Volume 9, Issue 3, March – 2024 International Journal of Innovative Science and Research Technology
ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/IJISRT24MAR2002

The Idea of an Integration Interface for Model-based Software Development's Processor-in-the-Loop (PiL) Simulation

Ganesh Kale[1]
Professional Engineer Embedded
Expleogroup
Berlin, Germany

Gregor Roering[2]
Professional Engineer System
Expleogroup
Cologne, Germany

Abstract:- Model-based design (MBD) has become a cornerstone in the development of embedded software, particularly in the automotive industry. Processor-in-the-Loop (PiL) simulation bridges the gap between virtual simulation environments and real hardware, enabling early verification and validation of control algorithms running on target processors. The design and implementation of toolchain is required for target specific code generation, compilation, and execution. Developing a toolchain specifically for the target architecture is crucial to preventing errors and ensuring smooth production. The configuration and designing of toolchain are one time effort of all Simulink model which want to be test on board.

This research investigates the importance of PiL for embedded systems in the automobile area. It describes the construction of a toolchain that integrates PiL simulation with Simulink, a popular MBD tool. The paper discusses the unique integration of TRACE32, a debugger and code analysis tool, with Simulink for testing programmes on Infineon devices.

TPT makes use of graphical test models that are easy to understand and have the capacity to automate complex closed loop tests in real time. It was Daimler Software Technology Research that initially developed TPT. Nowadays, vendors and automakers employ it in their development projects for production vehicles.

Keywords:- *PiL; MBD;Embedded; TPT; Software Development; Automotive.*

I. INTRODUCTION

The field of systems engineering unites several techniques and strategies for the creation of intricate systems. The diversity of the system's constituents is often the cause of a system's complexity. For instance, the intricacy of a product service system (PSS) results from the simultaneous development of services and the integrated assessment of product characteristics and functionalities in the immediate context of the services they are utilized in. On the other hand, coordinating the efforts of many specialized areas, including software, electronics, and mechanics, is challenging when it comes to mechatronic systems. The difficult coordination between several development strands or task areas within a development project is the cause of the complexity in both situations. [1]

The proliferation of complex automotive capabilities, along with high safety and performance standards, emphasizes the importance of sophisticated development processes. Model-Based Design (MBD) has emerged as a key method in this area, providing a formal framework for building, modelling, and evaluating automotive embedded systems. Despite the benefits of MBD, the shift from simulated models to actual hardware presents hurdles, notably in guaranteeing system authenticity. Compiler optimizations and hardware limits might cause inconsistencies, demanding creative ways to improve the integration of simulation and physical implementation. In this context, an integrated interface for Processor-in-the-Loop (PiL) simulation appears as a possible solution to these issues. [2]

II. MBD ACCORDING TO ISO26262

The International Electrotechnical Commission (IEC) and ISO (the International Organization for Standardization) work closely together. A 2011 formal publication of ISO 26262 specifications marked the evolution of IEC 61508, the general functional safety standard for E/E systems, into a specification. Adopting ISO 26262 contributes to ensuring that safety of automotive components is taken into account from the outset of development. It offers a thorough framework for handling safety during every stage of an automobile component's lifecycle, from initial risk assessment to ultimate decommissioning. By following ISO 26262, automotive manufacturers can ensure that their suppliers are meeting safety standards, preventing costly issues that might arise during production or after sale.[8] The standard considers the trend of increasing integration of hardware and software in automotive electronic systems. It acknowledges that hardware and software must be evaluated in tandem to attain the highest level of safety and offers comprehensive guidelines for the concurrent development and testing of both. This guarantees that every component of the system is taken into account and tested as a whole, encouraging a more complete and rigorous approach to functional safety. The development of highly integrated systems in the automotive industry typically requires proof of conformity to ISO 26262, the international standard for functional safety of motor vehicles. ISO 26262 classifies the various functions within the vehicle into safety requirement levels (Automotive Safety Integrity Levels (ASIL) from A to D, with ASIL-D being the highest level and entailing the most stringent requirements. [9] [10]

III. PROCESSOR IN THE LOOP (PIL)

Processor-in-the-Loop Testing (or PIL Testing) means that the code to be tested is built using a cross compiler and executed on the target processor. Smaller pieces of software (such as a unit) can be built separately and executed on the target using a Processor-in-the-Loop environment (PiL). To do this, an evaluation board is linked to the host PC (for example, via USB), and the PC controls the board's test execution. A PIL test not only exhibits the right functional behaviour, but it also shows that the tested function executes quickly enough on the target CPU and that the stack can manage the load.[4]

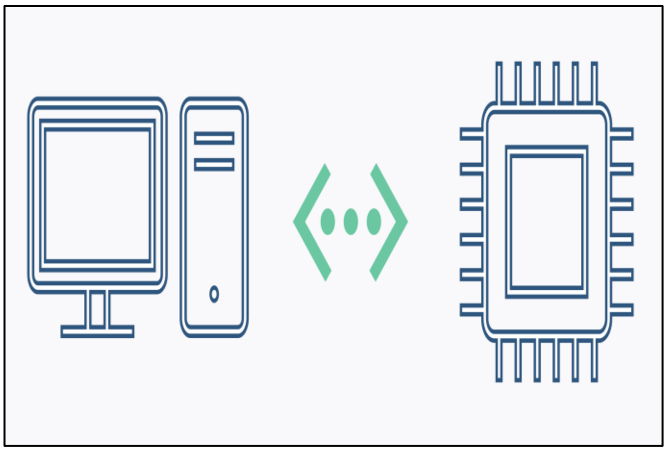
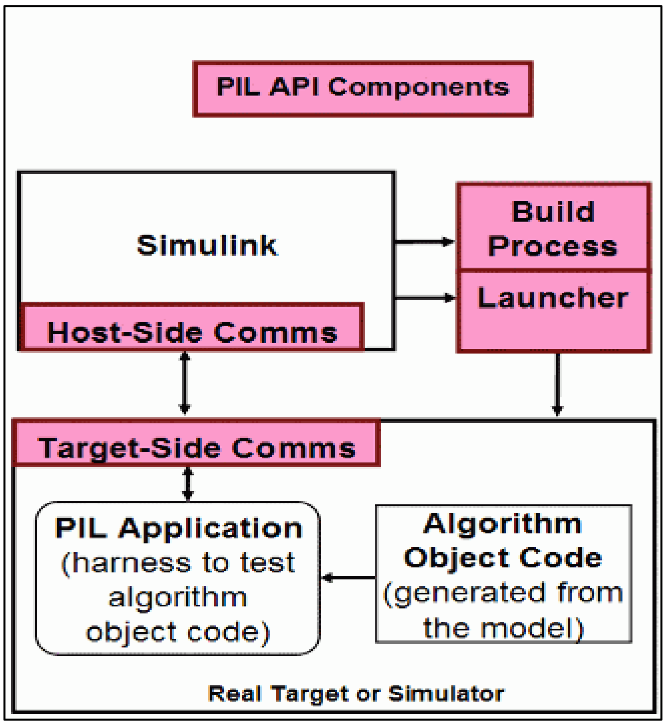


Fig 1 Interaction of PC and Processor

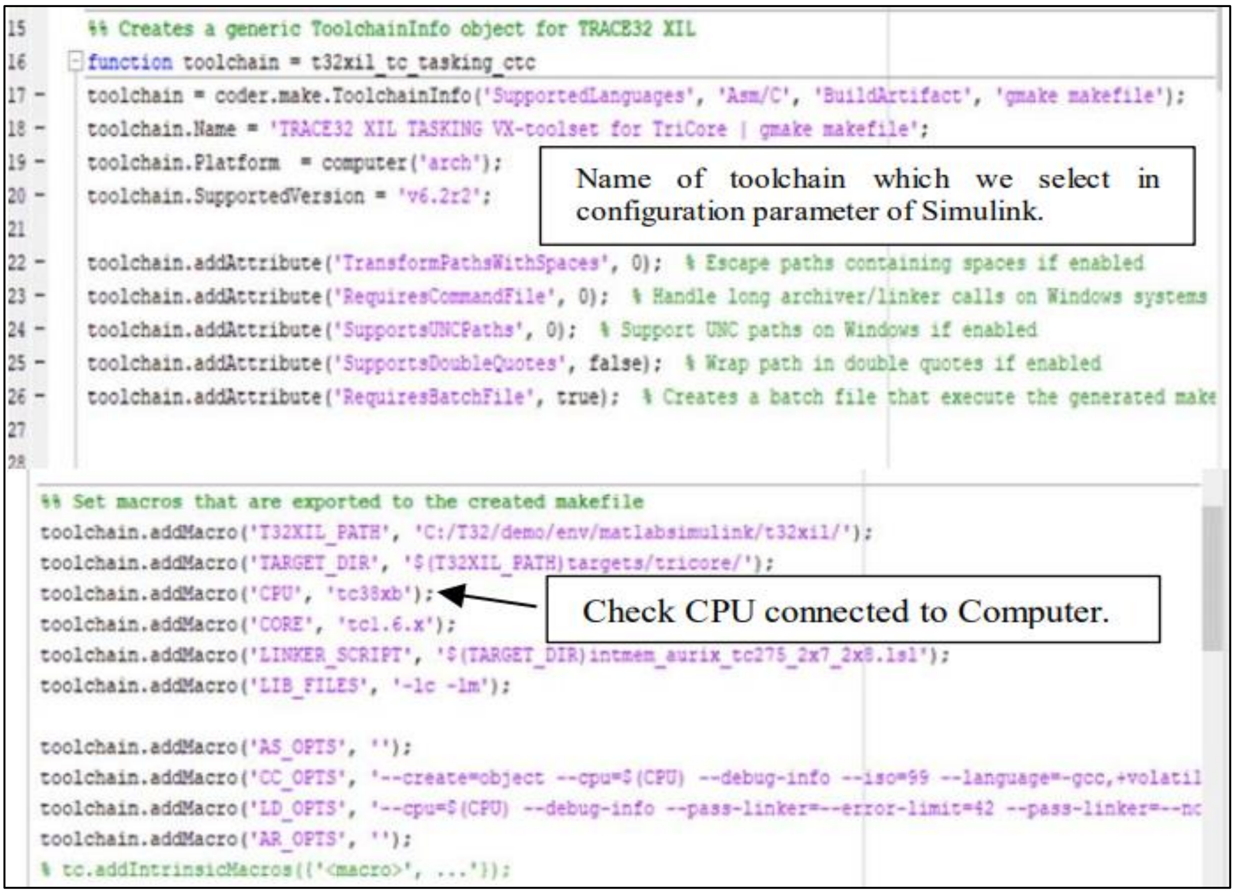
Processor-in-the-Loop (PiL) testing is the process of evaluating and verifying embedded software on the processor that will ultimately be used in the Electronic Control Unit (ECU). Typically, the algorithms and functions are created in a development environment on a PC, using either model-based or direct C or C++ programming. For instance, this may be a model from ASCET, TargetLink, Simulink, or ASCET-DEVELOPER. For the processor that will eventually be utilised in the vehicle's ECU, the generated C/C++ code has to be created using a unique "cross" compiler. To determine if the built code is also compatible with the target CPU, PiL tests are run. Typically, the PiL test control algorithms are run on an evaluation board. Occasionally, PiL testing are carried out using the actual ECU. In contrast to Software-in-Loop (SiL) testing, both versions make use of the actual processor found in the controller rather than the PC. The benefit of using the target processor is that compiler problems may be found. In PiL tests, "in-the-loop" refers to the integration of the controller into the simulation or testing environment.[11]

