LAUTERBACH
DEVELOPMENT TOOLS

Integration for Simulink

MANUAL

Integration for Simulink

Integration for Simulink

Version 11-Jan-2026

History

07-Jan-2026 Add section **"Use with Polyspace Test"**, page 35.

07-Jan-2026 Add section **"Settings File"**, page 11.

07-Jan-2026 Update list of predefined custom toolchains.

28-Mar-2024 Add QT version conflicts to **"Troubleshooting"**, page 36.

23-Jan-2024 Add support for Linux.

23-Jan-2024 Installer executable is replaced by MATLAB toolbox.

Introduction

This document describes how to install, configure, and use the TRACE32 Integration for Simulink.

It is now common to perform simulation and verification of designs before committing to hardware. Such solution can be achieved using tools like MATLAB and Simulink especially for the control engineering market. It can save a lot of time and effort if the control loop can be tested for the effects of many variables before finalizing the design. Simulink integrates the capability of generating code automatically from a model; this feature can be used to check that the program behaves the same way on the control hardware as in the simulation.

After creating the control algorithms and testing them with Simulink, the corresponding program code for the processor of the control hardware can be generated from the control blocks using the Embedded Coder. Using a TRACE32 debugger, the generated code can be loaded into the control hardware and tested in-situ. Polyspace Test can be utilized to execute tests directly on the target platform, virtual target or TRACE32 instruction set simulator.

Intended Audience

The users of the TRACE32 Integration for Simulink have to be familiar with TRACE32, MATLAB, Simulink, and Polyspace Test.

Installation

System Requirements

- Microsoft Windows or Linux
- TRACE32 installation **from 02/2024**
- MATLAB release **R2014b or newer [rtiostream API]**
- MATLAB release **R2021a or newer [DebugIOTool Debugger Abstraction Interface]**
- Additional MathWorks products:
  - MATLAB
  - Simulink or Polyspace Test
  - Embedded Coder
  - MATLAB Coder
  - Simulink Coder
  - Simulink Coder **[Code Coverage Measurement Only]**
- MATLAB-supported compiler for MEX files
- **TRACE32-supported cross compiler**
- The command line tool *t32cast*. For more information, see **"Application Note for t32cast"** (app_t32cast.pdf).

License Requirements

Using the TRACE32 Integration for Simulink requires additional licenses. Please contact your local sales office or **sales@lauterbach.com** for additional details.

Installing the Integration

To install the TRACE32 Integration for Simulink:

1. Start MATLAB.
2. Drag and drop the toolbox `~~/demo/env/matlabsimulink/t32xil.mltbx` into the MATLAB command window.

Updating the Integration

To update the TRACE32 Integration for Simulink:

1. Navigate to the subfolder `~~/demo/env/matlabsimulink` of your TRACE32 installation directory.
2. Place the newer version of the MATLAB toolbox into this subfolder.
3. Start MATLAB.
4. Drag and drop the toolbox `~~/demo/env/matlabsimulink/t32xil.mltbx` into the MATLAB command window to update the TRACE32 Integration for Simulink.

Deinstallation

To uninstall the TRACE32 Integration for Simulink:

1. Start MATLAB
2. Select the MATLAB add-on “t32xil” for deinstallation.

Custom Toolchains

A cross-compiler is required to build executables for the target hardware. New cross-compilers can be integrated into MATLAB as custom toolchains. TRACE32 XIL comes with a set of custom build toolchains, but it also supports user-defined ones.

The custom toolchains are included:

Toolchain	Toolchain Configuration
**TRACE32 XIL v1.0	gmake makefile**	t32xil_tc.m
**TRACE32 XIL GCC arm-none-eabi Cortex-R	gmake makefile**	t32xil_tc_gcc_arm_none_eabi_cortex_r.m
**TRACE32 XIL GCC arm-none-eabi Cortex-M	gmake makefile**	t32xil_tc_gcc_arm_none_eabi_cortex_m.m
**TRACE32 XIL TASKING VX-toolset for Tri-Core	gmake makefile**	t32xil_tc_tasking_ctc.mat
**TRACE32 XIL HighTec TriCore Development Platform	gmake makefile**	t32xil_tc_hightec_tricore_gcc.mat
**TRACE32 XIL Renesas RL78 Compiler CC-RL	gmake makefile**	t32xil_tc_renesas_rl78_ccrl.mat
**TRACE32 XIL Diab Compiler Power Architecture	gmake makefile**	t32xil_tc_ppc_diab.mat

These toolchain, together with a set of utility scripts, can be found in the directory *toolchain*:

Utility	Description
RegisterToolchains.m	This script creates and registers ToolchainInfo object for all custom toolchains located in the folder "*toolchain*". m-files containing custom toolchains need to start with "*t32xil_tc*".
rtwTargetInfo.m	Custom toolchains can be registered after creation of a ToolchainInfo object. To register an m-file with name "*rtwTargetInfo*" has to be on the MATLAB path.

To register your custom toolchain please create a new toolchain definition file and extend the provided utility scripts. Additional information can be found **here**.

Board Descriptions

TRACE32 XIL uses the MATLAB Target Framework to register the properties of the target hardware on which the PIL test should be executed. Users can add additional board description to extend the hardware support to new platforms.

These board descriptions are included:

Board	Processor
TRACE32 DebugIO for Arm	ARM Compatible-ARM Cortex
TRACE32 DebugIO for TriCore	Infineon-TriCore

The utility script with these board descriptions can be found in the directory *debugio*:

Utility	Description
t32boards.m	Custom boards that use the TRACE32 DebugIO interface can be registered here.

To register your board please create the required hardware information classes inside MATLAB and extend the provided utility script. Additional information can be found **here**.

Select Connectivity API

TRACE32 XIL supports both the MATLAB **rtiostream API** and **DebugIOTool debugger abstraction interface** for PIL target connectivity. After installation of TRACE32 XIL users can switch freely between both interfaces.

