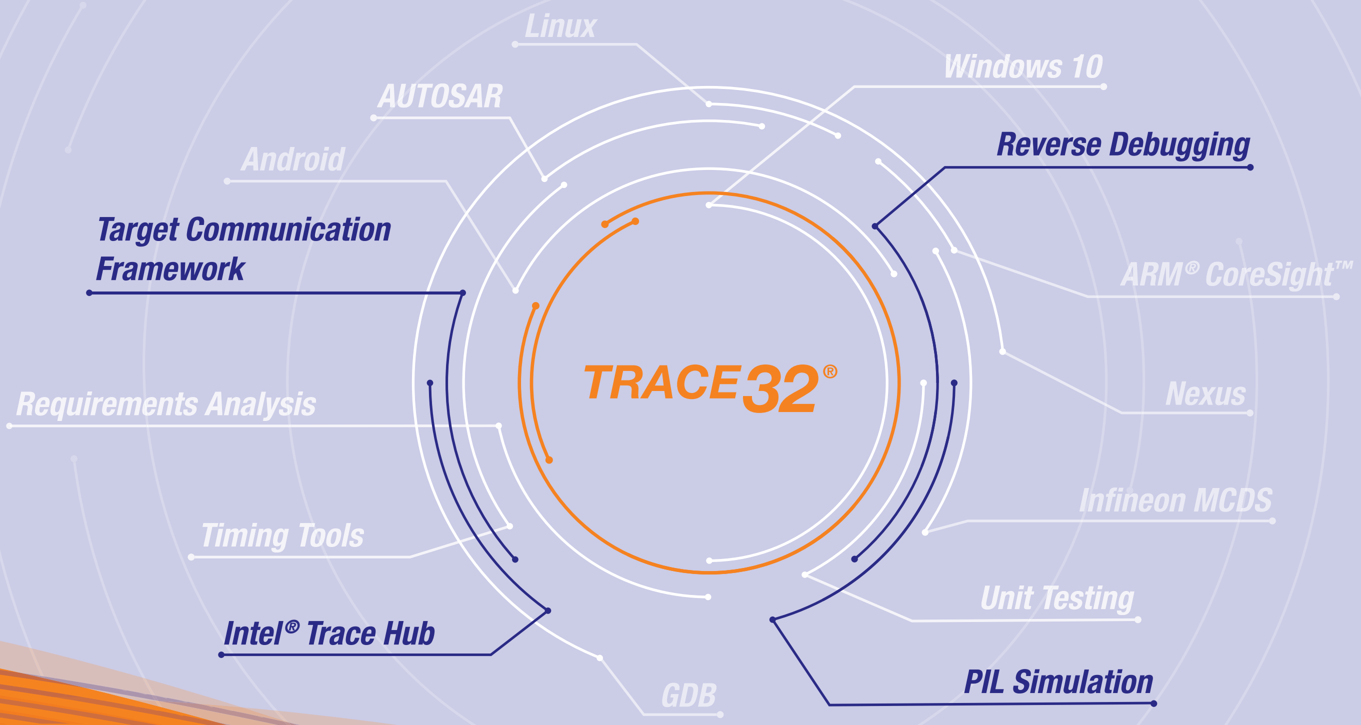
NEWS 2016



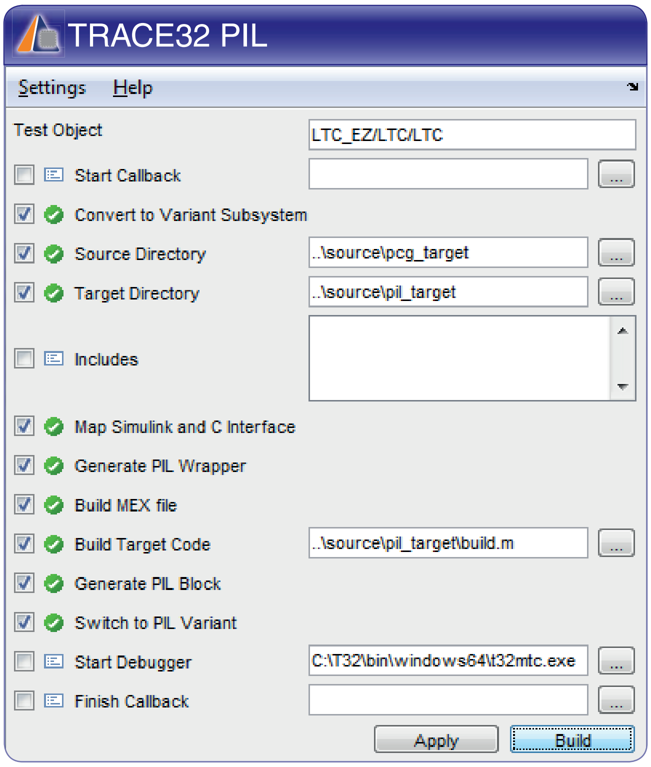
CONTENTS

LAUTERBACH
DEVELOPMENT TOOLS

PIL Simulation with TRACE32

In March 2016, Lauterbach will provide its Simulink plug-in for PIL (Processor-in-the-Loop) simulation. With this new plug-in, the modeling environment can communicate directly with the target through a TRACE32 debugger.

Over the course of the last few years, model-based methods have become more and more important in software development. The advantage of model-based methods is the continuous verification of the software design. The Processor-in-the-Loop simulation has become an important step in the design verification.



Test Object	LTC_EZ/LTC/LTC
☐ Start Callback	
☑ Convert to Variant Subsystem	
☑ Source Directory	..\source\pcg_target
☑ Target Directory	..\source\pil_target
☐ Includes	
☑ Map Simulink and C Interface	