IV. DEVELOPMENT OF TOOLCHAIN FOR PIL TESTING

We are using TRACE32 Tricore Debugger to connect our hardware to computer. To make communication between Simulink and Trace32 debugger we need to develop a tool chain that helps us to generate code from Simulink model and generate that integrates the cross compiler in order to create a flashable file directly from Simulink. [12]


PIL API Components
Simulink
Host-Side Comms
Build Process
Launcher
Target-Side Comms
PIL Application (harness to test algorithm object code)
Algorithm Object Code (generated from the model)
Real Target or Simulator
Fig 2 Steps API Communication [15]

We have several .m files for different compiler toolchains. In our project, we are using Tasking VX toolset. In the below figure, we have customized a toolchain according to our board specific requirements. [13]



```
%% Creates a generic ToolchainInfo object for TRACE32 XIL
function toolchain = t32xil_tc_tasking_ctc
toolchain = coder.make.ToolchainInfo('SupportedLanguages', 'Asm/C', 'BuildArtifact', 'gmake makefile');
toolchain.Name = 'TRACE32 XIL TASKING VX-toolset for TriCore | gmake makefile';
toolchain.Platform  = computer('arch');
toolchain.SupportedVersion = 'v6.2r2';

toolchain.addAttribute('TransformPathsWithSpaces', 0);  % Escape paths containing spaces if enabled
toolchain.addAttribute('RequiresCommandFile', 0);  % Handle long archiver/linker calls on Windows systems
toolchain.addAttribute('SupportsUNCPaths', 0);  % Support UNC paths on Windows if enabled
toolchain.addAttribute('SupportsDoubleQuotes', false);  % Wrap path in double quotes if enabled
toolchain.addAttribute('RequiresBatchFile', true);  % Creates a batch file that execute the generated make


%% Set macros that are exported to the created makefile
toolchain.addMacro('T32XIL_PATH', 'C:/T32/demo/env/matlabsimulink/t32xil/');
toolchain.addMacro('TARGET_DIR', '$(T32XIL_PATH)targets/tricore/');
toolchain.addMacro('CPU', 'tc38xb');
toolchain.addMacro('CORE', 'tc1.6.x');
toolchain.addMacro('LINKER_SCRIPT', '$(TARGET_DIR)intmem_aurix_tc275_2x7_2x8.lsl');
toolchain.addMacro('LIB_FILES', '-lc -lm');

toolchain.addMacro('AS_OPTS', '');
toolchain.addMacro('CC_OPTS', '--create=object --cpu=$(CPU) --debug-info --iso=99 --language=-gcc,+volatil
toolchain.addMacro('LD_OPTS', '--cpu=$(CPU) --debug-info --pass-linker=--error-limit=42 --pass-linker=--nc
toolchain.addMacro('AR_OPTS', '');
% tc.addIntrinsicMacros({'<macro>', ...'});
```

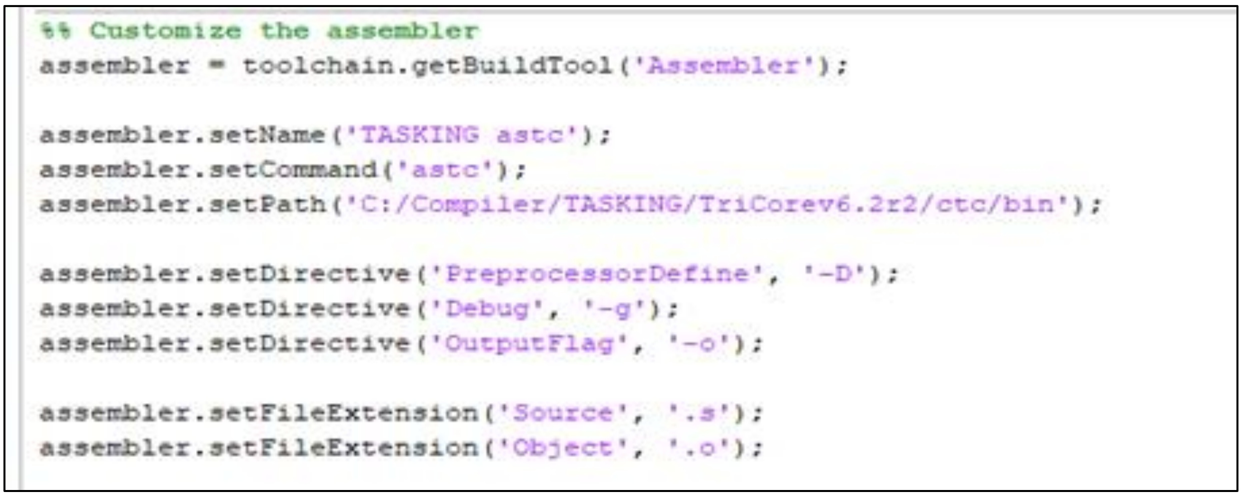
Name of toolchain which we select in configuration parameter of Simulink.

Check CPU connected to Computer.

Fig 3 Toolchain .m File Function

➢ *Assembler:*

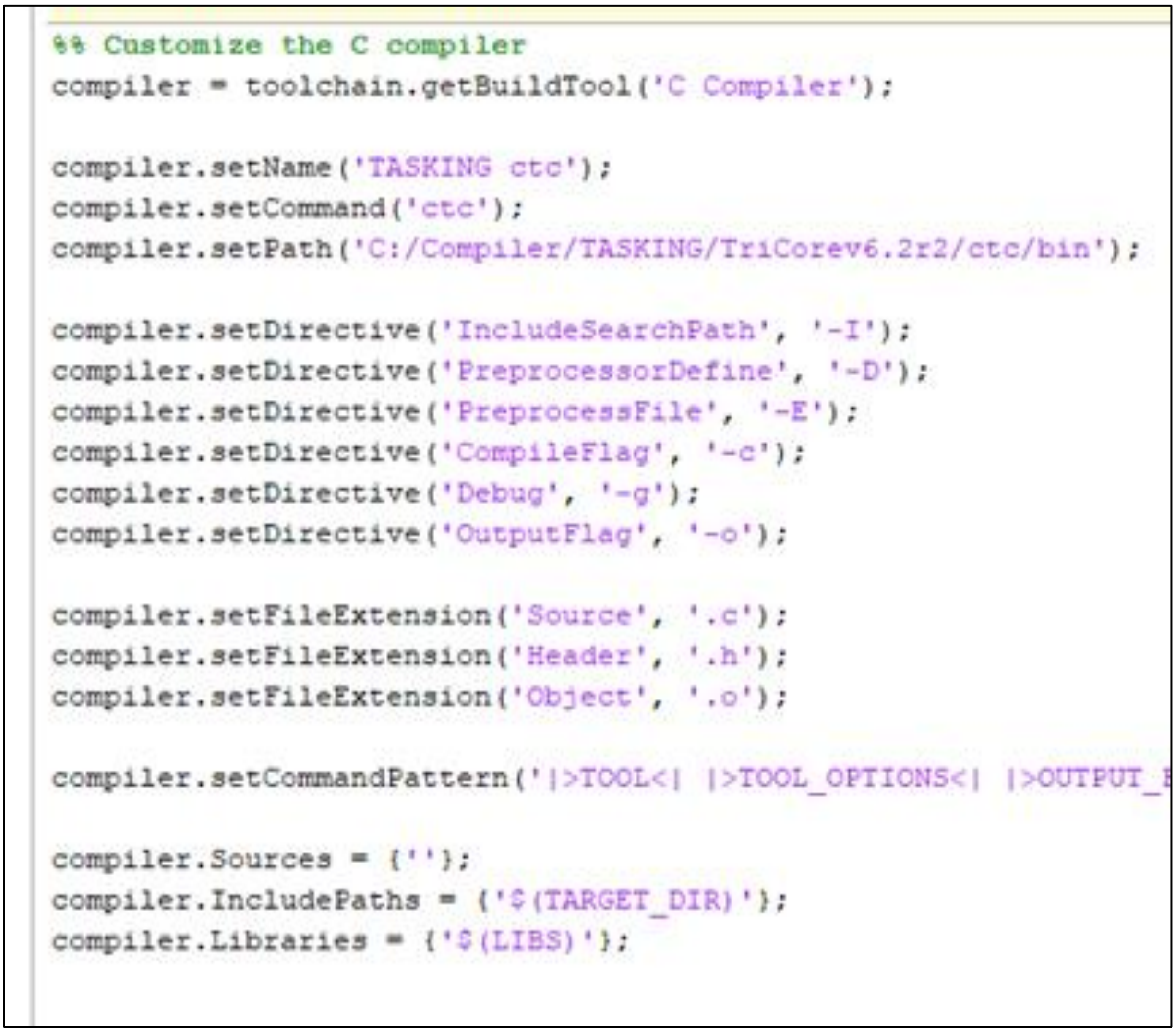
In the domain of computer architecture, assemblers play a critical role in translating human-readable assembly language instructions into machine-executable object code. This process involves the conversion of mnemonics, which represent operations and addressing modes, along with their associated syntax, into their corresponding numerical equivalents. The resulting object code typically comprises an operation code (opcode) responsible for specifying the instruction to be executed, along with additional control bits and relevant data. Assemblers further contribute by resolving symbolic names assigned to memory locations and other entities within the program. Additionally, they possess the capability to evaluate constant expressions, streamlining the programming process. A key advantage of assemblers lies in their extensive utilization of symbolic references. This approach significantly reduces the time required for manual calculations and address updates whenever program modifications are necessary. Furthermore, the inclusion of macro facilities within most assemblers allows for textual substitution. This functionality proves valuable in generating frequently used, short instruction sequences directly within the program, eliminating the need for separate subroutine calls. [15] [16]



```
%% Customize the assembler
assembler = toolchain.getBuildTool('Assembler');

assembler.setName('TASKING astc');
assembler.setCommand('astc');
assembler.setPath('C:/Compiler/TASKING/TriCorev6.2r2/ctc/bin');

assembler.setDirective('PreprocessorDefine', '-D');
assembler.setDirective('Debug', '-g');
assembler.setDirective('OutputFlag', '-o');

assembler.setFileExtension('Source', '.s');
assembler.setFileExtension('Object', '.o');
```

Fig 4 Customize Assembler

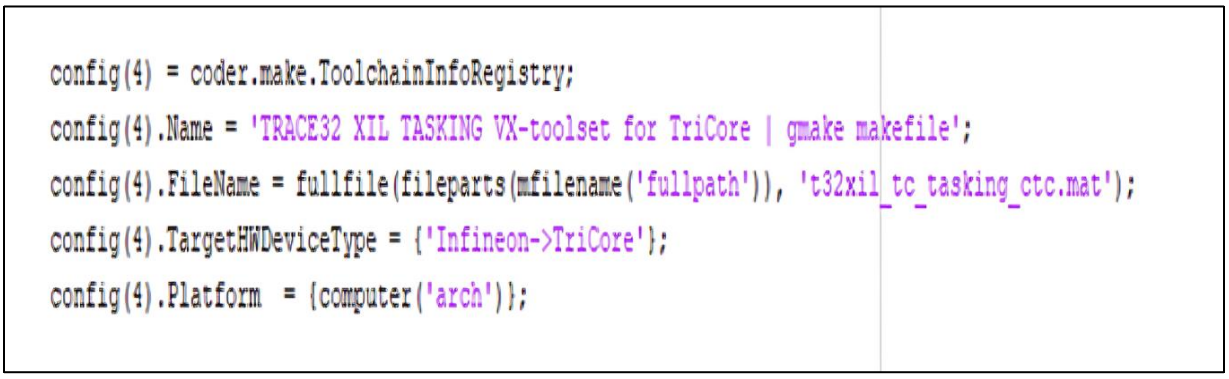
An application that transfers computer code written in one programming language (the source language) into another (the target language) is called a compiler. Programmes that convert source code from a high-level programming language (such as assembly, object, or machine code) to a low-level programming language (such as machine, assembly, or object code) in order to produce an executable programme are typically referred to as compilers. [16]



```
%% Customize the C compiler
compiler = toolchain.getBuildTool('C Compiler');

compiler.setName('TASKING ctc');
compiler.setCommand('ctc');
compiler.setPath('C:/Compiler/TASKING/TriCorev6.2r2/ctc/bin');

compiler.setDirective('IncludeSearchPath', '-I');
compiler.setDirective('PreprocessorDefine', '-D');
compiler.setDirective('PreprocessFile', '-E');
compiler.setDirective('CompileFlag', '-c');
compiler.setDirective('Debug', '-g');
compiler.setDirective('OutputFlag', '-o');

compiler.setFileExtension('Source', '.c');
compiler.setFileExtension('Header', '.h');
compiler.setFileExtension('Object', '.o');

compiler.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_

compiler.Sources = {''};
compiler.IncludePaths = {'$(TARGET_DIR)'};
compiler.Libraries = {'$(LIBS)'};
```

