR in any other failure or PLTD_SUCCESS otherwise.

PLTD_API int pltd_set_general_params(_pltd_general_params *pltd_general_params_t)

The pltd_set_general_params API function sets specific parameters that affect all DUTs. These parameters are the actual parameters that tell which tests will be performed, with what settings and what memory actions are going to be executed. The parameters are mainly used to program the u_dll.dll, the p_dll.dll and the instrument libraries. This API function returns PLTD_WRONG_DEV_PARAMS when an input parameter is invalid. It returns PLTD_ERROR in any other failure or PLTD_SUCCESS otherwise.

PLTD_API int pltd_start(void)

The pltd_start API function will perform all configured tests and memory programming actions in one sequence depending on the general parameter settings described before. It will update the device status on the higher layer software blocks (CFG, GUI or CLI) using callbacks. It will return when all device tests and memory actions have finished. It will return PLTD_SUCCESS when no system error occurred. If a test using external instrument is active and an error occurred during instrument initialization face the function will return PLTD_INSTR_ERROR.

PLTD_API int pltd_com_port_enum(uint32_t *com_port_dut);

The pltd_com_port_enum API function performs loopback in the PLT CPLD [1] hardware in order to identify the Windows COM port assigned to each DUT. This process is automatically being done in the first test execution after the start of the tool. However, it may be a case where the Windows re-enumerates the COM ports and therefore devices may take different COM port numbers. So, users can either restart the application or manually run the COM port enumeration to find the new DUT COM ports. When this API is used by the CFG or the CLI applications the returned DUT COM ports are saved in the params.xml file. Then, each time the test execution is started the DUT COM ports will not be enumerated again since the ports previously saved in the params.xml file will be used. Doing so, a lot of time can be saved when the CLI is used in batch commands. The API function returns PLTD_SUCCESS even if a device COM port was not found. The upper layer software will be notified with callbacks about the results of the COM port enumeration for each device. It returns PLTD_ERROR only if a system error occurs, if for example it cannot allocate memory.

PLTD_API int pltd_GU_com_find(int *gu_com_port);

The pltd_GU_com_find API function finds the Golden Unit (GU) Windows COM port and returns it as a function argument, in the gu_com_port. No callbacks are sent to the upper layer application. It returns PLTD_SUCCESS if the GU COM port was found, otherwise it returns PLTD_ERROR.

PLTD_API int pltd_GU_check_LED(int *gu_com_port);

The pltd_GU_check_LED API function is used to toggle the Golden Unit LED. The GU LED exists on the PLT hardware board [1]. When this function is called, the LED is toggled for 10 times with a 10ms period. Callbacks are sent to the upper layer software to update the GU status. The API function returns PLTD_SUCCESS if the process finished successfully or PLT_ERROR if a system error occurred.

PLTD_API int pltd_dbg_init(_pltd_dbg_params *pltd_dbg_params_t);

The pltd_dbg_init API function initializes the PLTD debug information. It also initializes the debug information for all the PLTD dynamic loaded DLLs (e.g., U_DLL, P_DLL, BARCODE_SCANNER.DLL, etc.) illustrated in Figure 1. The API function returns PLTD_SUCCESS if the initialization finished successfully or PLT_DBG_ERROR if a system error occurred.

PLTD_API char *pltd_get_volt_meter_instr_names(char *prev_name);

The pltd_get_volt_meter_instr_names API function is used in the DA14681-00 ADC gain calibration. It returns the names of the voltage meter DLLs found in the volt_meter_instr_plugins folder. The first time this function is called the `prev_name` parameter should be set to NULL. The second time it should take the return value of the first call. When all DLL names have been returned a NULL will be returned. This API function is used by the CFG application. By calling this function, the CFG will return a list of all instruments found in the designated folder. Users can then select a specific DLL to use. At this moment one instrument DLL exists, named `volt_meter_scpi.dll`. Users can create their own DLL, if specific instrument programming is required, other than what the default `volt_meter_scpi.dll` supports.

PLTD_API char *pltd_get_ble_tester_instr_names(char *prev_name);

The `pltd_get_ble_tester_instr_names` API function, like the `pltd_get_volt_meter_instr_names` described above, returns the names of the BLE tester instrument DLLs found in the `ble_tester_instr_plugins` folder. The first time this function is called the `prev_name` parameter should be set to NULL. The second time it should take the return value of the first call. When all DLL names have been returned a NULL will be returned. This API function is used by the CFG application. By calling this function, the CFG will return a list of all instruments found in the designated folder. Users can then select a specific DLL to use. At this moment two BLE tester instrument DLL exists, the `mt8852b.dll` and the `IQxelM.dll`. These are the DLLs used for the Anritsu MT8852B and the LitePoint IQxelM BLE tester instruments. Users can create their own DLL if support to other BLE tester instrument is required.

PLTD_API char * pltd_get_ammeter_instr_names(char *prev_name);

The `pltd_get_ammeter_instr_names` API function is used to return the current measurement instrument DLLs found under the `ammeter_instr_plugins` folder. The operation of this function is similar to the `pltd_get_ble_tester_instr_names` previously described.

PLTD_API char *pltd_get_temp_meas_instr_names(char *prev_name);

The `pltd_get_temp_meas_instr_names` API function, like the `pltd_get_volt_meter_instr_names` described above, returns the names of the temperature measurement instrument DLLs found in the `temp_meas_instr_plugins` folder. The first time this function is called the `prev_name` parameter should be set to NULL. The second time it should take the return value of the first call. When all DLL names have been returned a NULL will be returned. This API function is used by the CFG application. By calling this function, the CFG will return a list of all instruments found in the designated folder. Users can then select a specific DLL to use. At this moment two instrument DLL exist, the `ni_usb_tc01.dll` for the NI USB TC01 temperature sensor and the `tmu_temp_sens.dll` for the Papouch TMU sensor. Users can create their own DLL if support to other temperature sensors is required.

PLTD_API int pltd_vbat_uart_set (bool start, uint16_t duts);

The `pltd_vbat_uart_set` API function enables the VBAT and opens the UART to a specific set of devices. This is accomplished by instructing the GU to send an appropriate command to the CPLD on the PLT hardware. Callbacks for the GU status update are been send to the upper layer software. This feature could be used to communicate with the devices through UART by a user application other than PLT. The API function returns `PLTD_SUCCESS` if the process finished successfully or `PLT_ERROR` if a system error occurred.

PLTD_API int pltd_uart_coms_test(_pltd_uart_test *uart_test);

This function is used to test the communication path between the PLT host application and the devices under test. The test is performed at a user given UART baud rate. If the test fails then the communication to the device is not considered to be stable. A test failure could be due to long and unshielded cables between the PLT hardware and the DUT.

PLTD_API _pltd_versions *pltd_get_versions(void);

The `pltd_get_versions` API function will return the PLTD library version.

9.1.1 Production Line Tool DLL Function Input Arguments

Two of the PLTD API functions take pointers to data structures as arguments. Functions `pltd_set_device_params` and `pltd_set_general_params` provide all the necessary configuration setup for the PLTD to operate. After a successful configuration, calling the `pltd_start` function the device testing and programming will be performed.

In the next sections the function parameters are described in detail.

9.1.1.1 Function pltd_set_device_params Input Arguments

Function `pltd_set_device_params` takes a pointer as argument to the following data structure.

```
typedef struct __pltd_device_params
{
    _DUT_NUM dut_num;
    bool is_active;
    uint8_t bd_addr[BD_ADDR_SIZE];
    uint8_t OTP_customer_field[OTP_CUSTOMER_FIELD_SIZE];
    uint8_t xtal_trim_val[XTAL_TRIM_SIZE];
    float path_loss;
    uint32_t com_port;
    uint8_t mem_data[MAX_MEM_DATA_SIZE];
}_pltd_device_params;
```

```
_DUT_NUM dut_num;
```

The `dut_num` parameter can take values between 1 and 16. It serves as an index to the Production Line Tool hardware DUT connection [1]. No COM port number for the device is required as this is automatically found using a special loopback mechanism implemented in the CPLD and triggered by the GU device. However, if the `com_port` variable is not zero, then the automatic DUT com port find procedure is skipped and the particular COM port number will be used for this device.

The `_DUT_NUM` enumeration declaration is the following:

```
typedef enum __DUT_NUM
{
    DUT_1               = 1,
    DUT_2               = 2,
    DUT_3               = 3,
    DUT_4               = 4,
    DUT_5               = 5,
    DUT_6               = 6,
    DUT_7               = 7,
    DUT_8               = 8,
    DUT_9               = 9,
    DUT_10              = 10,
    DUT_11              = 11,
    DUT_12              = 12,
    DUT_13              = 13,
    DUT_14              = 14,
    DUT_15              = 15,
    DUT_16              = 16,
    INVALID_DUT_NUM
}_DUT_NUM;
```

```
bool is_active;
```

The `is_active` parameter enables or disables the device testing. The PLT hardware supports up to 16 devices. If some devices are not connected then they should be set to inactive using this variable. In that case, the PLTD will not perform any actions on disabled DUTs.

```
uint8_t bd_addr[BD_ADDR_SIZE];
```

The `bd_addr` parameter contains the device BD address.

The `BD_ADDR_SIZE` is defined as:

```
#define BD_ADDR_SIZE        4
```



```
uint8_t OTP_customer_field[OTP_CUSTOMER_FIELD_SIZE];
```

The `OTP_customer_field` parameter holds the data to be written in the DUT OTP customer header field. Different customer field per DUT is supported.

The `OTP_CUSTOMER_FIELD_SIZE` is defined as:

```
#define OTP_CUSTOMER_FIELD_SIZE    16
```

```
uint8_t xtal_trim_val[XTAL_TRIM_SIZE];
```

The `xtal_trim_val` parameter holds the XTAL trim value for each DUT. Users can manually fill the particular element with a valid device XTAL trim value, if the automatic XTAL trim procedure is not used. The value to set here could possibly be approximated by measuring the actual crystal frequency on a satisfied number of devices and calculating the correction needed for each one. An average value of all calculated correction values could then be used.

```
float path_loss;
```

The `path_loss` parameter holds the RF path losses between the GU antenna or the BLE tester antenna and the DUT antenna. It is used in the RF tests to compensate for differences in signal strength (RSSI) and TX power output between different device placements. This value can be between 0 and 40 dB.

```
uint32_t com_port;
```

If the `com_port` value is not zero for all active DUTs, then the automatic DUT COM port search will not be executed and the particular DUT COM ports defined in this variable will be used. This is mostly used to save time when the CLI application is used. A usual procedure using this value is explained below.

a. Call `pltd_set_general_params` to setup the active DUTs.
b. Execute `pltd_com_port_enum` function to get the DUT com ports for each DUT.
c. Export the new DUT COM ports to the `params.xml` file.
d. Each time the CLI starts, the COM ports from the `params.xml` file will be used, exported at step c. No automatic DUT COM port search will be executed.

This procedure needs to be executed only once on a particular test machine. The purpose will be to get the DUT COM ports and save them in the `params.xml` configuration file. Having done that, even if the CLI exits and restarts, the DUT COM ports will be read from the file and the automatic DUT COM port search procedure will not be executed, saving a considerable amount of time.

Check function `cli_dut_com_port_enum` in the `CLI_plt` project for an example.

```
uint8_t mem_data[MAX_MEM_DATA_SIZE];
```

This array keeps device specific data to be written to the device memory.

9.1.1.2 Function pltd_set_general_params Input Arguments

The function `pltd_set_general_params` takes a pointer to the following data structure as argument:

```
typedef struct __pltd_general_params
{
    _plt_log_info               log_info;
    _user_callback_pltd         user_callback_pltd;
    uint32_t                    GU_com_port;
    bool                        mem_wr_en;
    bool                        tests_en;
    bool                        vbat_uart_en;
    bool                        vbat_uart_rst;
    _vbat_rst_mode              vbat_rst_mode;
    uint16_t                    vbat_low_time;
    uint16_t                    reset_time;
    char                        flash_prog_fw_dir[FILE_PATH_SIZE];
    char                        prod_test_fw_dir[FILE_PATH_SIZE];
    char                        gu_fw_version[LOG_PARAM_STR_SIZE];
    bool                        gu_fw_version_check;
    bool                        use_programmer;
    _pltd_ic_specific           ic_spec;
}_pltd_general_params;

typedef struct __plt_log_info
{
    char                    station_num[LOG_PARAM_STR_SIZE];
    char                    caller_name[LOG_PARAM_STR_SIZE];
    char                    caller_ver[LOG_PARAM_STR_SIZE];
}_plt_log_info;
```

```
char station_num[LOG_PARAM_STR_SIZE];
```

The `station_num` parameter contains an identifier string indicating the testing station name. This string will be stored in the test result logs for easy identification of the testing station used.

