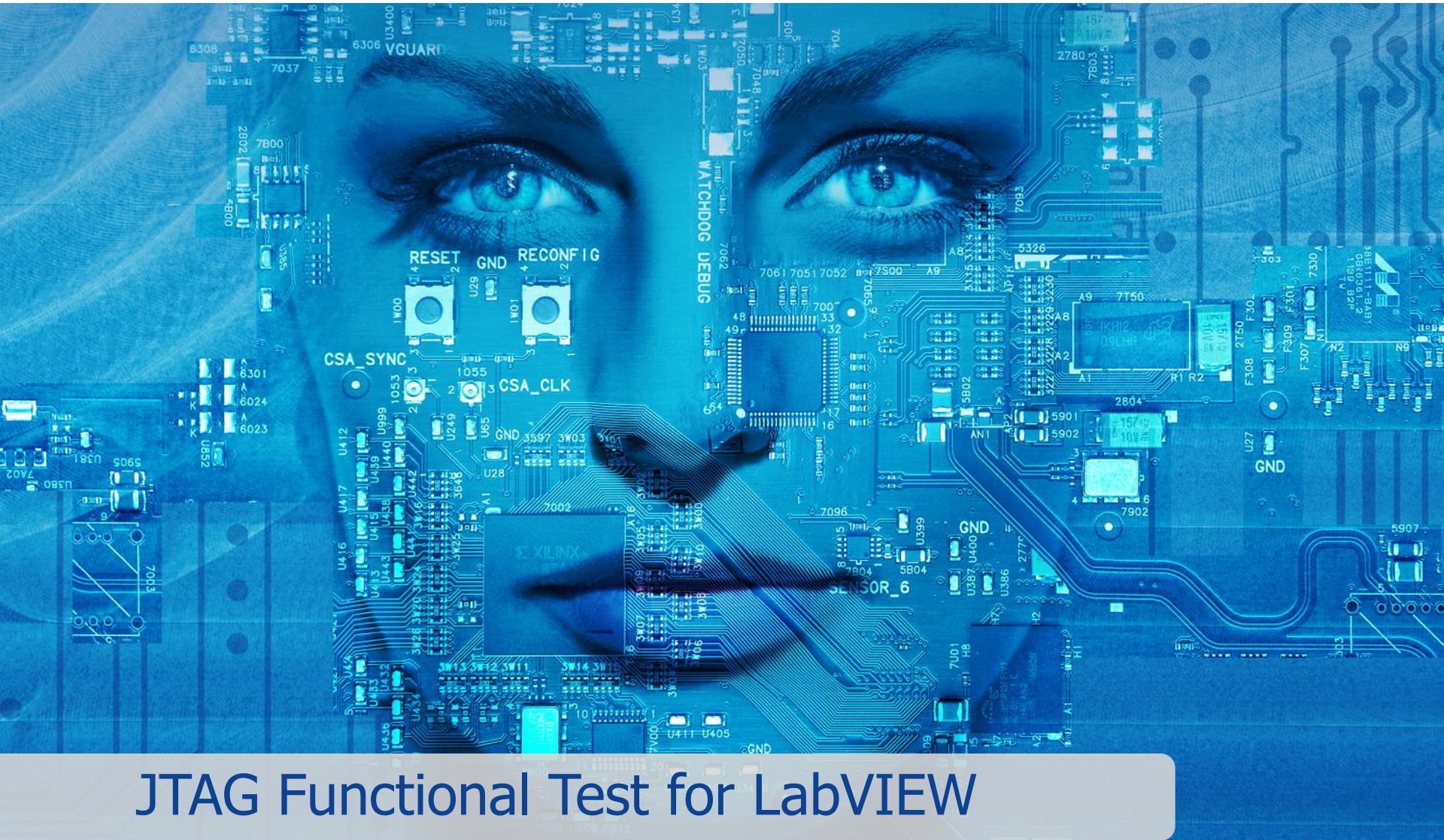


JTAG Functional Test for LabVIEW

Enhance your current tests with JFT LabVIEW

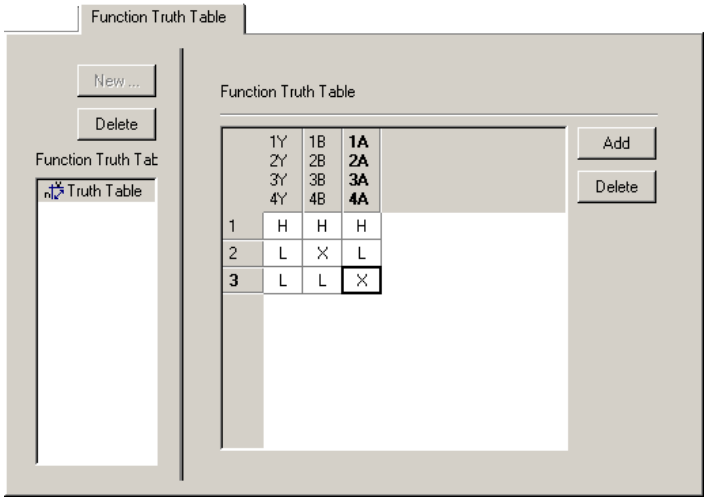
- Add complex cluster tests
- Drive peripherals and measure results
- Integrates into familiar environment
- Add Core commander and carry out system speed and Embedded tests

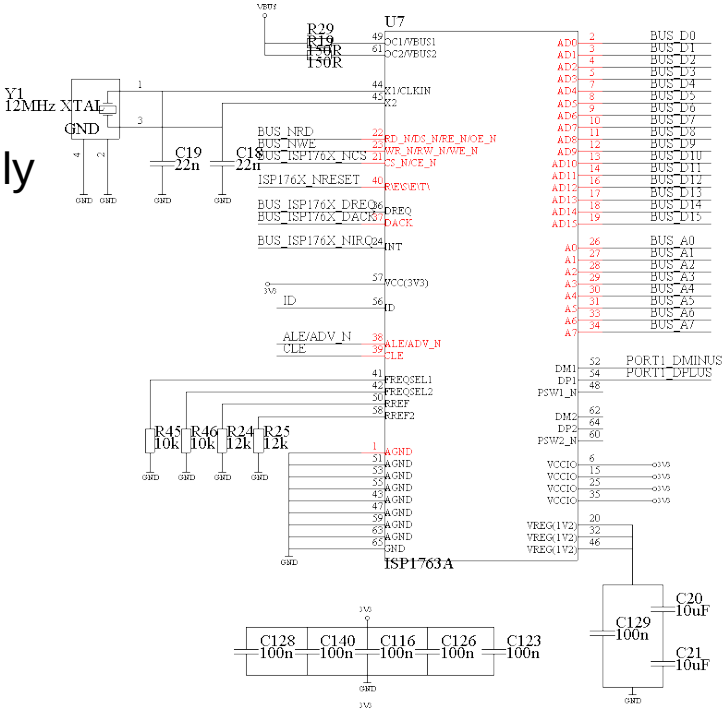
Low Cost Standalone development and execution

- JFT/xx enables your Boundary Scan chain.
- Drive and Sense pins on CPU/FPGA
- Low cost controller and software
- Many ready made functions available for both text based and graphical programming

Typical Boundary Scan Cluster Test

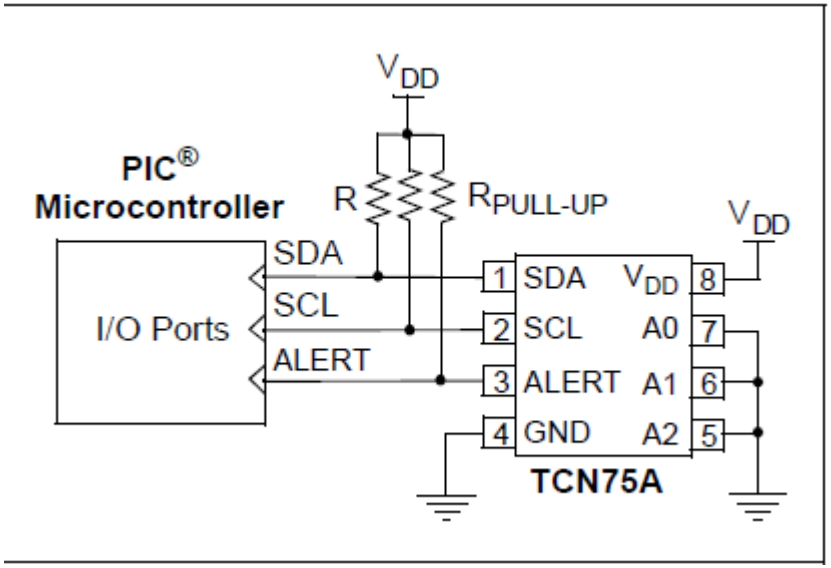
- Flat Vector based
- Drives or senses each pin individually
- Model based, Memories, Logic, Digital Peripheral devices.
- Automatic Generation possible





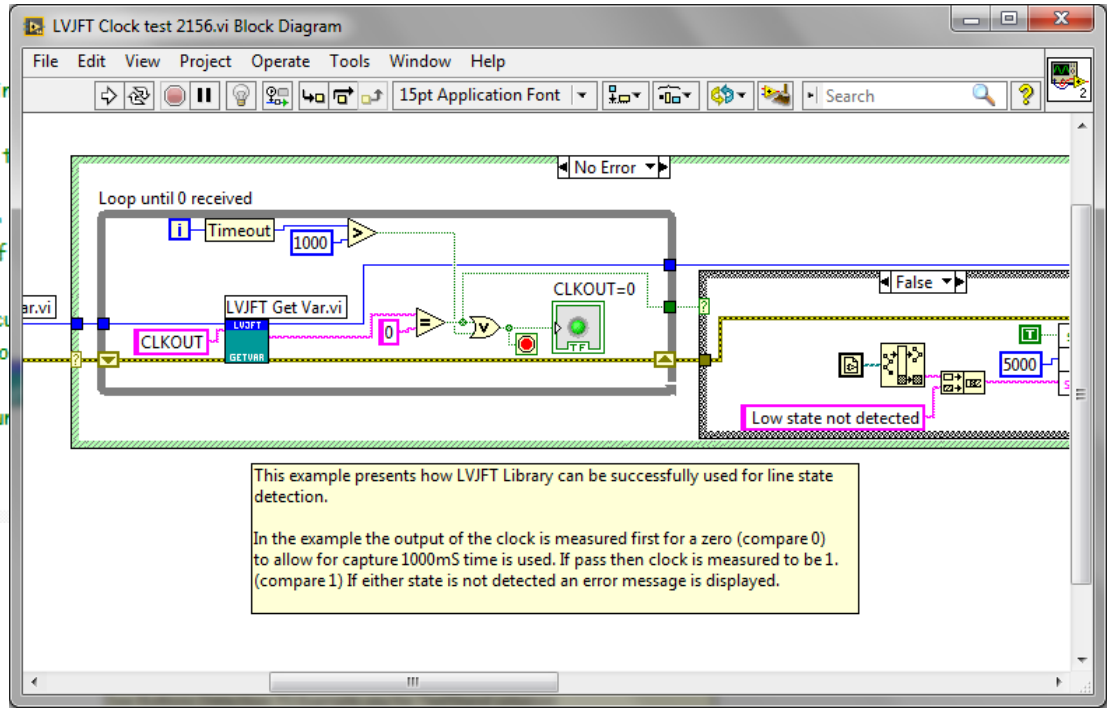
Complex Cluster Test

- Stacked Logic
- Flip Flops
- Analog Measurements
- Mixed Signal devices (DAC ADC)
- Drive sense of chip sensors and peripherals
- I2C and SPI devices