Fig 5 Customization of C Compiler [19]

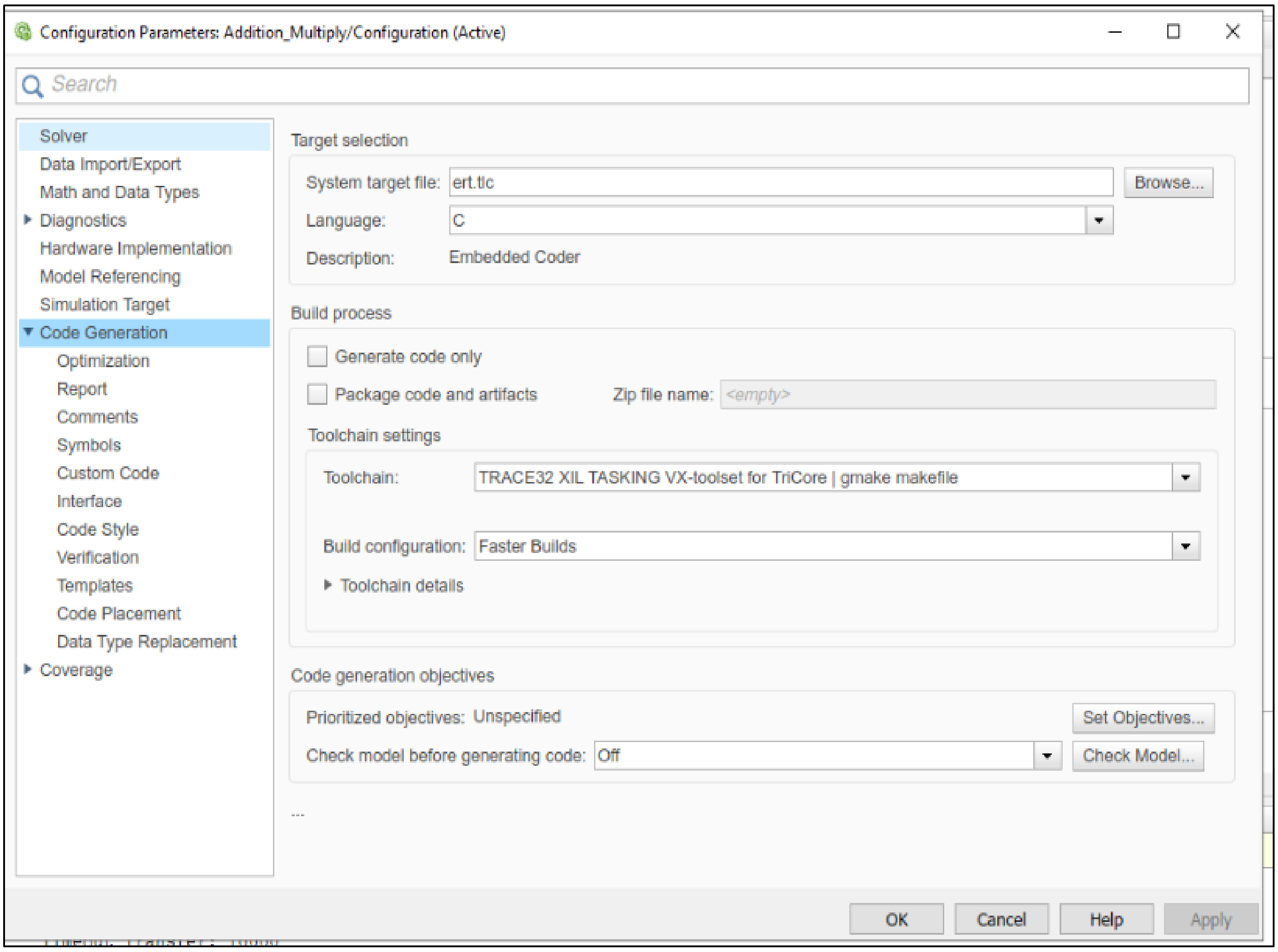
To register custom toolchain, we must make changes in rtwTargetInfo.m file to show our custom toolchain in Simulink configuration parameters. This file is provided by Lauterbach and helps to add custom toolchains.



```
config(4) = coder.make.ToolchainInfoRegistry;
config(4).Name = 'TRACE32 XIL TASKING VX-toolset for TriCore | gmake makefile';
config(4).FileName = fullfile(fileparts(mfilename('fullpath')), 't32xil_tc_tasking_ctc.mat');
config(4).TargetHWDeviceType = {'Infineon->TriCore'};
config(4).Platform  = {computer('arch')};
```

Fig 6 Toolchain

➢ *After Successfully Registering Toolchain, it will Reflect in the Configuration Parameters as Below.*


Configuration Parameters: Addition_Multiply/Configuration (Active)

Search

- Solver
- Data Import/Export
- Math and Data Types
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target
- Code Generation
  - Optimization
  - Report
  - Comments
  - Symbols
  - Custom Code
  - Interface
  - Code Style
  - Verification
  - Templates
  - Code Placement
  - Data Type Replacement
- Coverage

Target selection

System target file: ert.tlc Browse...

Language: C

Description: Embedded Coder

Build process

☐ Generate code only

☐ Package code and artifacts Zip file name: <empty>

Toolchain settings

Toolchain: TRACE32 XIL TASKING VX-toolset for TriCore | gmake makefile

Build configuration: Faster Builds

Toolchain details

Code generation objectives

Prioritized objectives: Unspecified Set Objectives...

Check model before generating code: Off Check Model...

...

OK Cancel Help Apply

Fig 7 Selection of Toolchain in Simulink

➢ *The Connection of Debugger from Lauterbach with PC and Target Hardware is Shown in Figure*

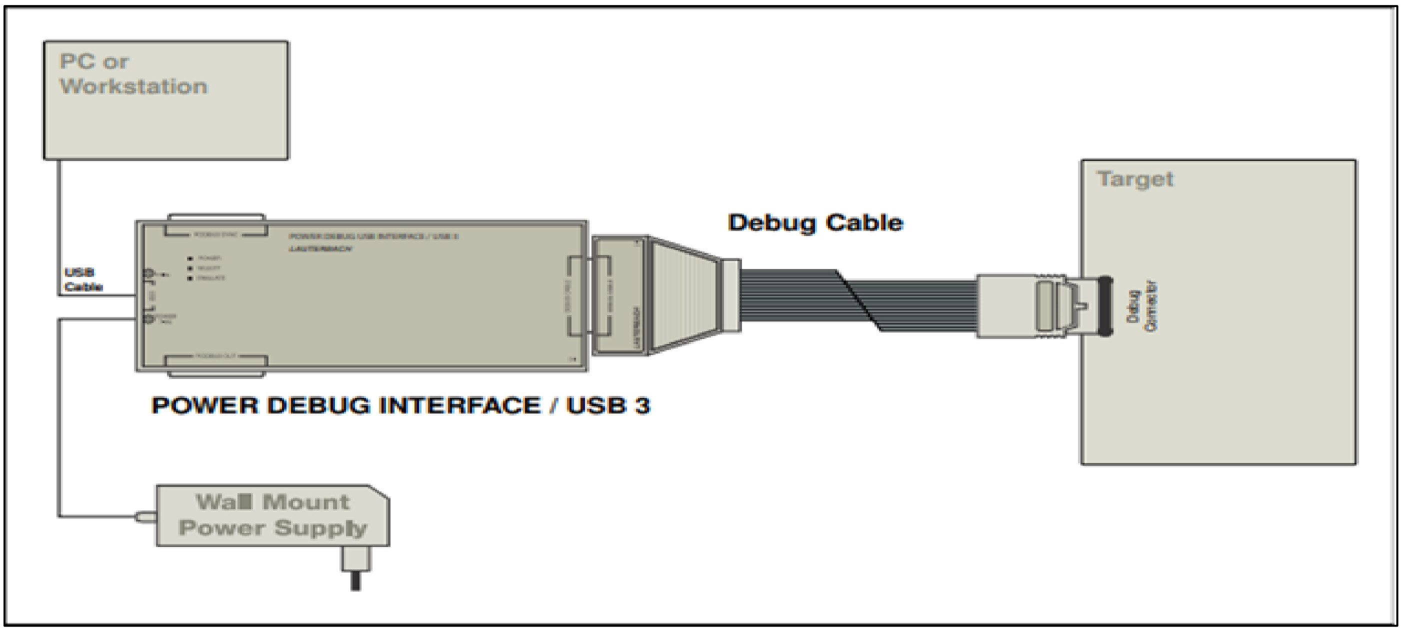
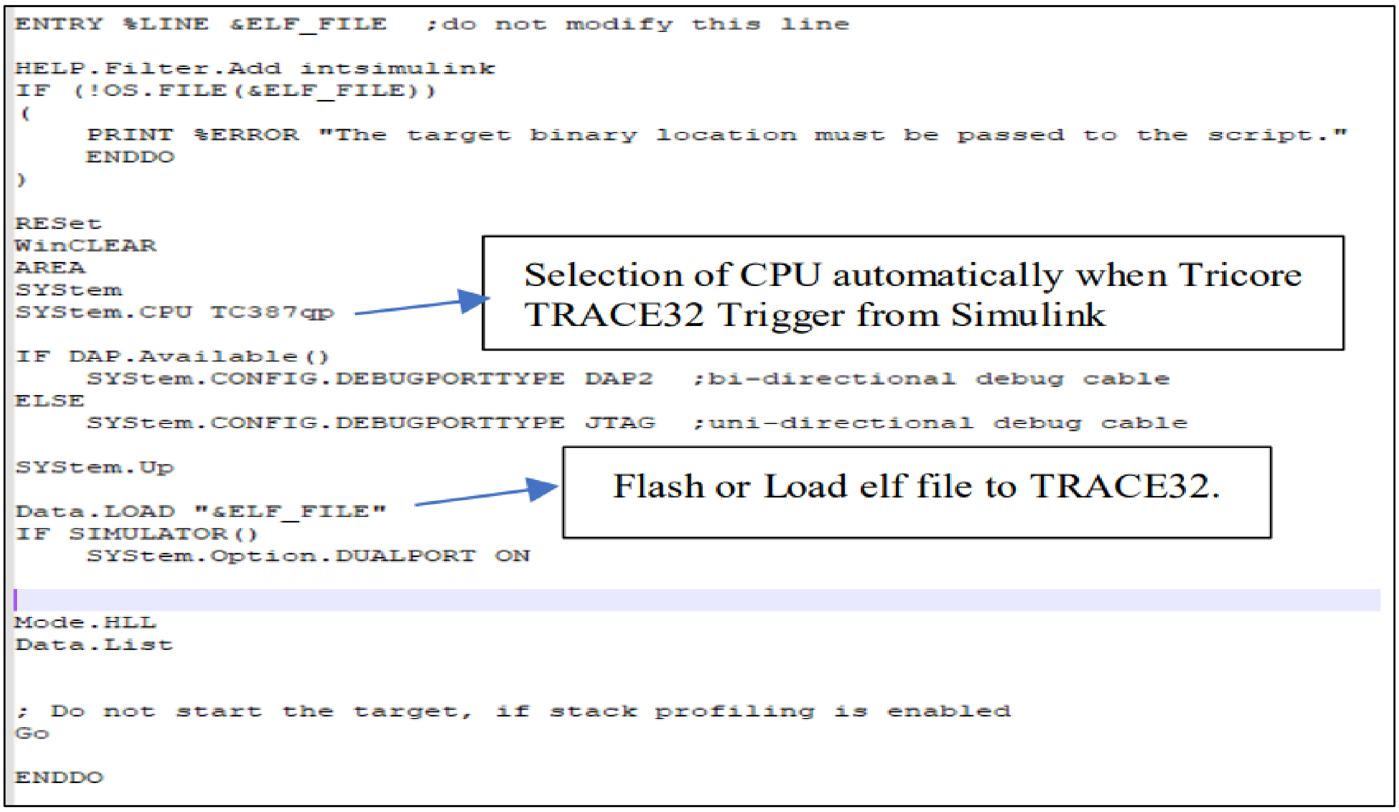