The `LOG_PARAM_STR_SIZE` is defined as:

```
#define LOG_PARAM_STR_SIZE          32
```

```
char caller_name[LOG_PARAM_STR_SIZE];
```

The `caller_name` parameter contains the name of the caller application. This string will be stored in the test result logs for easy identification of the application that called the PLTD API (e.g. a CLI or a GUI tool).

The `LOG_PARAM_STR_SIZE` is defined as:

```
#define LOG_PARAM_STR_SIZE          32
```

```
char caller_ver[LOG_PARAM_STR_SIZE];
```

The `caller_ver` parameter contains the version of the caller application. This string will be stored in the test result logs for easy identification of the version on which the tests were performed.

The `LOG_PARAM_STR_SIZE` is defined as:

```
#define LOG_PARAM_STR_SIZE          32
```

```
_user_callback_pltd user_callback_pltd;
```

The `user_callback_pltd` parameter is a pointer to a function. The PLTD will call the pointed function every time the DUT status changes. The actual declaration of this function is as follows:

```
typedef void (__stdcall *_user_callback_pltd) (_pltd_dut_results *pltd_dut_results_t);
```

The `pltd_dut_results` data structure will contain the results for all active DUTs.

```
uint32_t GU_com_port;
```

The `GU_com_port` parameter should contain the COM port number of the Golden Unit. This COM port definition is required, as no loopback functionality is supported for the GU. Users can call the `pltd_GU_com_find` function to automatically find the GU COM port. The `gu_com_port` argument of the `pltd_GU_com_find` function will contain the GU COM port found. This value can then be set to the `GU_com_port` parameter.

```
bool mem_wr_en;
```

The `mem_wr_en` parameter enables or disables the memory programming.

```
bool tests_en;
```

The `tests_en` parameter enables or disables the production tests.

```
bool vbat_uart_en;
```

The `vbat_uart_en` parameter enables or disables the device VBAT and opens the UART interface between the CPLD and the DUT so the DUTs can be accessed from any other PC application.

```
bool vbat_uart_rst;
```

The `vbat_uart_rst` parameter resets the device before the VBAT and UART are enabled. If this is not set then the PLT could download a test firmware to the device, open the VBAT and UART using the `vbat_uart_en` and access the device from an external host application.

```
_vbat_rst_mode vbat_rst_mode;
```

Three different modes of device boot are supported. These can be found in the next enumeration.

```
typedef enum __vbat_rst_mode
{
    VBAT_ONLY = 0,
    VBAT_ON_RST,
    VBAT_AS_RST,
    INVALID_VBAT_MODE
}_vbat_rst_mode;
```

The `VBAT_ONLY` mode is the default mode, where the PLT VBAT line should be connected to the DUT VBAT. The PLT will power-up the device using the VBAT line and the DUT will enter the boot mode so a test firmware can be downloaded. Figure 13 shows an example of this operation.

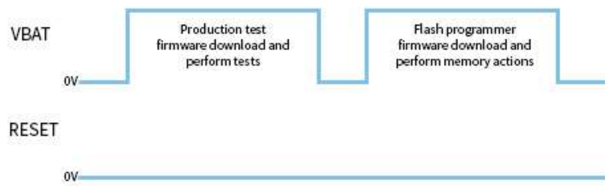


Figure 13: VBAT only operation

When the `VBAT_ON_RST` mode is selected, the VBAT PLT line stays always on but the PLT RST line is toggled high for a configurable amount of time. Figure 14 shows an example of this operation.

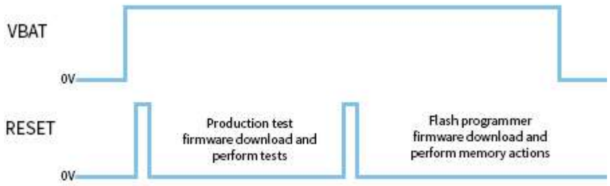


Figure 14: VBAT On with Reset

When the `VBAT_AS_RST` mode is selected, the VBAT line will toggle high for a configurable amount of time. The VBAT line can be connected to the DUT reset signal. Figure 15 shows an example of this operation.

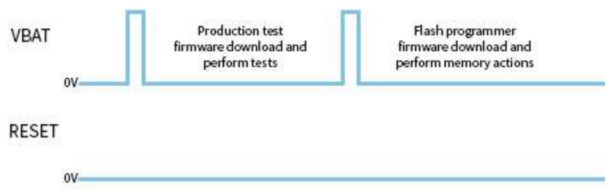


Figure 15: VBAT as Reset

```
uint16_t vbat_low_time;
```

This configures the time that the VBAT will stay low in the `VBAT_ONLY` mode. VBAT goes low to reboot the device in order to download the flash programmer firmware after production tests have been executed.

```
uint16_t reset_time;
```

This configures the time that the RST signal will be toggled high in the `VBAT_ON_RST` mode so the device can enter the boot mode. Additionally controls the time that the VBAT remains at a high level in the `VBAT_AS_RST` mode.

```
char flash_prog_fw_dir[FILE_PATH_SIZE];
```

The `flash_prog_fw_dir` parameter should contain the path to the `flash_programmer.bin` or `uartboot.bin` firmware.

The `FILE_PATH_SIZE` is defined as:

```
#define FILE_PATH_SIZE              256
```

```
char prod_test_fw_dir[FILE_PATH_SIZE];
```

The `prod_test_fw_dir` parameter should contain the path to the `prod_test_580.bin`, `prod_test_581.bin`, `prod_test_582.bin`, `prod_test_681_00.bin` or `prod_test_681_01.bin` firmware.

The `FILE_PATH_SIZE` is defined as:

```
#define FILE_PATH_SIZE              256
```

```
char gu_fw_version[LOG_PARAM_STR_SIZE];
```

This holds the value of the Golden Unit firmware version that the current PLT software supports. The GU firmware version supported by the current version of the tool is loaded from the configuration file. When the tests start the PLT reads the GU firmware version from the actual GU device and compares it to the value stored inside the `gu_fw_version`. If these do not match then an error will occur. This process helps to keep compatibility between the PLT software and the GU firmware burned in the GU SPI mounted on the PLT hardware board.

```
bool gu_fw_version_check;
```

When this is true the PLT will compare the GU version stored inside the `gu_fw_version` value to the one read from the GU device. If these do not match then an error will occur.

```
bool use_programmer;
```

If the `use_programmer` value is set to true, the production test firmware in DA1468x/DA1510x will be downloaded using the `uartboot.bin` and not the ROM bootloader. Doing so, fastest speed downloads can be achieved since the firmware can now be downloaded at the highest supported UART baud rate of 1Mbaud.

```
_pltd_ic_specific ic_spec;
```

The `ic_spec` parameter is a data structure union that contains specific DA1458x or DA1468x production test parameters. The union structure is shown below.

```
typedef union __pltd_ic_specific
{
     _dut_ic                        dut_ic;
     _pltd_580_params               params_580;
     _pltd_680_params               params_680;
}_pltd_ic_specific;
```

File `prod_line_tool_dll.h` in the `prod_line_tool_dll` Visual Studio 2015 project, contains a lot of information about the contents of the `_pltd_ic_specific` union data structures. Additionally, the HTML based help pages loaded after pressing the `source\production_line_tool\help\help.html` link can be used for more details.

9.2 PROD_LINE_TOOL_DLL API Details

More details on the PLTD API can be found in the API header file `source\production_line_tool\core_dlls\prod_line_tool_dll\prod_line_tool_dll.h`, or in the HTML based help pages loaded after pressing the `source\production_line_tool\help\help.html` link.

9.3 PROD_LINE_TOOL_DLL Example Procedures

In the next sections a procedure example will be given for performing a single RF test. The example briefly explains how the `prod_line_tool.dll` API can be used by a host application to perform this action.

9.3.1 PROD_LINE_TOOL_DLL RF Test Procedure

1. **PLTD initialization**

   Function call: `PLTD_API int pltd_init(void);`

   `Out: PLTD status codes`

2. **PLTD general parameter setup**

   Function call: `PLTD_API int pltd_set_general_params(_pltd_general_params *pltd_general_params_t);`

   `In: typedef struct __pltd_general_params`
   `Out: PLTD status codes`

   The data structure explained in section 9.1.1.2 should be filled and passed to this function as an argument. Example: consider the parameters shown in Table 14. Only the parameters necessary for the RF tests are shown. The other parameters should be initialized to zero.

Table 14: pltd_set_general_params Function Parameters for RF Test

Parameter	Value	Description
log_info.station_num	PC-10	The PC name of where the tool is running.
log_info.caller_name	DA1458x_DA1468x_GUI_ PLT.exe	The name of the host application.
log_info.caller_ver	v_3.0.7.500	The version of the host application.
user_callback_pltd	gplt_upgrade_datagrid	A callback function pointer that the DLL will call at every DUT status change.
GU_com_port	100	The COM port of the Golden Unit device. The GU is connected through the Production Line Tool hardware [3] directly to the PC with a dedicated FTDI USB cable. A virtual COM port will be assigned by Windows to the GU device. This port number should be entered here.
mem_wr_en	false	Disable the memory programming.
tests_en	True	Enable the production tests.
flash_prog_fw_dir	"binaries\ flash_programmer.bin"	Supply the flash programmer binary path.
prod_test_fw_dir	"binaries\prod_test_580.bin"	Supply the production test binary path.
ic_spec.dut_ic	DUT_IC_580	Set the device chipset to DA14580.
ic_spec.params_580.baud_rate	115200	Set the baud rate.
ic_spec.params_580.tests. xtal_trim_enable	false	Disable the automatic XTAL trim calibration.
ic_spec.params_580.tests. OTP_xtal_trim_burn	false	Disable the XTAL trim value burn in the OTP.
ic_spec.params_580.tests. xtal_trim_gpio	P0_5	Supply the XTAL trim GPIO input.
ic_spec.params_580.tests. rf_rx_test_enable[0]	true	Enable the first RF test.
ic_spec.params_580.tests. rf_rx_test_enable[1]	false	Disable the second RF test.
ic_spec.params_580.tests. rf_rx_test_enable[2]	false	Disable the third RF test.
ic_spec.params_580.tests. frequency[0]	2412	Set the RF channel for the first active RF test. The rest of the frequency channels do not matter since the tests are disabled.
ic_spec.params_580.tests. RSSI_limit[0]	-60	Set the RSSI threshold for the first RF test at which the DUTs will be marked as failed or passed.
ic_spec.params_580.tests. audio_test_enable	false	Disable the audio test.
ic_spec.params_580.tests. custom_data_test_enable	false	Disable the custom test.
ic_spec.params_580.tests. scan_enable	false	Disable the scan test.

Parameter	Value	Description
ic_spec.params_580.tests. gpio_led_test_enable[0],[1],[2]	false	Disable all GPIO LED tests.

For the rest of the parameters it does not matter what values they have, since we have enabled only a single RF test.

3. **PLTD device parameter setup**

   Function call: `PLTD_API int pltd_set_device_params(_pltd_device_params *pltd_device_params_t);`

   `In: typedef struct __pltd_device_params`
   `Out: PLTD status codes`

   This function should be called 16 times, once for each device. The data structure explained in section 9.1.1.1 should be filled and passed to this function as an argument. As an example consider the parameters shown in Table 15. The `pltd_set_device_params` function should be called for every DUT even if this is disabled. In that case `is_active` should be set to `false`. The following tables explain how the parameters may change among the different devices.

Table 15: pltd_set_device_params Function Parameters for RF Test

Parameter	Value	Description
dut_num	0-15	The device number.
is_active	true	Enable the tests for this device.
bd_addr	00:01:32:42:23:98	The BD address of this device. The BD address will be used in the test result logs as a DUT identifier.
OTP_customer_field	00:00:00:00…00	We will not write anything in OTP.
xtal_trim_val	00:00:00:00	XTAL trim value burn will not be performed, so this value can have just zeros.
com_port	0	Set to 0. Filled by the automatic DUT COM port find.