Higher Level of description for functions

```
22
23  DeclareVar("CLKOUT","U12.G2")          # CLKOUT = U12 pir
24
25  loopCounter=0                          #Set variable loopCounter t
26  failFlag=0                         #Set variable failFlag to 0
27  while (loopCounter<1000):              #While loopCounter
28      if (GetVar("CLKOUT"))==0:          #Sense CLKOUT. If
29          break                          #...exit the loop
30      if loopCounter==999:               #if the loop has occu
31          print("Error: Net not sensed low")   #...print erro
32          failFlag+=1                    #...and add 1 to failFlag
33      loopCounter+=1                     #increment loop cou
34  #End of while loop
35
```



- Drive/sense pins
- Drive/sense groups
- Assign Variables

These Variables can then be controlled, manipulated and displayed in either Text based or Graphical programming language

Available JFT integration languages

JTAG Functional Test

Adding JTAG to your tester

python powered

Visual Basic

Microsoft .net

NATIONAL INSTRUMENTS LabVIEW

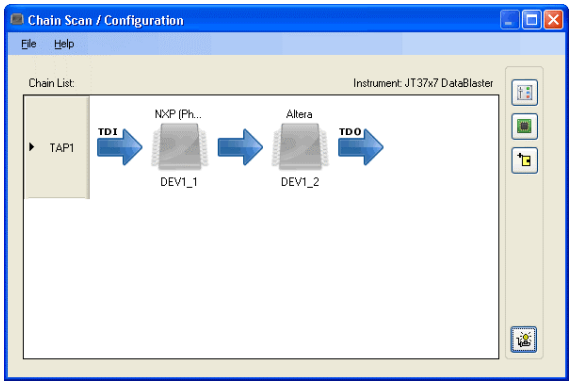
CVI

Requirements to develop LabVIEW code

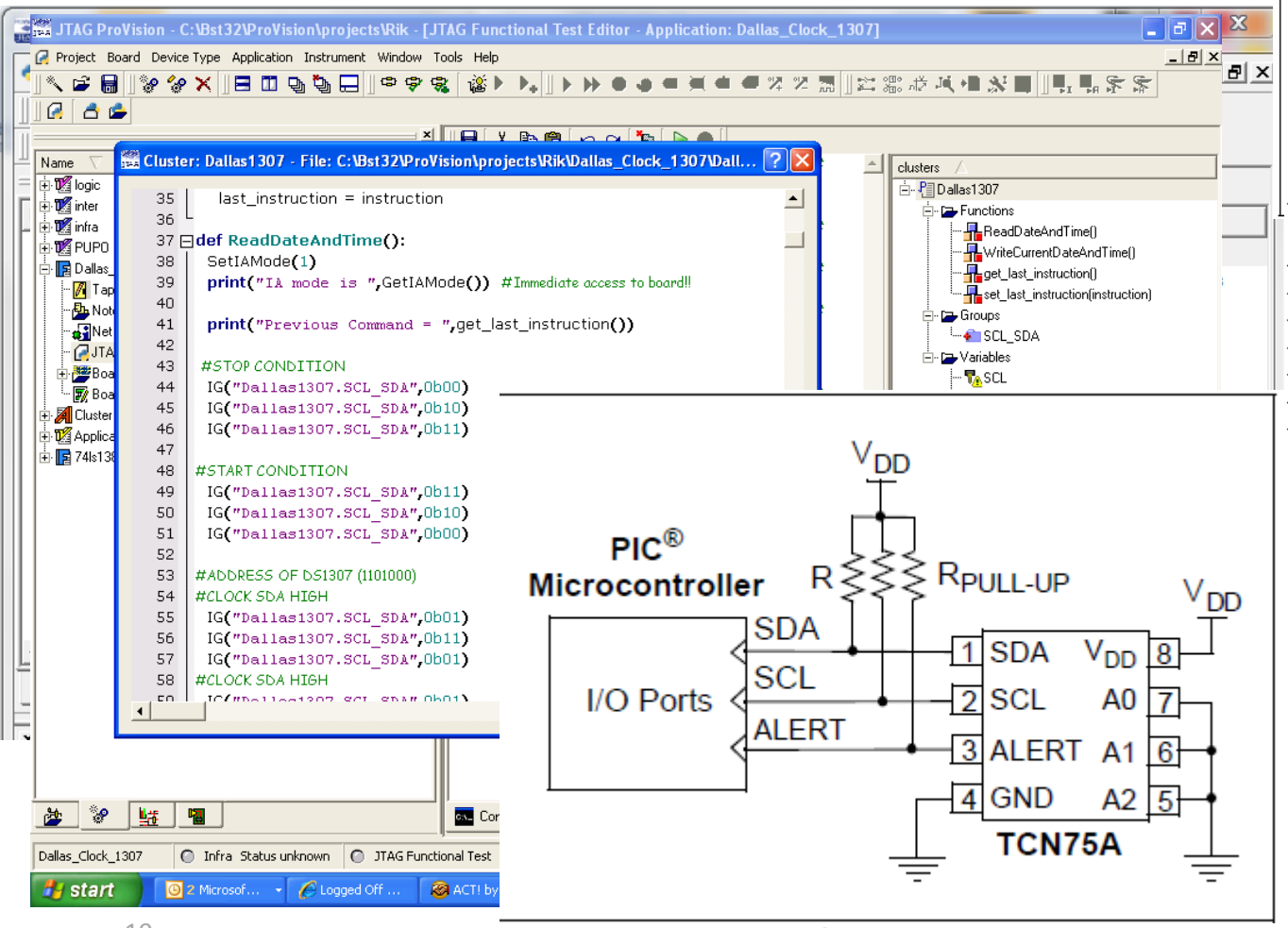
- Licensing
  - Included in both ProVision and PIP
- Software installation
  - Provision must have PIP/LabVIEW installed. (license not required)
  - PIP for execution (PIP License is required)
  - Standalone installation (JFT/LV license is required)

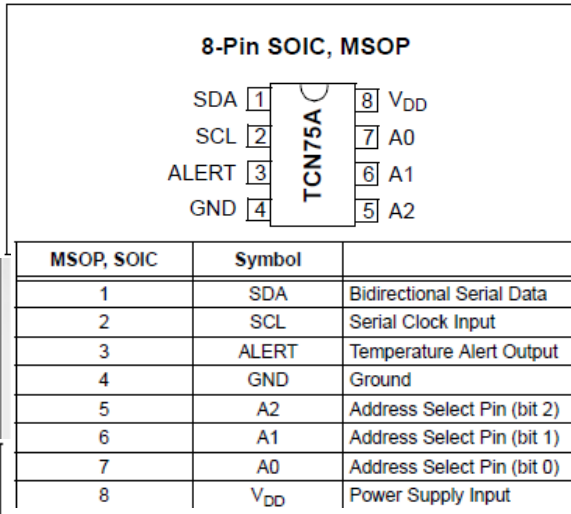
Requirements to develop LabVIEW code 2

- Files
  - Needs .gen and .vdf files (rename to LV VI name)
- Provision any application
- JTAG Live any application (including free buzz download)
- Standalone special chain builder tool



Typical cluster example for language approach





MSOP, SOIC	Symbol	
1	SDA	Bidirectional Serial Data
2	SCL	Serial Clock Input
3	ALERT	Temperature Alert Output
4	GND	Ground
5	A2	Address Select Pin (bit 2)
6	A1	Address Select Pin (bit 1)
7	A0	Address Select Pin (bit 0)
8	V_{DD}	Power Supply Input

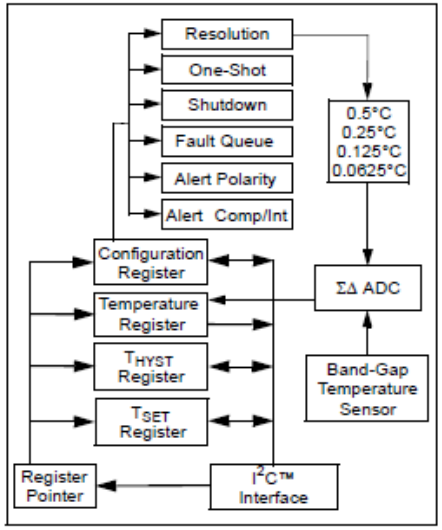
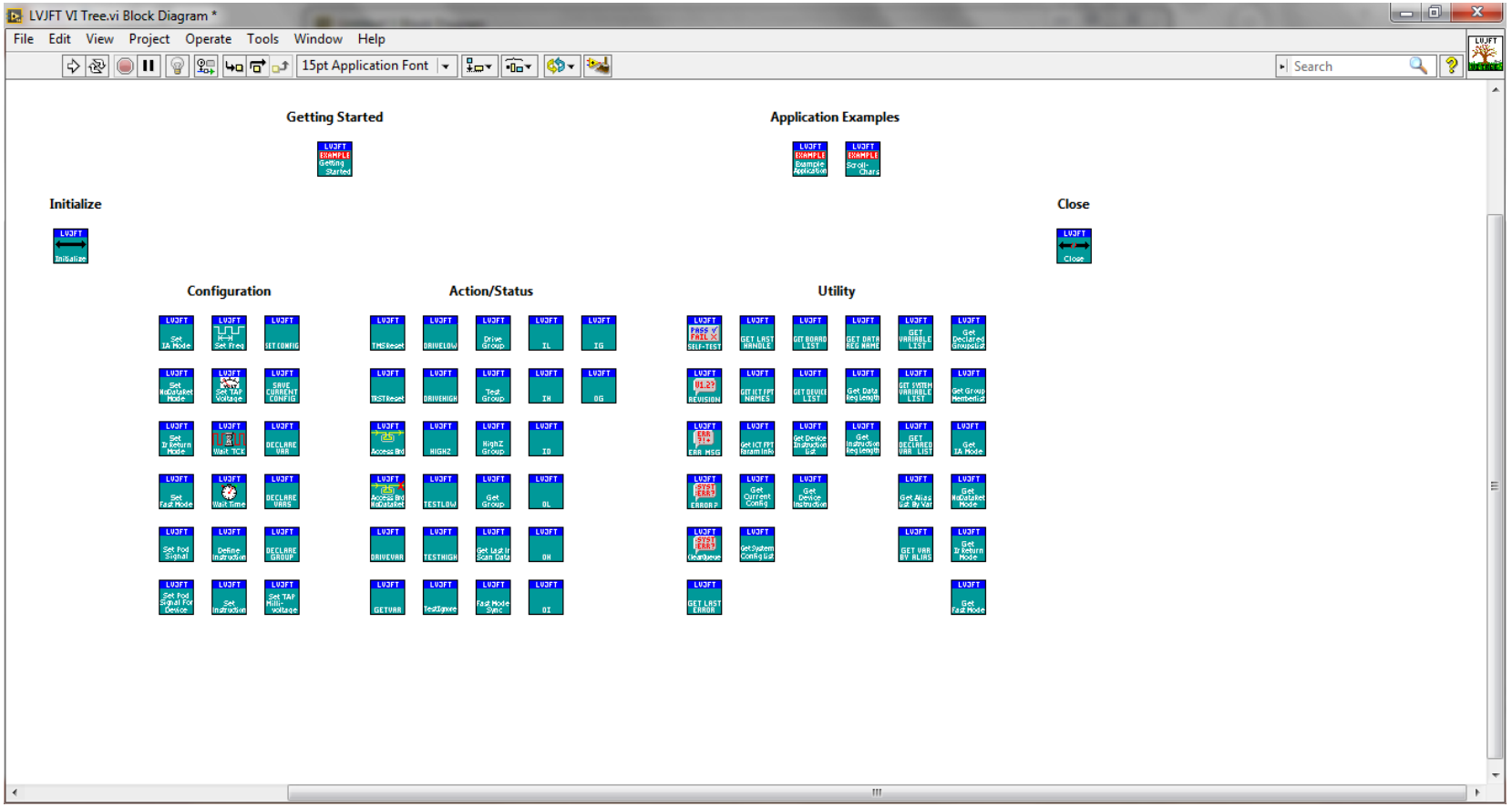