To select the rtiostream API:

1. Choose one of the files *sl_customization_custom_toolchain.p* or *sl_customization_template_makefile.p* and rename it to ***sl_customization.p***.
2. Restart MATLAB
3. Execute the MATLAB command:

```
sl_refresh_customizations
```

To select the DebugIOTool debugger abstraction interface:

1. Choose the file *sl_customization_debugio.p* and rename it to ***sl_customization.p***.
2. Restart MATLAB
3. Execute the MATLAB command:

```
sl_refresh_customizations
```

Settings File

The behavior of TRACE32 XIL can be configured by adding the settings file *trace32_settings.m* to the MATLAB search path. The settings file needs to return a struct that stores the configuration settings as field values:

```
function cfg = trace32_settings()
  cfg = struct;

  % T32_x: If given without full path, the current path of this file is
  %           searched for these files
  cfg.T32_Executable          = '<file>';
  cfg.T32_Startupscript       = '<script>';
  cfg.T32_Config              = '<config-file>';
  % Target_Binary_Extension: e.g. .elf, .coff, .ecoff, .xcoff, .out, ...
  cfg.Target_Binary_Extension = '<extension>';
  % Timeout_x: Timeouts in seconds
  cfg.Timeout_Init            = <number>;
  cfg.Timeout_Transfer        = <number>;
  % Specify timer object for code execution profiling.
  % If empty, execution profiling is disabled.
  cfg.Hardware_Timer          = '<name>';
  % Specify data for stack profiling.
  % If empty, stack profiling is disabled.
  cfg.Stack_Profiler          = '<mode>';
  cfg.PracticeHooks           = SetPracticeCallbacks();
end

function hooks = SetPracticeCallbacks()
  hooks = struct;

  % Triggered once immediately before the simulation is started
  hooks.PreInit  = {'<path-pre>', '<interface-pre>'};
  % Triggered once after the simulation has terminated
  hooks.PostTerm = {'<path-post>', '<interface-post>'};
end
```

Concrete sample files for various hardware platforms are available in the directory *demos*. Depending on the selected Connectivity API not all settings are mandatory.

rtiostream API

TRACE32 XIL is a fully integrated plug-in for Simulink that builds on the **MATLAB rtiostream API** for PIL Target Connectivity to run processor-in-the-loop simulations with Simulink. Generated code can be easily cross-compiled, deployed, executed and debugged on custom targets. Execution and stack profiling via code instrumentation and execution of PRACTICE callbacks is supported during simulation.To quickly switch between elements of your model and the corresponding sections of C/C++ and object code, you can use the navigation features of the TRACE32 XIL plug-in.

The key features are:

- TRACE32 Remote API for generic target support
- Compatible with TRACE32 Instruction Set Simulator for easy testing on virtual targets
- Support for built-in target connectivity capabilities like code execution profiling and code coverage
- Use TRACE32 features to diagnose errors on the target platform
- Bidirectional navigation between model and generated code

Demo Project

Lauterbach provides a demo project in the folder *demos*. We recommend that you familiarize yourself with the TRACE32 XIL plug-in for Simulink by taking your first steps in our demo project.

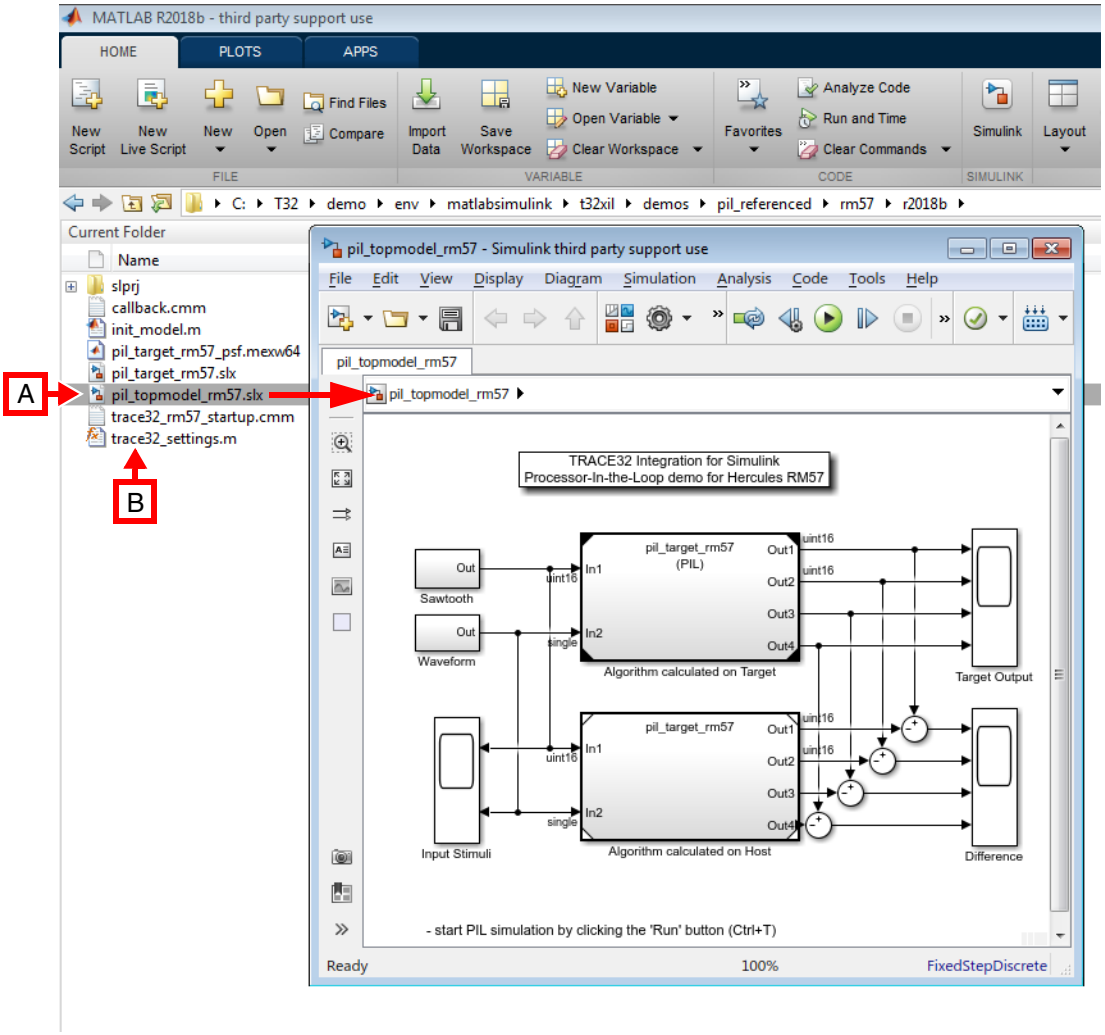
To run the demo project:

1. Double-click the *.exe to install the demo project.
2. Close all open TRACE32 instances.
3. Before you start TRACE32 for the first time from within Simulink, enable the port for code-to-model navigation in the TRACE32 configuration file by taking these steps:
   - Navigate to *simulinktemplate.config.t32*
   - Delete the two comment signs (**;**) from the block shown below.

```
; --------------------------(4)-------------------------------
SIMULINK=NETASSIST                  <-- delete comment sign here
; Port for code-to-model navigation
PORT=20000                          <-- delete comment sign here
```