4. **PLTD start operation**

   Function call: `PLTD_API int pltd_start(void);`

   `In: void`
   `Out: PLTD status codes`

   This function will start the RF test for all configured DUTs. It will call the `user_callback_pltd` callback at every device status change and then return. It will use the U_DLL and P_DLL library APIs, as explained in sections 7.1 and 8.1, to download the `prod_test_580.bin` to the DUTs and start the TX-RX operation between the GU and the devices.

   After the tests have finished and the `pltd_start` function has returned, steps 2 to 4 can be repeated without the need to re-initialize the library.

10 VOLT_METER_SCPI DLL

The volt_meter_scpi.dll is a DLL with a generic API interface used to take voltage measurements with instruments that support the SCPI commands of digital meters defined class. Currently, it has been tested with the Keithley 2000 [9] and the Agilent 34401A [8] digital multimeter instruments, using the GPIB interface. This particular DLL is only used in the DA14681-00 silicon ADC gain calibration procedure. Other DA1468x based chipsets do not require this calibration as this is performed during IC manufacturing. The VBAT going to the DUTs will be measured using this DLL. The measured VBAT voltage will be compared to the one read by the device's ADC, returned by the adc_read P_DLL test procedure. A gain difference will be calculated and stored either in QSPI Flash or OTP. This value will be used later by the product firmware.

For the DLL to operate, **NI-VISA** and **NI-488.2** software installations are needed. These, can be downloaded from the paths shown in Table 16.

Table 16: Software Installations for volt_meter_scpi.dll

Software	Installation Link
NI-VISA	http://www.ni.com/download/ni-visa-15.5/5846/en/
NI-488.2	http://www.ni.com/download/ni-488.2-15.5/5859/en/

Directory source\production_line_tool\instruments\voltmeter\volt_meter_scpi contains all the necessary source code of this particular DLL. The API of the DLL can be found in source\production_line_tool\voltmeter\volt_meter_driver\volt_meter_api.h.

Users can create similar DLLs in order to interface to voltage meter instruments that do not follow the SCPI commands of the digital meters defined class. These custom DLLs should make use of the API described next.

10.1 VOLT_METER_SCPI API Functions

The volt_meter_scpi.dll has the following user accessible functions:

```
VOLT_METER_API int volt_meter_api_dbg_init(_dbg_params *dbg_params_t);
VOLT_METER_API int volt_meter_api_dbg_close(void);
VOLT_METER_API int volt_meter_api_init(char *iface, _callback_volt_meter
callback_volt_meter);
VOLT_METER_API int volt_meter_api_close(void);
VOLT_METER_API int volt_meter_api_measure(void);
```

A short description of each API function follows.

VOLT_METER_API int volt_meter_api_dbg_init(_dbg_params *dbg_params_t);

The volt_meter_api_dbg_init API function initializes the debug print information of the particular DLL. The _dbg_params_t function parameter is a pointer to the debug parameter structure. It should contain the necessary debug print parameters. Details of this structure can be found in section 6.1.1.1. The API function returns VOLT_METER_API_INVALID_DBG_PARAMS if an error occurs or VOLT_METER_API_SUCCESS otherwise.

VOLT_METER_API int volt_meter_api_dbg_close(void);

The volt_meter_api_dbg_close API function closes the interface to the dbg_dll.dll library and all debug print information is disabled. This function returns always VOLT_METER_API_SUCCESS.

VOLT_METER_API int volt_meter_api_init(char *iface, _callback_volt_meter callback_volt_meter);

The `volt_meter_api_init` API function is used to initialize the voltage meter instrument. It opens a VISA session to the instrument, queries the instrument identification string, clears all queues and resets the instrument to a defined state.

Function input parameter `iface` is a string that defines the instrument interface, e.g. “GPIB0::11”. Additionally, the function input parameter `callback_volt_meter` sets the callback function to be called when the voltage meter results are available after the `volt_meter_api_measure` function described below is called. The function returns `VOLT_METER_API_ERROR` if an error occurs or `VOLT_METER_API_SUCCESS` otherwise.

VOLT_METER_API int volt_meter_api_close(void);

The `volt_meter_api_close` API function frees all DLL allocated resources. This function returns always `VOLT_METER_API_SUCCESS`.

VOLT_METER_API int volt_meter_api_measure(void);

When the `volt_meter_api_measure` API function is called, the instrument is instructed to perform voltage measurements at a 5 V range and with a 0.00001 V resolution. It sets the instrument to take four voltage samples. In parallel, a thread is created, which will fetch the four voltage meter results, when available, using the `FETCH?` SCPI query command. It will then average them and return the result using the callback function that was initialized when the `volt_meter_api_init` API function was called. Therefore, the function will return immediately but the upper layer software should wait for the result callbacks to be received from the thread created. The function will return `VOLT_METER_API_ERROR` on a system error or `VOLT_METER_API_SUCCESS` otherwise.

10.2 VOLT_METER_SCPI API Details

More details on the VOLT_METER_SCPI API can be found in the API header file in `source\production_line_tool\instruments\voltmeter\volt_meter_driver\volt_meter_api.h`, or in the HTML based help pages loaded after pressing the `source\production_line_tool\help\help.html` link.

11 VOLT_METER_DRIVER DLL

The volt_meter_driver.dll is a driver that can load all voltage meter DLLs from the volt_meter_instr_plugins folder. It is able to return all the DLL names existing in that folder and is able to load and use any one of them as long as they are following the correct API found under source\production_line_tool\instruments\voltmeter\volt_meter_driver\volt_meter_api.h.

The CFG application uses this driver, through PLTD, to display the available voltage meter instrument DLLs (e.g. volt_meter_scpi.dll). Users can then select a DLL name and all voltage meter functions will go through the one selected. The block describing this operation is shown in Figure 16.

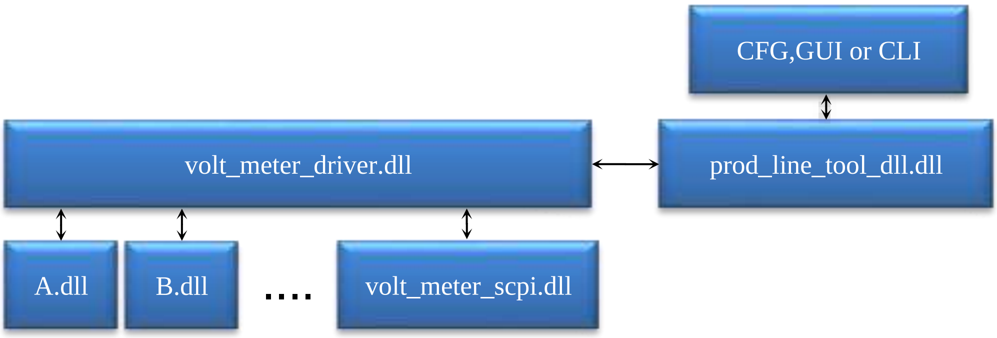


Figure 16: volt_meter_driver.dll Block Diagram

During the prod_line_tool_dll.dll initialization face (pltd_init) the volt_meter_driver.dll is also initialized. It loads all voltage meter instrument DLLs from the volt_meter_instr_plugins folder. The DLL function handlers and the DLL names are stored on a linked list inside the driver. Users can then get all voltage instrument DLL names by calling the pltd_get_volt_meter_instr_names API function from prod_line_tool.dll.

11.1 VOLT_METER_DRIVER API Functions

The volt_meter_driver.dll has the following user accessible functions.

```
VOLT_METER_DRV_API int volt_meter_drv_init(void);
VOLT_METER_DRV_API int volt_meter_drv_close(void);
VOLT_METER_DRV_API int volt_meter_drv_dbg_init(_dbg_params *dbg_params_t);
VOLT_METER_DRV_API int volt_meter_drv_dbg_close(void);
VOLT_METER_DRV_API volt_meter_drv_instr_hdl
      volt_meter_drv_get_instr_hdl(volt_meter_drv_instr_hdl prev_instr_hdl);
VOLT_METER_DRV_API int volt_meter_drv_get_instr_name(volt_meter_drv_instr_hdl
      instr_hdl, char *name, int size);
VOLT_METER_DRV_API int volt_meter_drv_instr_init(volt_meter_drv_instr_hdl instr_hdl,
      char *iface, _callback_volt_meter callback_volt_meter);
VOLT_METER_DRV_API int volt_meter_drv_instr_close(volt_meter_drv_instr_hdl instr_hdl);
VOLT_METER_DRV_API int volt_meter_drv_instr_measure(volt_meter_drv_instr_hdl
      instr_hdl);
```

A detailed description of the API functions will be given next.

VOLT_METER_DRV_API int volt_meter_drv_init(void);

The volt_meter_drv_init API function initializes the DLL. It searches the folder volt_meter_instr_plugins to find voltage meter instrument DLLs and saves them to an internal linked list. If no instrument DLL is found the function will return VOLT_METER_DRV_NO_INSTR_PLUGIN.

On a system error the function will return VOLT_METER_DRV_ERROR or VOLT_METER_DRV_SUCCESS otherwise.

VOLT_METER_DRV_API int volt_meter_drv_close(void);

The volt_meter_drv_close API function frees the allocated resources of the voltage meter plugins created during the initialization process. The function will always return VOLT_METER_DRV_SUCCESS.

VOLT_METER_DRV_API int volt_meter_drv_dbg_init(_dbg_params *dbg_params_t);

The volt_meter_drv_dbg_init API function initializes the debug information. Details on the dbg_params_t input parameter can be found in section 6.1.1.1. This function will return VOLT_METER_DRV_INVALID_DBG_PARAMS if an invalid dbg_params_t input parameter is given. If an error occurred during dbg_dll.dll initialization, the function will return VOLT_METER_DRV_DBG_DLL_ERROR. On success the function will return VOLT_METER_DRV_SUCCESS.

VOLT_METER_DRV_API int volt_meter_drv_dbg_close(void);

The volt_meter_drv_dbg_close API function closes the debug session. It calls the dbg_close API function from dbg_dll.dll. If an error occurred in the dbg_dll.dll then VOLT_METER_DRV_DBG_DLL_ERROR will be returned. The function will return VOLT_METER_DRV_SUCCESS on success.

VOLT_METER_DRV_API volt_meter_drv_instr_hdl volt_meter_drv_get_instr_hdl(volt_meter_drv_instr_hdl prev_instr_hdl);

The volt_meter_drv_get_instr_hdl API function returns the handles of the voltage meter DLLs created when the volt_meter_drv_init API function was called. First call of this function must be made with the prev_instr_hdl input parameter set to NULL. All subsequent calls must be made with the prev_instr_hdl input parameter set to the handle returned from the previous function call. When this function returns NULL no more instrument handles exist.

VOLT_METER_DRV_API int volt_meter_drv_get_instr_name(volt_meter_drv_instr_hdl instr_hdl, char *name, int size);

The volt_meter_drv_get_instr_name API function returns the voltage meter instrument DLL names as recorded when the volt_meter_drv_init API function was called. It is mainly used to display the names in the CFG application, so users can select a specific DLL to use for the DA14681-00 ADC gain calibration.

VOLT_METER_DRV_API int volt_meter_drv_instr_init(volt_meter_drv_instr_hdl instr_hdl, char *iface, _callback_volt_meter callback_volt_meter);

The volt_meter_drv_instr_init API function initializes a voltage meter instrument DLL with instr_hdl handle. It actually calls the volt_meter_api_init API function in the voltage meter DLL pointed by the instr_hdl handle. The input callback_volt_meter parameter is used to return measurement status and results through callbacks. A pointer to a callback function is passed that will be called when a voltage measurement is ready. If the function succeeds it will return VOLT_METER_DRV_SUCCESS, otherwise it will return VOLT_METER_DRV_ERROR on failure.

VOLT_METER_DRV_API int volt_meter_drv_instr_close(volt_meter_drv_instr_hdl instr_hdl);

The volt_meter_drv_instr_close API function closes the voltage meter instrument DLL. It actually calls the volt_meter_api_close API function in the DLL pointed by the instr_hdl handle. If the function succeeds it will return VOLT_METER_DRV_SUCCESS otherwise it will return VOLT_METER_DRV_ERROR on failure.