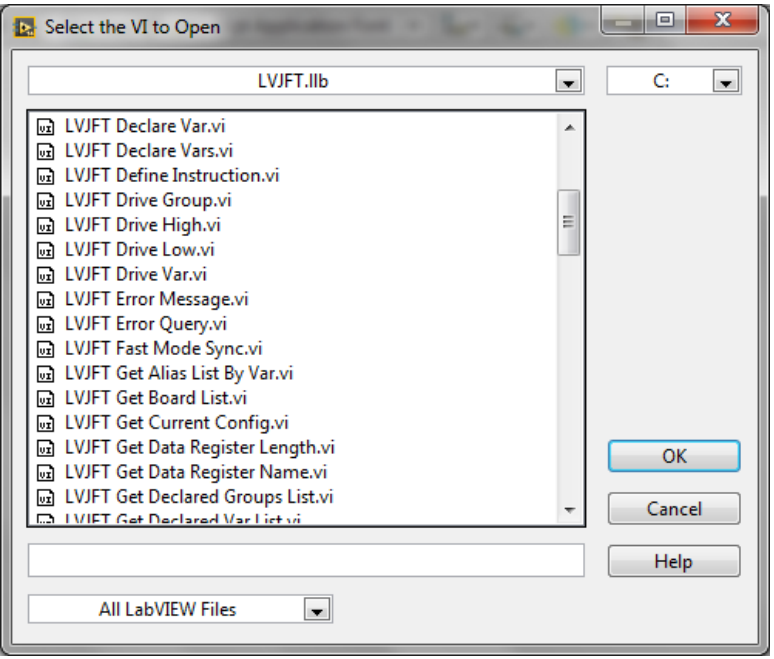
FIGURE 5-1: Functional Block Diagram.

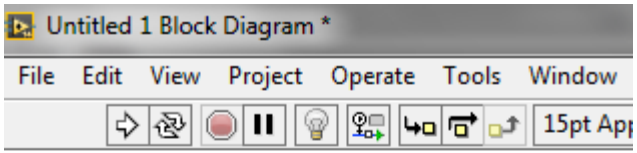
JTAG Functional Test with

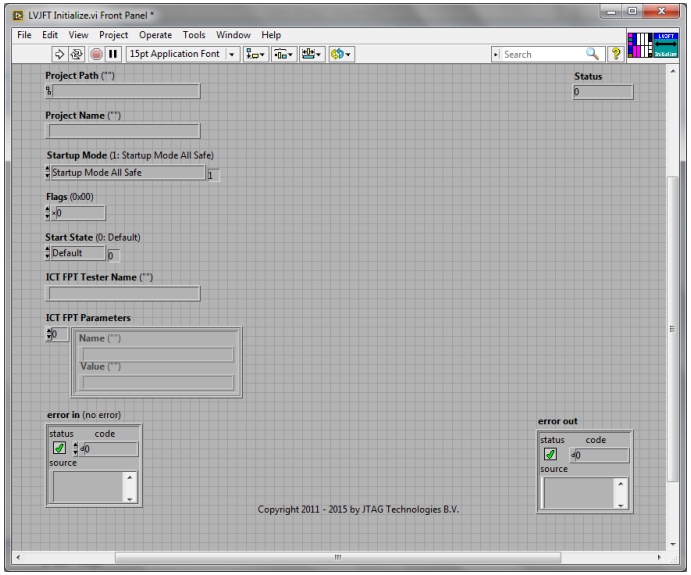
Overview of VI's (LVJFT VI Tree)