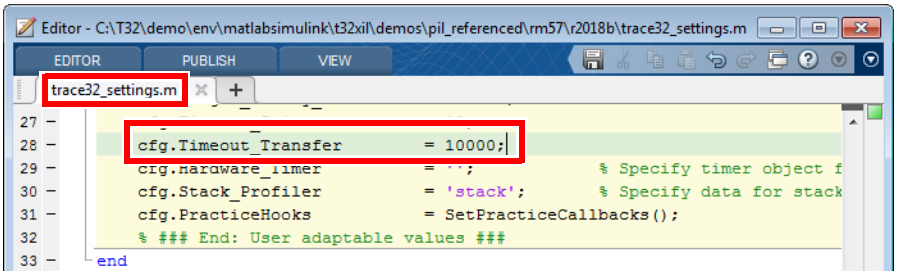
4. Start the demo project in Simulink by double-clicking this file *demos/pil_referenced/rm57/r2018b/init_model.m*

5. Once Simulink has started, double-click the file pil_topmodel_*<target_board>*.slx [**A**]:

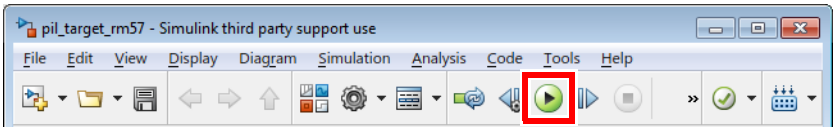


6. Open the file trace32_settings.m [**B**] and increase the communication timeout value of `cfg.Timeout_Transfer`.

This is to prevent error messages in Simulink while you are debugging an application in TRACE32. The default timeout value is 10 seconds. Here, we have set the timeout value to 10000 seconds.



7. In Simulink, click the **Run** button to compile the Simulink model.



- Simulink establishes a connection to the required TRACE32 instance.
- TRACE32 displays the state “stopped” in the state line. TRACE32 stays in the state “running” if you have set the simulation stop time to **`inf`** in Simulink. To end the simulation in Simulink and the application execution in TRACE32, click **Stop** *in Simulink*.

Additionally in our demo project, the following happens after the connection has been established:

- The PRACTICE script trace32_<*target_board*>_startup.cmm is executed in TRACE32. For convenient access to the script file, double-click it in Simulink’s **Current Folder** window pane.
- A **List.Mix** window opens in the main window of TRACE32 PowerView because the **List.Mix** command is included in the PRACTICE script trace32_<*target_board*>_startup.cmm. The **List.Mix** window displays the source code Simulink has generated based on your model.

Next:

- **Functional Overview**

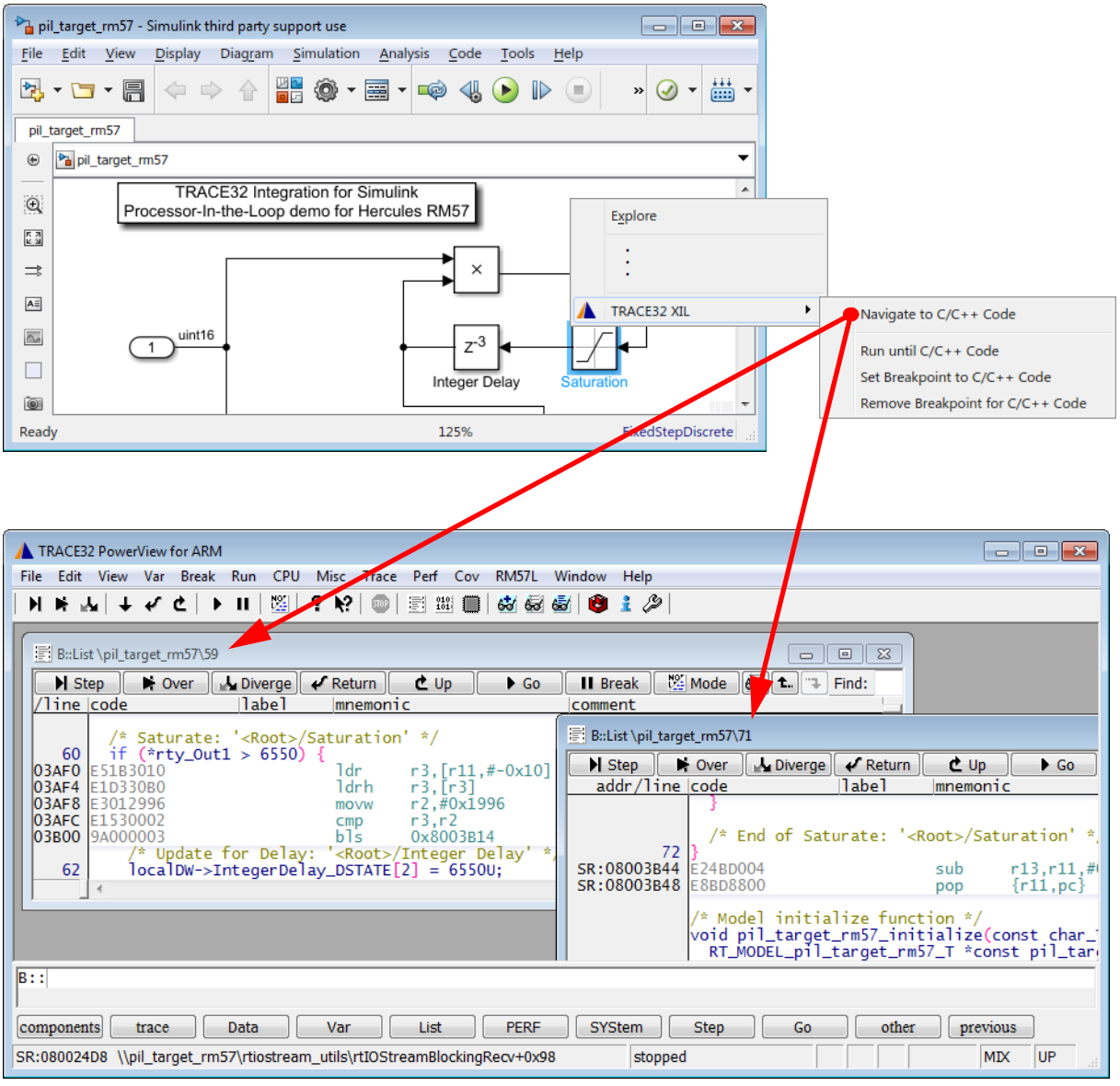
In this section:

You can navigate from a Simulink block to TRACE32 to view the code generated from this block.

- Right-click a Simulink block, and then select **TRACE32 XIL** > **Navigate to C++ Code**.

  A **List.auto** window opens in TRACE32 PowerView, displaying the code generated from the selected Simulink block.