VOLT_METER_DRV_API int volt_meter_drv_instr_measure(volt_meter_drv_instr_hdl instr_hdl);

The `volt_meter_drv_instr_measure` API function starts the voltage measurements. It actually calls the `volt_meter_api_measure` API function in the voltage meter instrument DLL pointed by the `instr_hdl` input parameter handle. A thread is created that waits for the instrument to return some measurements. After averaging the measurements, the result is returned to the upper layer software using callbacks. The function returns immediately but the upper layer software has to wait for the callback that will come later as a thread is created. If the function succeeds to create the measurement thread it will return `VOLT_METER_DRV_SUCCESS` otherwise it will return `VOLT_METER_DRV_ERROR` on failure.

11.2 VOLT_METER_DRIVER API Details

More details on the VOLT_METER_DRIVER API can be found in the API header file in `source\production_line_tool\instruments\voltmeter\volt_meter_driver\volt_meter_driver.h` or in the HTML based help pages loaded after pressing the `source\production_line_tool\help\help.html` link.

12 MT8852B and IQxelM DLLs

The mt8852b.dll and IQxelM.dll are DLLs that support BLE DTM measurements using the Anritsu MT8852B and the Litepoint IQxelM Bluetooth test set instruments [7] [15]. The API used by both DLLs is designed such that it can be used by other BLE tester instruments as well. It is able to perform TX power, TX modulation index, TX frequency drift and offset and RX RSSI measurements. All test settings and pass/fail limits are configurable giving a great flexibility to the user.

For the DLLs to operate, **NI-VISA** and **NI-488.2** software installations are needed. These can be downloaded from the paths shown in Table 16.

Directory source\production_line_tool\instruments\ble_testers\mt8852b contains all the necessary source code of the mt8852b.dll DLL. Directory source\production_line_tool\instruments\ble_testers\IQxelM contains all the necessary source code of the IQxelM.dll DLL.

The API of the BLE tester DLLs is defined in source\production_line_tool\instruments\ble_testers\ble_tester_driver\ble_instr_api.h.

12.1 BLE Tester API Functions

The BLE tester API has the following user accessible functions:

```
BLE_INSTR_API int ble_instr_dbg_init(_dbg_params *dbg_params_t);
BLE_INSTR_API int ble_instr_dbg_close(void);
BLE_INSTR_API int ble_instr_init(void *data, _callback_ble_instr callback_ble_instr);
BLE_INSTR_API int ble_instr_close(void);
BLE_INSTR_API int ble_instr_set_path_loss(float path_loss);
BLE_INSTR_API int ble_instr_set_pwr_range(_ble_instr_pwr_range pwr_range);
BLE_INSTR_API int ble_instr_set_tx_pwr_h_lim(float avg_high_limit);
BLE_INSTR_API int ble_instr_set_tx_pwr_l_lim(float avg_low_limit);
BLE_INSTR_API int ble_instr_set_tx_pwr_pk_lim(float pk_avg_limit);
BLE_INSTR_API int ble_instr_do_tx_pwr(uint32_t freq);
BLE_INSTR_API int ble_instr_set_freq_offs_h_lim(uint32_t pos_freq_limit);
BLE_INSTR_API int ble_instr_set_freq_offs_l_lim(uint32_t neg_freq_limit);
BLE_INSTR_API int ble_instr_set_freq_drift_pkt_lim(uint32_t drift_pkt_limit);
BLE_INSTR_API int ble_instr_set_freq_drift_rate_lim(uint32_t drift_rate_limit);
BLE_INSTR_API int ble_instr_do_freq_offs(uint32_t freq);
BLE_INSTR_API int ble_instr_set_mod_idx_f1_min(uint32_t f1_min_limit);
BLE_INSTR_API int ble_instr_set_mod_idx_f1_max(uint32_t f1_max_limit);
BLE_INSTR_API int ble_instr_set_mod_idx_f2_max(uint32_t f2_max_limit);
BLE_INSTR_API int ble_instr_set_mod_idx_f1f2_ratio(float f1f2_ratio_limit);
BLE_INSTR_API int ble_instr_do_mod_idx(uint32_t freq);
BLE_INSTR_API int ble_instr_set_rx_sens_tx_pat(uint8_t pattern);
BLE_INSTR_API int ble_instr_set_rx_sens_pkt_space(uint16_t spacing);
BLE_INSTR_API int ble_instr_set_rx_sens_pkt_num(uint16_t num_of_pkts);
BLE_INSTR_API int ble_instr_set_rx_sens_tx_pwr(float tx_power);
BLE_INSTR_API int ble_instr_set_rx_sens_tx_dirty(bool dirty);
BLE_INSTR_API int ble_instr_set_rx_sens_tx_crc(bool crc_state);
BLE_INSTR_API int ble_instr_do_rx_sens(uint32_t freq);
```

A detailed description of the API functions will be given next.

BLE_INSTR_API int ble_instr_dbg_init(_dbg_params *dbg_params_t)

The ble_instr_dbg_init API function is used to initialize the debug print operation. Details on the dbg_params_t input parameter can be found in section 6.1.1.1. This function will return BLE_INSTR_INVALID_DBG_PARAMS if an invalid dbg_params_t input parameter is given. If an error occurs during dbg_dll.dll initialization, the function will return BLE_INSTR_DBG_DLL_ERROR. On success the function will return BLE_INSTR_SUCCESS.

BLE_INSTR_API int ble_instr_dbg_close(void)

The ble_instr_dbg_close API function closes the debug session. It calls the dbg_close API function from dbg_dll.dll library to free all the resources acquired. It always returns BLE_INSTR_SUCCESS.

BLE_INSTR_API int ble_instr_init(void *data, _callback_ble_instr callback_ble_instr);

The ble_instr_init API function resets and initializes the instrument and the NI VISA resource manager. The data input pointer parameter points to the interface string, which by default is set to "GPIB0::27" in the configuration parameter file, params.xml. The callback_ble_instr is a pointer to a callback function that the DLL will call to update measurement status and results. If the function succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

BLE_INSTR_API int ble_instr_close(void)

The ble_instr_close API function deallocates all NI VISA and other local resources used during measurements. It closes the measurement thread, if this has remained opened for some reason, and disables any GPIB service requests. If it succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

BLE_INSTR_API int ble_instr_set_path_loss(float path_loss)

The ble_instr_set_path_loss API function sets the path losses between the device antenna and the BLE tester instrument antenna. It is recommended that the RF measurements take place in a shielded box and that the antennas are as close as possible. The path_loss input parameter is a positive number from 0 to 40 dB. This value is added in the TX power and RSSI measurements. If the function succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

BLE_INSTR_API int ble_instr_set_pwr_range(_ble_instr_pwr_range pwr_range);

The ble_instr_set_pwr_range API function is used to set the TX input power range. It is used in all the TX measurements. It pre-sets the instrument to a specific input power scale. The input pwr_range parameter is an enumeration that provides different power scale ranges. It is suggested to use the AUTO_PWR_RANGE selection as it will allow the instrument to select the appropriate TX input power range scale. If the function succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

BLE_INSTR_API int ble_instr_set_tx_pwr_h_lim(float avg_high_limit)

The ble_instr_set_tx_pwr_h_lim API function sets the average high power limit for the TX output power measurements. The avg_high_limit input parameter units are in dBm and the allowable range is between -80 and 30 dBm. If the function succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR on failure.

BLE_INSTR_API int ble_instr_set_tx_pwr_l_lim(float avg_low_limit)

The ble_instr_set_tx_pwr_l_lim API function sets the average low power limit for the TX output power measurements. The avg_low_limit input parameter units are in dBm and the allowable range is between -80 and 30 dBm. If the function succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

BLE_INSTR_API int ble_instr_set_tx_pwr_pk_lim(float pk_avg_limit)

The `ble_instr_set_tx_pwr_pk_lim` API function sets the peak-to-average power limit for the TX output power measurements. The `pk_avg_limit` input parameter units are in dB and the allowable range is between 0 and 10 dB. If the function succeeds it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_do_tx_pwr(uint32_t freq)

The `ble_instr_do_tx_pwr` API function starts the TX output power measurements at the frequency given to the `freq` input parameter. This function enables GPIB service requests, which provide measurement completion information. It also creates a thread for the service requests to be handled. Appropriate callbacks are sent to the upper layer software when measurements are ready. The callback function has the following type.

```
typedef void (*_callback_ble_instr)(int status, char *data);
```

The `status` argument is of `BLE_INSTR_STATUS_CODES` type and describes the status of the measurement. For a TX power measurement, if the callback has `status = BLE_INSTR_TX_PWR_PASSED` or `status = BLE_INSTR_TX_PWR_FAILED` then the `data` output argument will contain the measurement results.

The `ble_instr_do_tx_pwr` API function returns immediately after setting up the measurement. Actual measurement results are returned using callbacks. If the function succeeds to initialize the measurement it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_set_freq_offs_h_lim(uint32_t pos_freq_limit)

The `ble_instr_set_freq_offs_h_lim` API function sets the maximum positive offset limit in kHz for the TX carrier frequency offset measurements. The `pos_freq_limit` input parameter units are in kHz and the allowable range is between 0 and 250 kHz. If the function succeeds it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_set_freq_offs_l_lim(uint32_t neg_freq_limit)

The `ble_instr_set_freq_offs_l_lim` API function sets the maximum negative offset limit in kHz for the TX carrier frequency offset measurements. The `neg_freq_limit` input parameter units are in kHz and the allowable range is between 0 and 250 kHz. If the function succeeds it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_set_freq_drift_pkt_lim(uint32_t drift_pkt_limit)

The `ble_instr_set_freq_drift_pkt_lim` API function sets the overall packet drift limit in kHz for the TX carrier frequency offset measurements. The `drift_pkt_limit` input parameter units are in kHz and the allowable range is between 0 and 200 kHz. If the function succeeds it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_set_freq_drift_rate_lim(uint32_t drift_rate_limit)

The `ble_instr_set_freq_drift_rate_lim` API function sets the drift rate limit in kHz/50 µs. It is used in the TX carrier frequency offset and drift measurements. The `drift_rate_limit` input parameter allowable range is between 1 and 90 kHz/50 µs. If the function succeeds it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_do_freq_offs(uint32_t freq)

The `ble_instr_do_freq_offs` API function performs the TX carrier frequency offset and drift measurements at the frequency given to the `freq` input parameter. This function enables the GPIB service requests, which provide measurement completion information. It also creates a thread for the service requests to be handled. Appropriate callbacks are sent to the upper layer software when the measurements are ready. The callback function has the following type.

```
typedef void (*_callback_ble_instr)(int status, char *data);
```

The `status` argument is of `BLE_INSTR_STATUS_CODES` type and describes the status of the measurement. For a TX carrier frequency offset and drift measurement, if the callback has `status = BLE_INSTR_TX_FREQ_OFFS_PASSED` or `status = BLE_INSTR_TX_FREQ_OFFS_FAILED` then the `data` output argument will contain the measurement results.

The `ble_instr_do_freq_offs` API function returns immediately after setting up the measurement. Actual measurement results are returned using callbacks. If the function succeeds to initialize the measurement it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_set_mod_idx_f1_min(uint32_t f1_min_limit)

The `ble_instr_set_mod_idx_f1_min` API function sets the F1 minimum average limit in kHz for the TX modulation index measurements. The `f1_min_limit` input parameter units are in kHz and the allowable range is between 0 and 300 kHz. If the function succeeds it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_set_mod_idx_f1_max(uint32_t f1_max_limit)

The `ble_instr_set_mod_idx_f1_max` API function sets the F1 maximum average limit in kHz for the TX modulation index measurements. The `f1_max_limit` input parameter units are in kHz and the allowable range is between 0 and 300 kHz. If the function succeeds it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_set_mod_idx_f2_max(uint32_t f2_max_limit)

The `ble_instr_set_mod_idx_f2_max` API function sets the F2 maximum average limit in kHz for the TX modulation index measurements. The `f2_max_limit` input parameter units are in kHz and the allowable range is between 0 and 300 kHz. If the function succeeds it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_set_mod_idx_f1f2_ratio(float f1f2_ratio_limit)

This API function sets the F1/F2 maximum average ratio limit for the TX modulation index measurements. The `f1f2_ratio_limit` input parameter allowable range is between 0 and 1. If the function succeeds it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_do_mod_idx(uint32_t freq)

The `ble_instr_set_mod_idx_f1f2_ratio` API function starts the TX modulation index measurement at the frequency given to the `freq` input parameter. This function enables GPIB service requests, which provide measurement completion information. It also creates a thread for the service requests to be handled. Appropriate callbacks are sent to the upper layer software when measurements are ready. The callback function has the following type.

```
typedef void (*_callback_ble_instr)(int status, char *data);
```

The `status` argument is of `BLE_INSTR_STATUS_CODES` type and describes the status of the measurement. For a TX carrier frequency offset and drift measurement, if the callback has `status = BLE_INSTR_TX_MOD_IDX_PASSED` or `status = BLE_INSTR_TX_MOD_IDX_FAILED` then the `data` output argument will contain the measurement results.

