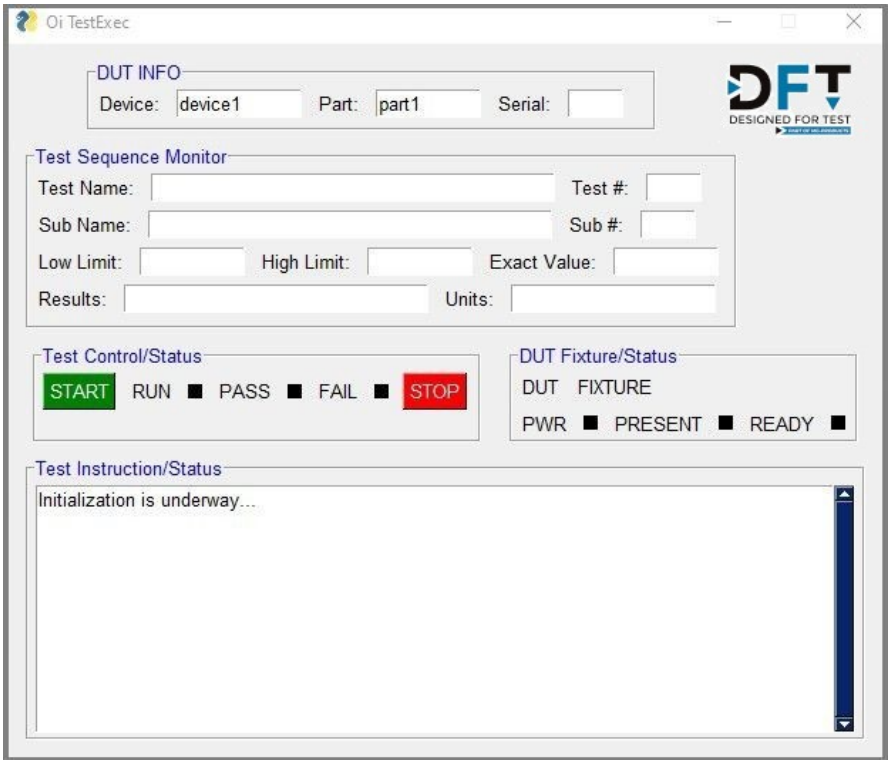
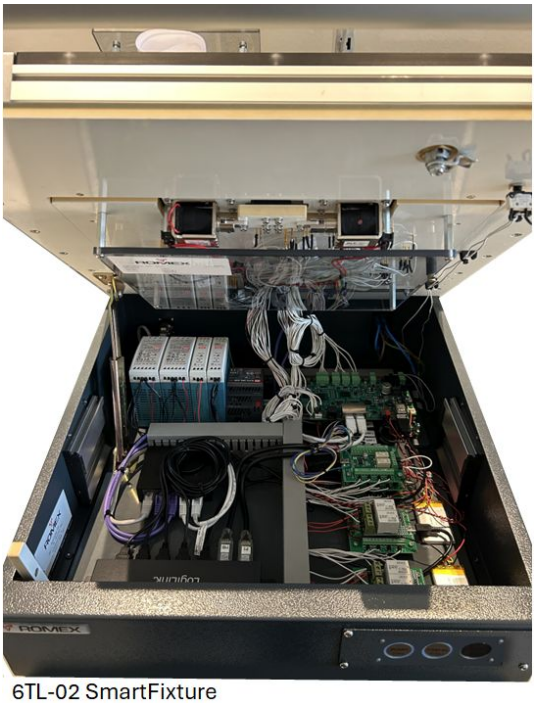
OI, Test Sequence Manager®



for the, DFT SmartFixture.



6TL-02 SmartFixture

MG-Products, Rijkevoortsedijk 27A, 5447BD, Rijkevoort, (NL) Phone: +31(0)485382133 email: info@MG-Products.com
www.designedfortest.com www.mg-products.com