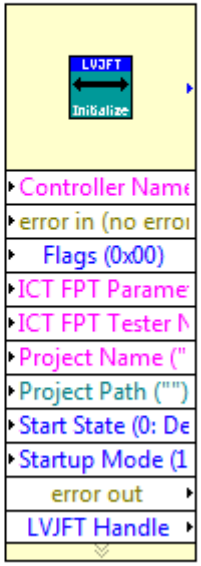


Example configuration Vi's Initialise

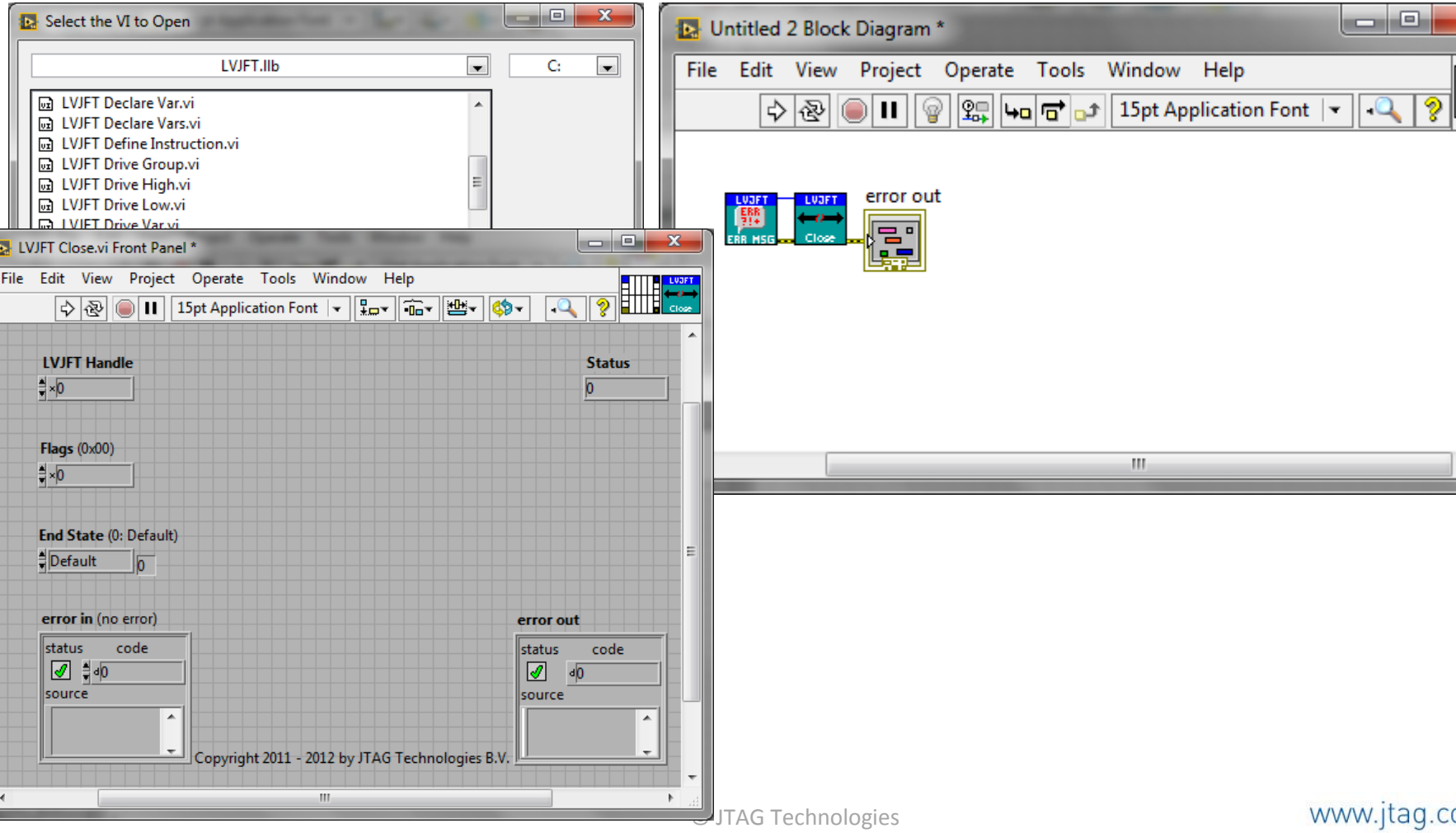




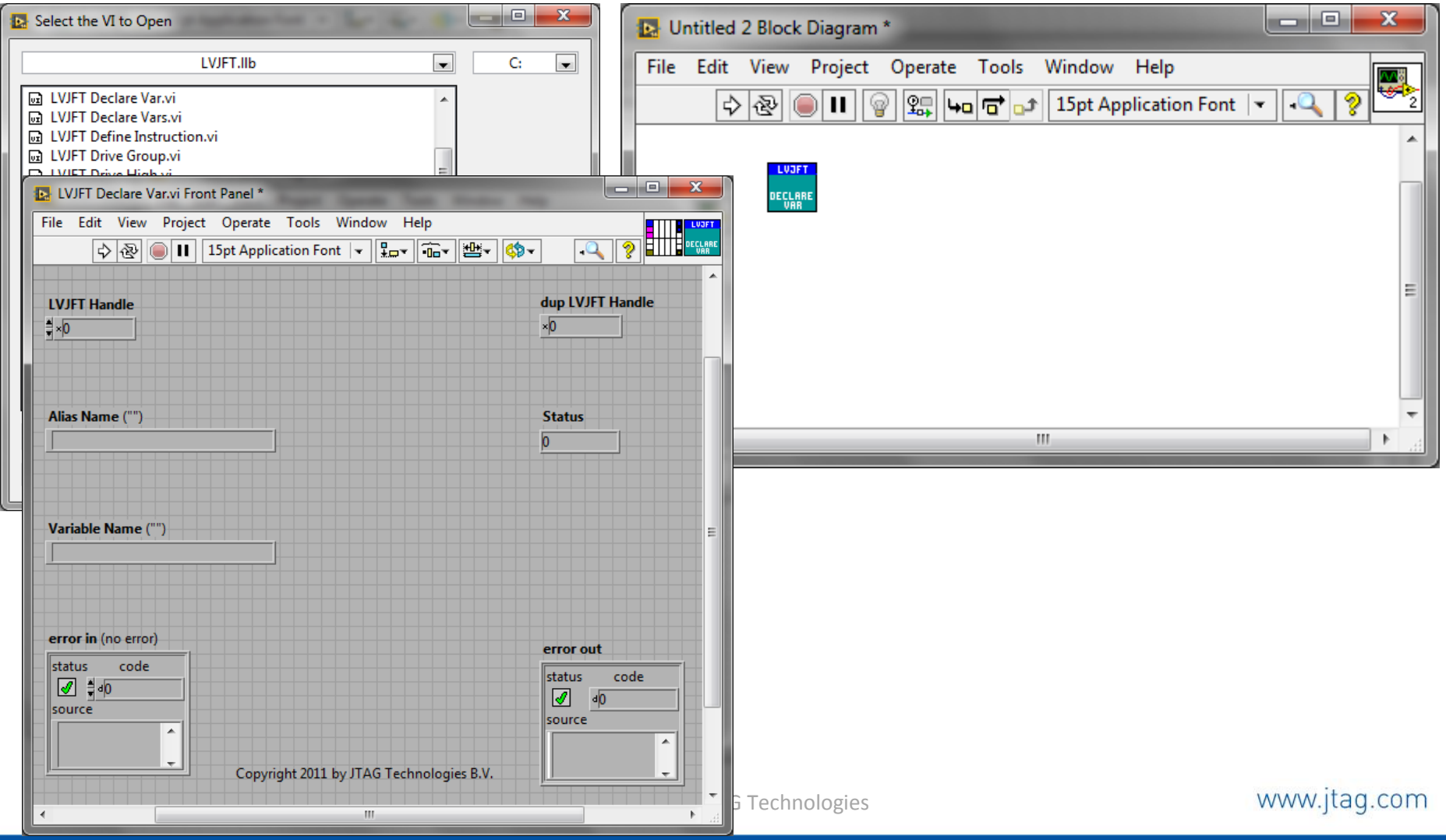




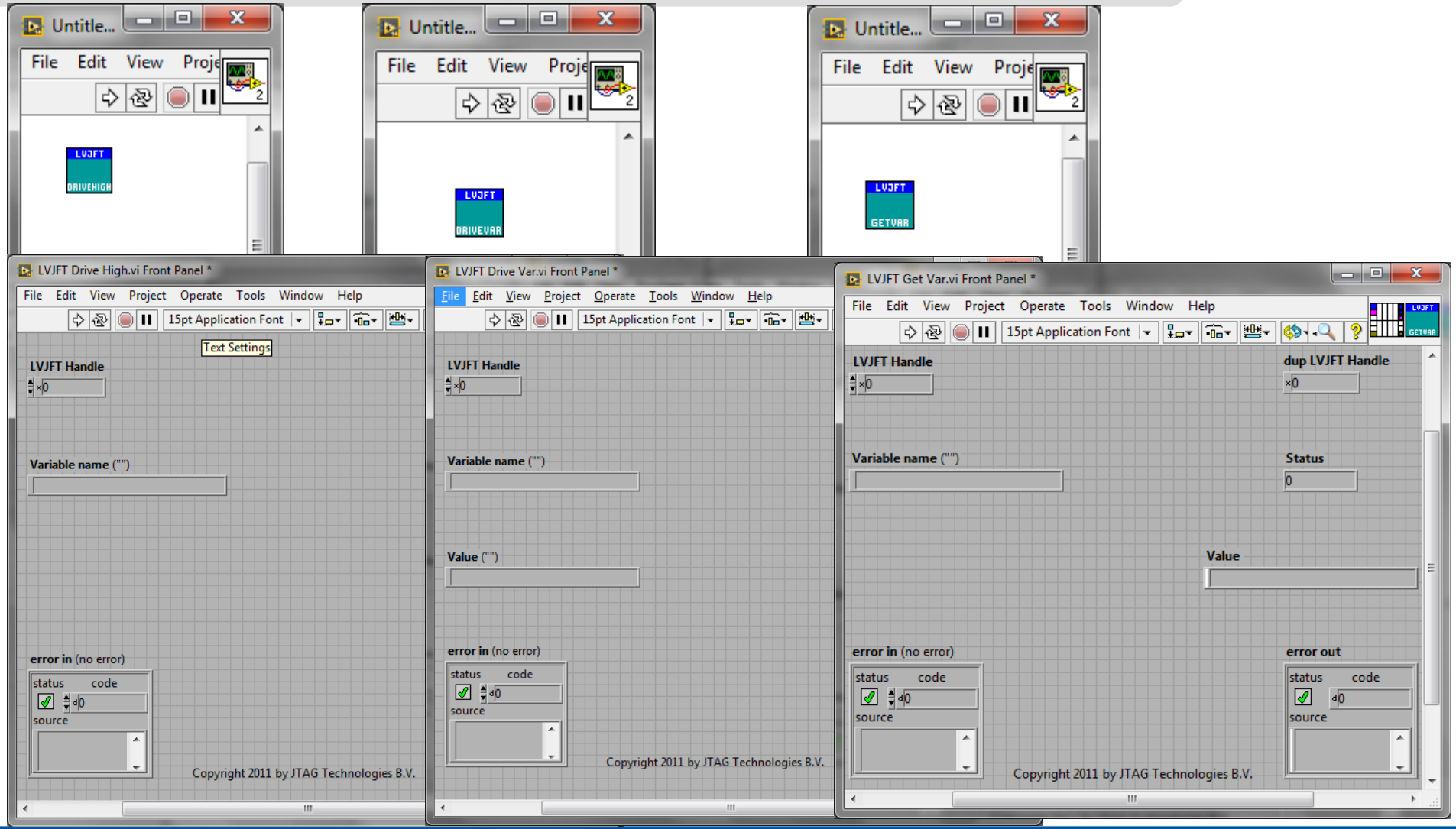
Close and error handling



Example action Vi's 1 Declare Variable

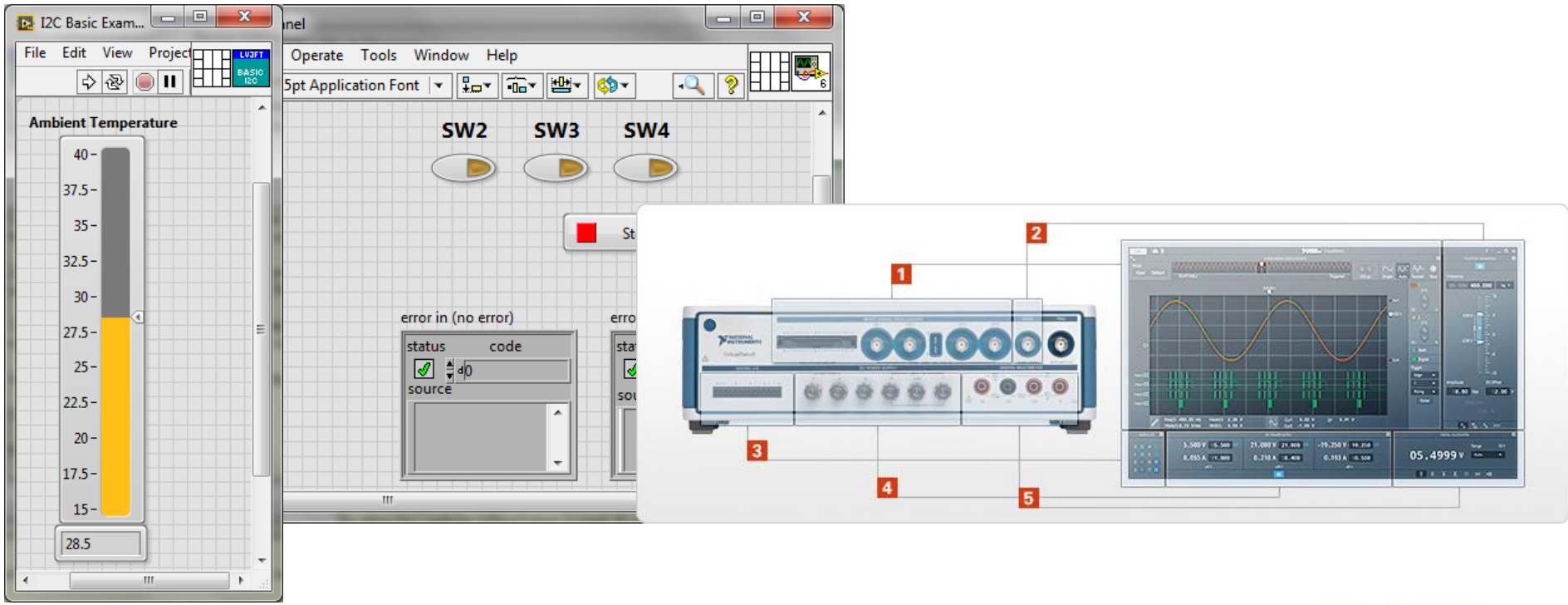


Action Vi's 2 Drive and sense

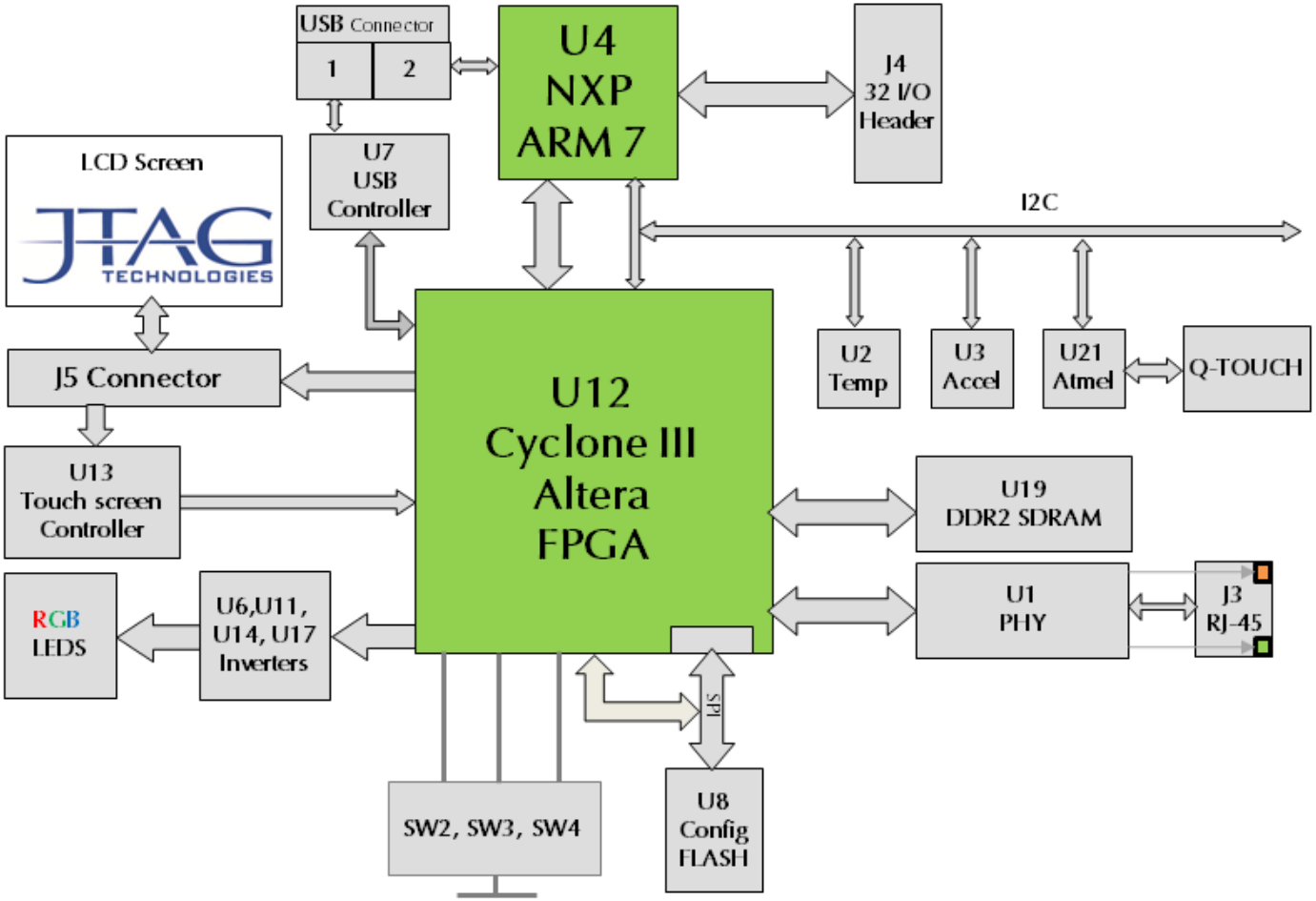


Combining LVJFT with measurements

- LVJFT gives capability to drive or sense the Boundary Scan cells.
- The result of the read or driven cells can then be used to manipulate controls or take measurements using the standard LabVIEW tools and Hardware.



JT2156 Functional Block diagram

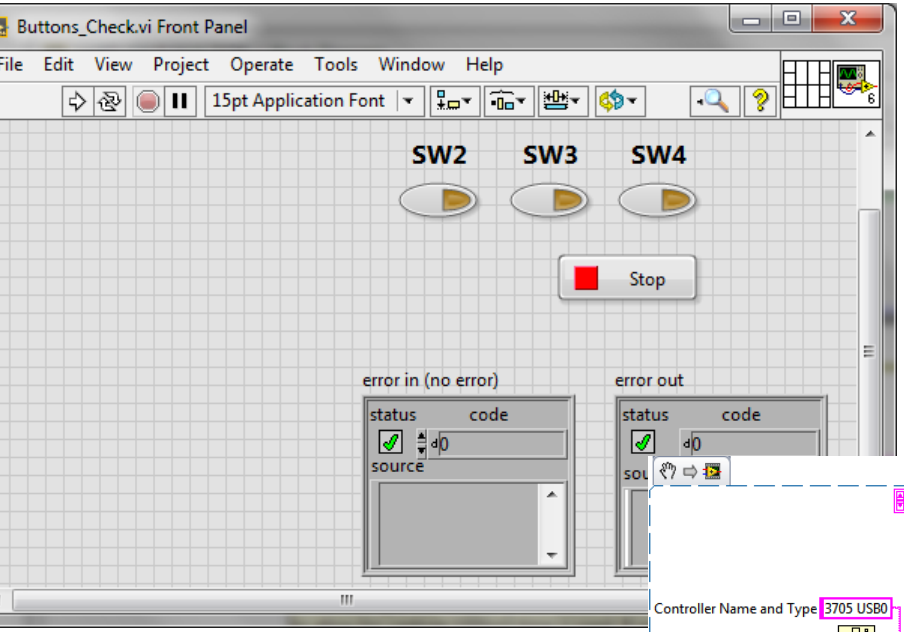


Read switch and display result to on-board LED

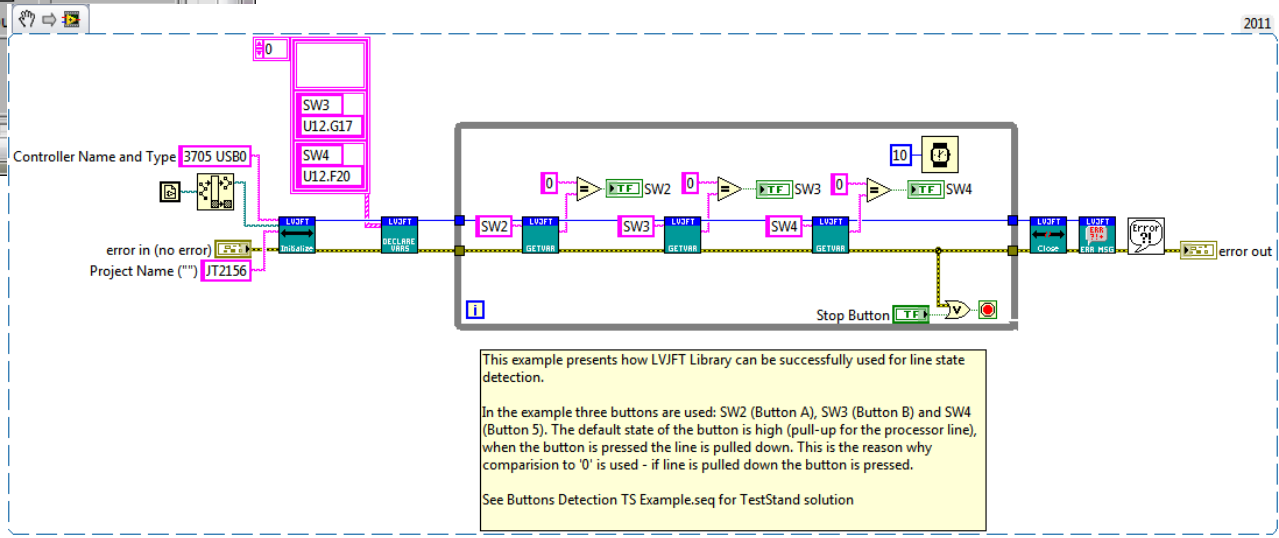
- Using text based language
- Read the switch value
- Display result on UUT LED

```
DeclareVar("SW2", "U12.H20") #SW2 is now a JFT variable and linked to Bscan pin H20 from U12
DeclareVar("D1-Red", "U12.F14") #D1-Red is now a JFT variable and linked to Bscan pin F14 from U12
DeclareVar("D1-Green","U12.E22") #D1-Green is now a JFT variable and linked to Bscan pin E22 from U12

print("Press Switch 2")
Wait_Time(2000000) # Wait 2 seconds to give the operator time to find Switch 2 on the JT2156
print("If Switch 2 is pressed RGB LED D1 will turn Red")
print("If Switch 2 is NOT pressed RGB LED D1 will be Green")
Wait_Time(3000000) # Wait 3 seconds before moving on

Switch2 = GetVar("SW2") # Switch2 is now a Python variable which can be used in the script

if (Switch2)==0:
    DriveVar("D1-Red",1) # Make RGB LED D1 Red
    print ("Switch2 is pressed")
    Wait_Time(2000000) # Wait 2 seconds
if (Switch2)==1:
    DriveVar("D1-Green",1) # Make RGB LED D1 Green
    print ("Switch2 is NOT pressed")
    Wait_Time(2000000) # Wait 2 seconds

print("value of Switch 2 was:", Switch2)
```