The `ble_instr_do_mod_idx` API function returns immediately after setting up the measurement. Actual measurement results are returned using callbacks. If the function succeeds to initialize the measurement it returns `BLE_INSTR_SUCCESS`, otherwise it returns `BLE_INSTR_ERROR`.

BLE_INSTR_API int ble_instr_set_rx_sens_tx_pat(uint8_t pattern)

The ble_instr_set_rx_sens_tx_pat API function sets the TX packet pattern type that the BLE tester instrument will transmit when performing RX RSSI measurements. The pattern input parameter can take the following values.

- 0: pseudo random binary sequence 9 (PRBS9)
- 1: alternate 0s and 1s.
- 2: alternation between 0000 and 1111.

If the function succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

BLE_INSTR_API int ble_instr_set_rx_sens_pkt_space(uint16_t spacing)

The ble_instr_set_rx_sens_pkt_space API function sets the TX packet spacing that the BLE tester instrument will transmit when performing RX RSSI measurements. The spacing input parameter has a measurement unit of microseconds (μs) and can take values from 625 μs to 65535 μs. If the function succeeds it returns BLE_INSTR_SUCCESS otherwise it returns BLE_INSTR_ERROR on failure.

BLE_INSTR_API int ble_instr_set_rx_sens_pkt_num(uint16_t num_of_pkts)

The ble_instr_set_rx_sens_pkt_num API function sets the TX packets number that the BLE tester instrument will transmit when performing RX RSSI measurements. The num_of_pkts input parameter can take values from 1 to 65535. If the function succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

BLE_INSTR_API int ble_instr_set_rx_sens_tx_pwr(float tx_power)

The ble_instr_set_rx_sens_tx_pwr API function sets the TX output power that the BLE tester instrument will transmit when performing RX RSSI measurements. The tx_power input parameter measurements unit is in dBm and can take values from 0 to -90 dBm. If the function succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

BLE_INSTR_API int ble_instr_set_rx_sens_tx_dirty(bool dirty)

The ble_instr_set_rx_sens_tx_dirty API function enables or disables the 'TX dirty' option used in the RX RSSI measurements. When enabled the BLE tester packet generator uses an internal dirty table to be transmitted. If the function succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

BLE_INSTR_API int ble_instr_set_rx_sens_tx_crc(bool crc_state)

The ble_instr_set_rx_sens_tx_crc API function enables or disables the 'TX CRC alternate state' option used in the RX RSSI measurements. When enabled the BLE tester packet generator will alternate the CRC of the transmitted packets. It will send a packet with CRC error, one without and so on. If the function succeeds it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

BLE_INSTR_API int ble_instr_do_rx_sens(uint32_t freq);

The ble_instr_do_rx_sens API function starts the BLE tester packet generator, used in the RX RSSI measurements. This function enables GPIB service requests, which provide measurement completion information. It also creates a thread for the service requests to be handled. Appropriate callbacks are sent to the upper layer software when the measurements are ready. The callback function has the following type.

```
typedef void (*_callback_ble_instr)(int status, char *data);
```


The status argument is of BLE_INSTR_STATUS_CODES type and describes the status of the measurement. For the RX RSSI measurements, status = BLE_INSTR_TX_RX_SENS_OK denotes that the packet generator has finished transmitted the programmed packets.

The ble_instr_do_rx_sens API function returns immediately after initializing the packet generator. The status of the packets transmitted is reported to the upper layer software using callbacks as noted above. If the function succeeds to initialize the measurement it returns BLE_INSTR_SUCCESS, otherwise it returns BLE_INSTR_ERROR.

12.2 MT8852B and IQxelM API Details

More details on the MT8852B and IQxelM API can be found either in the API header file in source\production_line_tool\instruments\ble_testers\ble_tester_driver\ble_instr_api.h or in the HTML based help pages loaded after pressing the source\production_line_tool\help\help.html link.

13 BLE_TESTER_DRIVER DLL

The ble_tester_driver.dll is a DLL driver used to load and access different BLE tester instrument DLLs from ble_tester_instr_plugins folder. It is able to return all the DLL names existing in that folder and able to use any one of them as long as they are following the correct API found under source\production_line_tool\instruments\ble_tester\ble_tester_driver\ble_instr_api.h. The API in the ble_instr_api.h file was actually described in section 12 since it is the one that the mt8852b.dll and IQxelM.dll are using. The CFG application uses this driver through the PLTD to display the available BLE tester instrument DLLs (e.g. mt8852b.dll). Users can then select a DLL name and all the BLE measurements will go through the one selected. The actual block describing this operation is shown in Figure 17.

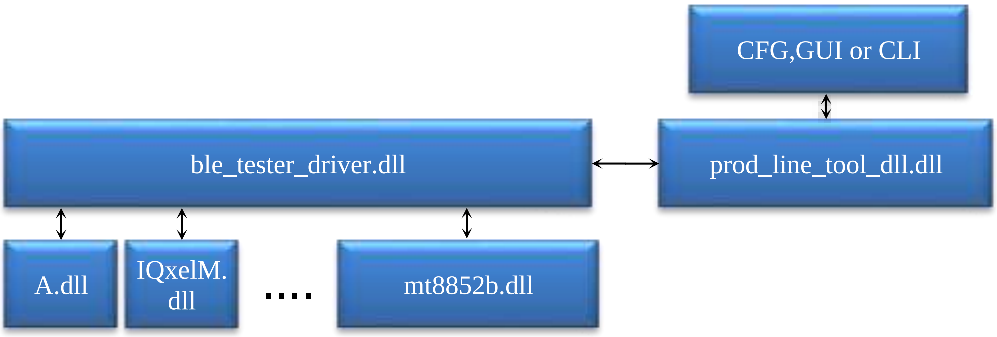


Figure 17: ble_tester_driver.dll Block Diagram

During the prod_line_tool_dll.dll initialization face (pltd_init) the ble_tester_driver.dll is also initialized. The BLE tester instrument driver DLL loads all the DLLs found in ble_tester_instr_plugins folder. The driver creates function handles to all the loaded DLL APIs and each DLL name is stored on a linked list in the driver. Users can then get all BLE tester instrument DLL names by calling the pltd_get_ble_tester_instr_names prod_line_tool.dll API function.

13.1 BLE_TESTER_DRIVER API Functions

The ble_tester_driver.dll has the following user accessible functions:

```
BLE_TESTER_DRV_API int ble_tester_drv_init(void);
BLE_TESTER_DRV_API int ble_tester_drv_close(void);
BLE_TESTER_DRV_API int ble_tester_drv_dbg_init(_dbg_params *dbg_params_t);
BLE_TESTER_DRV_API int ble_tester_drv_dbg_close(void);
BLE_TESTER_DRV_API int ble_tester_drv_get_instr_name(ble_drv_instr_hdl instr_hdl, char
      *name, int size);
BLE_TESTER_DRV_API ble_drv_instr_hdl ble_tester_drv_get_instr_hdl(ble_drv_instr_hdl
      prev_instr_hdl);
BLE_TESTER_DRV_API int ble_tester_drv_instr_init(ble_drv_instr_hdl instr_hdl, void
      *data, _callback_ble_instr callback_ble_instr);
BLE_TESTER_DRV_API int ble_tester_drv_instr_close(ble_drv_instr_hdl instr_hdl);
BLE_TESTER_DRV_API int ble_tester_drv_set_path_loss(ble_drv_instr_hdl instr_hdl, float
      path_loss);
BLE_TESTER_DRV_API int ble_tester_drv_set_pwr_range(ble_drv_instr_hdl instr_hdl,
      _ble_instr_pwr_range pwr_range);
BLE_TESTER_DRV_API int ble_tester_drv_set_tx_pwr_h_lim(ble_drv_instr_hdl instr_hdl,
      float avg_high_limit);
BLE_TESTER_DRV_API int ble_tester_drv_set_tx_pwr_l_lim(ble_drv_instr_hdl instr_hdl,
      float avg_low_limit);
BLE_TESTER_DRV_API int ble_tester_drv_set_tx_pwr_pk_lim(ble_drv_instr_hdl instr_hdl,
      float pk_avg_limit);
```



```
BLE_TESTER_DRV_API int ble_tester_drv_do_tx_pwr(ble_drv_instr_hdl instr_hdl, uint32_t
      freq);
BLE_TESTER_DRV_API int ble_tester_drv_set_freq_offs_h_lim(ble_drv_instr_hdl instr_hdl,
      uint32_t pos_freq_limit)
BLE_TESTER_DRV_API int ble_tester_drv_set_freq_offs_l_lim(ble_drv_instr_hdl instr_hdl,
      uint32_t neg_freq_limit);
BLE_TESTER_DRV_API int ble_tester_drv_set_freq_drift_pkt_lim(ble_drv_instr_hdl
      instr_hdl, uint32_t drift_pkt_limit);
BLE_TESTER_DRV_API int ble_tester_drv_set_freq_drift_rate_lim(ble_drv_instr_hdl
      instr_hdl, uint32_t drift_rate_limit);
BLE_TESTER_DRV_API int ble_tester_drv_do_freq_offs(ble_drv_instr_hdl instr_hdl,
      uint32_t freq);
BLE_TESTER_DRV_API int ble_tester_drv_set_mod_idx_f1_min(ble_drv_instr_hdl instr_hdl,
      uint32_t f1_min_limit);
BLE_TESTER_DRV_API int ble_tester_drv_set_mod_idx_f1_max(ble_drv_instr_hdl instr_hdl,
      uint32_t f1_max_limit);
BLE_TESTER_DRV_API int ble_tester_drv_set_mod_idx_f2_max(ble_drv_instr_hdl instr_hdl,
      uint32_t f2_max_limit);
BLE_TESTER_DRV_API int ble_tester_drv_set_mod_idx_f1f2_ratio(ble_drv_instr_hdl
      instr_hdl, float f1f2_ratio_limit);
BLE_TESTER_DRV_API int ble_tester_drv_do_mod_idx(ble_drv_instr_hdl instr_hdl, uint32_t
      freq);
BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_tx_pat(ble_drv_instr_hdl instr_hdl,
      uint8_t pattern);
BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_pkt_space(ble_drv_instr_hdl
      instr_hdl, uint16_t spacing);
BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_pkt_num(ble_drv_instr_hdl instr_hdl,
      uint16_t num_of_pkts);
BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_tx_pwr(ble_drv_instr_hdl instr_hdl,
      float tx_power);
BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_tx_dirty(ble_drv_instr_hdl
      instr_hdl, bool dirty);
BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_tx_crc(ble_drv_instr_hdl instr_hdl,
      bool crc_state);
BLE_TESTER_DRV_API int ble_tester_drv_do_rx_sens(ble_drv_instr_hdl instr_hdl, uint32_t
      freq);
```

A detailed description of the API functions will be given next.

BLE_TESTER_DRV_API int ble_tester_drv_init(void)

The ble_tester_drv_init API function initializes the DLL. It searches ble_tester_instr_plugins folder to find BLE instrument DLLs. It loads the DLLs found and saves the names and the function handles to an internal linked list. If no instrument DLLs are found the function will return BLE_TESTER_DRV_NO_INSTR_PLUGINS. On a system error the function will return BLE_TESTER_DRV_ERROR or BLE_TESTER_DRV_SUCCESS otherwise.

BLE_TESTER_DRV_API int ble_tester_drv_close(void)

The ble_tester_drv_close API function frees the allocated resources of all the BLE tester DLL plugins created during the initialization process. The function will always return BLE_TESTER_DRV_SUCCESS.