☑ Generate PIL Wrapper	
☑ Build MEX file	
☑ Build Target Code	..\source\pil_target\build.m
☑ Generate PIL Block	
☑ Switch to PIL Variant	
☐ Start Debugger	C:\T32\bin\windows64\t32mtc.exe
☐ Finish Callback	

PIL Simulation

PIL simulations are performed to ensure the developed algorithms provide the correct functionality in the target environment. This verification step can be executed on one of the following target systems:

- Final target hardware / evaluation board
- Virtual target / core simulator
- *TRACE32 Instruction Set Simulator*

To execute the PIL simulation, the algorithm to be tested is replaced by a PIL block in the modeling environment.

TRACE32 PIL

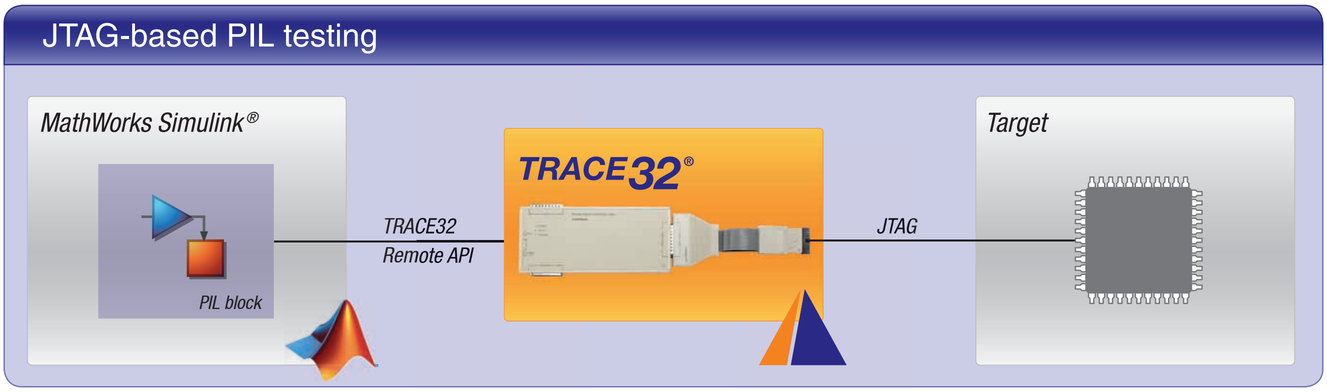
The TRACE32 PIL plug-in (see the screenshot on this page) is used to configure the PIL simulation. The most important configuration steps of this plug-in will be presented briefly in the following.