Fig 8 Connection of Debugger, PC and Processor [18]

V. ROLE OF TRACE32_TC387QP_STARTUP.CMM FILE

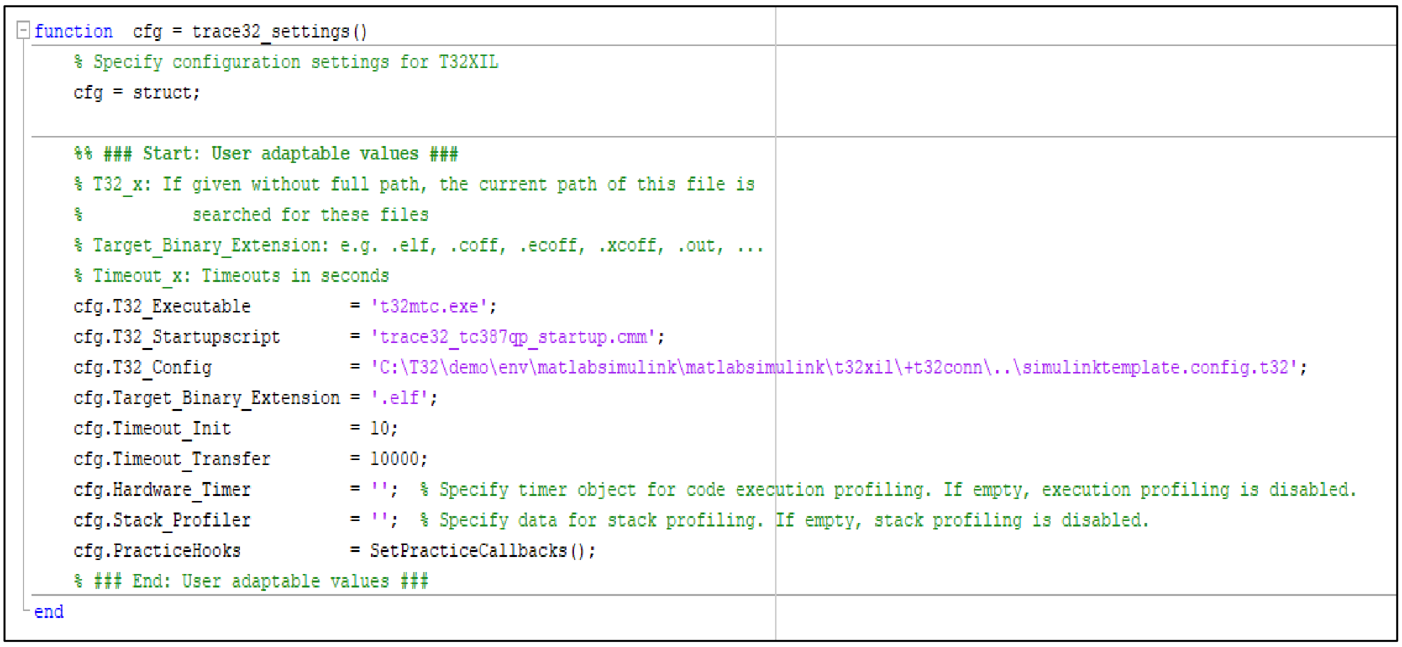
In TRACE32, a .cmm file, which stands for "Command Module," is a script file that contains a series of TRACE32 debugger commands written in the TRACE32 script language. These script files play a crucial role in automating and streamlining various debugging and analysis tasks.[14]


Selection of CPU automatically when Tricore TRACE32 Trigger from Simulink
Flash or Load elf file to TRACE32.

```
ENTRY %LINE &ELF_FILE  ;do not modify this line

HELP.Filter.Add intsimulink
IF (!OS.FILE(&ELF_FILE))
(
    PRINT %ERROR "The target binary location must be passed to the script."
    ENDDO
)

RESet
WinCLEAR
AREA
SYStem
SYStem.CPU TC387qp

IF DAP.Available()
    SYStem.CONFIG.DEBUGPORTTYPE DAP2  ;bi-directional debug cable
ELSE
    SYStem.CONFIG.DEBUGPORTTYPE JTAG  ;uni-directional debug cable

SYStem.Up

Data.LOAD "&ELF_FILE"
IF SIMULATOR()
    SYStem.Option.DUALPORT ON


Mode.HLL
Data.List


; Do not start the target, if stack profiling is enabled
Go

ENDDO
```

Fig 9 Trace32_Tc387qp_Startup.Cmm Script

The trace32_settings.m file is a configuration file used in MATLAB and Simulink when working with the TRACE32 XIL (Processor-in-the-Loop) target connectivity. This file defines settings and parameters that specify how Simulink interacts with the TRACE32 debugger when conducting PiL testing. It's a critical part of setting up and configuring the PiL environment for testing and debugging embedded software on real or emulated hardware with TRACE32. [14]



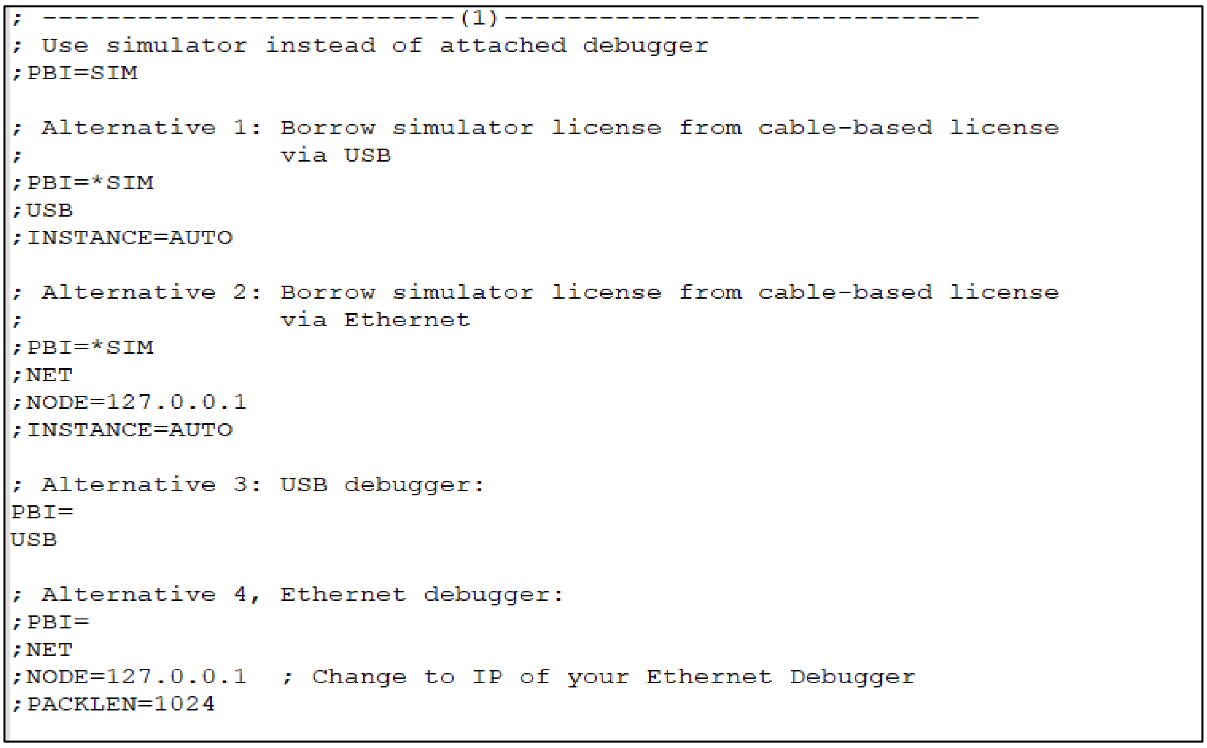
```
function  cfg = trace32_settings()
    % Specify configuration settings for T32XIL
    cfg = struct;

    %% ### Start: User adaptable values ###
    % T32_x: If given without full path, the current path of this file is
    %            searched for these files
    % Target_Binary_Extension: e.g. .elf, .coff, .ecoff, .xcoff, .out, ...
    % Timeout_x: Timeouts in seconds
    cfg.T32_Executable          = 't32mtc.exe';
    cfg.T32_Startupscript       = 'trace32_tc387qp_startup.cmm';
    cfg.T32_Config              = 'C:\T32\demo\env\matlabsimulink\matlabsimulink\t32xil\+t32conn\..\simulinktemplate.config.t32';
    cfg.Target_Binary_Extension = '.elf';
    cfg.Timeout_Init            = 10;
    cfg.Timeout_Transfer        = 10000;
    cfg.Hardware_Timer          = '';  % Specify timer object for code execution profiling. If empty, execution profiling is disabled.
    cfg.Stack_Profiler          = '';  % Specify data for stack profiling. If empty, stack profiling is disabled.
    cfg.PracticeHooks           = SetPracticeCallbacks();
    % ### End: User adaptable values ###
end
```

Fig 10 Trace32_Settings.m Script

This script helps to execute t32mtc.exe file which is executable product of Trace32 for tricore boards. User also add startup script file name that user already configure. In cfg.T32.Config, specifying the exact file path for simulinktemplate.config.t32 is optional. Alternatively, leaving it empty prompts the script to dynamically acquire the configuration file from the MATLAB setup path. These settings help to automatically trigger trace32 application which is suitable for tricore board.