The code generated from a single Simulink block may be found at more than one location within the source code. For this reason more than one **List.auto** window opens if **Navigate to C++ Code** is executed for some Simulink blocks.



You can navigate from the source code displayed in a **List.auto** window of TRACE32 PowerView back to the corresponding Simulink block.

This navigation is made possible by navigation tags within the source code. A navigation tag is a special comment containing the name of the Simulink block to which the generated code belongs. The required navigation tags are created and updated by Simulink during the code generation phase.

Prerequisite(s):

Add this command to your PRACTICE start-up script (*.cmm) if code-to-model navigation is frequently used.

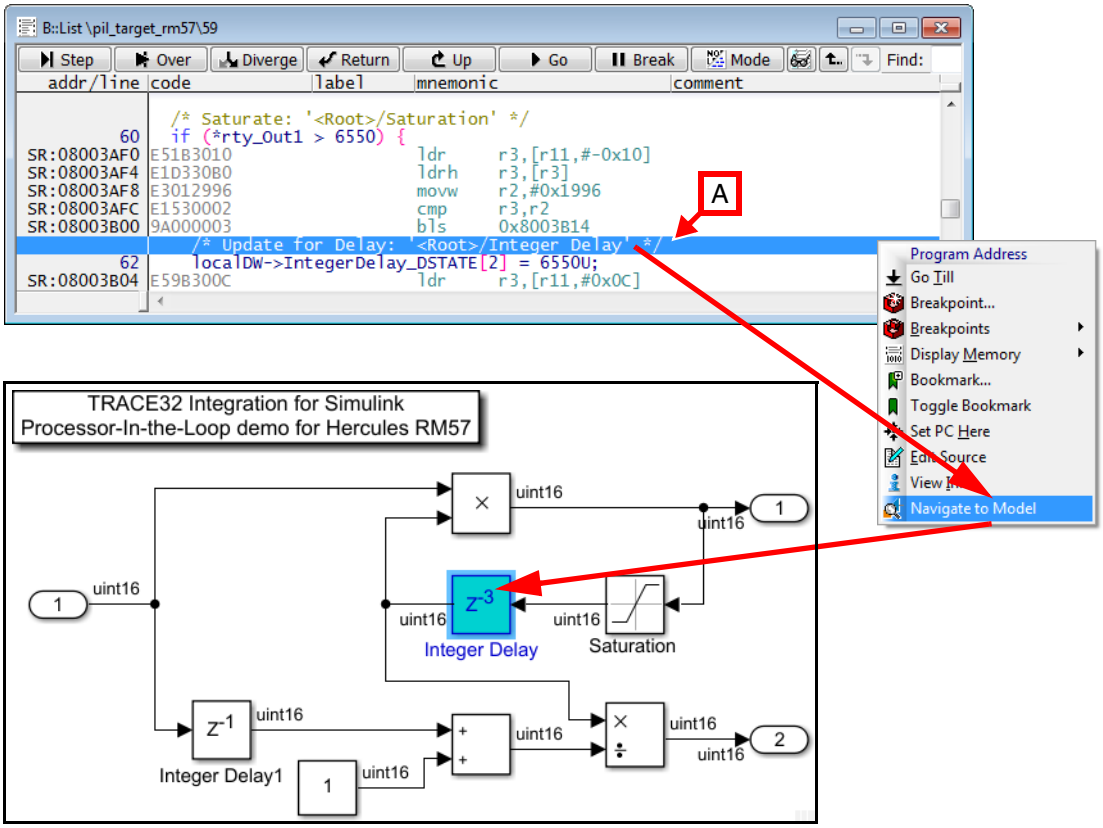
```
sYmbol.ECA.LOADALL /SkipErrors
```

In our demo project, the PRACTICE start-up script is called trace32_*<target_board>*_startup.cmm. For convenient access to the script file, double-click it in Simulink's Current Folder window pane.

To navigate from TRACE32 to the respective block of the Simulink model:

- In a TRACE32 **List.auto** window, right-click a comment line containing a navigation tag [**A**], and then select **Navigate to Model** from the popup menu.

  If the ECA data made available to TRACE32 is out of date, then TRACE32 disables the navigation back to Simulink. That is, the **Navigation to Model** option is hidden in the popup menu.



The Simulink window is brought to the front, and the corresponding Simulink block blinks briefly.

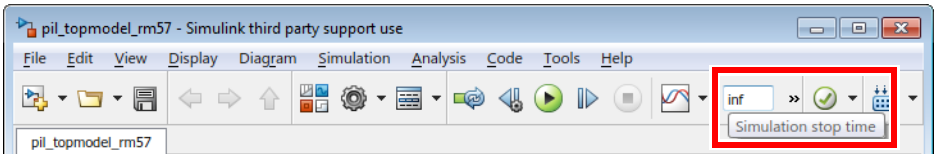
From within Simulink, you can instruct TRACE32 PowerView to execute the software up to the code which was generated from the corresponding Simulink block. This is useful for debugging an individual time step in the Simulink model while the application is running in TRACE32.

On the TRACE32 PowerView side, this is done by automatically setting temporary breakpoint(s) and executing a **Go**. Temporary breakpoints are automatically deleted once the temporary breakpoint has been reached.

To run until C/C++ Code using temporary breakpoints:

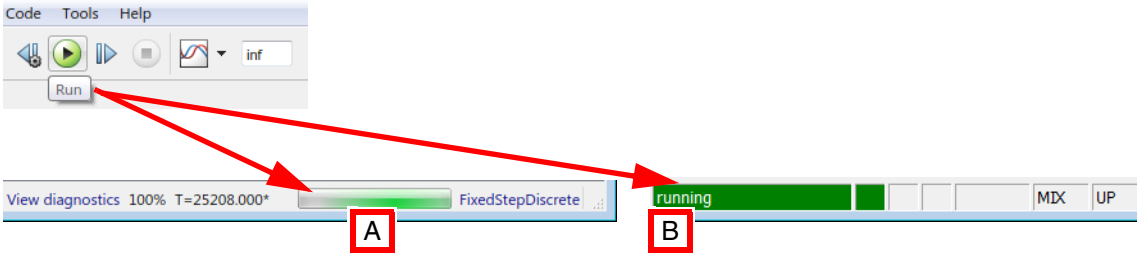
1. In Simulink, enter the simulation stop time.

   In case of our demo project, enter **`inf`** as the simulation stop time in the window of the pil_topmodel_*<target_board>*.



2. In Simulink, click **Run**.

   In case of our demo project, click **Run** in the window of the pil_topmodel_*<target_board>*.