Checkbox: Map Simulink and C Interface

The interface between Simulink and the target application needs to be configured before generating the PIL block. For configuration purposes, the Mapping GUI dialog (see the screenshot on page 3) automatically generates a suggestion for mapping the callbacks of the S-function (user-defined block) to the corresponding C functions. This suggestion can be checked and corrected in the dialog. The same map must also be created between the model parameters and the C variables.

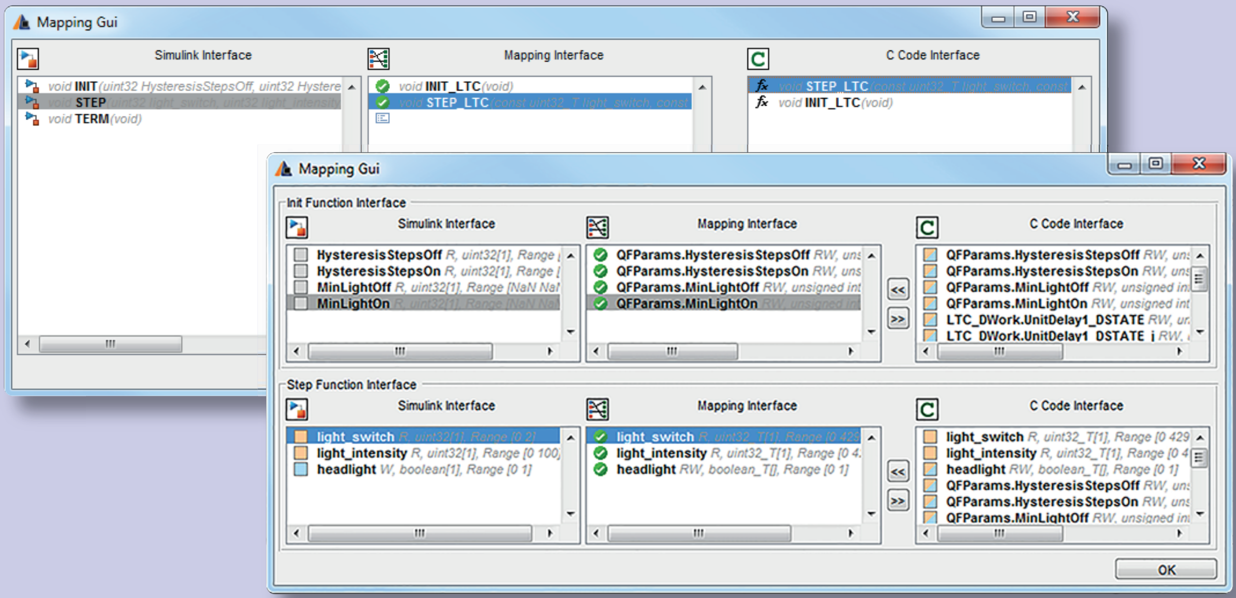
Checkbox: Generate PIL Wrapper

Based on the maps of the callbacks and the model parameters, the following interfaces are created automatically in this configuration step:



Mapping Simulink and C Interface

Mapping Simulink callbacks $\longrightarrow$ *C functions*



Simulink Interface:
void INIT(uint32 HysteresisStepsOff, uint32 Hystere
void STEP(uint32 light_switch, uint32 light_intensity
void TERM(void)

Mapping Interface:
void INIT_LTC(void)
void STEP_LTC(const uint32_T light_switch, const

C Code Interface:
void STEP_LTC(const uint32_T light_switch, const
void INIT_LTC(void)

Init Function Interface

Simulink Interface	Mapping Interface	C Code Interface
HysteresisStepsOff R, uint32[1], Range	QFParams.HysteresisStepsOff RW, uns	QFParams.HysteresisStepsOff RW, uns
HysteresisStepsOn R, uint32[1], Range [	QFParams.HysteresisStepsOn RW, uns	QFParams.HysteresisStepsOn RW, uns
MinLightOff R, uint32[1], Range [NaN NaN	QFParams.MinLightOff RW, unsigned int	QFParams.MinLightOff RW, unsigned int
MinLightOn R, uint32[1], Range [NaN NaN	QFParams.MinLightOn RW, unsigned int	QFParams.MinLightOn RW, unsigned int
		LTC_DWork.UnitDelay1_DSTATE RW, ur
		LTC_DWork.UnitDelay1_DSTATE_i RW,