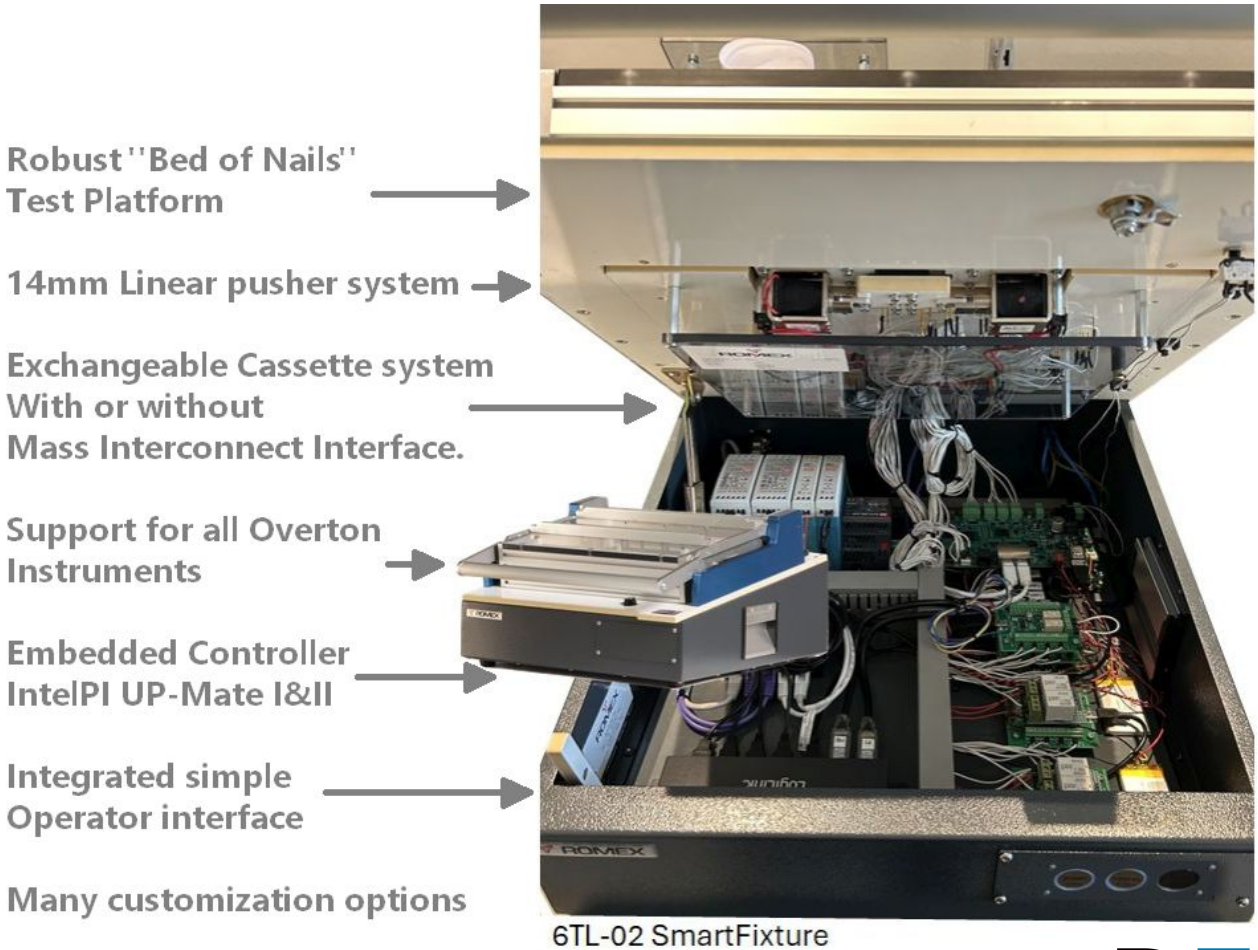
Table of Contents

1. Introduction

Welcome to the wonderful world of low-cost “automated” hardware test development. The purpose for this document is to acquaint the User with the many attributes of the Test Sequence Manager (TSM), and provide a guide to implement an actual Test Application which runs on the DFT-02 SmartFixture Test System. The next section provides a general description of the DFT-02 SmartFixture Test System, and beyond that we provide a deeper dive into the opera-tion of the TSM.

2. DFT-02 SmartFixture

As the diagram below illustrates, the DFT-02 SmartFixture is a highly integrated PCB Functional Test System. The system was designed to significantly reduce the high-cost of automated test, and (in doing so) allow "automatic test" to be cost-justified for PCB's that could not previously. Starting with a robust bed-of-nails test platform, the system will support a wide variety of PCB's (regardless of the size or complexity). Next, is the Sure Clamp (over-clamp assembly), which is a patented design that prevents PCB wobble or flap. The Probe Plate (the assembly that houses the pogo pins), features a unique capability that allows it to be removed and replaced (which provides a quick change over to accommodate a different DUT). The embedded instrumentation is provided by Overton Instruments, and includes our extensive line of test instrument modules. The Operator support function is provided by a simple I/O solution (located on the front panel). Finally, the complete test sequence process can be driven by a low-cost selection of embedded IntelPi computers (the UP-MATE I & II).