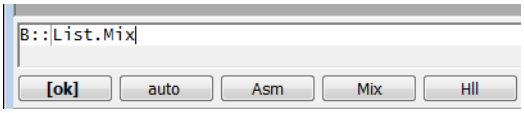
   - The model is now being executed in Simulink [**A**].
   - The application code is now running in TRACE32, see state line at the bottom of the TRACE32 PowerView window [**B**].

3. In Simulink, right-click a block, and then select **TRACE32 XIL** > **Run until C/C++ Code** from the popup menu.

   - TRACE32 stops the application execution at the *temporary breakpoint* you have just set from within Simulink. The TRACE32 state line displays **stopped at breakpoint**.

   - The GUI controls (popup menu, buttons, etc.) in Simulink are deactivated.
   - If the code generated from the Simulink block is located at different source code positions, software execution will stop in TRACE32 at the first temporary breakpoint which is hit.
   - If the code generated from the block cannot be reached at all, the software execution won't stop, i.e. the target stays in the state **running**.

4. To view the current location of the PC (program counter), open a **List.Mix** window by typing **`List.Mix`** at the TRACE32 command line.



5. To restore the full GUI control to Simulink, you need to stop the simulation as described in **"Stop the Simulation"**, page 23.

Variation:

- For the **List.Mix** window to open automatically in TRACE32, include the **List.Mix** command in your PRACTICE start-up script (*.cmm); see **"Demo Project"**, page 12.

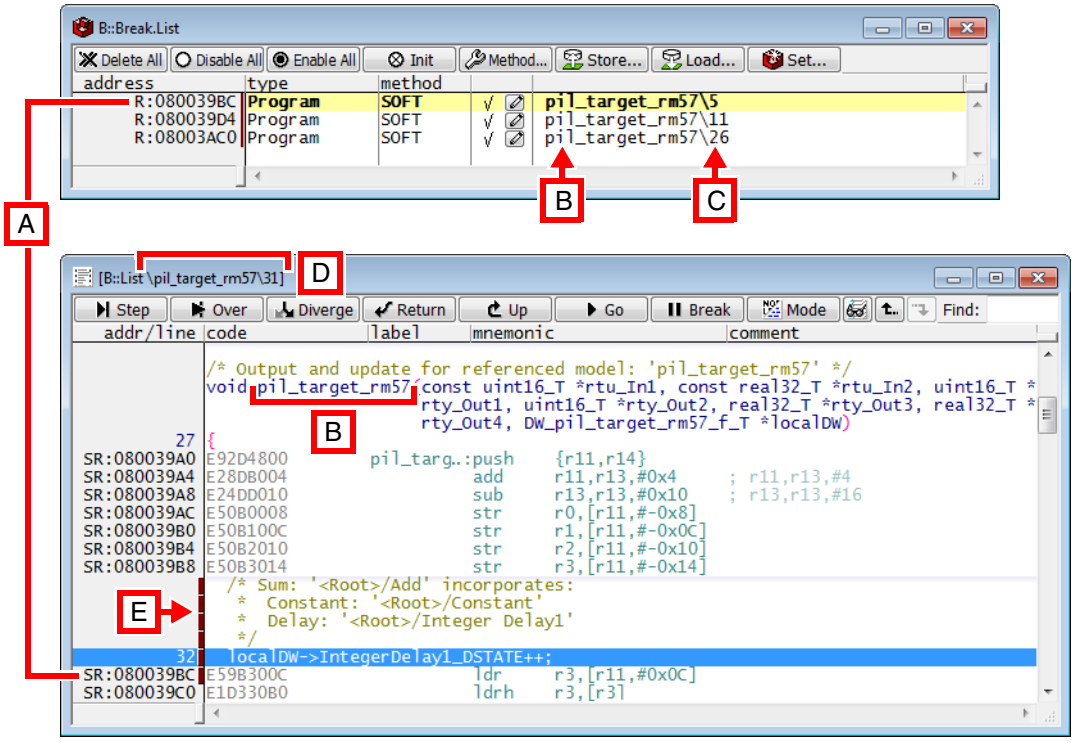
From within Simulink, you can instruct TRACE32 PowerView to set breakpoints at the code sections which were generated from the corresponding Simulink block. You can view the successful execution of this task by opening a **Break.List** window in TRACE32.

Breakpoints set with the function **Set Breakpoint to C/C++ Code** are *not deleted automatically* by TRACE32 once such a breakpoint has been reached in TRACE32.

NOTE:

- Clicking **Run** in Simulink deletes all existing breakpoints in TRACE32 because the entire code is re-generated. Therefore, you should set breakpoints from within Simulink while the simulation is running in Simulink.
- Alternatively, you can specify breakpoints in your PRACTICE start-up script (*.cmm). In this case you need to set the breakpoints to symbols rather than addresses. These breakpoints are re-set when the model is re-built.
  Example: `Break.Set pil_target_rm57 /Program`

By double-clicking a breakpoint in the **Break.List** window, you can view code and breakpoint in a **List.auto** window.



A Breakpoint address.

B Function name (which is the same as the \<*module_name*> in our demo project)

C Number of lines between function name and breakpoint (line number offset).

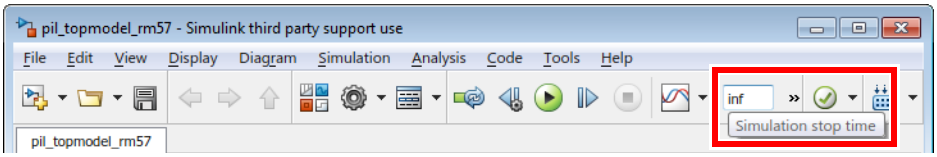
D \\<*module_name*>\\<*absolute_line_number*>. Please note the leading backslash at the module name.

E Red bars visualize breakpoints. Here a breakpoint in line 32.

To set a breakpoint to C/C++ code:

1. In Simulink, enter the simulation stop time.

   In case of our demo project, enter **`inf`** as the simulation stop time in the window of the pil_topmodel_<*target_board*>.