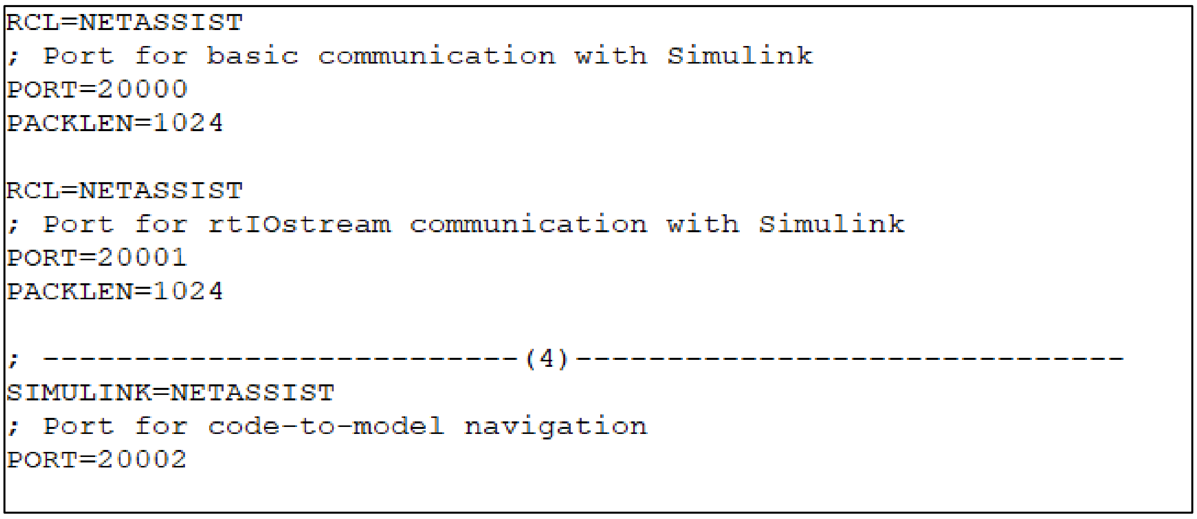
VI. TRACE32 CONFIGURATION FILE FOR TRACE32 INTEGRATION IN SIMULINK

Lauterbach has provided simulinktemplate.config.32 file to adapt communication between trace32 debugger and Simulink. In below figure, we have different alternatives to select attached trace32 debugger.



```
; --------------------------(1)------------------------------
; Use simulator instead of attached debugger
;PBI=SIM

; Alternative 1: Borrow simulator license from cable-based license
;                via USB
;PBI=*SIM
;USB
;INSTANCE=AUTO

; Alternative 2: Borrow simulator license from cable-based license
;                via Ethernet
;PBI=*SIM
;NET
;NODE=127.0.0.1
;INSTANCE=AUTO

; Alternative 3: USB debugger:
PBI=
USB

; Alternative 4, Ethernet debugger:
;PBI=
;NET
;NODE=127.0.0.1  ; Change to IP of your Ethernet Debugger
;PACKLEN=1024
```

Fig 11 Alternative to Select Debugger Attachment



```
RCL=NETASSIST
; Port for basic communication with Simulink
PORT=20000
PACKLEN=1024

RCL=NETASSIST
; Port for rtIOstream communication with Simulink
PORT=20001
PACKLEN=1024

; --------------------------(4)------------------------------
SIMULINK=NETASSIST
; Port for code-to-model navigation
PORT=20002
```

Fig 12 Communication Port to TRACE32

TRACE32 XIL is a fully integrated Simulink plug-in for processor-in-the-loop simulations using the MATLAB rtiostream API for PIL Target Connectivity. The generated code may be cross compiled, deployed, run, and debugged on a custom target. During simulation, PRACTICE callbacks and stack profiling via code instrumentation are supported. The TRACE32 XIL plug-in's navigation functions allow us to swiftly transition between model components and the appropriate parts of C/C++ and object code. [14]

VII. INTEGRATION OF TOOLCHAIN WITH SIMULINK EXAMPLE

➢ *A Counter is a Digital or Electronic Device used to Count Occurrences or Events.*

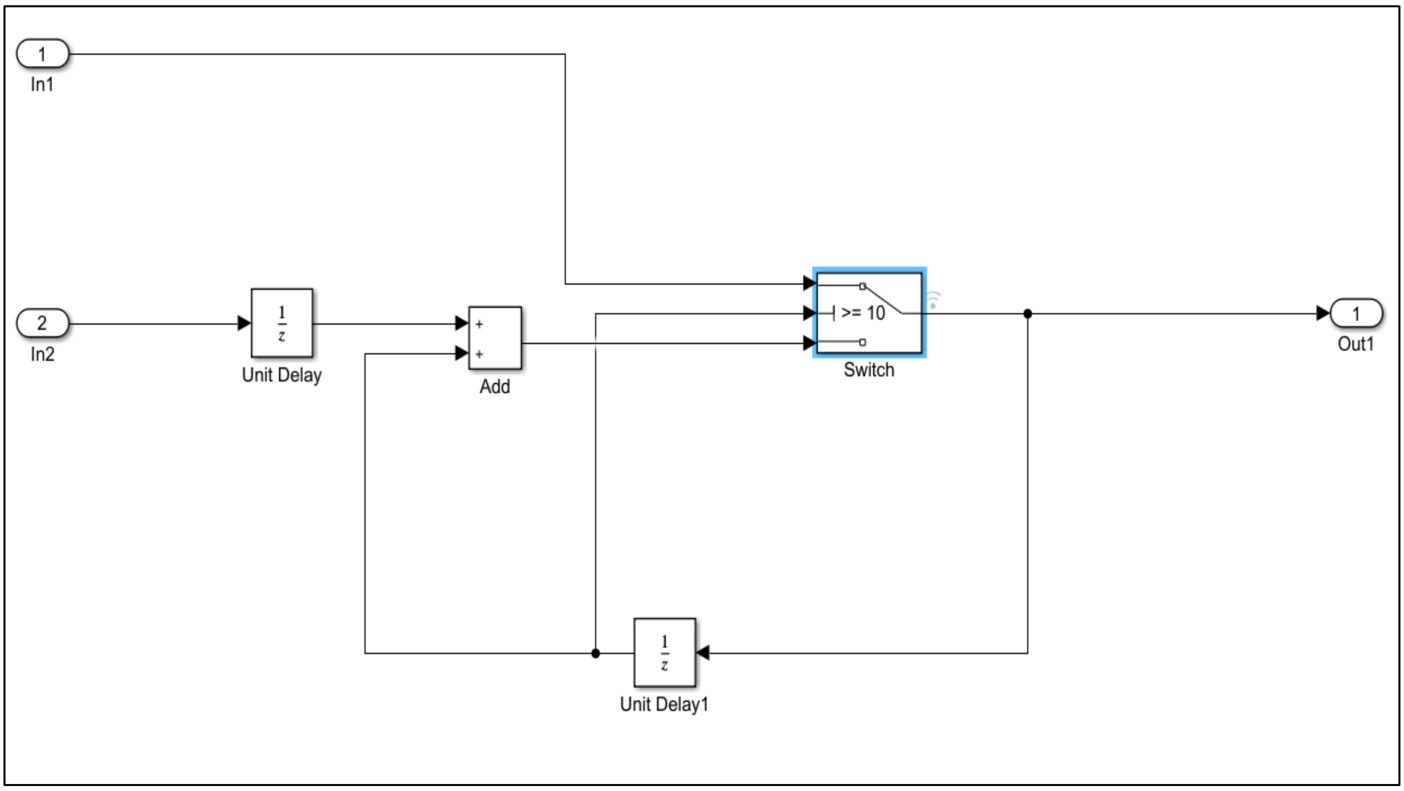


Fig 13 Up Counter Subsystem

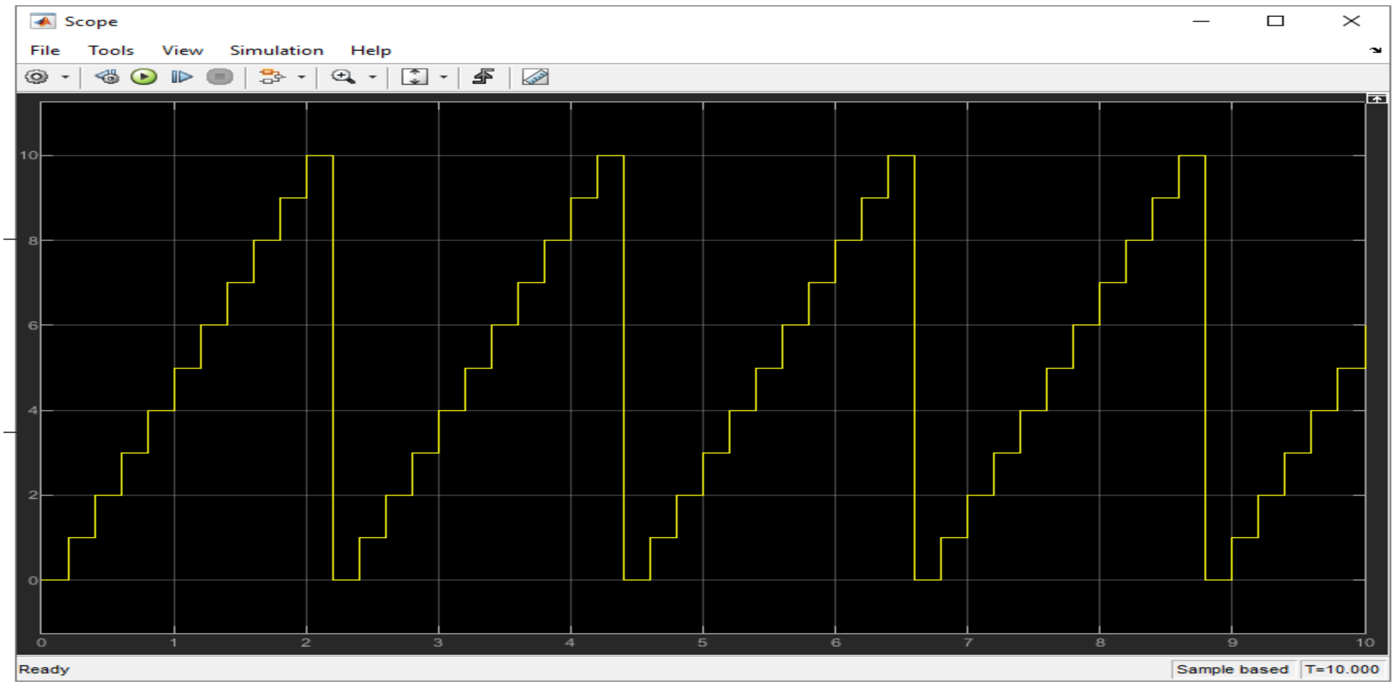
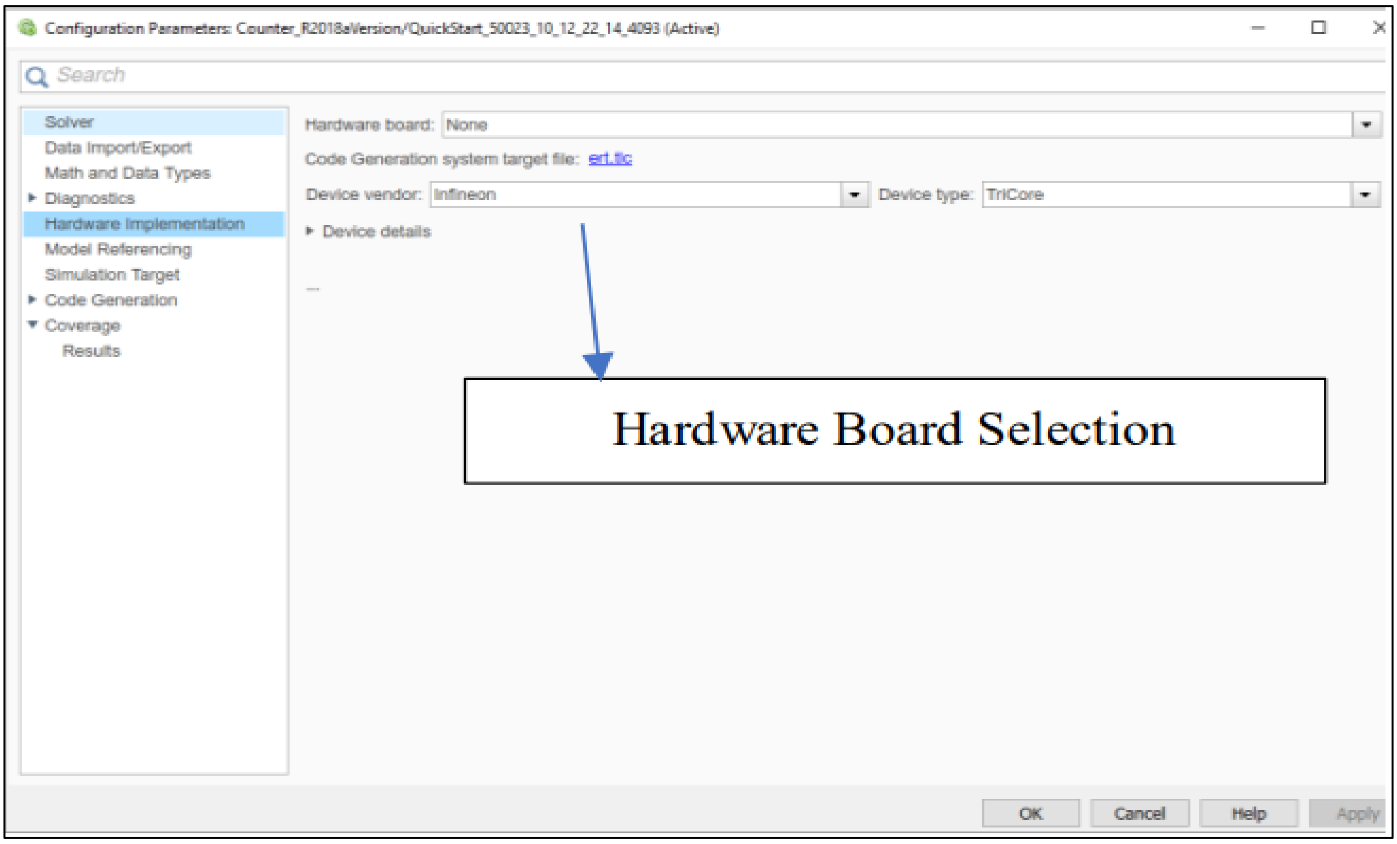