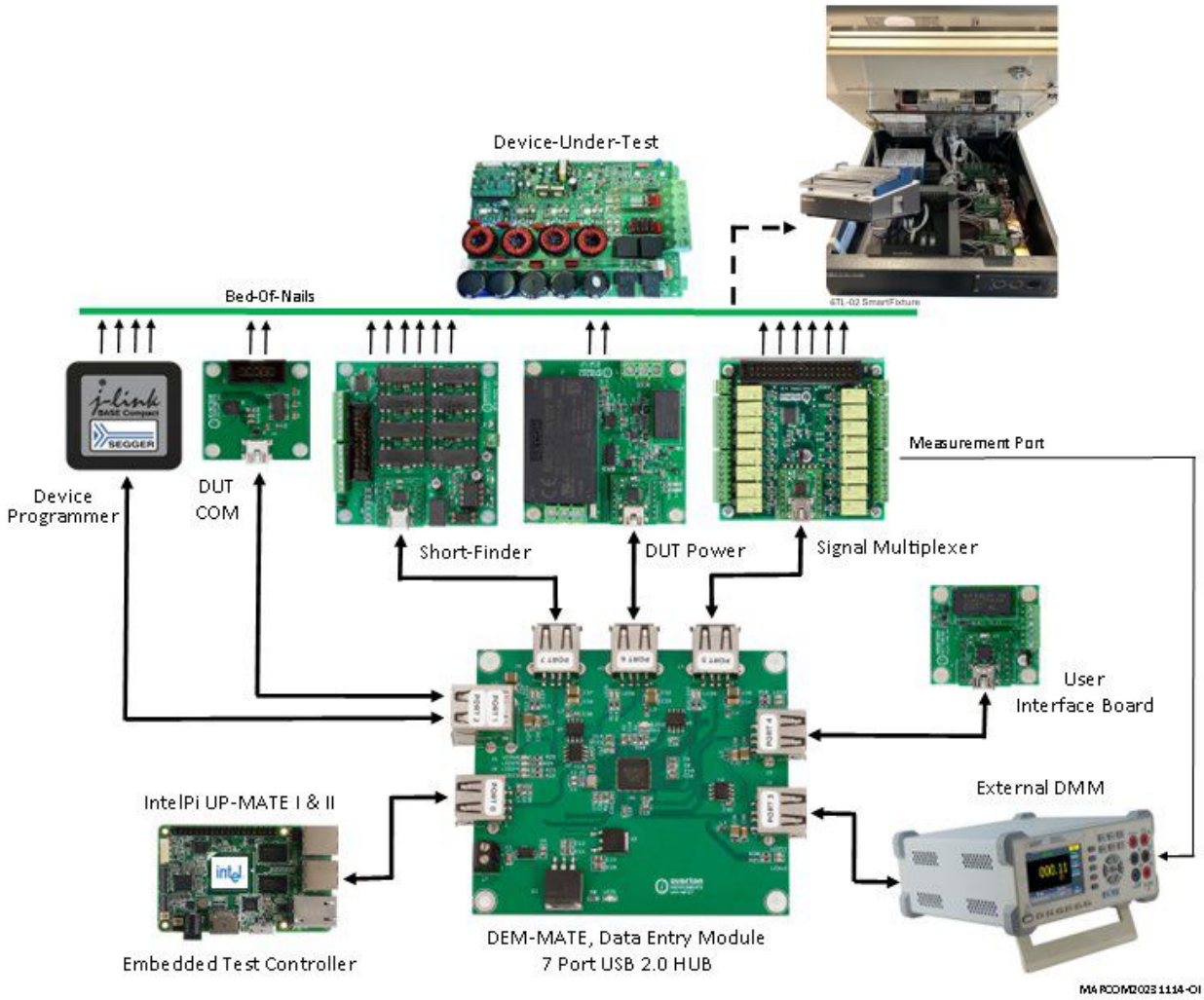


6TL-02 SmartFixture

DFT DESIGNED FOR TEST PART OF MG-PRODUCTS

2. DFT-02 SmartFixture - cont.,

As the diagram below illustrates, the DFT-02 SmartFixture can be quickly and easily transformed into what we call a ***"Smart Test Fixture".*** The goal of the configuration is to carry out a Pre Power Test (identify "short" conditions), a DUT Power Test (power the DUT and check "key" voltages and current at specific test points), and finally upload the DUT Production Code (use the flash programming to upload the code and verify the checksum at the end).