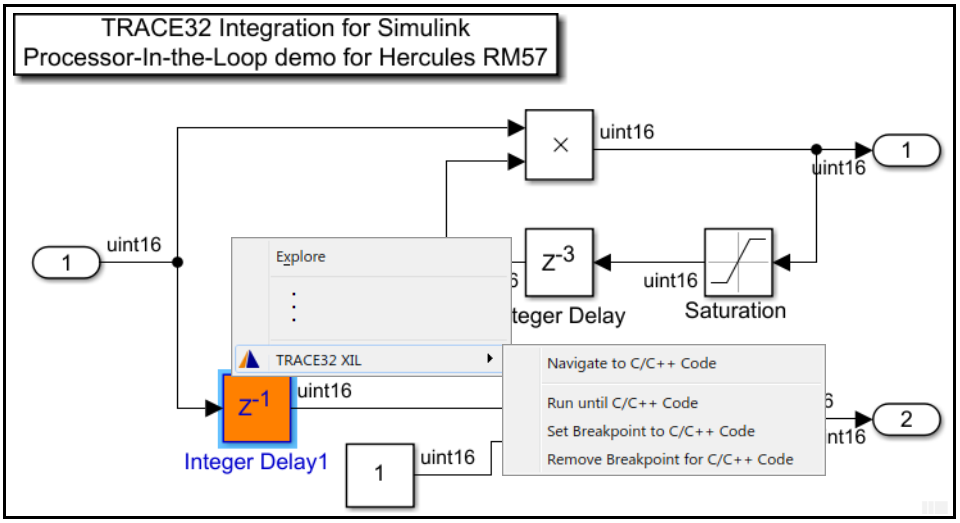


2. In Simulink, click the **Run** button.

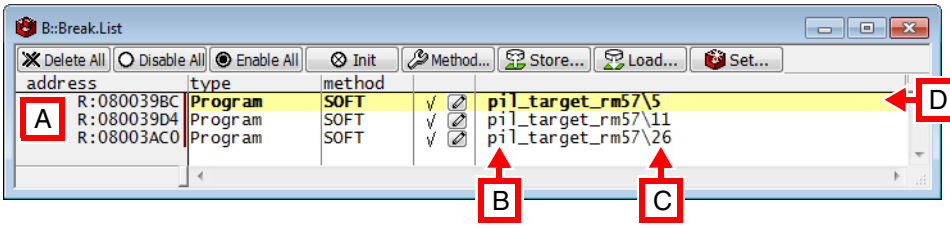
   In case of our demo project, click the **Run** button in the window of the pil_topmodel_<*target_board*>.

3. In Simulink, right-click a block, and then select **TRACE32 XIL** > **Set Breakpoint to C/C++ Code** from the popup menu.

   The block to which you have set a breakpoint is highlighted in orange.



4. In TRACE32, open a **Break.List** window to view the result.



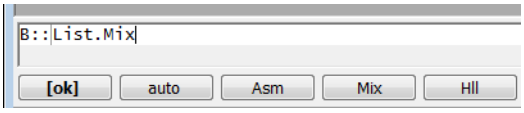
A Breakpoint addresses. **B** Function names.

C Number of lines between function name and breakpoint (line number offset).

D Any breakpoint that stops the application execution in TRACE32 is highlighted in yellow in the **Break.List** window.

- The code generated from a single Simulink block may be found at more than one location within the source code. For this reason more than one breakpoint is set if **Set Breakpoint to C/C++ Code** is executed for some Simulink blocks [**A**].
- The GUI controls (popup menu, buttons, etc.) in Simulink are deactivated.

5. To view the current location of the PC (program counter), open a **List.Mix** window by typing **`List.Mix`** at the TRACE32 command line.



6. To continue in TRACE32, click **Go**.
7. To restore the full GUI control to Simulink, you need to stop the simulation as described in **"Stop the Simulation"**, page 23.

Variation:

- For the **List.Mix** window to open automatically in TRACE32, include the **List.Mix** command in your PRACTICE start-up script (*.cmm); see **"Demo Project"**, page 12.
- The same tip applies to the **Break.List** window.

Remove Breakpoint for C/C++ Code

Orange Simulink blocks indicate that breakpoints exist for the associated source code in TRACE32. If you no longer need a breakpoint in the source code for particular orange block, you can delete this breakpoint from within Simulink.

To delete breakpoints in the source code of TRACE32 from within Simulink:

- Right-click an orange Simulink block, and then select **TRACE32 XIL** > **Remove Breakpoint for C/C++ Code** from the popup menu.

  Result:

  - In Simulink, a previously orange block then turns white again.
  - In TRACE32, the deleted breakpoints are removed from the **Break.List** window.

Stop the Simulation

The GUI controls (popup menu, buttons, etc.) in Simulink are deactivated if the following two conditions apply:

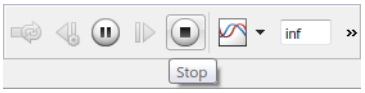
- The simulation is running in Simulink.
- The application execution in TRACE32 is stopped by a breakpoint or a temporary breakpoint.

To restore the full GUI control to Simulink, you need to stop the simulation as described in the steps below.

To stop the simulation:

1. Open the **Break.List** window to check if any breakpoints are still enabled.
   - Enabled breakpoints: Their font color is black.
   - Disabled breakpoints: Their font color is gray.
2. Disable all breakpoints by clicking **Disable All**.
3. Click **Go** in any **List.*** window or click **Go** on the TRACE32 toolbar.

   Since TRACE32 is now in the state **running** again, the GUI controls in Simulink are also re-activated.
4. In Simulink, click **Stop**.



The stop of the simulation in Simulink also causes TRACE32 to stop the application execution. The TRACE32 state line displays **stopped**.

TRACE32 XIL has built-in support for both template makefiles used for building legacy models and custom toolchains. Custom toolchains require MATLAB R2015b and newer.

To switch between both configurations the plug-in registration of TRACE32 XIL has to be altered. The following steps are required:

1. Select the plug-in registration:

Build Method	Plug-in Registration
Template Makefile	sl_customization_template_makefile.p
Custom Toolchain	sl_customization_custom_toolchain.p

2. Delete or rename the current plug-in registration *sl_customization.p*.
3. Change the filename of the selected plug-in registration to *sl_customization.p*.

NOTE:	Starting with MATLAB R2018b TRACE32 XIL is configured to use custom toolchains during the build process. However, template makefiles are used by default for older versions.

TRACE32 XIL is automatically selected for the execution of PIL simulations when top level model and its sub and referenced modules are configured properly.