Fig 14 Output of Model

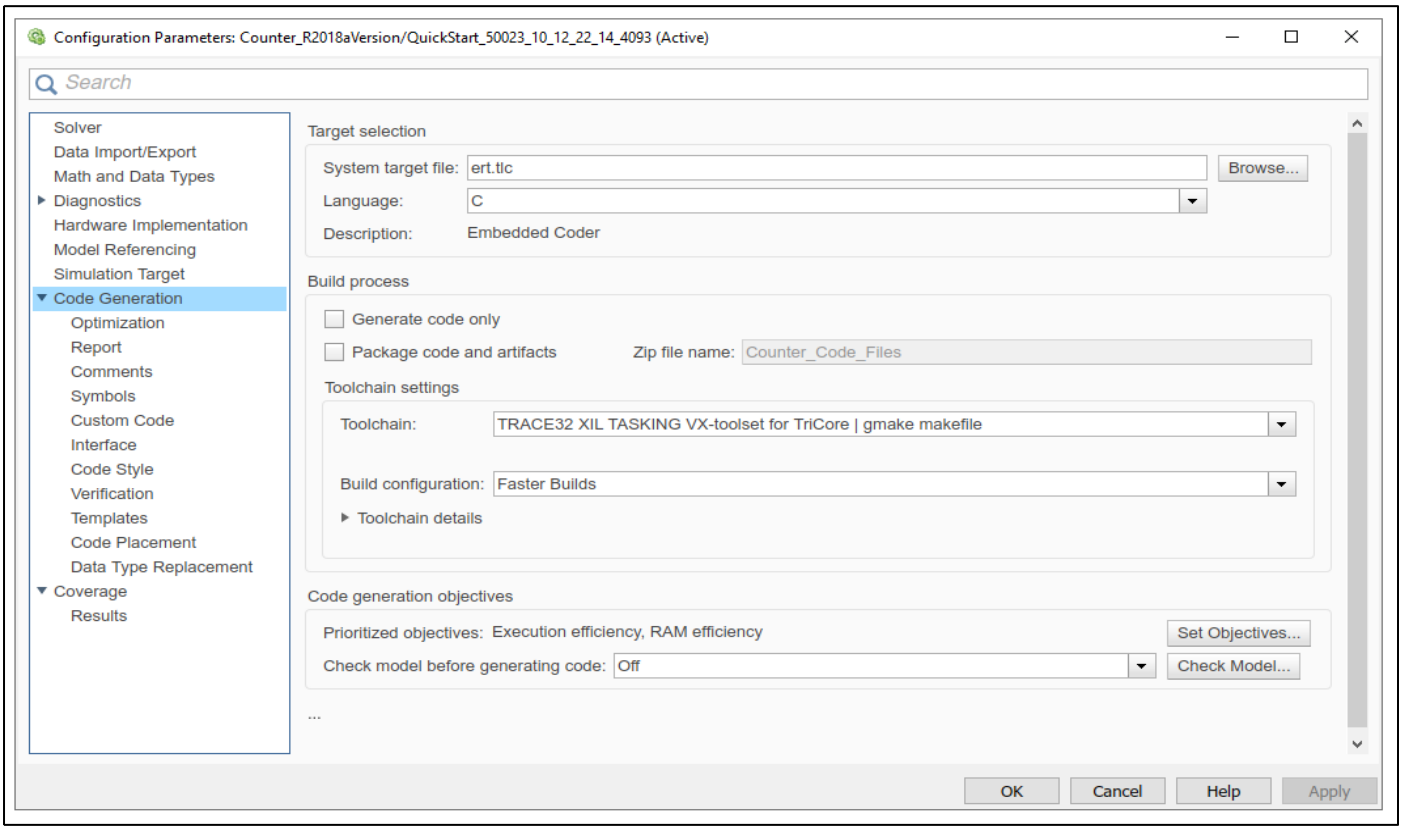
The Model configuration parameter settings for PiL simulation are as follows: The solver is always used as fixed step solver because of the real time application. The below figure shows the selection of hardware implementation.



Hardware board: None
Code Generation system target file: ert.tlc
Device vendor: Infineon
Device type: TriCore

Fig 15 Selection of Hardware Implementation

In code generation, selection of toolchain that is developed for PiL simulation i.e. TRACE32 XIL TASKING VX-toolset for TriCore|gmake| makefile. An explanation how the toolchain is developed can be found in the above section. This toolchain helps to generate all required files for board specific code generation process.



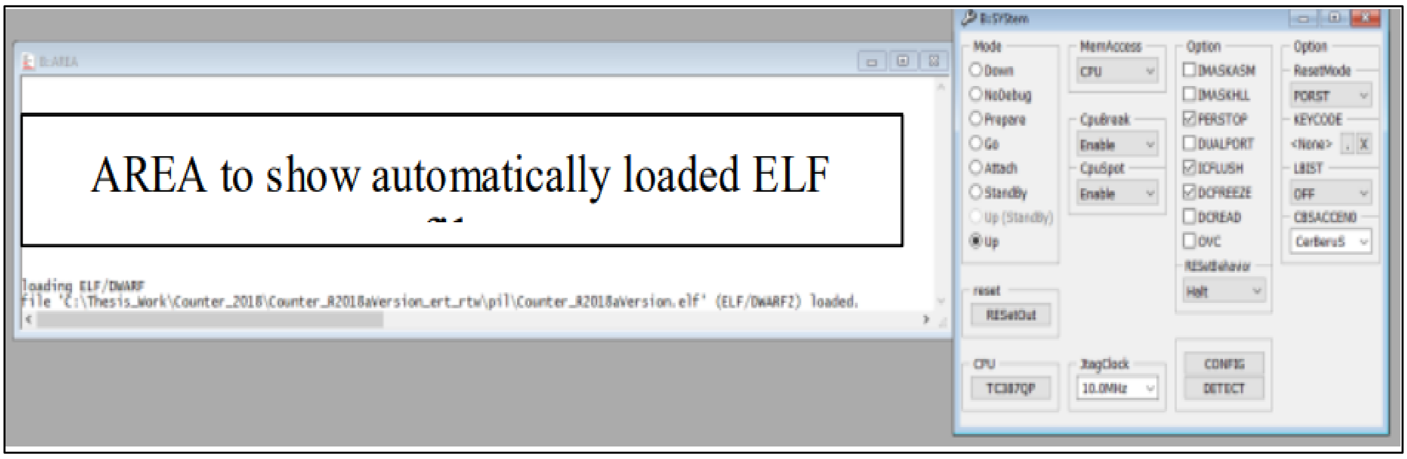
Target selection
System target file: ert.tlc
Language: C
Description: Embedded Coder
Build process
Generate code only [ ]
Package code and artifacts [ ] Zip file name: Counter_Code_Files
Toolchain settings
Toolchain: TRACE32 XIL TASKING VX-toolset for TriCore | gmake makefile
Build configuration: Faster Builds
Toolchain details
Code generation objectives
Prioritized objectives: Execution efficiency, RAM efficiency
Check model before generating code: Off

Fig 16 Selection of Toolchain and Embedded Coder

➢ *Run PiL Simulation in Simulink:*

The selection of simulation mode is as follows in figure. When model run in PiL simulation mode, it will generate code out of model automatically with the help of embedded coder as per configuration. The toolchain helps to generate code and trigger Lauterbach debugger for Tricore simulation.