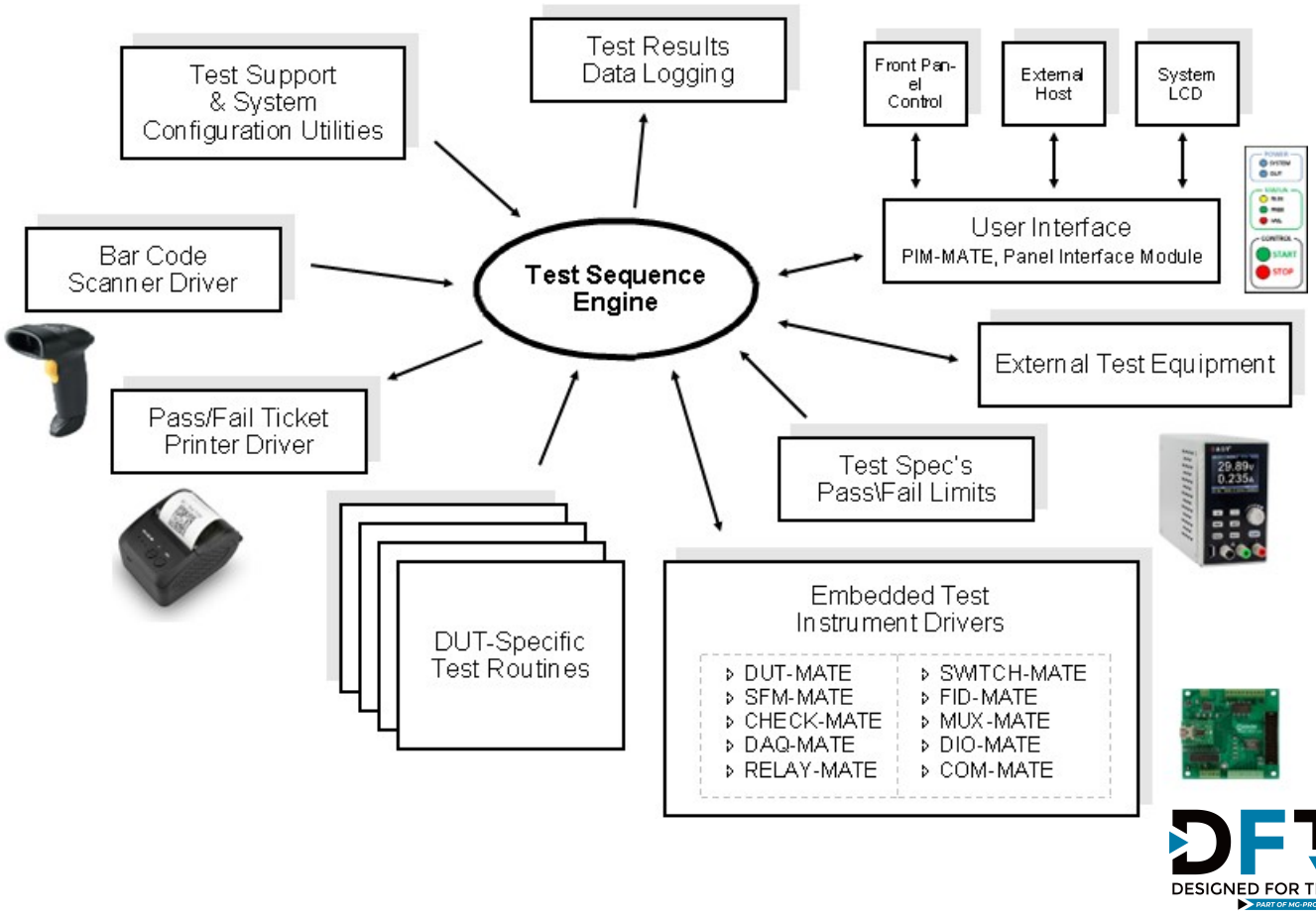


3. TSM, Test Sequence Manager

The TSM (Test Sequence Manager), is a “free” (open source) Test Executive for the DFT-02 SmartFixture Test System. The TSM is written in Python and includes all of the software components shown in the block diagram below. The TSM can run on either a Windows or Linux platform, or remotely (via external PC), or embedded (with the UP-MATE computers).

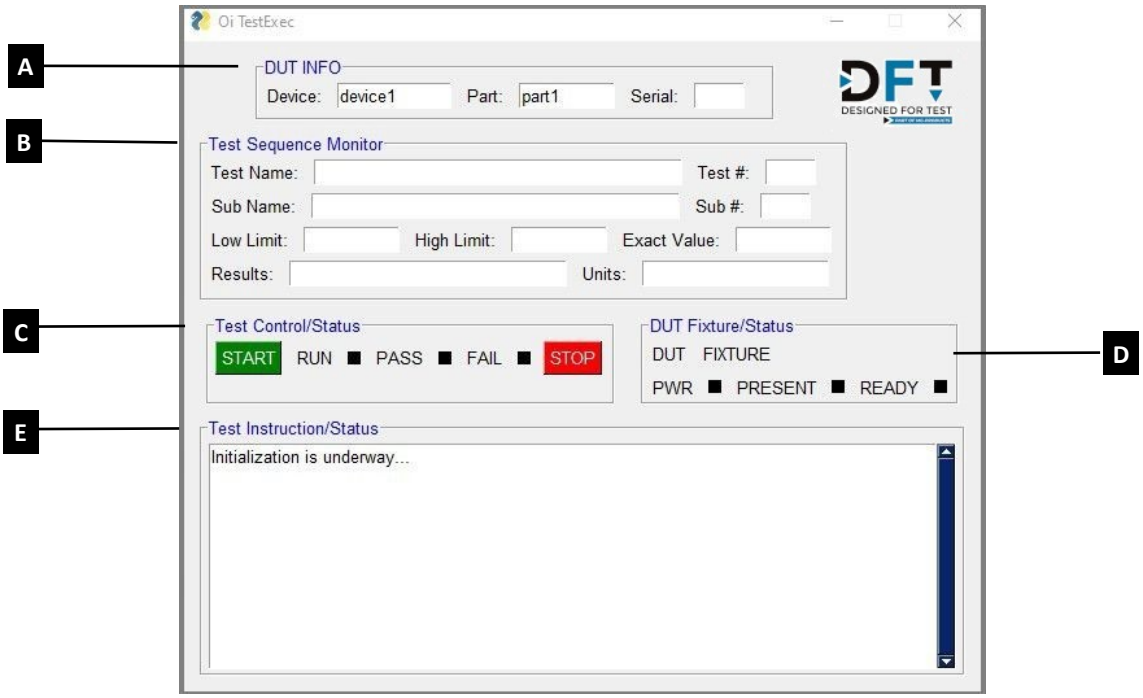
The TSM includes a unique Test Sequence Engine which is a special software routine that manages the complete Functional Test Process - by controlling the test operation, configuring the test equipment, acquiring a test measurement, determining Pass or Fail status, and logging the test results.