To configure a model for use, the following steps are necessary:

1. Open the dialog **Model Configuration Parameters**.
2. Configure the settings in the dialog **Hardware Implementation** for your target.
3. Select a supported system target file in the dialog **Code Generation**:

Model Type	System Target File
AUTOSAR	autosar.tlc
Embedded Coder	ert.tlc
	trace32_target_ert.tlc

4. Configure the build process in the dialog **Code Generatio**n.
   - Select a supported template makefile and a make command when building using template makefile:

Architecture	Template Makefiles
ARM	trace32_arm_tmf.tmf
C2000	trace32_c2000_tmf.tmf
Power Architecture	trace32_mpc_tmf.tmf
RH850	trace32_rh850_tmf.tmf
TriCore	trace32_tc_tmf.tmf
V800	trace32_v850_tmf.tmf

- Select a supported toolchain when building using custom toolchains:

Toolchain	Toolchain Configuration
**TRACE32 XIL v1.0	gmake makefile**	t32xil_tc.m
**TRACE32 XIL GCC arm-none-eabi Cortex-R	gmake makefile**	t32xil_tc_gcc_arm_none_eabi_cortex_r.m
**TRACE32 XIL GCC arm-none-eabi Cortex-M	gmake makefile**	t32xil_tc_gcc_arm_none_eabi_cortex_m.m
**TRACE32 XIL TASKING VX-toolset for Tri-Core	gmake makefile**	t32xil_tc_tasking_ctc.m
**TRACE32 XIL HighTec TriCore Development Platform	gmake makefile**	t32xil_tc_hightec_tricore_gcc.m

5. Add the template makefile to the MATLAB search path and configure it for your build toolchain.

NOTE: This step is only required if you have configured the build process for template makefiles. It can be omitted when a custom toolchain is used.

6. Create a PRACTICE start-up script for your target:

```
HELP.FILTER.Add intsimulink

ENTRY %LINE &ELF_FILE  ;do not modify this line

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

...

ENDDO
```

7. Create a TRACE32 configuration file for your target:
   - Select a TRACE32 operation mode (Instruction Set Simulator, ICD via USB, ...).
   - Configure two ports for the TRACE32 Remote API.
8. Create a TRACE32 XIL settings file and add it to the search path.

Ready-to-run examples for the TRACE32 Instruction Set Simulator are located in the directory `demos`.

Code Coverage Measurement

To measure the code coverage during simulation Simulink Coverage is required. It can be configured via the dialog “Coverage” in “Model Configuration Parameters”.

Code Execution Profiling

To measure the duration of function calls and tasks code execution profiling can be used. The measurement is performed during simulation by supplementing the generated code with instrumentation probes that track the execution time by evaluation of the target's hardware timers. After completion an execution profile of functions and tasks can be viewed within MATLAB.

NOTE:	TRACE32 XIL version 2.1747 or newer is required for code coverage measurement. To check your version enter the command `ver` in the MATLAB command prompt.

Customize Execution Profiling

Enabling execution profiling requires the following steps:

1. Specify a hardware timer for the PIL target connectivity API via the MATLAB Code Replacement Tool.
2. Specify the name of the timer object in the TRACE32 XIL settings file.

Please refer to the MATLAB documentation for additional information on how to set up code execution profiling for PIL simulations.

Stack Profiling

To measure the utilization of the stack memory on the target platform stack profiling can be applied. Stack profiling reports the maximum amount of used stack memory after completion of the simulation.
The measurement is performed by marking the complete stack memory area with a byte pattern prior to the start and verifying its integrity after the simulation has been completed.

Customize Stack Profiling

Enabling stack profiling requires the following configuration steps:

1. Create a variable on the MATLAB workspace with stack memory information for the current target platform.
2. Specify the name of the variable in the TRACE32 XIL settings file.
3. Modify the PRACTICE start-up script to halt the target after completion of the initialization.

Report of Profiling Results

After the end of the simulation a report of the stack profiling is displayed:

```
Maximum stack usage:

System Stack: 255/257 (99%) Bytes used.
```

In addition the contents of the stack memory information are updated with the profiling results.

Stack Memory Information

Information about the stack memory section of the target is provided as MATLAB structure array:

```
>> stack

stack =

                name: 'System Stack'
        startAddress: '807ff00'
          endAddress: '8080000'
      growthDirection: 'DOWN'
             pattern: 'a5'
      maxUsageInBytes: ''
    maxUsageInPercent: ''
```

Field description:

name	Sets the display name of the stack memory section
startAddress	Sets the starting address of the stack memory section in hexadecimal notation
endAddress	Sets the end address of the stack memory section in hexadecimal notation
growthDirection	Sets the growth direction of used stack memory during execution. The values `UP` and *Down* are supported.
pattern	Sets the byte pattern for marking unused stack memory in hexadecimal notation.
maxUsageInBytes	Is updated after completion of the simulation with the maximum number of used stack memory bytes.
maxUsageInPercent	Is updated after completion of the simulation with the percentage of used stack memory bytes.

PRACTICE Callbacks

To support complex analysis tasks with TRACE32, a number of callbacks can be configured to trigger the execution of PRACTICE scripts at certain events during a simulation run. Passing and returning data between MATLAB and TRACE32 is useful to model complex workflows.

Customizing Callbacks

Setting up PRACTICE callbacks requires the following steps:

1. Create one or more containers on the MATLAB workspace that reflect the callback interface for data exchange with TRACE32.
2. Create PRACTICE script(s) that provide the implementation for the required callbacks.
3. Enable the callbacks in the TRACE32 XIL settings file.

Callback Interface

The callback interface is represented by a MATLAB structure array that consists of key-value pairs. Fields can be either defined as input whose value is transferred to TRACE32 or as output whose value is updated by TRACE32.

```
interface = struct('in1', 'input1', ...
                   'in2', 'input2', ...
                   'in3', 'input3', ...,
                   'out1', [], ...
                   'out2', []);
```

The example above defines the following callback interface:

variable	type	value
in1	Input	input1
in2	Input	input2
in3	Input	input3
out1	Output	-
out2	Output	-

The value type determines if a filed is interpreted as input or output. Input values must have type string, whereas all outputs must be defined as empty arrays.

Callback Implementation

The PRACTICE script that is executed is responsible for processing the input arguments and returning its results. Input and output argument are exchanged as a single string with key-value pairs.

```
PRIVATE &in1
PRIVATE &in2
PRIVATE &in3

PARAMETERS &parameters
&in1=STRing.SCANAndExtract("&parameters","IN1=","")
&in2=STRing.SCANAndExtract("&parameters","IN2=","")
&in3=STRing.SCANAndExtract("&parameters","IN3=","")

; Callback implementation here

PRINT " OUT1=output1  OUT2=output2 "

ENDDO
```

Callback Events

The following simulation events are currently supported:

PreInit	Simulation run is prepared, but has not started.
PostTerm	Simulation run is completed.

For each event callback implementation and callback interface can be defined via the TRACE32 XIL settings file.

Enabling Callbacks

Callbacks can be configured via the function *SetPracticeCallbacks()* in the TRACE32 XIL settings file:

```
%% ### Start: User adaptable values ###
hooks.PreInit  = {'', ''};  % Triggered once immediately before the
                            %  simulation is started
hooks.PostTerm = {'callback', 'interface'};  % Triggered once after the
                                             % simulation has
                                             % terminated
% ### End: User adaptable values ###
```

Each callback has the following parameters:

Callback Implementation	Path of the PRACTICE script implementing the callback
Callback Interface	Name of the MATLAB structure array implementing the callback interface

To enable a callback both parameters must be provided. Callbacks are disabled by using two empty strings as parameters.