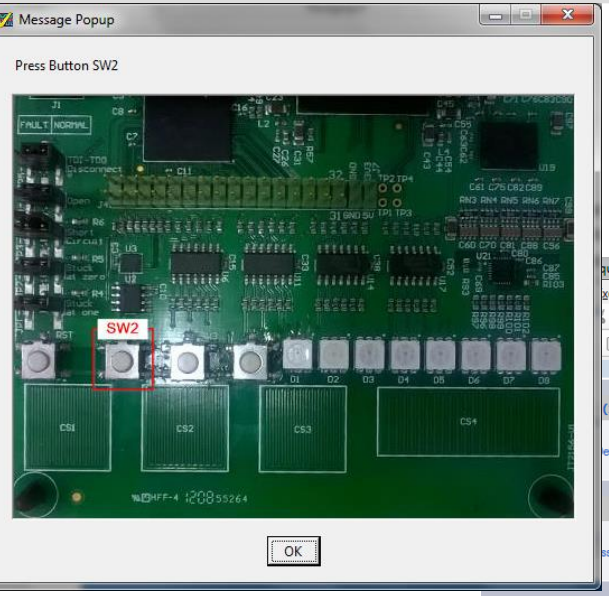
Using LabVIEW



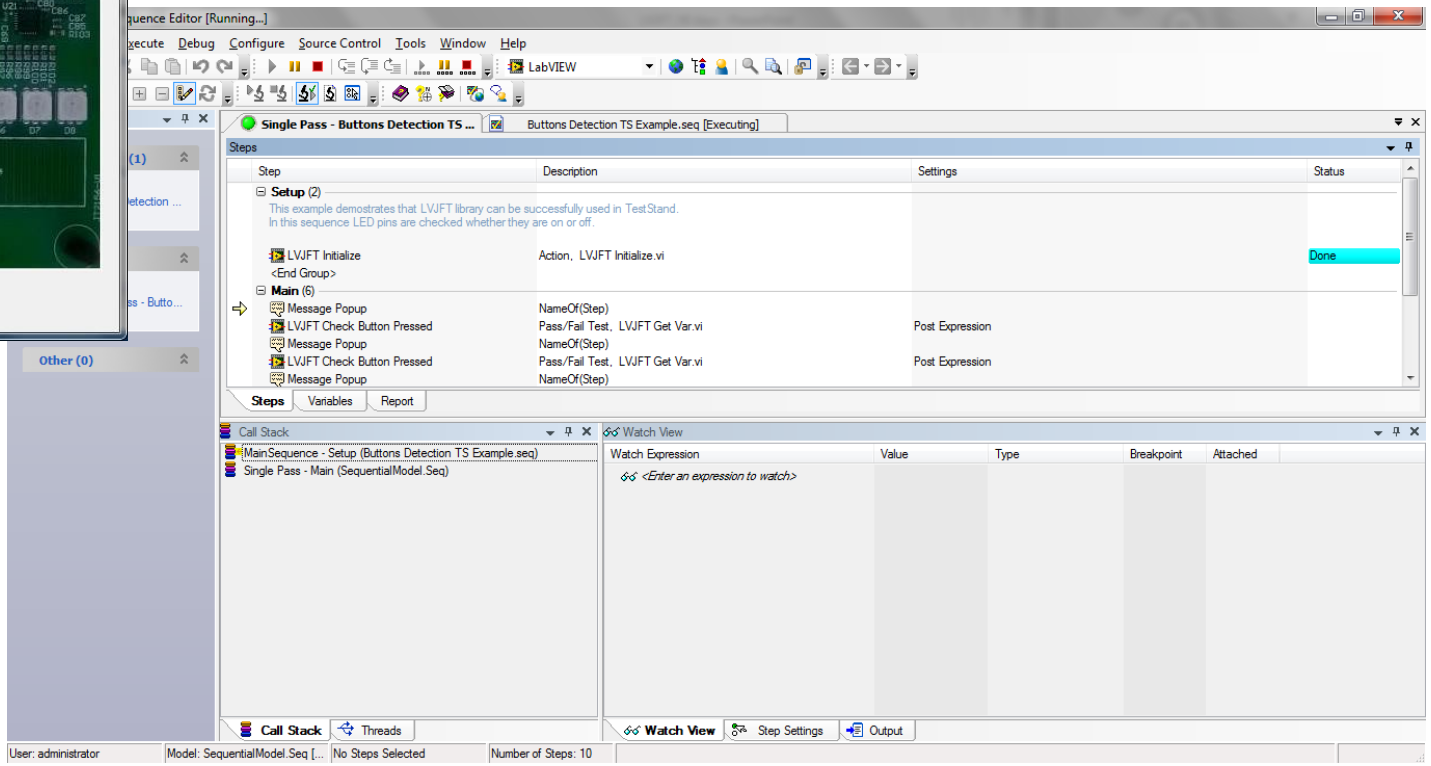
- Read Switches
- Display results on indicator
- Results can be manipulated



Build into a TestStand sequence

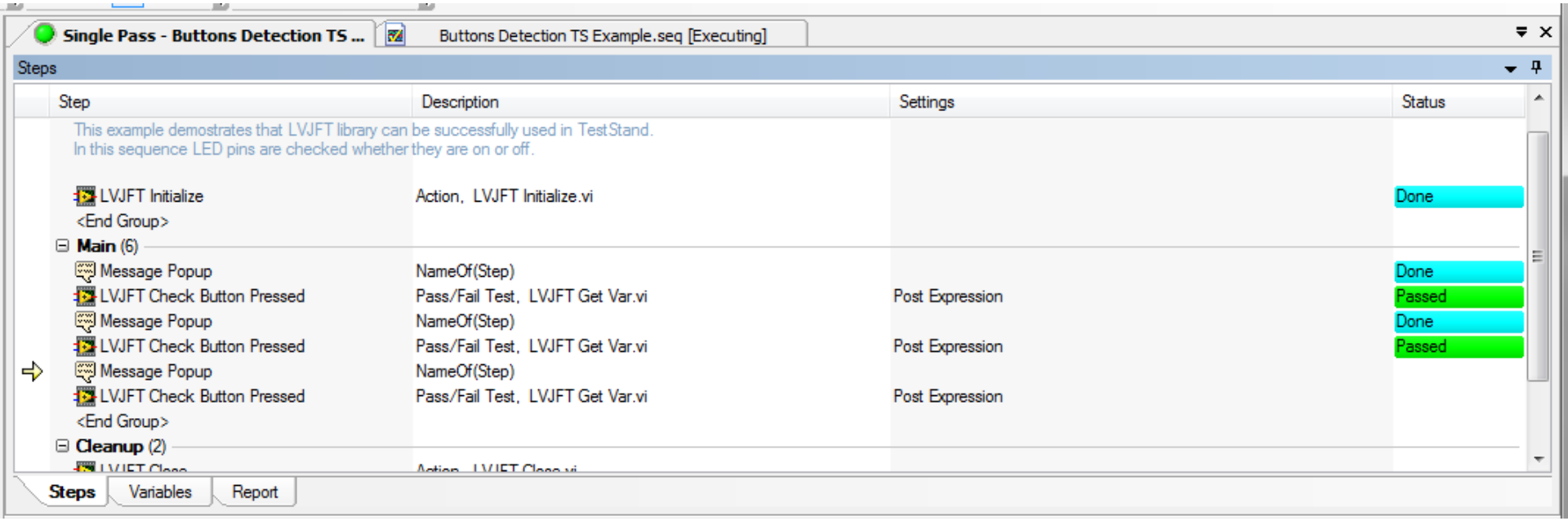


- Setup steps
- Display information to operator
- Execute Tests



Execute Sequence

- Complex sequences can be developed
- Combine Boundary Scan with standard Instrument measurements in sequence.



Setup and read an I2C device

- U2 is an I2C temperature sensor type TCN75A
- By manipulating the SCL and SDA boundary scan bits it is possible to set up and then to read the data register from the temperature sensor.
- The information received can then be calculated and output as a numeric (text based language) or as an indicator in LabVIEW.
- To read the register we need to toggle the clock (SCL) and during the positive to negative clock cycle either write or read information to the Data pin (SDA)
- Declare I2C pins (SCL, SDA)
- Write to device address and receive Acknowledge (ACK)
- Set address pointer and slave address.
- Read the upper and lower byte values and display.

Using text based language

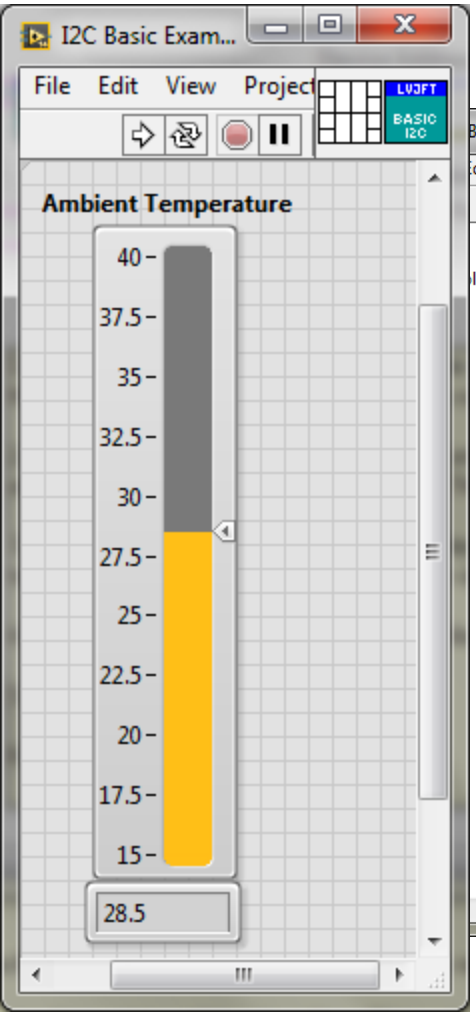
- Declare_I2C_Pins("U12.E21","U12.F22") #I2C.py function. Declares SCL and SDA nets
- Set_I2C_Address("1001000") #Set the I2C address. Datasheet for TCN75A shows address is 0x48
- WRITE("00000000") #Access Register 0x00 (Ambient temperature register)
- Ambient_Temp_Reg_Byte1 = READ()
- Ambient_Temp_Reg_Byte2 = READ()
- print("Ambient temp register upper half = ",Ambient_Temp_Reg_Byte1) # Print the contents of this register
- print("Ambient temp register lower half = ",Ambient_Temp_Reg_Byte2) # Print the contents of this register

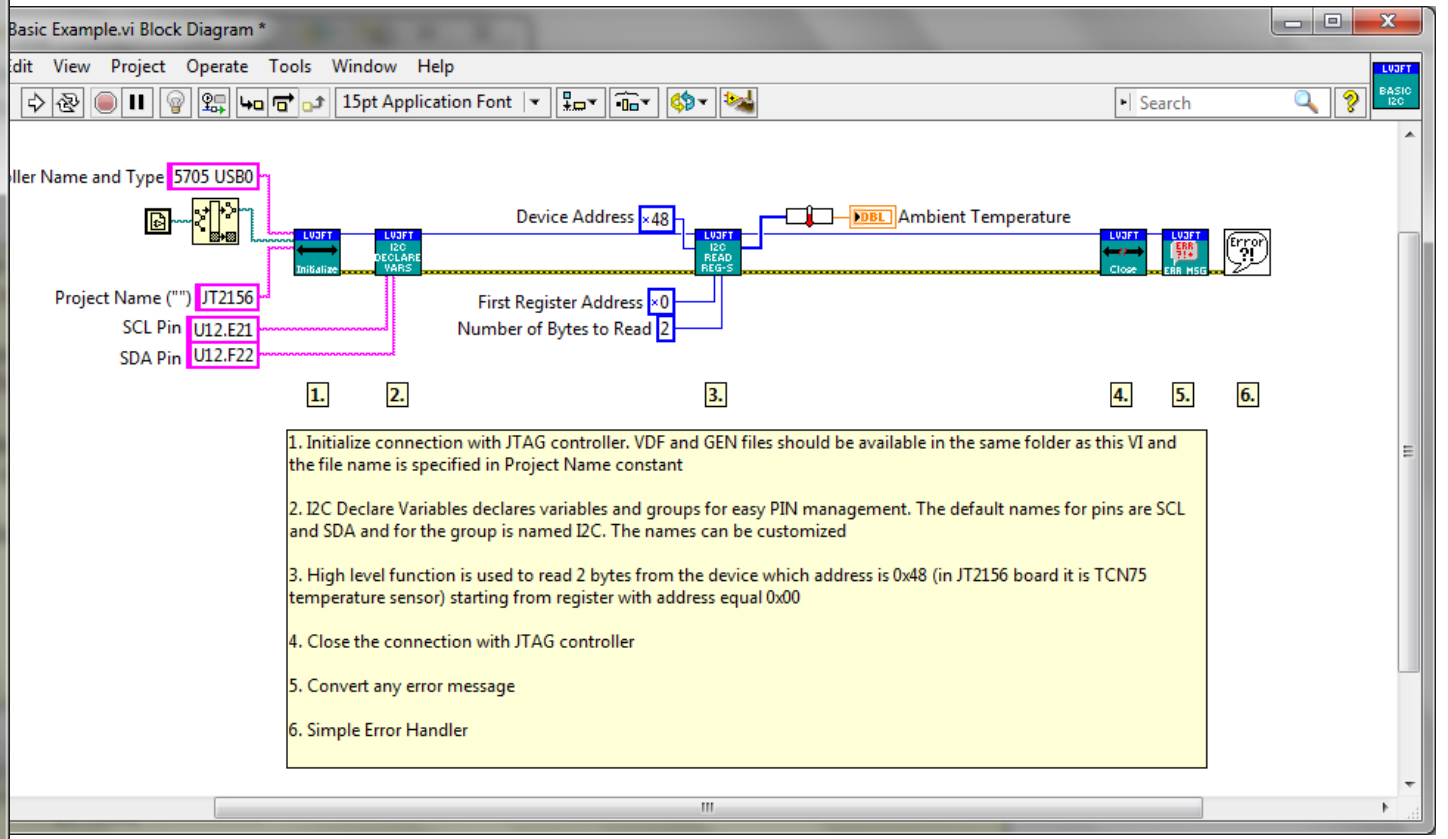
- Temp = (Ambient_Temp_Reg_Byte1 & 0x7F) + (0.5 * (Ambient_Temp_Reg_Byte2 >> 7))