BLE_TESTER_DRV_API int ble_tester_drv_dbg_init(_dbg_params *dbg_params_t);

The ble_tester_drv_dbg_init API function initializes the debug information. Details on the dbg_params_t input parameter can be found in section 6.1.1.1. This function will return BLE_TESTER_DRV_INVALID_DBG_PARAMS if an invalid dbg_params_t input parameter is given. If an error occurred during dbg_dll.dll initialization, the function will return BLE_TESTER_DRV_DBG_DLL_ERROR. On success the function will return BLE_TESTER_DRV_SUCCESS.

BLE_TESTER_DRV_API int ble_tester_drv_dbg_close(void)

The ble_tester_drv_dbg_close API function closes the debug session. It calls the dbg_close API function from the dbg_dll.dll library. If an error occurs in the dbg_dll.dll then BLE_TESTER_DRV_DBG_DLL_ERROR will be returned. Otherwise the function will return BLE_TESTER_DRV_SUCCESS on success.

BLE_TESTER_DRV_API int ble_tester_drv_get_instr_name(ble_drv_instr_hdl instr_hdl, char *name, int size)

The ble_tester_drv_get_instr_name API function returns the voltage meter instrument DLL names as these were taken when the ble_tester_drv_init API function was called. It is mainly used to display the DLL names in the CFG PLT application, so users can select a specific DLL to use together with a specific BLE tester instrument.

BLE_TESTER_DRV_API ble_drv_instr_hdl ble_tester_drv_get_instr_hdl(ble_drv_instr_hdl prev_instr_hdl)

The ble_tester_drv_get_instr_hdl API function returns the handles of the BLE tester instrument DLLs created when the ble_tester_drv_init API function was called. First call of this function must be made with the prev_instr_hdl input parameter set to NULL. All subsequent calls must be made with the prev_instr_hdl input parameter set to the handle returned from the previous function call. When this function returns NULL no more instrument handles exist.

BLE_TESTER_DRV_API int ble_tester_drv_instr_init(ble_drv_instr_hdl instr_hdl, void *data, _callback_ble_instr callback_ble_instr)

The ble_tester_drv_instr_init API function initializes a specific BLE tester instrument DLL that is pointed by the instr_hdl handle. It actually calls the ble_instr_init API function for the instrument DLL pointed by the instr_hdl handle. The input callback_ble_instr parameter is used to return measurement status and results through callbacks. A pointer to a callback function is passed that will be called when a BLE RF measurement is ready. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_instr_close(ble_drv_instr_hdl instr_hdl);

The ble_tester_drv_instr_close API function closes the BLE tester instrument DLL. It actually calls the ble_instr_close API function for the instrument DLL pointed by the instr_hdl handle. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_path_loss(ble_drv_instr_hdl instr_hdl, float path_loss)

The ble_tester_drv_set_path_loss API function sets the path losses between the device antenna and the antenna of the BLE instrument used. It actually calls the ble_instr_set_path_loss API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_pwr_range(ble_drv_instr_hdl instr_hdl, _ble_instr_pwr_range pwr_range);

The ble_tester_drv_set_pwr_range API function is used to set the TX input power range in the BLE instrument DLL that has a handle pointed by the instr_hdl input parameter. This function calls the ble_instr_set_pwr_range API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS otherwise, it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_tx_pwr_h_lim(ble_drv_instr_hdl instr_hdl, float avg_high_limit)

The ble_tester_drv_set_tx_pwr_h_lim API function is used to set the average high power limit for the TX output power measurements. This function calls the ble_instr_set_tx_pwr_h_lim API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_tx_pwr_l_lim(ble_drv_instr_hdl instr_hdl, float avg_low_limit)

The ble_tester_drv_set_tx_pwr_l_lim API function is used to set the average low power limit for the TX output power measurements. This function calls the ble_instr_set_tx_pwr_l_lim API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_tx_pwr_pk_lim(ble_drv_instr_hdl instr_hdl, float pk_avg_limit)

The ble_tester_drv_set_tx_pwr_pk_lim API function is used to set the peak to average power limit for the TX output power measurements. This function calls the ble_instr_set_tx_pwr_pk_lim API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_do_tx_pwr(ble_drv_instr_hdl instr_hdl, uint32_t freq)

The ble_tester_drv_do_tx_pwr API function is used to start the TX output power measurements at the frequency given to the freq input parameter. This function calls the ble_instr_do_tx_pwr API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_freq_offs_h_lim(ble_drv_instr_hdl instr_hdl, uint32_t pos_freq_limit)

The ble_tester_drv_set_freq_offs_h_lim API function is used to set the maximum positive offset limit in kHz for the TX carrier frequency offset measurements. This function calls the ble_instr_set_freq_offs_h_lim API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_freq_offs_l_lim(ble_drv_instr_hdl instr_hdl, uint32_t neg_freq_limit)

The ble_tester_drv_set_freq_offs_l_lim API function is used to set the maximum negative offset limit in kHz for the TX carrier frequency offset measurements. This function calls the ble_instr_set_freq_offs_l_lim API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_freq_drift_pkt_lim(ble_drv_instr_hdl instr_hdl, uint32_t drift_pkt_limit)

The ble_tester_drv_set_freq_drift_pkt_lim API function is used to set the overall packet drift limit in kHz for the TX carrier frequency offset measurements. This function calls the ble_instr_set_freq_drift_pkt_lim API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_freq_drift_rate_lim(ble_drv_instr_hdl instr_hdl, uint32_t drift_rate_limit)

The ble_tester_drv_set_freq_drift_rate_lim API function is used to set the drift rate limit in kHz/50us for the TX carrier frequency offset and drift measurements. This function calls the ble_instr_set_freq_drift_rate_lim API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_do_freq_offs(ble_drv_instr_hdl instr_hdl, uint32_t freq)

The ble_tester_drv_do_freq_offs API function is used to start the TX carrier frequency offset and drift measurements at the frequency given to the freq input parameter. This function calls the ble_instr_do_freq_offs API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_mod_idx_f1_min(ble_drv_instr_hdl instr_hdl, uint32_t f1_min_limit)

The ble_tester_drv_set_mod_idx_f1_min API function is used to set the F1 minimum average limit in kHz for the TX modulation index measurements. This function calls the ble_instr_set_mod_idx_f1_min API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_mod_idx_f1_max(ble_drv_instr_hdl instr_hdl, uint32_t f1_max_limit)

The ble_tester_drv_set_mod_idx_f1_max API function is used to set the F1 maximum average limit in kHz for the TX modulation index measurements. This function calls the ble_instr_set_mod_idx_f1_max API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_mod_idx_f2_max(ble_drv_instr_hdl instr_hdl, uint32_t f2_max_limit)

The ble_tester_drv_set_mod_idx_f2_max API function is used to set the F2 maximum average limit in kHz for the TX modulation index measurements. This function calls the ble_instr_set_mod_idx_f2_max API function for the instrument DLL pointed by the instr_hdl input parameter. If the function succeeds it will return BLE_TESTER_DRV_SUCCESS, otherwise it will return BLE_TESTER_DRV_ERROR.

BLE_TESTER_DRV_API int ble_tester_drv_set_mod_idx_f1f2_ratio(ble_drv_instr_hdl instr_hdl, float f1f2_ratio_limit)

The `ble_tester_drv_set_mod_idx_f1f2_ratio` API function is used to set the F1/F2 maximum average ratio limit for the TX modulation index measurements. This function calls the `ble_instr_set_mod_idx_f1f2_ratio` API function for the instrument DLL pointed by the `instr_hdl` input parameter. If the function succeeds it will return `BLE_TESTER_DRV_SUCCESS`, otherwise it will return `BLE_TESTER_DRV_ERROR`.

BLE_TESTER_DRV_API int ble_tester_drv_do_mod_idx(ble_drv_instr_hdl instr_hdl, uint32_t freq)

The `ble_tester_drv_do_mod_idx` API function is used to start the TX modulation index measurement at the frequency given to the `freq` input parameter. This function calls the `ble_instr_do_mod_idx` API function for the instrument DLL pointed by the `instr_hdl` input parameter. If the function succeeds it will return `BLE_TESTER_DRV_SUCCESS`, otherwise it will return `BLE_TESTER_DRV_ERROR`.

BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_tx_pat(ble_drv_instr_hdl instr_hdl, uint8_t pattern)

The `ble_tester_drv_set_rx_sens_tx_pat` API function is used to set the TX packet pattern type that the BLE tester instrument will transmit when performing RX RSSI measurements. This function calls the `ble_instr_set_rx_sens_tx_pat` API function for the instrument DLL pointed by the `instr_hdl` input parameter. If the function succeeds it will return `BLE_TESTER_DRV_SUCCESS`, otherwise it will return `BLE_TESTER_DRV_ERROR`.

BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_pkt_space(ble_drv_instr_hdl instr_hdl, uint16_t spacing)

The `ble_tester_drv_set_rx_sens_pkt_space` API function is used to set the TX packet spacing that the BLE tester instrument will transmit when performing RX RSSI measurements. This function calls the `ble_instr_set_rx_sens_pkt_space` API function for the instrument DLL pointed by the `instr_hdl` input parameter. If the function succeeds it will return `BLE_TESTER_DRV_SUCCESS` otherwise it will return `BLE_TESTER_DRV_ERROR` on failure.

BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_pkt_num(ble_drv_instr_hdl instr_hdl, uint16_t num_of_pkts)

The `ble_tester_drv_set_rx_sens_pkt_num` API function is used to set the TX packets number that the BLE tester instrument will transmit when performing RX RSSI measurements. This function calls the `ble_instr_set_rx_sens_pkt_num` API function for the instrument DLL pointed by the `instr_hdl` input parameter. If the function succeeds it will return `BLE_TESTER_DRV_SUCCESS`, otherwise it will return `BLE_TESTER_DRV_ERROR`.

BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_tx_pwr(ble_drv_instr_hdl instr_hdl, float tx_power)

The `ble_tester_drv_set_rx_sens_tx_pwr` API function is used to set the TX output power that the BLE tester instrument will transmit when performing RX RSSI measurements. This function calls the `ble_instr_set_rx_sens_tx_pwr` API function for the instrument DLL pointed by the `instr_hdl` input parameter. If the function succeeds it will return `BLE_TESTER_DRV_SUCCESS`, otherwise it will return `BLE_TESTER_DRV_ERROR`.

BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_tx_dirty(ble_drv_instr_hdl instr_hdl, bool dirty)

The `ble_tester_drv_set_rx_sens_tx_dirty` API function enables or disables the TX dirty option used in the RX RSSI measurements. This function calls the `ble_instr_set_rx_sens_tx_dirty` API function for the instrument DLL pointed by the `instr_hdl` input parameter. If the function succeeds it will return `BLE_TESTER_DRV_SUCCESS`, otherwise it will return `BLE_TESTER_DRV_ERROR`.

BLE_TESTER_DRV_API int ble_tester_drv_set_rx_sens_tx_crc(ble_drv_instr_hdl instr_hdl, bool crc_state)

The `ble_tester_drv_set_rx_sens_tx_crc` API function enables or disables the TX CRC alternate state option used in the RX RSSI measurements. This function calls the `ble_instr_set_rx_sens_tx_crc` API function for the instrument DLL pointed by the `instr_hdl` input parameter. If the function succeeds it will return `BLE_TESTER_DRV_SUCCESS`, otherwise it will return `BLE_TESTER_DRV_ERROR`.

BLE_TESTER_DRV_API int ble_tester_drv_do_rx_sens(ble_drv_instr_hdl instr_hdl, uint32_t freq)

The `ble_tester_drv_do_rx_sens` API function is used to start the BLE tester packet generator at the frequency given to the `freq` input parameter. This function calls the `ble_instr_do_rx_sens` API function for the instrument DLL pointed by the `instr_hdl` input parameter. If the function succeeds it will return `BLE_TESTER_DRV_SUCCESS`, otherwise it will return `BLE_TESTER_DRV_ERROR`.

13.2 BLE_TESTER_DRIVER API Details

More details on the BLE_TESTER_DRIVER API can be found either in the API header file in `source\production_line_tool\instruments\ble_testers\ble_tester_driver\ble_tester_driver.h`, or in the HTML based help pages loaded after pressing the `source\production_line_tool\help\help.html` link.