Step Function Interface

Simulink Interface	Mapping Interface	C Code Interface
light_switch R, uint32[1], Range [0 2]	light_switch R, uint32_T[1], Range [0 429	light_switch R, uint32_T[1], Range [0 429
light_intensity R, uint32[1], Range [0 100]	light_intensity R, uint32_T[1], Range [0 4	light_intensity R, uint32_T[1], Range [0 4
headlight W, boolean[1], Range [0 1]	headlight RW, boolean_T[], Range [0 1]	headlight RW, boolean_T[], Range [0 1]
		QFParams.HysteresisStepsOff RW, uns
		QFParams.HysteresisStepsOn RW, uns
		QFParams.MinLightOff RW, unsigned int

Mapping Simulink parameters $\longrightarrow$ *C variables*

- Implementation of the callbacks on the target system
- Interface of the model to the *TRACE32 Remote API*

After that, the executable code can be created.

Checkboxes:
Generate PIL Block and Switch to PIL Variant

After the function interface between the model and the *TRACE32 Remote API* is created, the PIL block can be created and displayed in the simulation model. The configuration of the PIL simulation is complete as soon as TRACE32 starts.

Advantages

1. All methods of code generation can be used

Through the ability to dynamically configure the interface for the callbacks, the Lauterbach solution can be used with all code generators as well as with code created manually.

2. Direct adaptation to new target systems

TRACE32 supports a very wide range of processor architectures and compilers. Start-up scripts for the target systems can be created directly by the developer. You don't have to wait for a software update from any tool provider. As soon as the debugger is able to communicate with the target, the basis for the PIL simulation is ready.

3. Immediate debugging

If the results of a simulation deviate from what was expected, then it is possible to debug the C functions directly.

4. License policy

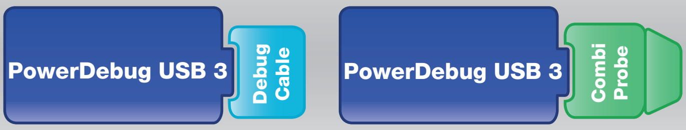
Developers who are already using a hardware-based TRACE32 debugger or a virtual target with a *TRACE32 Floating License,* need only a *TRACE32 PIL Simulation License.* Customers who want to use a *TRACE32 Instruction Set Simulator* for the simulation additionally need the new *TRACE32 Simulator License*.

Conversion to New Base Modules Completed in 2015

All TRACE32 PowerDebug modules are now equipped with a USB 3 interface. The module PowerDebug PRO also offers a Gigabit Ethernet interface as well as a PodBus Express interface which is used for connecting the trace modules TRACE32 PowerTrace PX (new) or TRACE32 PowerTrace II. The previous generation of base modules can still be used for new chips/processors without any restrictions.