The TSM also includes a simple Graphical User Interface (GUI), which is highlighted on page 7, and is divided into 6 separate sections (A thru F). The **DUT Info** section, provides DUT related information (that is acquired from the UIM-MATE module). The **TEST Sequence Monitor,** provides pertinent information related to the current Test Step. The **TEST Control** section allows the Operator to START (resume), or STOP (abort) the test process. The **TEST Fixture** section, provides a collection of LED’s that indicate Fixture & DUT PWR status. The **TEST Instruction** section is used to display specific Operator instructions. The **POPUP Alerts**, are a collection of individual popup messages that are used to convey important alerts.



3. TSM, Test Sequence Manager - cont.,

TSM GUI, Graphical User Interface



A	DUT INFO
DUT NAME	Identifies the DUT by the part name
PART #	Indicates the DUT generic part number
SERIAL #	Indicates the DUT assigned serial number

B	TEST SEQUENCE MONITOR
TEST NAME	Identifies the test name
TEST #	Indicates the test number
SUB NAME	Identifies the sub-test name
SUB #	Indicates the sub-test number
LOW LIMIT	Indicates the lower P/F limit
HIGH LIMIT	Indicates the highest P/F limit
EXACT #	Indicates the exact value to measure

E	TEST INSTRUCTIONS/STATUS
	Display Operator instructions and general status information

C	TEST CONTROL/STATUS
START	Pushbutton for START or resume
RUN	Yellow LED, to indicate test is underway
PASS	Green LED, to indicate test PASS
FAIL	RED LED, to indicate test FAIL
STOP	Pushbutton for STOP or abort

D	DUT FIXTURE/STATUS
DUT PWR	Green LED, to indicate DUT power is ON
FIXTURE READY	Green LED, to indicate fixture over-clamp is engaged
DUT PRESENT	Green LED, to indicate the DUT is properly installed in the test fixture

F	POP UP MESSAGES/EVENTS
	Display a series or collection of high-level pop up alerts and messages, as needed.

4. How to build a Test Application

STEP 1, Setup your development environment

4.1.1 Download TSM_beta_xxx / Distribution.zip;
https://www.testprobes.nl/downloads/oi/Distribution.zip

4.1.2 Open and copy the TSM_beta_xxx folder to your PC

4.1.3 Download Python, https://www.python.org/downloads/

4.1.4 Download Visual Studio (optional), https://
code.visualstudio.com download

4.1.5 Configure Visual Studio to run Python code, https://

www.datacamp.com/tutorial/setting-up-vscode-python

4.1.6 Download VCP to run Overton Instruments, https://
www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers?
tab=downloads

4.1.7 Download Pyserial, pypi.org/project/pyserial/

4.1.8 Download NI VISA, https://www.ni.com/en/support/
downloads/drivers/download.ni-visa.html#521671

4.1.9 Download Py Simple GUI, https://pypi.org/project/
PySimpleGUI/

4. How to build a Test Application - cont.,

STEP 2, Review the Ebike example

The best way to understand how a Test Application is built, you should first review and examine a working example. On page 10, we have included a block diagram for a test system that is designed to validate a Motor Control Unit (MCU), which is part of an Ebike (electronic shooter). There are 2 items that are critical to the development of the Test Sequence Manager, and they are the Test Sequence Map and the TestLimits file. In the Ebike example, the Test Sequence Manager is programmed to run 4 separate tests, which are highlighted in both the Test Sequence Map and the TestLimits file.

Test Sequence Map

The Test Sequence Map is a document that contains a standard spreadsheet format and is unique to every Test Application. The objective is to provide a step-by-step list of all the relevant tests and sub-tests that comprise the Test Application. In addition, it attempts to itemize all of the interactions between the test instruments and the test process. Ultimately, the Test Sequence Map is designed to provide a clear "road map" for the Python programmer to update and modify the Test Sequence Manager (for a given Test Application). It should also be noted that the original information for the Test Sequence Map emanates from the Test Specification or Manufacturing Test Procedure for the PCB.

To view the Test Sequence Map for the Ebike example, go to the Application Development folder and click on the Examples folder, and then click the Ebike folder. Finally, use a text editor to open the Ebike Test Sequence Map file.

TestLimits

The second important item is the TestLimits file. The TestLimits file, holds a specific batch of "," delimited data values that are used to feed the Test Sequence Manager. During program execution, the Test Sequence Manager reads the TestLimits file line-by- line, and updates the GUI Display accordingly. In addition, the TestLimits file contains the actual Pass/Fails limits for each test.