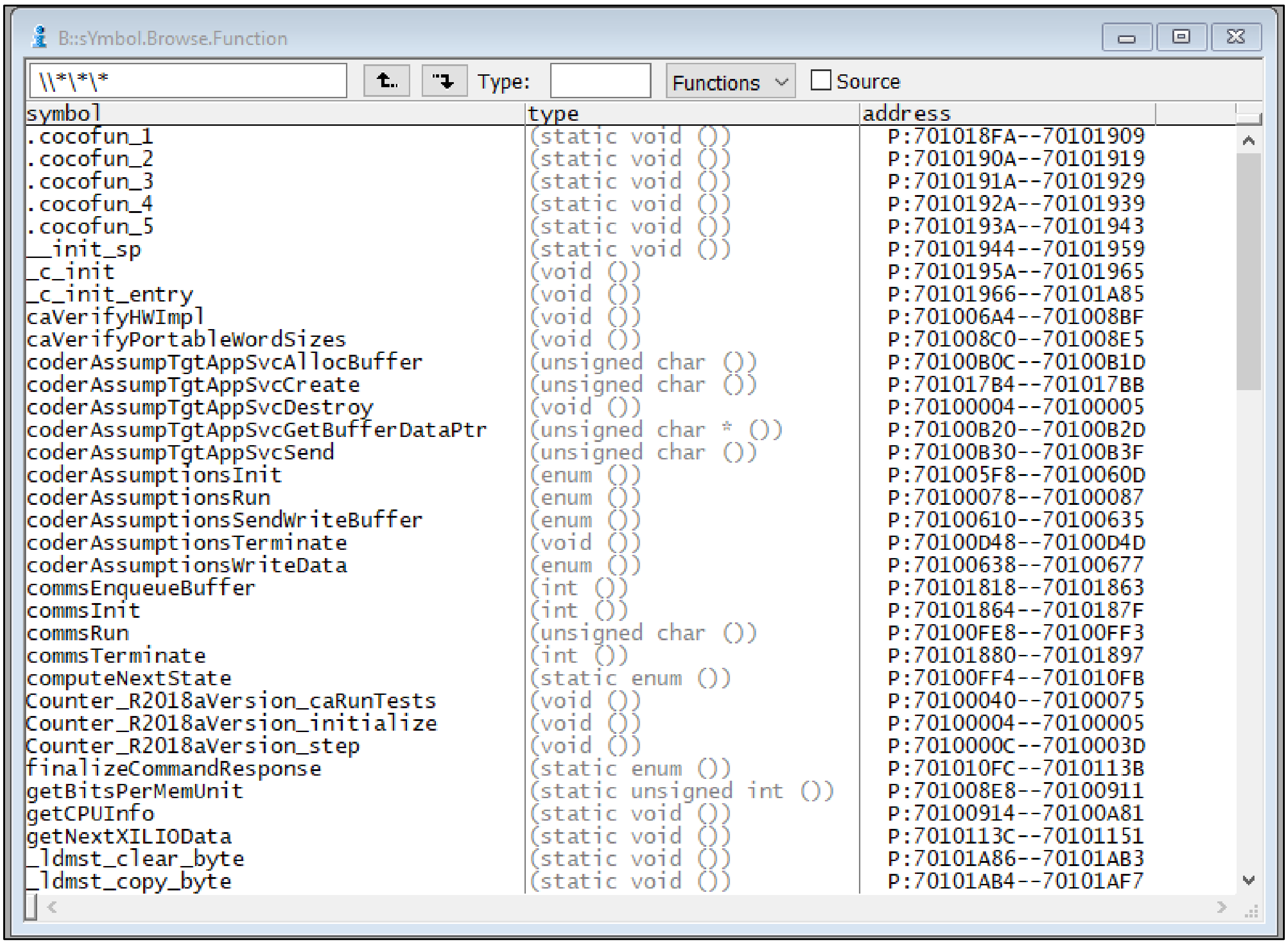
The model run in PiL simulation mode and then it automatically triggers TRACE32 PowerView for TriCore. The .elf file is automatically loaded to actual hardware with help of Trace32 debugger and it is showing under AREA section.



```
loading ELF/DWARF
File 'C:\Thesis_Work\Counter_2018\Counter_R2018aVersion_ert_rtw\pil\Counter_R2018aVersion.elf' (ELF/DWARF2) loaded.
```

Fig 17 Automatically Loading elf File

The confirmation for checking loaded elf file is checking the functions in Symbol view. The below figure shows the function automatically loaded in TRACE32.

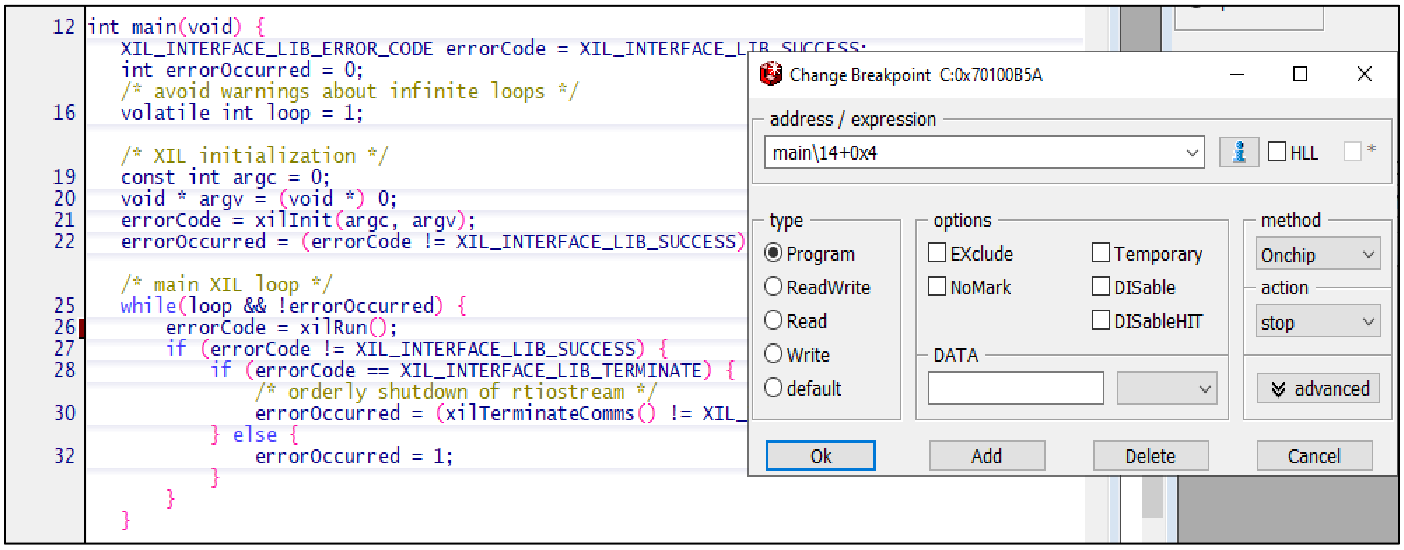


```
symbol                                  type                              address
.cocofun_1                              (static void ())                  P:701018FA--70101909
.cocofun_2                              (static void ())                  P:7010190A--70101919
.cocofun_3                              (static void ())                  P:7010191A--70101929
.cocofun_4                              (static void ())                  P:7010192A--70101939
.cocofun_5                              (static void ())                  P:7010193A--70101943
__init_sp                               (static void ())                  P:70101944--70101959
_c_init                                 (void ())                         P:7010195A--70101965
_c_init_entry                           (void ())                         P:70101966--70101A85
caVerifyHWImpl                          (void ())                         P:701006A4--701008BF
caVerifyPortableWordSizes               (void ())                         P:701008C0--701008E5
coderAssumpTgtAppSvcAllocBuffer         (unsigned char ())                P:70100B0C--70100B1D
coderAssumpTgtAppSvcCreate              (unsigned char ())                P:701017B4--701017BB
coderAssumpTgtAppSvcDestroy             (void ())                         P:70100004--70100005
coderAssumpTgtAppSvcGetBufferDataPtr    (unsigned char * ())              P:70100B20--70100B2D
coderAssumpTgtAppSvcSend                (unsigned char ())                P:70100B30--70100B3F
coderAssumptionsInit                    (enum ())                         P:701005F8--7010060D
coderAssumptionsRun                     (enum ())                         P:70100078--70100087
coderAssumptionsSendWriteBuffer         (enum ())                         P:70100610--70100635
coderAssumptionsTerminate               (void ())                         P:70100D48--70100D4D
coderAssumptionsWriteData               (enum ())                         P:70100638--70100677
commsEnqueueBuffer                      (int ())                          P:70101818--70101863
commsInit                               (int ())                          P:70101864--7010187F
commsRun                                (unsigned char ())                P:70100FE8--70100FF3
commsTerminate                          (int ())                          P:70101880--70101897
computeNextState                        (static enum ())                  P:70100FF4--701010FB
Counter_R2018aVersion_caRunTests        (void ())                         P:70100040--70100075
Counter_R2018aVersion_initialize        (void ())                         P:70100004--70100005
Counter_R2018aVersion_step              (void ())                         P:7010000C--7010003D
finalizeCommandResponse                 (static enum ())                  P:701010FC--7010113B
getBitsPerMemUnit                       (static unsigned int ())          P:701008E8--70100911
getCPUInfo                              (static void ())                  P:70100914--70100A81
getNextXILIOData                        (static void ())                  P:7010113C--70101151
_ldmst_clear_byte                       (static void ())                  P:70101A86--70101AB3
_ldmst_copy_byte                        (static void ())                  P:70101AB4--70101AF7
```

Fig 18 Symbol Functions

➢ *Breakpoints:*

In the context of software development and debugging, a "breakpoint" is a marker or instruction set by a developer within the source code of a program. Breakpoints are used to pause the execution of a program at a specific line or location in the code, allowing the developer to inspect the program's state, variables, and behavior at that point. Breakpoints are a crucial tool for debugging and troubleshooting software.



```
int main(void) {
    XIL_INTERFACE_LIB_ERROR_CODE errorCode = XIL_INTERFACE_LIB_SUCCESS;
    int errorOccurred = 0;
    /* avoid warnings about infinite loops */
    volatile int loop = 1;

    /* XIL initialization */
    const int argc = 0;
    void * argv = (void *) 0;
    errorCode = xilInit(argc, argv);
    errorOccurred = (errorCode != XIL_INTERFACE_LIB_SUCCESS)

    /* main XIL loop */
    while(loop && !errorOccurred) {
        errorCode = xilRun();
        if (errorCode != XIL_INTERFACE_LIB_SUCCESS) {
            if (errorCode == XIL_INTERFACE_LIB_TERMINATE) {
                /* orderly shutdown of rtiostream */
                errorOccurred = (xilTerminateComms() != XIL_
            } else {
                errorOccurred = 1;
            }
        }
    }
```

Change Breakpoint C:0x70100B5A

address / expression: main\14+0x4 HLL

type: [✓ Program] ReadWrite Read Write default

options: EXclude Temporary NoMark DISable DISableHIT

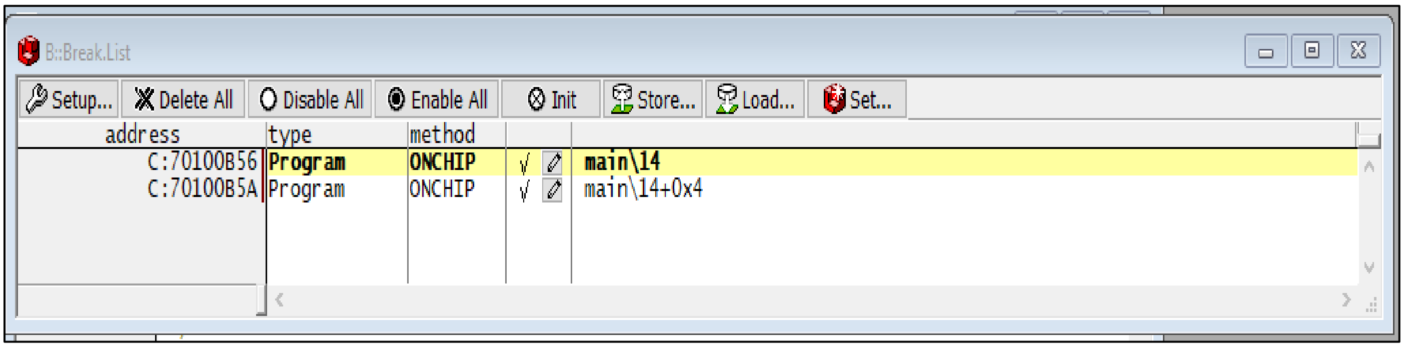
DATA

method: Onchip

action: stop

Fig 19 Setting Breakpoint On Chip

The developer can set breakpoint to functions or code according to requirements or checking main function. When the developer sets a breakpoint at a particular location it is highlighted in Break List window of TRACE32. Due to this highlight, TRACE32 gives message Stopped at breakpoint instead of running.[14]



```
     address  type     method
  C:70100B56  Program  ONCHIP   √   main\14
  C:70100B5A  Program  ONCHIP   √   main\14+0x4
```