- if(Ambient_Temp_Reg_Byte1 & 0x80): #If Ambient temp reg upper half bit 8 = 1 (sign bit)
- Temp = -1 * (128-Temp) #Temperature is negative and therefore two's compliment

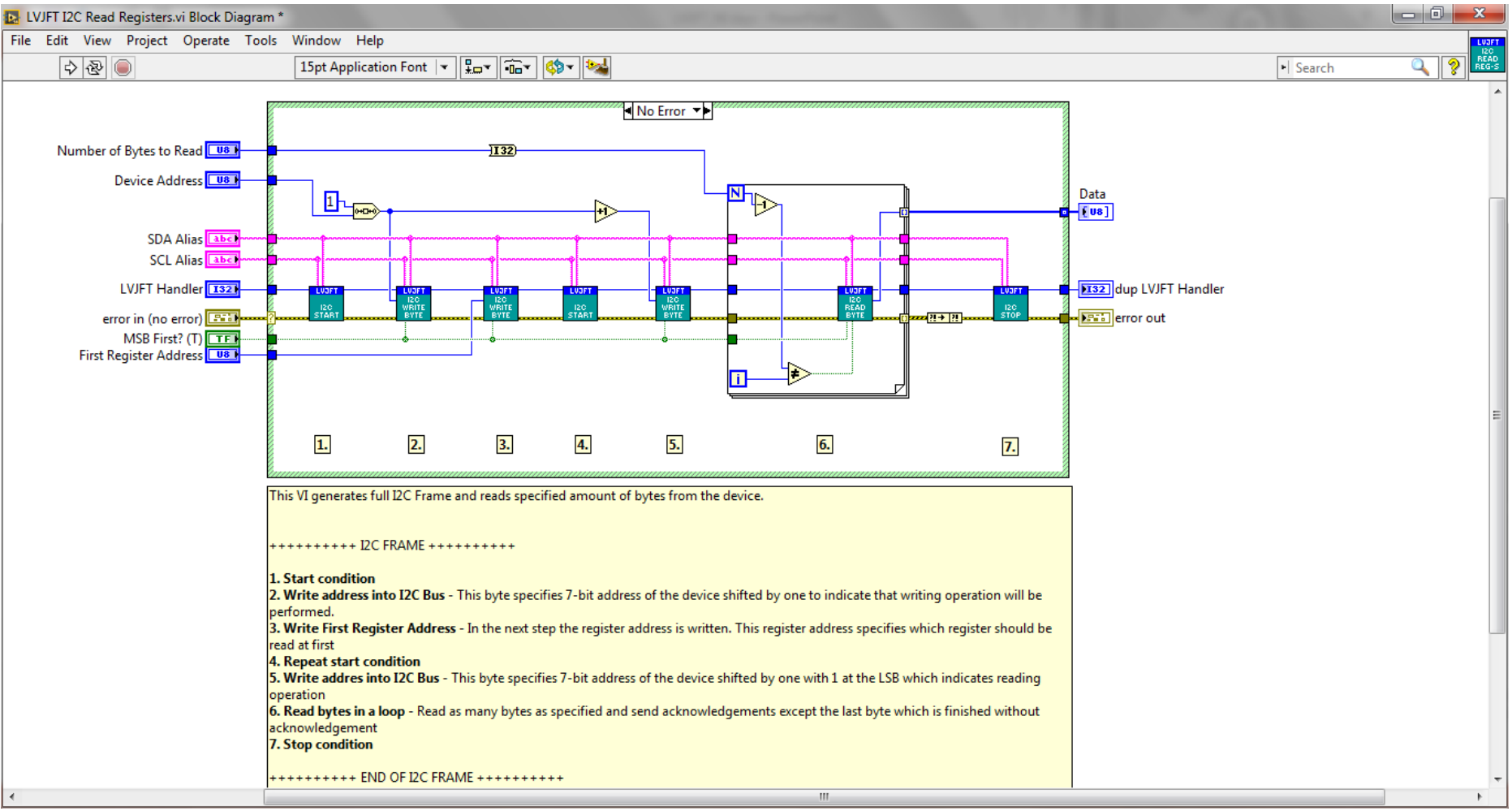
- print("Temperature is", Temp, "degrees")

Using LabVIEW with indicator



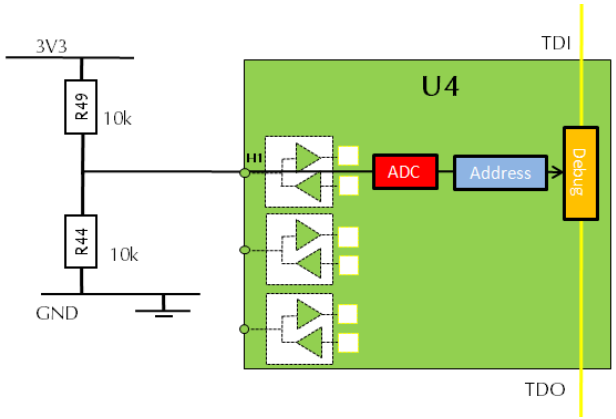


Generic VI for I2C devices



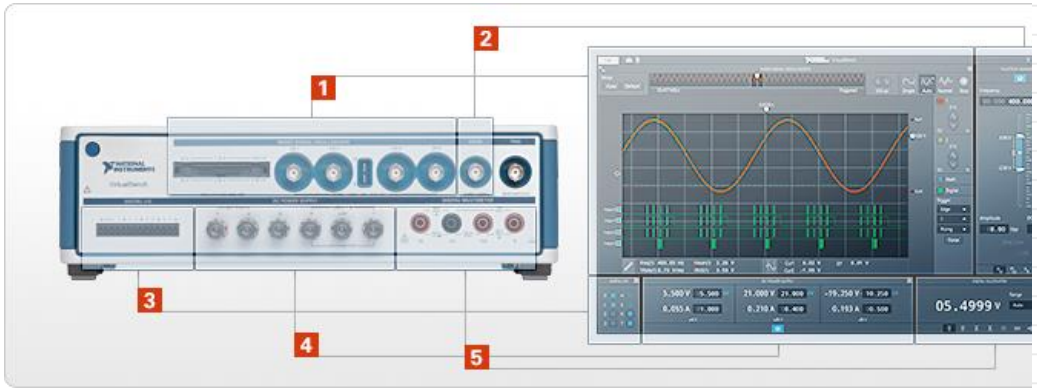
Use CPU to read ADC and display output

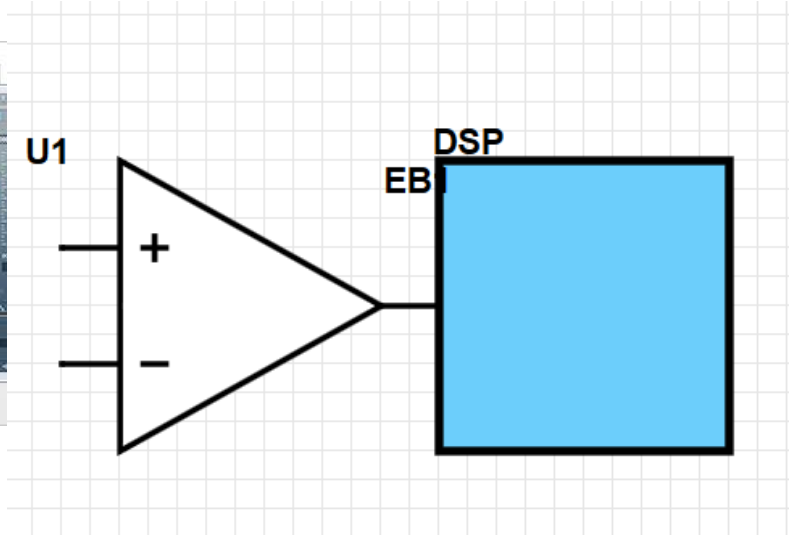
- The CPU of the board has a built in ADC 2 resistors of equal value are connected to the ADC pin between +3.3V and GND



DSP with internal ADC

- Supply Amplifier with fixed voltage
- Read internal DSP or Microcontroller ADC for dynamic (at speed) tests.