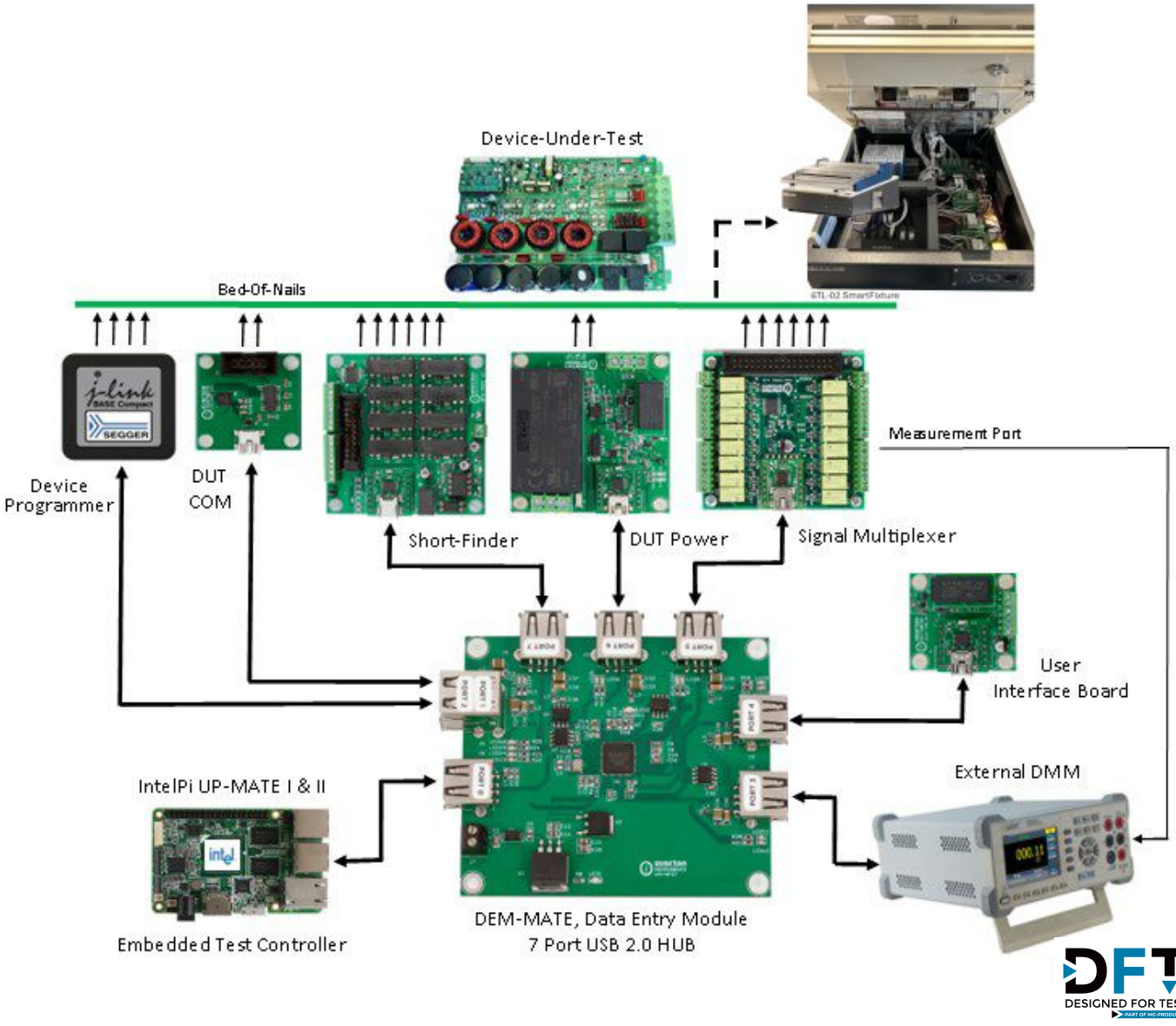
To view the TestLimits file for the Ebike example, go to the Application Development folder and click on the Examples folder, and then click Ebike folder. Finally, use a text editor to open the Ebike TestLimits file.