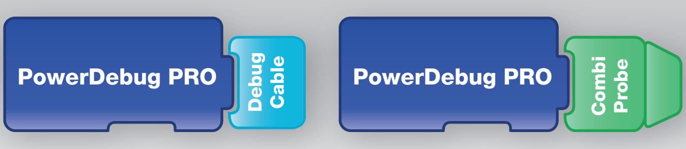
PowerDebug USB 3

Previous generation PowerDebug USB 2



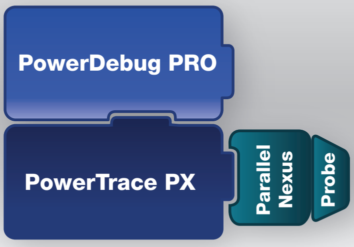
PowerDebug PRO

Previous generation PowerDebug Ethernet or PowerDebug II



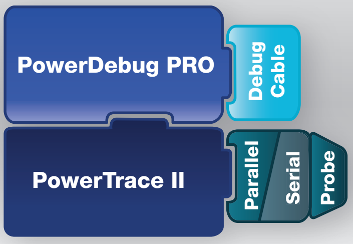
PowerDebug PRO + PowerTrace PX

Previous generation PowerTrace Ethernet



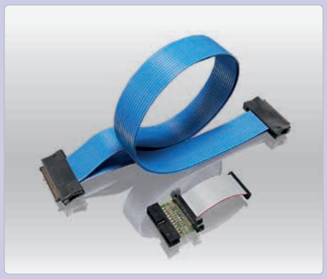
PowerDebug PRO + PowerTrace II

Previous generation PowerDebug II + PowerTrace II



LAUTERBACH DEVELOPMENT TOOLS

PowerTrace Serial



TRACE32 Accessory Set

The new TRACE32 base module PowerTrace Serial will be available for delivery starting in Q2/2016.

Lauterbach has supported serial trace interfaces since 2008 with a serial trace probe for various processor architectures. The probe has the following features:

- Up to 4 RX channels
- 6.25 Gbit/s per channel for up to 3 channels
- 3.125 Gbit/s per channel for up to 4 channels
- For trace protocols that use Aurora

The bandwidths of current serial trace interfaces on the processors are not always high enough to enable the developer to fully visualize the internal operations in complex multicore systems. This is why some processor manufacturers are starting to develop interfaces with higher data rates and more channels. In addition, PCI Express is being mentioned more and more often as a trace export interface.

PowerTrace Serial Features

The TRACE32 PowerTrace Serial base module is designed to meet the latest requirements.

- Up to 8 channels
- Up to 12.5 Gbit/s per channel
- Xilinx Aurora as well as other protocols especially PCI Express
- 4 gigabyte trace memory

Since these features could only be implemented by using a very large and high-performance FPGA, the new PowerTrace Serial has been designed as an all-in-one solution. This means the serial trace probe previously available separately is now integrated into the PowerTrace Serial. Lauterbach offers various accessory sets to connect the PowerTrace Serial to the target. The accessory sets will normally consist of an appropriate flex cable and the necessary adapters.

The PowerTrace Serial is licensed for decoding the core trace information of one processor architecture upon delivery. Trace decoding functionality for other architectures can simply be licensed later on at any time.

PowerTrace Serial Connectors

The PowerTrace Serial is equipped with the following interfaces:

Serial trace port 0 (Samtec ERF8, 40-pin)

- For trace protocols that use Aurora
- 6 RX channels
- Reference clock 0.325 - 6.25 GHz

Target Debug Port (34-Pin MIPI Connector)

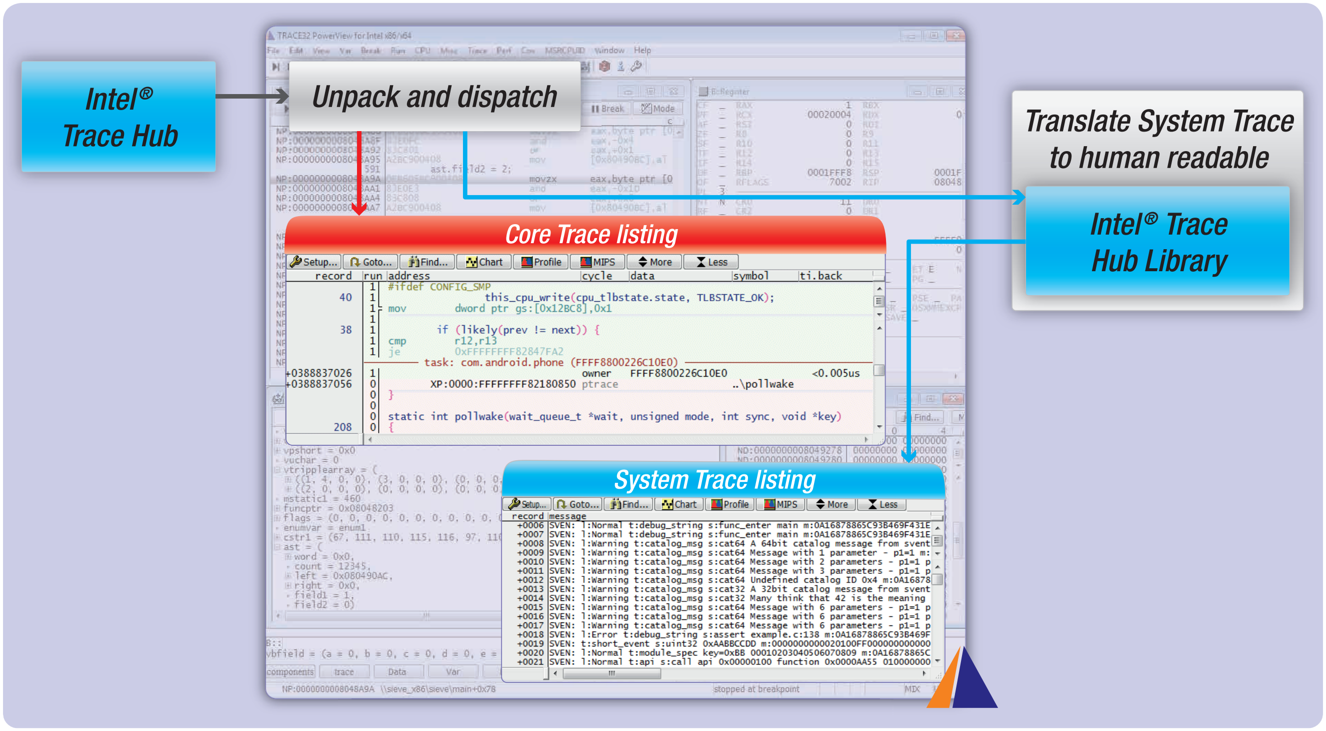
If debug signals are also routed to the 40-pin Samtec connector (JTAG/SWD/cJTAG), then a TRACE32 debug cable can also be connected here.