14 NI_USB_TC01 DLL

The ni_usb_tc01.dll is a DLL with an API used to take ambient temperature measurements using the NI USB TC01 temperature sensor [11]. The API used in the ni_usb_tc01.dll is designed in such a way so it can be used by other temperature sensor measurement instruments as well. All temperature measurement settings are configurable giving a great flexibility to the user.

For the DLL to operate the **NI-VISA** software installation is needed. It can be downloaded from the path shown in Table 16.

Folder source\production_line_tool\instruments\temp_sensors\ni_usb_tc01 contains all the necessary source code for the particular DLL. The API of the DLL is defined in source\production_line_tool\instruments\temp_sensors\temp_meas_driver\temp_meas_api.h.

14.1 NI_USB_TC01 API Functions

The ni_usb_tc01.dll has the following user accessible functions:

```
TEMP_MEAS_API int temp_meas_api_dbg_init(_dbg_params *dbg_params_t);
TEMP_MEAS_API int temp_meas_api_dbg_close(void);
TEMP_MEAS_API int temp_meas_api_init(char *iface, _callback_temp_meas
callback_temp_meas);
TEMP_MEAS_API int temp_meas_api_close(void);
TEMP_MEAS_API int temp_meas_api_measure(void);
```

A detailed description of the API functions will be given next.

TEMP_MEAS_API int temp_meas_api_dbg_init(_dbg_params *dbg_params_t)

The temp_meas_api_dbg_init API function is used to initialize the debug print operation. Details on the dbg_params_t input parameter can be found in section 6.1.1.1. This function will return TEMP_MEAS_API_INVALID_DBG_PARAMS if an invalid dbg_params_t input parameter is given. If an error occurred during dbg_dll.dll initialization, the function will return TEMP_MEAS_API_DBG_DLL_ERROR. On success the function will return TEMP_MEAS_API_SUCCESS.

TEMP_MEAS_API int temp_meas_api_dbg_close(void)

The temp_meas_api_dbg_close API function closes the debug session. It calls the dbg_close API function from dbg_dll.dll to free all the resources acquired. It always returns TEMP_MEAS_API_SUCCESS.

TEMP_MEAS_API int temp_meas_api_init(char *iface, _callback_temp_meas callback_temp_meas)

The temp_meas_api_init API function resets and initializes the instrument. The iface input pointer parameter points to the interface string, which even if not used should have a valid pointer to an allocated space. The ni_udb_tc01.dll does not use this parameter, because the interface is automatically found by the software that uses appropriate NI VISA functions. On the other hand, the tmu_temp_sens.dll temperature measurement DLL for the Papouch TMU temperature sensor [10] is using this parameter. The tmu_temp_sens.dll does not use NI VISA but RS232 serial interface to communicate. In that case the iface parameter provides the COM port. The callback_temp_meas is a pointer to a callback function that the DLL will call to update measurement status and results. If the function succeeds it returns TEMP_MEAS_API_SUCCESS otherwise it returns TEMP_MEAS_API_ERROR on failure.

TEMP_MEAS_API int temp_meas_api_close(void)

The temp_meas_api_close API function deallocates all resources allocated during the temperature measurements. Closes the measurement thread, if this has been remained opened for some reason, disables any GPIB service requests if NI VISA is used or closes any used COM port resources. If the function succeeds it returns TEMP_MEAS_API_SUCCESS, otherwise it returns TEMP_MEAS_API_ERROR.

TEMP_MEAS_API int temp_meas_api_measure(void)

The temp_meas_api_measure API function is used to start the temperature measurement operation. It creates a thread for the measurements to take place and initializes NI VISA resources if needed. It then returns immediately with TEMP_API_SUCCESS if the measurement operation started successfully or TEMP_API_ERROR if an error occurred. The actual measurement results will be returned using the callback function registered when the temp_meas_api_init API function was called. The callback function has the following type.

```
typedef void (*_callback_temp_meas)(int status, float temp);
```

When the status input parameter is set to TEMP_MEAS_API_READ_OK the output temperature result is valid and it is returned in the temp argument. The upper layer software handling the callback can then read the result.

14.2 NI_USB_TC01 API Details

More details on the NI_USB_TC01 API can be found either in the API header file in source\production_line_tool\instruments\temp_sensors\temp_meas_driver\temp_meas_api.h or in the HTML based help pages loaded after pressing the source\production_line_tool\help\help.html link.

15 TMU_TEMP_SENS DLL

The `tmu_temp_sens.dll` is a DLL with an API used to take ambient temperature measurements using the Papouch TMU temperature sensor [10]. The API is the same as the one described for the NI USB TC01 sensor in section 14.

Folder `source\production_line_tool\instruments\temp_sensors\tmu_temp_sens` contains all the necessary source code for the particular DLL. The API of the DLL is defined in `source\production_line_tool\instruments\temp_sensors\temp_meas_driver\temp_meas_api.h`.

15.1 TMU_TEMP_SENS API Functions

The `tmu_temp_sens.dll` has the same API functions as the NI USB TC01 sensor described in section 14.1.

The API is the same in order for the upper layer software, the `TEMP_MEAS_DRIVER` described next in section 16, to be able to load and access both. By using function pointers to those two DLLs the `TEMP_MEAS_DRIVER` can operate with any one.

15.2 TMU_TEMP_SENS API Details

More details on the TMU_TEMP_SENS API can be found either in the API header file in `source\production_line_tool\instruments\temp_sensors\temp_meas_driver\temp_meas_api.h` or in the HTML based help pages loaded after pressing the `source\production_line_tool\help\help.html` link.

16 TEMP_MEAS_DRIVER DLL

The temp_meas_driver.dll is a DLL driver used to load and access different temperature measurement DLLs from the temp_meas_instr_plugins folder. It is able to return all the DLL names existing in that folder and is able to use any one of them, as long as they are following the correct API found under source\production_line_tool\instruments\temp_sensors\temp_meas_driver\temp_meas_api.h.

The API in the temp_meas_api.h file was actually described in section 14 since it is the one that ni_usb_tc01.dll and tmu_temp_sens.dll are using. The CFG application uses the temp_meas_driver.dll through the PLTD to display the available temperature measurement DLLs (e.g. tmu_temp_sens.dll). Users can then select a DLL name and all the temperature measurements will go through the one selected. The actual block diagram describing this operation is shown in Figure 18.

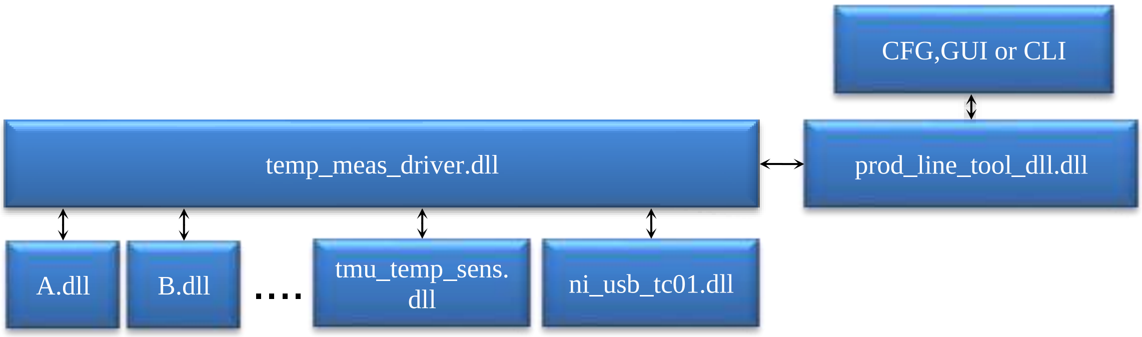


Figure 18: temp_meas_driver.dll Usage Block Diagram

During the prod_line_tool_dll.dll initialization phase (pltd_init) the temp_meas_driver.dll is also initialized. During temp_meas_driver.dll initialization all the DLLs found in the temp_meas_instr_plugins folder are loaded. The driver stores the loaded DLL names and creates function handles in a linked list of the driver. Users can then get all the temperature measurement DLL names by calling the pltd_get_temp_meas_instr_names prod_line_tool.dll API function.

16.1 TEMP_MEAS_DRIVER API Functions

The temp_meas_driver.dll has the following user accessible functions:

```
TEMP_MEAS_DRV_API int temp_meas_drv_init(void);
TEMP_MEAS_DRV_API int temp_meas_drv_close(void);
TEMP_MEAS_DRV_API int temp_meas_drv_dbg_init(_dbg_params *dbg_params_t);
TEMP_MEAS_DRV_API int temp_meas_drv_dbg_close(void);
TEMP_MEAS_DRV_API int temp_meas_drv_get_instr_name(temp_meas_drv_instr_hdl instr_hdl,
char *name, int size);
TEMP_MEAS_DRV_API temp_meas_drv_instr_hdl
temp_meas_drv_get_instr_hdl(temp_meas_drv_instr_hdl prev_instr_hdl);
TEMP_MEAS_DRV_API int temp_meas_drv_instr_init(temp_meas_drv_instr_hdl instr_hdl, char
*iface, _callback_temp_meas callback_temp_meas);
TEMP_MEAS_DRV_API int temp_meas_drv_instr_close(temp_meas_drv_instr_hdl instr_hdl);
TEMP_MEAS_DRV_API int temp_meas_drv_instr_measure(temp_meas_drv_instr_hdl instr_hdl);
```

A detailed description of the API functions will be given next.

TEMP_MEAS_DRV_API int temp_meas_drv_init(void)

The temp_meas_drv_init API function initializes the DLL. It searches the temp_meas_instr_plugins folder to find temperature measurement instrument DLLs. It loads the DLLs found and saves the names and the function handles in an internal linked list. If no instrument DLLs are found the function will return `TEMP_MEAS_DRV_NO_INSTR_PLUGINS`. On a system error the function will return `TEMP_MEAS_DRV_ERROR` or `TEMP_MEAS_DRV_SUCCESS` otherwise.

TEMP_MEAS_DRV_API int temp_meas_drv_close(void)

The `temp_meas_drv_close` API function frees the allocated resources of all the temperature measurement DLL plugins created during the initialization process. The function will always return `TEMP_MEAS_DRV_SUCCESS`.

TEMP_MEAS_DRV_API int temp_meas_drv_dbg_init(_dbg_params *dbg_params_t)

The `temp_meas_drv_dbg_init` API function initializes the debug information. Details on the `dbg_params_t` input parameter can be found in section 6.1.1.1. This function will return `TEMP_MEAS_DRV_INVALID_DBG_PARAMS` if an invalid `dbg_params_t` input parameter is given. If an error occurred during `dbg_dll.dll` initialization, the function will return `TEMP_MEAS_DRV_DBG_DLL_ERROR`. On success the function will return `TEMP_MEAS_DRV_SUCCESS`.

TEMP_MEAS_DRV_API int temp_meas_drv_dbg_close(void)

The `temp_meas_drv_dbg_close` API function closes the debug session. It calls the `dbg_close` API function from `dbg_dll.dll`. If an error occurred in the `dbg_dll.dll` then `TEMP_MEAS_DRV_DBG_DLL_ERROR` will be returned. Otherwise the function returns `TEMP_MEAS_DRV_SUCCESS` on success.

TEMP_MEAS_DRV_API int temp_meas_drv_get_instr_name(temp_meas_drv_instr_hdl instr_hdl, char *name, int size)

The `temp_meas_drv_get_instr_name` API function returns the temperature measurement instrument DLL names as these were taken when the `temp_meas_drv_init` API function was called. It is mainly used to display the DLL names in the CFG PLT application, so users can select one to use together with a specific temperature measurement instrument.

TEMP_MEAS_DRV_API temp_meas_drv_instr_hdl temp_meas_drv_get_instr_hdl(temp_meas_drv_instr_hdl prev_instr_hdl)

The `temp_meas_drv_get_instr_hdl` API function returns the handles of the temperature measurement instrument DLLs created when the `temp_meas_drv_init` API function was called. First call of this function must be made with the `prev_instr_hdl` input parameter set to NULL. All subsequent calls must be made with the `prev_instr_hdl` input parameter set to the handle returned from the previous function call. When this function returns NULL, no more instrument handles exist.