4. How to build a Test Application - cont.,

STEP 2, Review the Ebike example

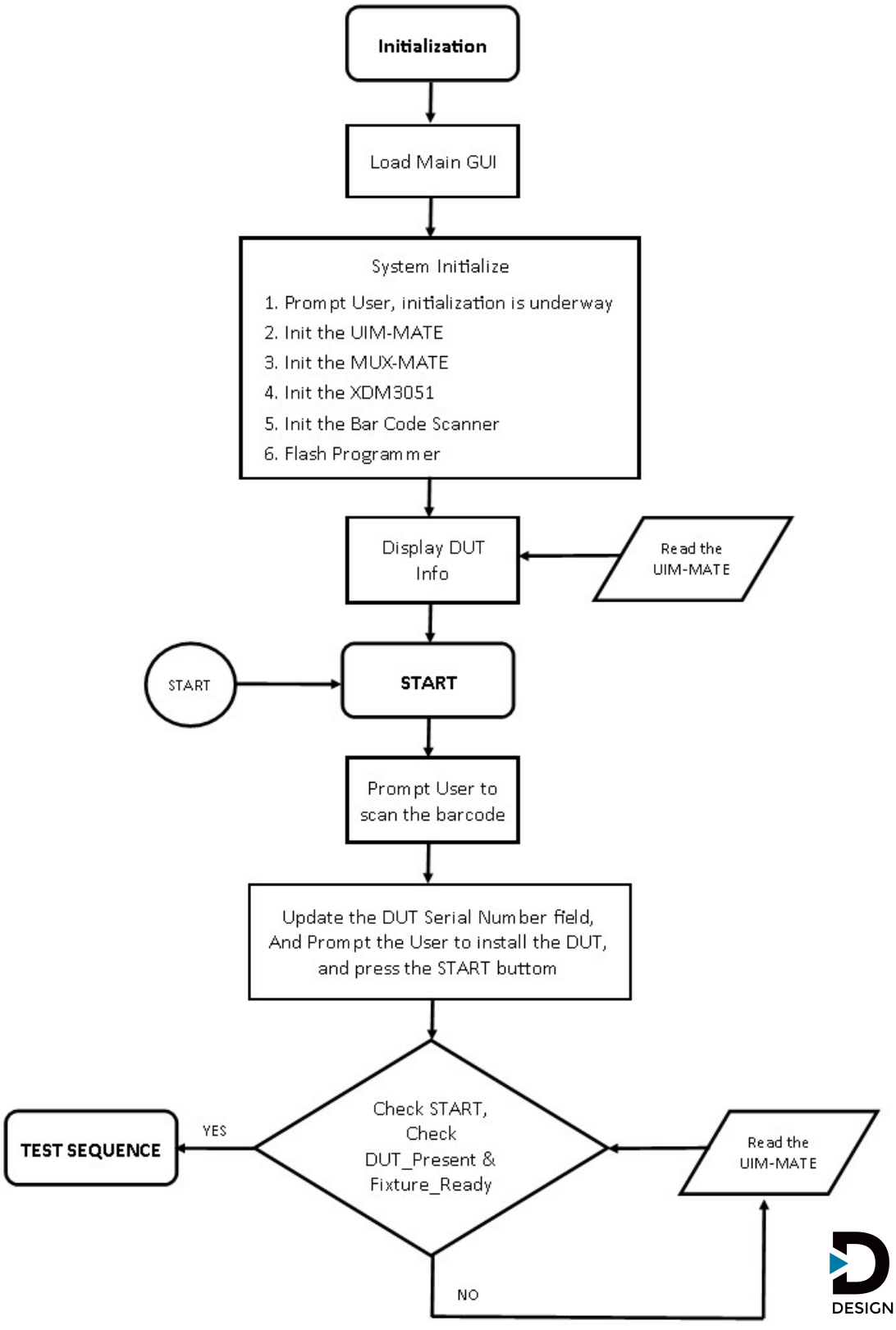
The Ebike Functional Test System shown below is driven by the intelPi (embedded controller), which also runs the Test Sequence Manager. The goal of the test is to perform a quick Go/NoGo test process, and end the test by uploading the Production Code. On pages 11 & 12, is a flowchart that highlights the overall test process.

To view the Test Sequence Manager (Python code), for the Ebike example, go to the Application Development folder and click on the Examples folder, and then click the Ebike folder. Finally, select the TSM_main_Ebike, and Visual Studio should launch and load the Ebike/ Python program.



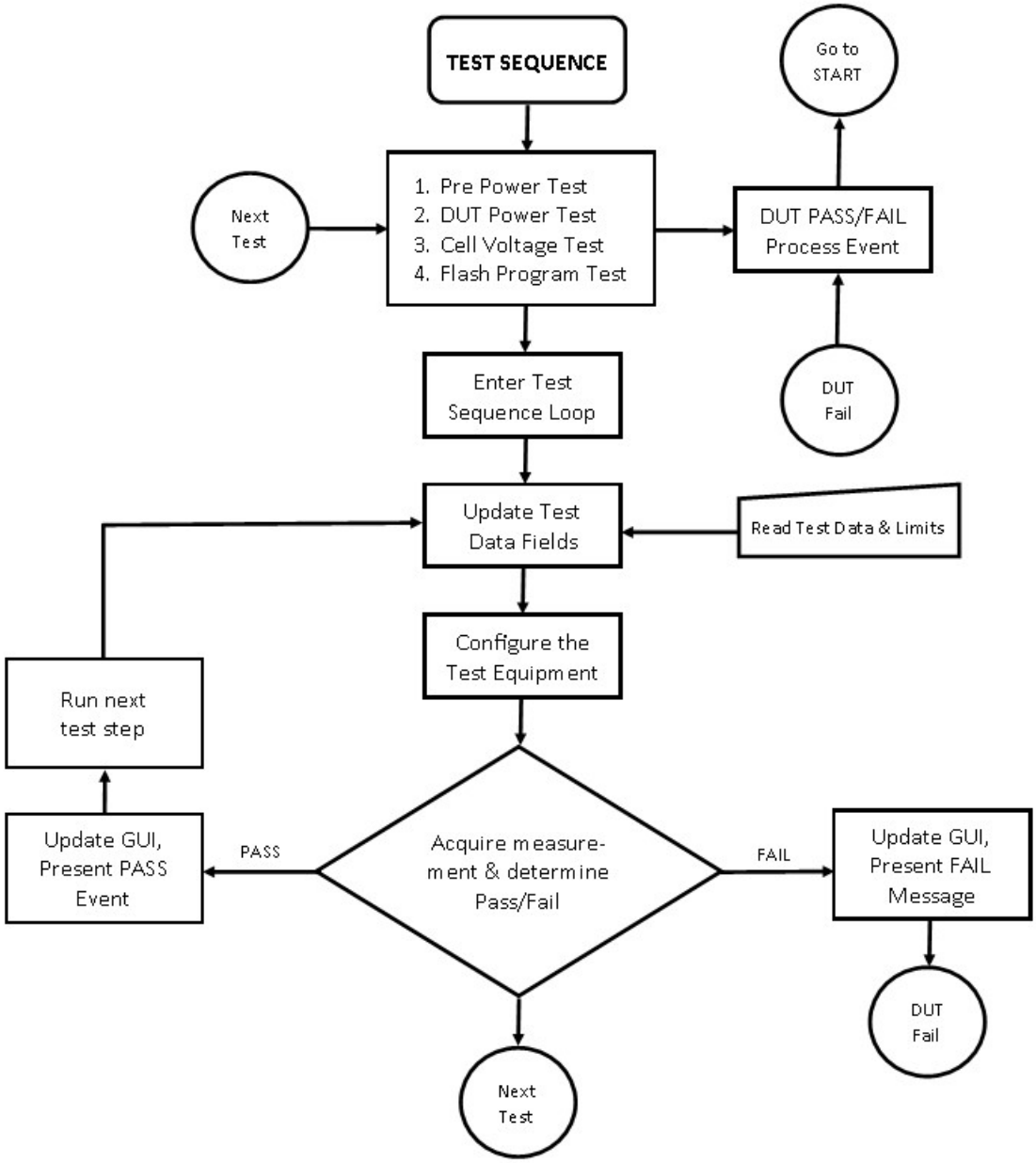
4. How to build a Test Application - cont.,