Serial trace port 1 (Samtec ERM-ERF, 80-pin)

- For all protocols
- 8 RX/TX channels
- Reference clock 0.325 - 6.25 GHz or 10 - 500 MHz

By providing both trace ports, the PowerTrace Serial can be used in current designs as well as in future projects.

Support for Intel® Trace Hub



```
   record  run address                                      cycle   data              symbol      ti.back
            1  #ifdef CONFIG_SMP
       40   1                this_cpu_write(cpu_tlbstate.state, TLBSTATE_OK);
            1  mov         dword ptr gs:[0x12BC8],0x1
            1
       38   1          if (likely(prev != next)) {
            1  cmp         r12,r13
            1  je          0xFFFFFFFF82847FA2
           ——— task: com.android.phone (FFFF8800226C10E0) ———
+0388837026 1                                               owner   FFFF8800226C10E0              <0.005us
+0388837056 0          XP:0000:FFFFFFFF82180850 ptrace                            ..\pollwake
            0  }
            0
            0  static int pollwake(wait_queue_t *wait, unsigned mode, int sync, void *key)
      208   0  {
```

```
  record message
   +0006 SVEN: l:Normal t:debug_string s:func_enter main m:0A16878865C93B469F431E
   +0007 SVEN: l:Normal t:debug_string s:func_enter main m:0A16878865C93B469F431E
   +0008 SVEN: l:Warning t:catalog_msg s:cat64 A 64bit catalog message from svent
   +0009 SVEN: l:Warning t:catalog_msg s:cat64 Message with 1 parameter - p1=1 m:
   +0010 SVEN: l:Warning t:catalog_msg s:cat64 Message with 2 parameters - p1=1 p
   +0011 SVEN: l:Warning t:catalog_msg s:cat64 Message with 3 parameters - p1=1 p
   +0012 SVEN: l:Warning t:catalog_msg s:cat64 Undefined catalog ID 0x4 m:0A16878
   +0013 SVEN: l:Warning t:catalog_msg s:cat32 A 32bit catalog message from svent
   +0014 SVEN: l:Warning t:catalog_msg s:cat32 Many think that 42 is the meaning
   +0015 SVEN: l:Warning t:catalog_msg s:cat64 Message with 6 parameters - p1=1 p
   +0016 SVEN: l:Warning t:catalog_msg s:cat64 Message with 6 parameters - p1=1 p
   +0017 SVEN: l:Warning t:catalog_msg s:cat64 Message with 6 parameters - p1=1 p
   +0018 SVEN: l:Error t:debug_string s:assert example.c:138 m:0A16878865C93B469F
   +0019 SVEN: t:short_event s:uint32 0xAABBCCDD m:000000000020100FF000000000000
   +0020 SVEN: l:Normal t:module_spec key=0xBB 00010203040506070809 m:0A16878865C
   +0021 SVEN: l:Normal t:api s:call api 0x00000100 function 0x0000AA55 010000000
```

In May 2016, TRACE32 will support the Intel® Trace Hub and its associated framework.