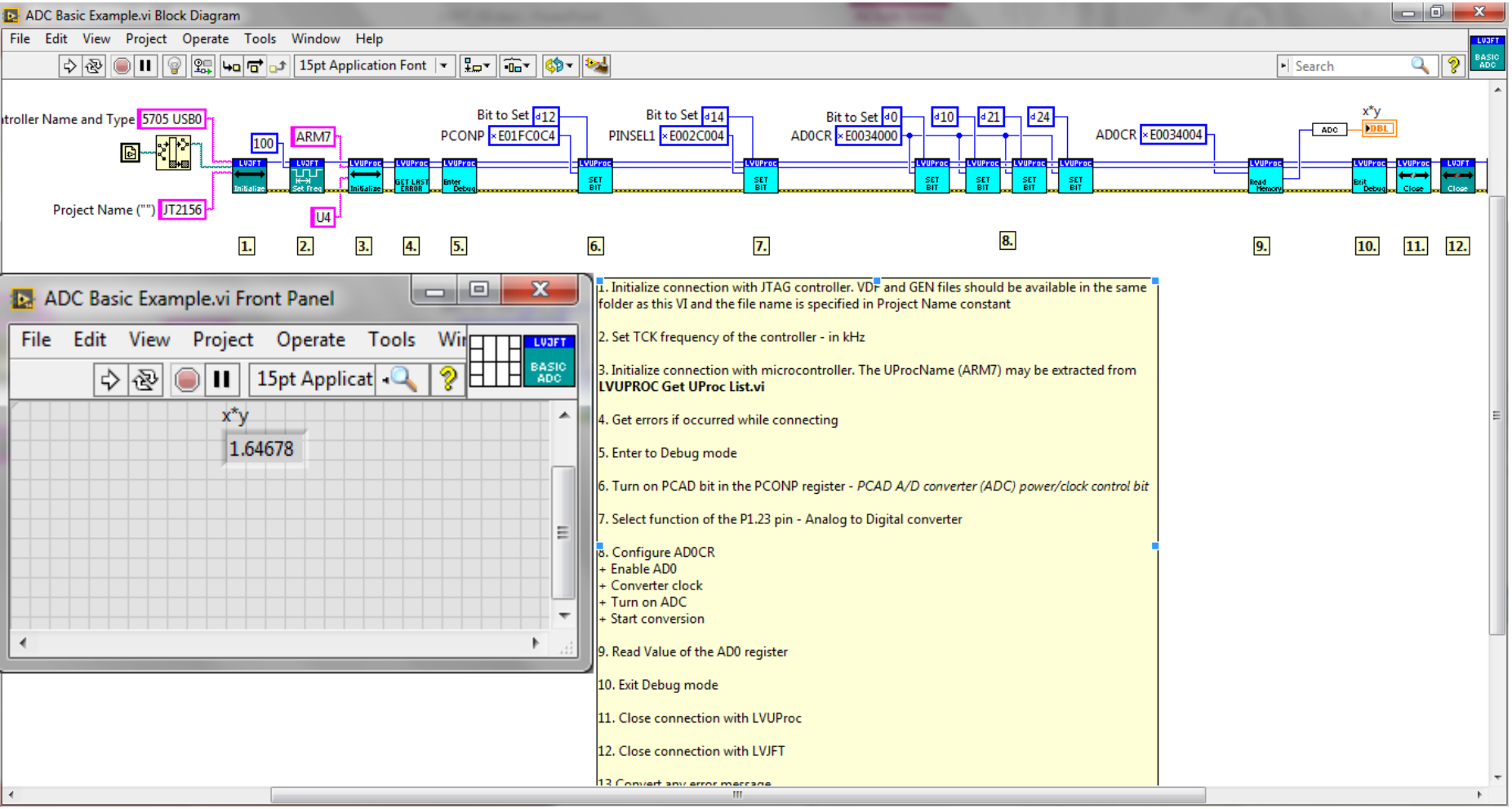




Using text based language

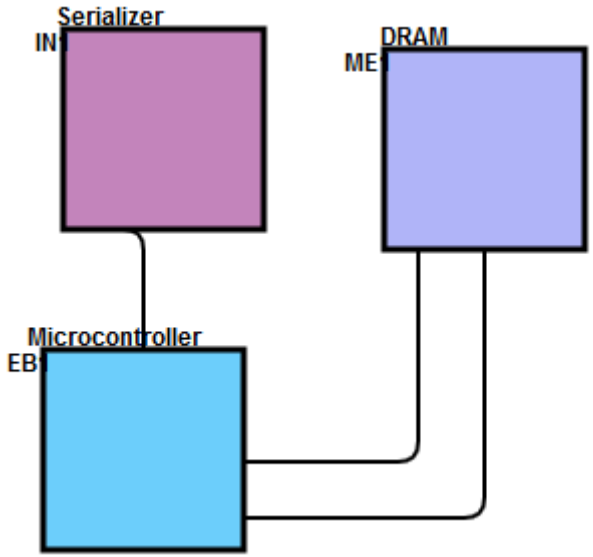
```
SetScanFrequency(100) #TCK set to 100kHz needed for NXP2468 Debug mode
Init("ARM7", "", "U4")
EnterDebug()

def CheckStatus(Value, Expected, Mask):
    if((Value & Mask)!= Expected):
        print("Invalid status", hex(Value), ": Expected", hex(Expected))
        return 1
    return 0

def ReadADC():
    pconp = ReadMemory(0xe01fc0c4)
    WriteMemory(0xe01fc0c4, pconp | 0x1000)  # Power on ADC block
    WriteMemory(0xe002c004, 0x00004000)      # Set function of pin to ADC
    WriteMemory(0xe0034000, 0x00200401)      # Power on ADC, select AD0.0, devide clock by 5
    WriteMemory(0xe0034000, 0x01200401)      # Start conversion
    Status = ReadMemory(0xe0034004)          # Read status
    if CheckStatus(Status, 0x80000000, 0xc0000000) == 0:
        ADC_value = ((Status & 0xffc0) >> 6)
        print("ADC voltage =", ((3.3/1024)*ADC_value), "V")
        voltage=((3.3/1024)*ADC_value)
    else:
        print ("Status ADC is not correct")
        voltage=0
    return voltage

voltage=ReadADC()
expected_voltage=3.3/2
tolerance=0.1 #10%
if voltage>expected_voltage*(1-tolerance) and voltage<expected_voltage*(1+tolerance):
  print ("within tolerance")
  teststatus=0
else:
  print ("not within  tolerance")
  teststatus=1


ExitDebug()
Close()
exit(teststatus)
```

Using LabVIEW with indicator



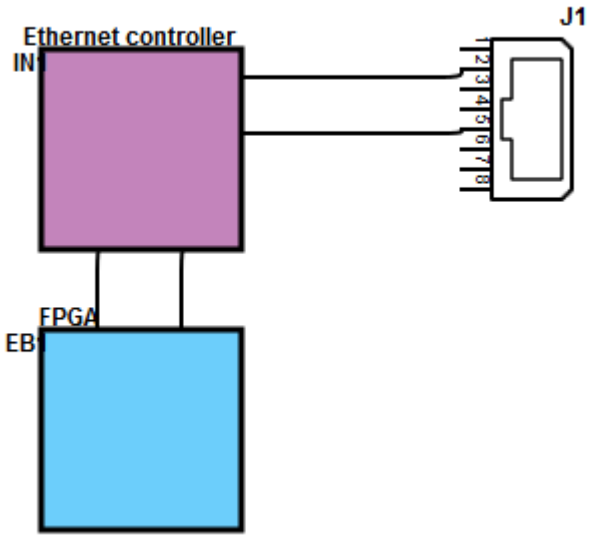
Other peripheral devices Ser_Des, DRAM

- Use either Boundary Scan chain (static tests)
- Or use Microcontroller core for dynamic (at speed) tests.

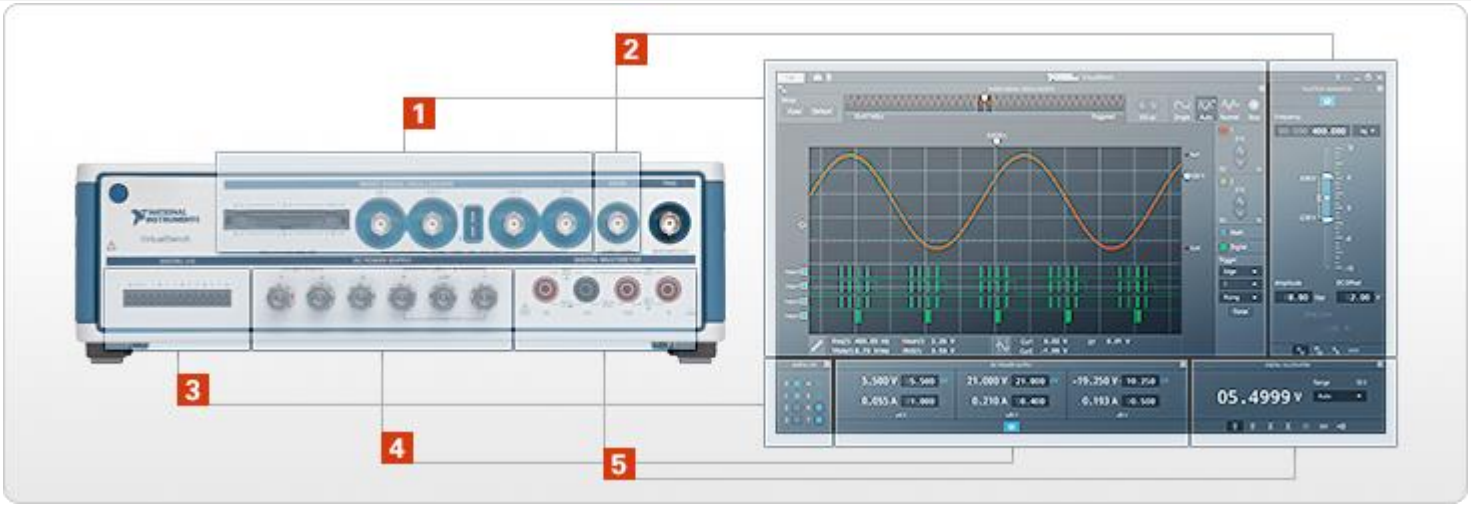


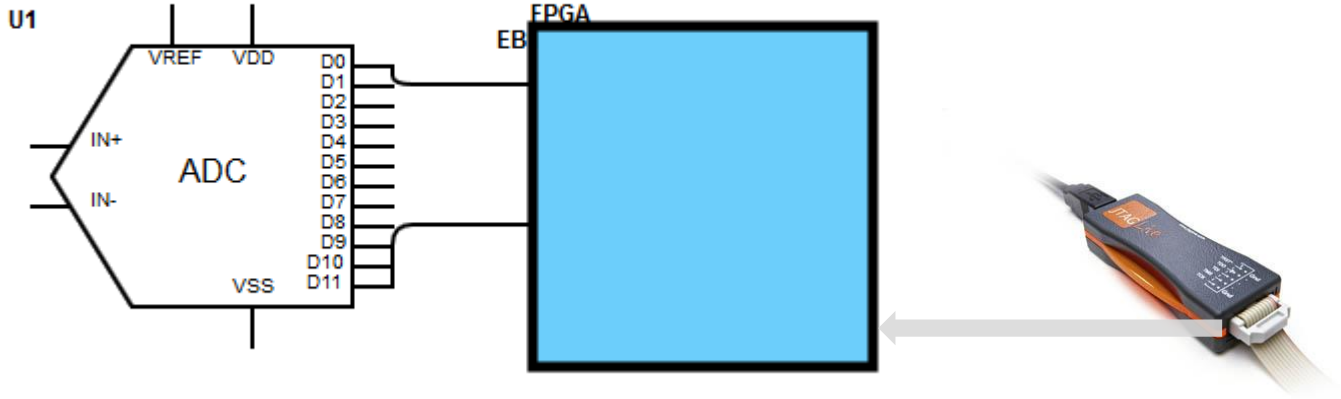
Other peripheral devices Ethernet

- Test Ethernet ID
- Carry out loopback test



Combine with functional tests





Next steps

Etteplan

- Add SPI Library.
- Enhance embedded Microprocessor Library.
- Publish the libraries to LabVIEW tools network

Conclusions

- Combination of JTAG and Functional measurements provides a powerful test strategy.
- Use either external instruments or JTAG MIOS module
- Simple JTAG controller and actions can be combined with functional test instruments or LabVIEW indicators.
- Add CoreCommander to read internal ADC values.
- No need for Software or Firmware inside FPGA or CPU.
- Suitable for
  - simple bench testing (R&D)
  - Mass production´(Bench top or fixture based)
  - Repair centre.

We *are* boundary-scan.™