TEMP_MEAS_DRV_API int temp_meas_drv_instr_init(temp_meas_drv_instr_hdl instr_hdl, char *iface, _callback_temp_meas callback_temp_meas)

The `temp_meas_drv_instr_int` API function initializes a specific temperature measurement instrument DLL that is pointed by the `instr_hdl` handle. It actually calls the `temp_meas_api_init` API function from the DLL pointed by the `instr_hdl` handle. The input `callback_temp_meas` parameter is used to return measurement status and results through callbacks. A pointer to a callback function is passed, which will be called when a temperature measurement is ready. If the function succeeds it will return `TEMP_MEAS_DRV_SUCCESS`, otherwise it will return `TEMP_MEAS_DRV_ERROR`.

TEMP_MEAS_DRV_API int temp_meas_drv_instr_close(temp_meas_drv_instr_hdl instr_hdl)

The `temp_meas_drv_instr_close` API function closes the temperature measurement instrument DLL. It actually calls the `temp_meas_api_close` API function from the DLL pointed by the `instr_hdl` handle. If the function succeeds it will return TEMP_MEAS_DRV_SUCCESS, otherwise it will return TEMP_MEAS_DRV_ERROR.

TEMP_MEAS_DRV_API int temp_meas_drv_instr_measure(temp_meas_drv_instr_hdl instr_hdl)

The temp_meas_drv_instr_measure API function starts the temperature measurements. It actually calls the temp_meas_api_measure API function from the instrument DLL pointed by the instr_hdl input parameter handle. In the currently supported instrument DLLs, the temp_meas_api_measure API function creates a thread that waits for the instrument to return some measurements. After averaging the measurements, the result is returned using callbacks. The function returns immediately but the upper layer software has to wait for the callback to arrive, in order to get the temperature results. If the function succeeds to start the measurement and create the measurement thread it will return TEMP_MEAS_DRV_SUCCESS, otherwise it will return TEMP_MEAS_DRV_ERROR.

16.2 TEMP_MEAS_DRIVER API Details

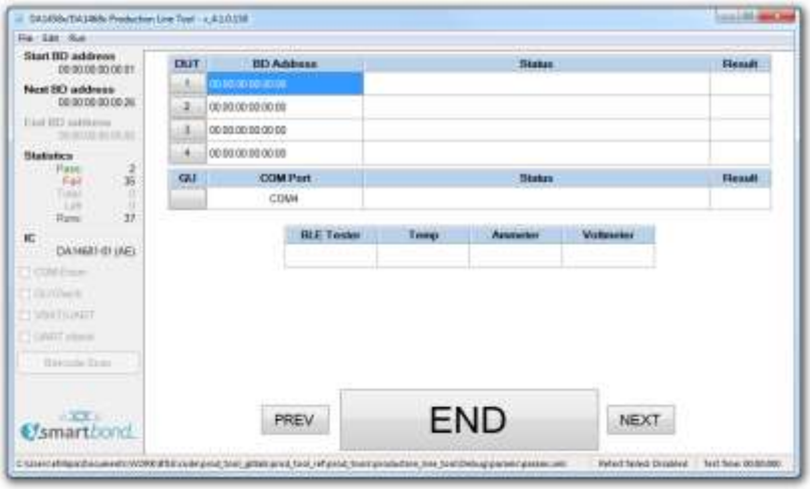
More details on the TEMP_MEAS_DRIVER API can be found in the API header file in source\production_line_tool\instruments\temp_sensors\temp_meas_driver\temp_meas_driver.h or in the HTML based help pages loaded after pressing the source\production_line_tool\help\help.html link.

17 BARCODE_SCANNER DLL

The `barcode_scanner.dll` is a DLL that is directly loaded and used by the GUI PLT. It provides a generic interface to scan device BD addresses and memory data. The barcode scanner instrument should be connected to the PC using a USB to RS232 or plain RS232 interface. Ideally, any barcode scanner instrument that has USB to RS232 or plain RS232 interface could be used with the `barcode_scanner.dll`. However, the particular DLL has been tested with the Honeywell Xenon 1900 USB to RS232 barcode scanner [12] and the Motorola LS2208 [13]. The DLL supports two modes of operation as described in Table 17.

Table 17: Barcode Scanner Modes of Operation

<table>
<tr><th>Barcode Scan Operation</th><th>Description</th></tr>
<tr><td>Auto</td><td>When the auto mode is selected, users only need to scan device BD addresses one after the other. The device to be scanned is automatically selected by the upper layer software (GUI). The GUI automatically increments the device selected to be scanned. Additionally, in the GUI there are user accessible controls where any of the devices to be scanned can be selected manually.

The following picture illustrates the barcode scan screen. Buttons NEXT or PREV can be pressed by the users to select any of the four active devices. Currently, device 1 is selected. If user presses the NEXT button device 2 will be selected and if the barcode scanner scans a BD address it will be passed to device 2. If Custom Memory Data are also to be scanned using a barcode scanner then the PLT will select the next cell named Memory Data, in the same DUT row.


After the BD address of the 2nd device has been scanned, the GUI tool will automatically select the 3rd device. If a BD address is scanned again, the BD address will be saved in the third device and the GUI tool will automatically select the next device, device 4.</td></tr>
<tr><td>Scan position</td><td>In this mode the GUI PLT waits for a specific callback status code from the barcode scanner in order to select a different device to be scanned. The callback status code is the `BARCODE_SCANNER_API_POS_READ_OK`. This callback status code comes with data that denote which device to select. The barcode scanner should scan a special word with the following format: ‘`TEST POSITION XXX`’. Word ‘`XXX`’ should be replaced with the actual device position.

For example, if the user would like to scan the BD address for device 10, then the word to scan should be ‘`TEST POSITION 010`’. When this word is scanned and if the automatic mode is disabled, the `barcode_scanner.dll` will send a callback to the upper layer software, the GUI in that case, with status code `BARCODE_SCANNER_API_POS_READ_OK` and data value from 1 to 16 indicating the device to be selected.</td></tr>
</table>

17.1 BARCODE_SCANNER API Functions

The `barcode_scanner.dll` has the following user accessible functions.

```
BARCODE_SCANNER_API int barcode_scanner_api_dbg_init((_dbg_params *dbg_params_t);
BARCODE_SCANNER_API int barcode_scanner_api_dbg_close(void);
BARCODE_SCANNER_API int barcode_scanner_api_init(_barcode_scanner_cfg *cfg);
BARCODE_SCANNER_API int barcode_scanner_api_close(void);
BARCODE_SCANNER_API int barcode_scanner_api_get_BDA(void);
BARCODE_SCANNER_API int barcode_scanner_api_get_mem_data(uint32_t size);
```

A detailed description of the API functions will be given next.

BARCODE_SCANNER_API int barcode_scanner_api_dbg_init(_dbg_params *dbg_params_t)

The `barcode_scanner_api_dbg_init` API function initializes the debug information. Details on the `dbg_params_t` input parameter can be found in section 6.1.1.1. This function will return `BARCODE_SCANNER_API_INVALID_DBG_PARAMS` if an invalid `dbg_params_t` input parameter is given. If an error occurs during `dbg_dll.dll` initialization, the function will return `BARCODE_SCANNER_API_DBG_DLL_ERROR`. On success the function will return `BARCODE_SCANNER_API_DBG_DLL_SUCCESS`.

BARCODE_SCANNER_API int barcode_scanner_api_dbg_close(void)

The `barcode_scanner_api_dbg_close` API function closes the debug session. It calls the `dbg_close` API function from `dbg_dll.dll`. The function always returns `BARCODE_SCANNER_API_SUCCESS`.

BARCODE_SCANNER_API int barcode_scanner_api_init(_barcode_scanner_cfg *cfg);

The `barcode_scanner_api_init` API function initializes the barcode scanner DLL. It takes as argument the following data structure.

```
typedef struct __barcode_scanner_cfg
{
    char iface[256];
    bool mode_auto;
    _callback_barcode_scanner callback_barcode_scanner;
}_barcode_scanner_cfg;
```

In the `iface` parameter a COM port number is expected. This should be the COM port number of the barcode scanner instrument. The API function will open the PC COM port and keep the COM port handle open to be used during the scan operation. The PC COM port initialization settings are illustrated in the following Table 18.

Table 18: Barcode Scanner COM Port Settings

Parameter	Value
Baud rate	115200
Byte size	8
Parity	Disabled
Stop bits	One
Flow control	Disabled

The second function parameter `mode_auto`, is used to set the DLL scan operation mode as described in Table 17. The last parameter, `callback_barcode_scanner`, is a pointer to the upper layer software, pointing to the function to be called when a valid scan takes place.

BARCODE_SCANNER_API int barcode_scanner_api_get_BDA(void)

The barcode_scanner_api_get_BDA API function is used to start the barcode scan operation. When called, the DLL will create a thread for the scan messages to be received. It will then return immediately with BARCODE_SCANNER_API_SUCCESS if the thread was successfully created or BARCODE_SCANNER_API_ERROR if an error occurred. The upper layer software should wait for the scan results through the callback function pointer passed in the barcode_scanner_api_init API function.

The callback function has the following type.

```
typedef void (_stdcall *_callback_barcode_scanner)(int status, uint32_t size, uint8_t
*data);
```

BARCODE_SCANNER_API int barcode_scanner_api_get_mem_data(uint32_t size)

The barcode_scanner_api_get_mem_data API function is used to start the barcode scan operation in order to scan data to be burned into the device memory. When called, the DLL will create a thread for the scan messages to be received. It will then return immediately with BARCODE_SCANNER_API_SUCCESS if the thread was successfully created or BARCODE_SCANNER_API_ERROR if an error occurred. The upper layer software should wait for the scan results through the callback function pointer passed in the barcode_scanner_api_init API function.

The callback function has the following type.

```
typedef void (_stdcall *_callback_barcode_scanner)(int status, uint32_t size, uint8_t
*data);
```

The status parameter can take one of the following enumeration values.

```
typedef enum _BARCODE_SCANNER_API_STATUS_CODES
{
    BARCODE_SCANNER_API_SUCCESS = 0,
    BARCODE_SCANNER_API_ERROR,
    BARCODE_SCANNER_API_INVALID_DBG_PARAMS,
    BARCODE_SCANNER_API_DBG_DLL_ERROR,
    BARCODE_SCANNER_API_START,
    BARCODE_SCANNER_API_BDA_READ_OK,
    BARCODE_SCANNER_API_BDA_READ_ERROR,
    BARCODE_SCANNER_API_POS_READ_OK,
    BARCODE_SCANNER_API_DATA_READ_OK,
    BARCODE_SCANNER_API_DATA_READ_ERROR
}BARCODE_SCANNER_API_STATUS_CODES;
```

Callback data passed to the data callback parameter pointer are only valid when the BARCODE_SCANNER_API_BDA_READ_OK or the BARCODE_SCANNER_API_POS_READ_OK callback status is received. If BARCODE_SCANNER_API_DATA_READ_OK is received then data contains a valid scanned data with up to 256 bytes of allowable size. If BARCODE_SCANNER_API_POS_READ_OK is received then data contain a valid device position number from 1 to 16.

17.2 BARCODE_SCANNER API Details

More details on the BARCODE_SCANNER API can be found in the API header file in source\production_line_tool\instruments\barcode_scanner\barcode_scanner.h or in the HTML based help pages loaded after pressing the source\production_line_tool\help\help.html link.

Revision History

Revision	Date	Description
1.0	26-Jan-2015	Initial release version for DA1458x_Production_Line_Tool_v_3.0.7.494 release.
2.0	14-Jul-2015	Updated for DA14580_Production_Line_Tool_v_3.170.2 release.
3.0	28-Apr-2016	Updated for DA1458x_68x_PLT_v3 release.
3.1	05-Aug-2016	Updated for DA1458x_DA1468x_PLT_v3.1 release. Template and spelling updated to latest branding guidelines.
4.0	13-Dec-2016	Updated for DA1458x_DA14658x_PLT_v4 release.
4.2	19-Oct-2017	Updated for DA1458x_DA14658x_PLT_v4.2 release.
4.3	17-Jan-2022	Updated logo, disclaimer, copyright.

DA1458x/DA1468x Production Line Tool Libraries

Status Definitions

Status	Definition
DRAFT	The content of this document is under review and subject to formal approval, which may result in modifications or additions.
APPROVED or unmarked	The content of this document has been approved for publication.