Fig 20 Highlighting Breakpoint

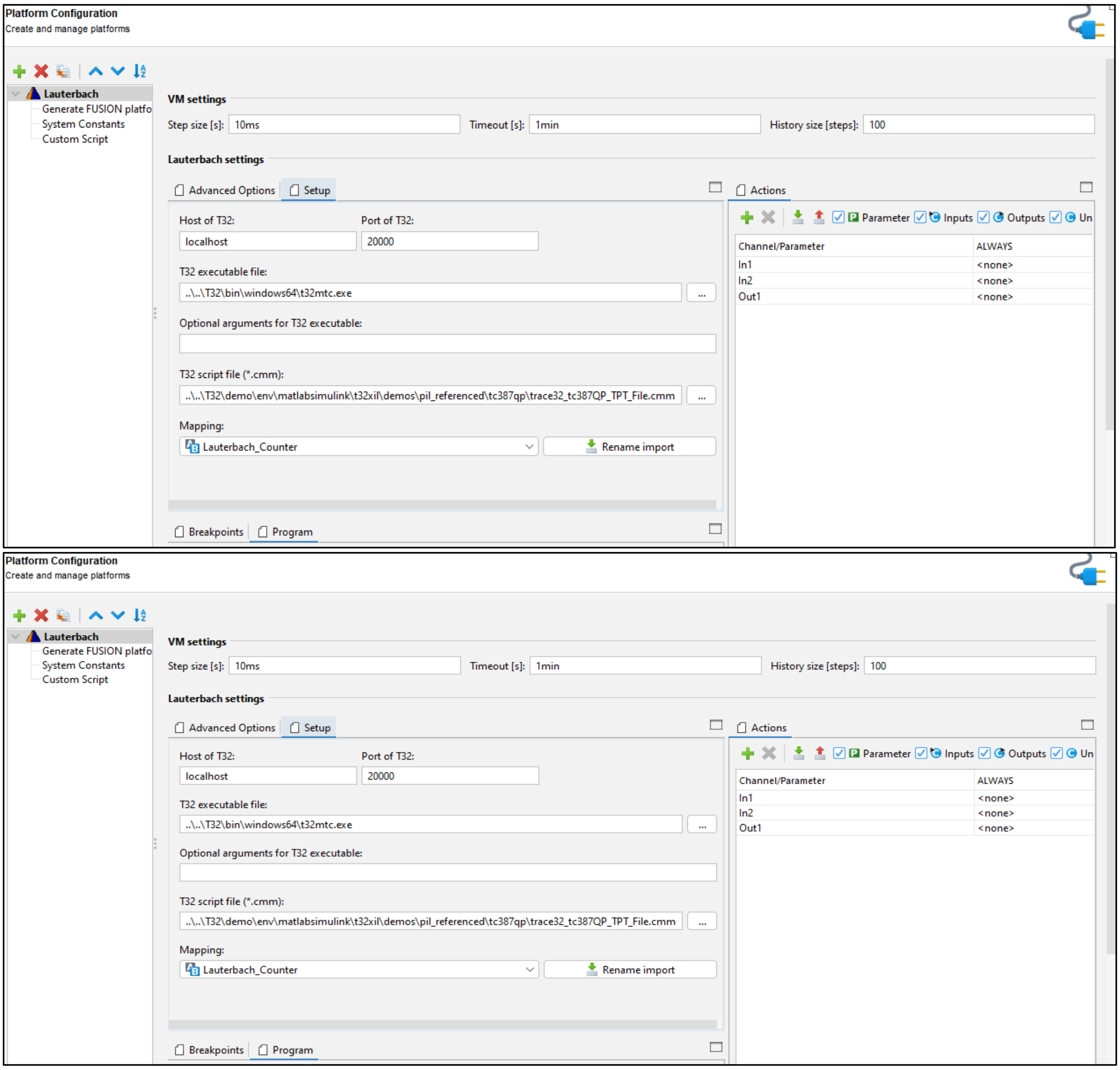
There may be several instances of code generated by a single Simulink block in the source code. Therefore, if you set Breakpoint to C/C++, more than one breakpoint is set. For some Simulink blocks, code is run.

VIII. PIL TESTING WITH TPT TESTING PLATFORM

PikeTec TPT (Time Partition Testing) testing software is designed for software installation testing. It supports various tests like Model-in-the-loop (MiL), Software-in-the-loop (SiL), Processor-in-the-loop (PiL), Hardware-in-the-loop (HiL) Examine. TPT allows testing of ECU software and embedded control systems at various stages of development. It provides facilities for easy and flexible testing, whether simple module testing or complex system testing. TPT also supports security standards such as ISO 26262 and provides test case design/generation, test execution, analysis and reporting capabilities.

Especially for PiL testing, TPT enables embedded software to be tested in the PiL environment. It supports testing activities such as test case design, test execution, and PiL environment-specific analysis.

The figure below illustrates the platform configuration of Lauterbach in the TPT software. The TPT environment calls the executable file of TRACE32 to execute test cases on real hardware. The crucial configuration details include specifying the path of the TRACE32 executable file in the T32 executable file section. The path of .exe file should be provided in T32 executable file portion. The communication between TPT and TRACE32 relies on a host and port configuration. In this integration, the local host is used, and the default port is set to 20000. This standard port serves as the communication bridge between TPT and TRACE32, ensuring a robust and reliable connection. [23]


Platform Configuration
Create and manage platforms

Lauterbach
- Generate FUSION platfo
- System Constants
- Custom Script

VM settings
Step size [s]: 10ms Timeout [s]: 1min History size [steps]: 100

Lauterbach settings
Advanced Options | Setup

Host of T32: localhost Port of T32: 20000

T32 executable file: ..\..\T32\bin\windows64\t32mtc.exe

Optional arguments for T32 executable:

T32 script file (*.cmm): ..\..\T32\demo\env\matlabsimulink\t32xil\demos\pil_referenced\tc387qp\trace32_tc387QP_TPT_File.cmm

Mapping: Lauterbach_Counter Rename import

Breakpoints | Program

Actions
Parameter Inputs Outputs Un

Channel/Parameter	ALWAYS
In1	<none>
In2	<none>
Out1	<none>

Platform Configuration
Create and manage platforms

Lauterbach
- Generate FUSION platfo
- System Constants
- Custom Script

VM settings
Step size [s]: 10ms Timeout [s]: 1min History size [steps]: 100

Lauterbach settings
Advanced Options | Setup

Host of T32: localhost Port of T32: 20000

T32 executable file: ..\..\T32\bin\windows64\t32mtc.exe

Optional arguments for T32 executable:

T32 script file (*.cmm): ..\..\T32\demo\env\matlabsimulink\t32xil\demos\pil_referenced\tc387qp\trace32_tc387QP_TPT_File.cmm

Mapping: Lauterbach_Counter Rename import

Breakpoints | Program

Actions
Parameter Inputs Outputs Un

Channel/Parameter ALWAYS
In1 <none>
In2 <none>
Out1 <none>

Fig 21 Platform Configuration

➢ *Advantages of Integrating TRACE32 and TPT:*

- *Real Hardware Testing:*
  The integration allows for the execution of test cases directly on real hardware, providing a more accurate representation of the system's behavior.

- *Comprehensive Scenario Testing:*
  Users can define and execute complex testing scenarios by leveraging the features of both TPT and TRACE32.

- *Efficient Debugging:*
  The integration facilitates efficient debugging through the TRACE32 environment, enhancing the identification and resolution of issues during testing. [23]

IX. CONCLUSION

In conclusion, the development of a comprehensive and robust Toolchain for Processor-in-the-Loop (PiL) testing, integrating MATLAB/Simulink, Lauterbach, and various other essential components, is an imperative achievement. This master’s thesis has addressed a critical need within the domain of embedded system testing, and its implications are significant for both academia and industry. It has been meticulously designed, implemented, and tested to cater to the specific requirements of PiL testing, a crucial step in ensuring the safety and reliability of software in safety-critical systems. The incorporation of MATLAB scripts, Simulink models, and C language laments into a cohesive framework has the potential to greatly enhance the efficiency and effectiveness of PiL testing.

The developed toolchain, which includes the generation of ELF files for the hardware and the flashing of these files into a real Electronic Control Unit (ECU), has proved to be vital for conducting realistic and reliable testing scenarios. The PiL approach enables the simulation of the embedded system in a realistic environment, allowing for early detection and resolution of potential issues during the development process. By integrating Simulink with the Lauterbach debugger, the thesis has demonstrated a powerful method of monitoring and debugging the embedded system in real-time. The debugger’s capabilities, such as real-time trace and breakpoints, have provided valuable insights into the system’s behavior, aiding in identifying and analyzing potential errors or performance bottlenecks.

Furthermore, the versatility of this toolchain is evident in its potential for integration with other testing software, such as TPT from Piketec. This adaptability extends the toolchain's utility beyond the immediate scope of this master’s thesis, making it an asset for a broad range of testing scenarios and environments. The toolchain's exemplary compatibility and collaborative capabilities underscore its potential to significantly elevate testing and validation processes within organizations.

REFERENCES

[1]. Systems Engineering: Principles and Practice" by Alexander Kossiakoff, William N. Sweet, Sam Seymour, and Steven M. Biemer https://books.google.de/books?id=MRZoj0yAm9oC&printsec=frontcover&redir_esc=y#v=onepage&q&f=false
[2]. "Model-Based Engineering for Complex Electronic Systems" by Peter Wilson
[3]. JTAG Interface Training JTAG Interface (https://www.lauterbach.com/)
[4]. BTC Embedded System Processor-in-the-Loop (PIL) - Testing (https://www.btc-embedded.com/) , 2023
[5]. Lauterbach Product and Expertise TRACE32 Debugger | Lauterbach 2023
[6]. System Engineering What is Systems Engineering? https://www.incose.org/about-systems-engineering/what-is-systems-engineering
[7]. Modelling Integrated Product Development Processes ProcessModel– (gfse.de), 1999
[8]. ISO 26262 ISO 26262-1:2011 - Road vehicles — Functional safety — Part 1: Vocabulary
[9]. MBD according to ISO 26262 ISO 26262 Support in MATLAB and Simulink - Automotive Standards - MATLAB & Simulink (https://de.mathworks.com/)
[10]. ASIL Level What is ASIL (Automotive Safety Integrity Level)? – Overview | Synopsys Automotive
[11]. "A Real-Time Testing System Based on the Model-in-the-Loop and Processor-in-the-Loop Techniques" by Zeyad T. Almashhadany and Ahmad S
[12]. PiL Testing https://www.plexim.com/sites/default/files/flyers/flyer_pil_a4.pdf https://de.mathworks.com/solutions/automotive/standards/iso-26262.html
[13]. Integration of Pil Simulation https://repo.lauterbach.com/design_of_a_flexible_integration_interface_for_pil_tests.pdf (lauterbach.com) T. Erkkinen and M. Conrad. Verification, Validation, and Test with Model-Based Design. SAE Technical Paper. 2008.
[14]. Integration of TRACE32 for Simulink https://www2.lauterbach.com/pdf/int_simulink.pdf
[15]. https://de.mathworks.com/help/ecoder/ug/create-pil-target-connectivity-configuration.html
[16]. C language toolchain Embedded System Build Process https://microcontrollerslab.com/embedded-systems-build-process-using-gnu-toolchain/
[17]. Assembly Language for x86 Processors" by Kip R. Irvine https://broman.dev/download/Assembly%20Language%20for%20x86%20Processors%207th%20Edition.pdf
[18]. PiL Target Connectivity https://de.mathworks.com/help/ecoder/ug/create-pil-target-connectivity-configuration.html
[19]. Tasking Compiler https://www.infineon.com/cms/en/tools/aurix-tools/Compilers/TASKING/
[20]. TRACE32 Installation Guide https://www2.lauterbach.com/pdf/installation.pdf
[21]. TRACE Trace Tutorial (https://www.lauterbach.com/)
[22]. Model Based Testing for real time embedded system automotive https://d-nb.info/993865100/34
[23]. TPT Testing https://piketec.com/tpt/