STEP 3, Analyze the Ebike test sequence flowchart



4. How to build a Test Application - cont.,

STEP 3, Analyze the Ebike test sequence flowchart



4. How to build a Test Application - cont.,

STEP 4, Build-It-Yourself

To jump-start and accelerate your effort to build a Test Application, we have provided a TSM template file (called the TSM_main_template), which provides a basic framework for building a custom Functional Test process.

It is strongly recommended that you prepare a Test Sequence Map for your DUT. The Test Sequence Map, can be built from an existing Manufacturing Test Procedure. You can also reference the Ebike example for additional direction. Remember, the goal is to identify each-and-every test step in the test process, and determine what equipment is needed and how it should be configured.

From the Test Sequence Map, you can generate the TestLimits.txt file, which consolidates the names and titles for the each test and sub-test, and includes critical pass/fail limit values. The TSM reads the TestLimits file, and processes the data accordingly.

When the Test Application is launched, the first thing it does is initialize the OI test instruments and external test equipment. So, in your program you must include the equipment you require and add the the appropriate instrument drivers. Follow the Test Sequence Map, it will include that information in the "Initiate" section.

Next, the program should identify the DUT by querying the UIM-MATE module to receive the DUT name and part number. Then, (assuming your application requires it), the program will prompt the Operator to scan the DUT serial number. Next, the program should prompt the Operator to install the DUT in the test fixture, close the over clamp, and press the START button. To verify the test fixture is ready, the program will then query the UIM-MATE to receive DUT_present & Fixture_ready status. If the status is OK, then the program can proceed to run the test sequence.

When creating a test sequence, one must understand that at the core of each test operation is a basic 4-step "loop" construct. The 4 steps include, 1) update the GUI, load the Test Sequence Monitor, 2) configure the OI instrument modules or external test equipment to satisfy the current test step, 3) trigger the test instrument to acquire a measurement, and 4) analyze the measurement to determine pass/fail and log the test results. Keep in mind, this same 4-step loop, is repeated for each sub-test (that is defined in the Test Sequence Map and the TestLimits file). On page 14, we provided a code snippet that illustrates a typical test operation.

To support the overall test flow, the Test Sequence Manager also includes a number of important housekeeping utilities as well. Such as the mechanism to read & process the TestLimits file, the mechanism to determine Pass/Fail and data logging, the mechanism to support the user panel and the Operator prompts, and the mechanism to support the START/STOP control buttons.

DFT DESIGNED FOR TEST PART OF MG-PRODUCTS

4. How to build a Test Application - cont.,

STEP 4, Build-It-Yourself (sample code)

```
def pre_power_test(self, subTestNum, retryCount):
    # initialize
    if self.owon.get_function() != 'RES':
        self.owon.change_function("RES")

    channel = subTestNum[3].strip()
    lowVal = self.parse_string_to_int(subTestNum[4])
    hiVal = self.parse_string_to_int(subTestNum[6])
    lowValstr = subTestNum[4].strip()
    hiValstr = subTestNum[6].strip()
    while retryCount >= 0:                          # loop retry count or test pass

        channel = "0" + channel if len(channel) == 1 else channel   # add 0 before single digits
        self.mux_mate.select_relay(channel, "1")
        print(self.mux_mate.get_relay_status())
        exactVal = float(self.owon.get_measurement())
        result = ""
        retryCount = retryCount - 1
        if exactVal < lowVal or exactVal > hiVal:
            if retryCount == -1:
                result = "F"
                self.fill_monitor(subTestNum, result)
                self.save_test_results(lowValstr, hiValstr, self.float_to_sn(exactVal), result)
                self.fail_dut()
                break
            else:
                continue
        else:
            result = "P"
            self.fill_monitor(subTestNum, result)
            self.save_test_results(lowValstr, hiValstr, self.float_to_sn(exactVal), result)
            self.pass_dut()
            break
```