Headless Mode

When running in headless mode the operation of TRACE32 XIL is optimized for use with environment for continuous integration/deployment. Functions of TRACE32 XIL that are not required in those environments are disabled. Headless mode can be enabled via the TRACE32 XIL settings file:

```
cfg.HeadlessMode            = struct('Enabled', true);
```

DebugIOTool Debugger Abstraction Interface

TRACE32 XIL supports use of the **DebugIOTool debugger abstraction interface** for the execution of PIL simulations. At the time of writing the debugger abstraction interface offers fewer capabilities than the rtiostream API.

The key features are:

- TRACE32 Remote API for generic target support
- Compatible with TRACE32 Instruction Set Simulator for easy testing on virtual targets
- Support for built-in target connectivity capabilities like code execution profiling and code coverage
- Uses breakpoints for data transfer. No support for cache-coherent runtime memory access is required.

TRACE32 XIL is automatically selected for the execution of PIL simulations when top level model and its sub and referenced modules are configured properly.

To configure a model for use, the following steps are necessary:

1. Open the dialog **Model Configuration Parameters**.
2. Configure the settings in the dialog **Hardware Implementation** for your target.
3. Select a supported board description with the dialog element **Hardware board**.
4. Open the dialog **Code Generation**.
5. Select a supported toolchain with the dialog element **Toolchain**.
6. Create a PRACTICE start-up script for your target:

```
HELP.FILTER.Add intsimulink

ENTRY %LINE &ELF_FILE  ;do not modify this line

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

...

ENDDO
```

7. Create a TRACE32 configuration file for your target:
   - Select a TRACE32 operation mode (Instruction Set Simulator, ICD via USB, ...).
   - Configure two ports for the TRACE32 Remote API.
8. Create a TRACE32 XIL settings file and add it to the search path.

Use with Polyspace Test

TRACE32 XIL can be used in combination with Polyspace Test to execute tests on the target hardware. Please refer to **this page** for additional information.

Switching from Toolchain Approach to Makefile-based Build Process for T32XIL by modifying System Target Files

To permanently switch from toolchain approach for builds to a makefile-based build process the header of the active system target file must be modified. The toolchain approach is selected with a header format as shown below:

```
%% SYSTLC: <file>
%%    TMF: ert_default_tmf MAKE: make_rtw EXTMODE: no_ext_comm
```

To select a makefile-based build process please modify the header of the system target file as shown below:

```
%% SYSTLC: <file>
%%    TMF: trace32_target_ert MAKE: make_rtw EXTMODE: no_ext_comm
```

Please reselect the active system target file in the model configuration dialog before using the makefile-based build process.

Switching from Toolchain Approach to Makefile-based Build Process for T32XIL without modifying System Target Files

To switch from toolchain approach for builds to a makefile-based build process without modifying the active system target file the model configuration has to be changed. The following configuration parameters of the model determine the build process:

- GenerateMakefile
- MakeCommand
- TemplateMakefile

Settings one of these parameters to a non-default value unlocks the makefile-based build process e. g.

```
set_param(getActiveConfigSet(gcs),'MakeCommand', 'make_rtw "USE_TMF=1"')
```

Additional modifications can be performed as required via command line or graphical configuration dialog.

Known Issues

This section is updated regularly with a list of common issues and workarounds.

TRACE32 XIL Cannot Be Used for BigEndian Targets [MATLAB R2016a]

Simulation runs may fail in case of BigEndian targets with the error message:

```
Error: Invalid payload size (16777216) received during SIL/PIL
communication between Simulink and the target application. Check the
rtiostream implementation for the target application.
```

This behavior is a known issue of MATLAB R2016a. Please see bug report #1404465 on the MathWorks homepage for additional details and an official workaround.

TRACE32 XIL Cannot Be Used with Code Coverage Measurement and Diab Compiler [MATLAB R2016b]

Simulation runs may fail in case of code coverage measurement and Diab Compiler:

```
Error: The dialect 'diab' is unknown
```

Please carry out the following steps as workaround for this issue:

1. Go to the “<MATLABROOT>*polyspace**configure**compiler_configuration*” folder.
2. Open the original “*diab.xml”* file and search for the line: "*<dialect>diab</dialect>*".
3. Replace "*<dialect>diab</dialect>*" with "*<dialect>default</dialect>*".
4. Restart MATLAB.
5. Backup the file “*<MATLABROOT>**polyspace**verifier**extensions**diab**tmw_builtins**powerpc.h*”
6. Replace the file “*<MATLABROOT>**polyspace**verifier**extensions**diab**tmw_builtins**powerpc.h*” file with the version in “t32xil\targets\ppc\powerpc.h”.

NOTE:	The modified header file is intended for use with the processor type PPCE200Z0VES. The use of the function “alloca()” is not supported.

TRACE32 with QT Screendriver Cannot Be Started [Linux]

Simulation runs may fail with these error messages, if TRACE32 uses the QT screendriver:

```
Call to service method "open" failed.

TRACE32 DebugIO: Cannot retrieve license.
```

```
TRACE32 XIL: Cannot establish connection with TRACE32.
```

Root cause for this error is that the QT version shipping with MATLAB is not compatible with TRACE32. To circumvent this error, please create a wrapper script for TRACE32 that overloads the environment variable "*LD_LIBRARY_PATH*":

```
#!/bin/bash
EXENAME=$(readlink -e "$0")

LD_LIBRARY_PATH=/lib/x86_64-linux-gnu/:$LD_LIBRARY_PATH exec ${EXENAME%-
matlab} "$@"
```

Please convert "*/lib/x86_64-linux-gnu/*" to the corresponding path for your operating system.

Help Us Help You - Export TRACE32 Information

To help us help you, we need some data about your TRACE32 installation: which TRACE32 revision, which operating system (32-bit or 64-bit variant), which Lauterbach hardware, which firmware version you use, which target architecture and CPU you are debugging with etc.

To automatically collect this data, please follow these steps:

1. Download **`support.cmm`** from **https://www.lauterbach.com/support/static/support.cmm**
2. Start TRACE32 as usual. If possible, connect to the target and stop on a breakpoint.
3. Execute the downloaded support script (in TRACE32) with **`DO support.cmm`**
   (on Windows you can drag and drop it from the Windows Explorer window into the TRACE32 command line).

   The script will first show a form for contact data.
4. If this is your first inquiry, please fill in the form.
   (Name, Address and Email are vital, to make sure we can reach you with our response).
5. Click the **Save to File** button.
6. Attach the generated output (the saved text file) in your support request.