Intel® Trace Hub

Intel® Trace Hub (Intel® TH) is the name of the trace infrastructure Intel provides in its new hardware platforms. This trace infrastructure makes it possible:

1. To provide a common timestamp to *Intel® Processor Trace* data from individual cores and *System Trace* information from a variety of sources.
2. To merge all data into a single trace stream by using the MIPI STPv2.1 protocol.
3. To convey this stream to the selected trace destination.

To enable debug and trace tools such as TRACE32 to support this trace infrastructure easily, Intel also provides a software framework.

Intel® Trace Hub Configuration API

The aim of the *Intel® Trace Hub Configuration API* is to simplify the configuration of the trace infrastructure through the debug tool. TRACE32 does not have to know the platform-specific programming sequence, but can send a configuration request to the *Intel® TH Configuration API* instead. The API then provides the appropriate programming sequence, and TRACE32 writes this sequence to the control registers via the JTAG interface.

Intel® Trace Hub Library

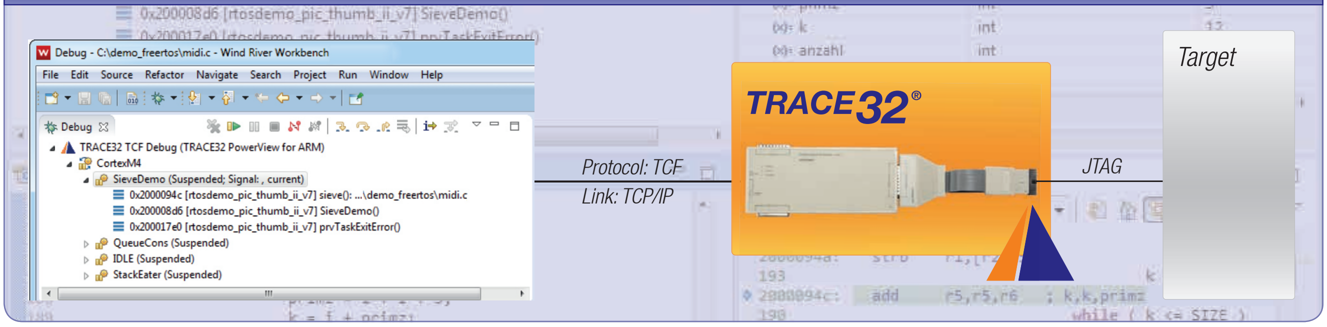
The aim of the *Intel® TH Library* is to convert the *System Trace* packets into human readable trace messages. Since the *Intel® TH* has packed all trace information in the MIPI STPv2.1 protocol, it must first be unpacked and then distributed for decoding (see picture above):

- *Intel® Processor Trace packets* remain in TRACE32, where they are directly decoded for display and evaluation.
- The *System Trace* packets are forwarded to the *Intel® TH Library*. The returned decoded and human readable trace messages can then be displayed and evaluated within TRACE32 with a dedicated command group.

The simple availability of a time-correlated view of all trace information in TRACE32 allows a quick overview of the platform-wide activities.

TRACE32 as a TCF-Agent

JTAG-based TCF Toolchain



```
Debug - C:\demo_freertos\midi.c - Wind River Workbench
File  Edit  Source  Refactor  Navigate  Search  Project  Run  Window  Help
Debug
  TRACE32 TCF Debug (TRACE32 PowerView for ARM)
    CortexM4
      SieveDemo (Suspended; Signal: , current)
        0x2000094c [rtosdemo_pic_thumb_ii_v7] sieve(): ...\demo_freertos\midi.c
        0x200008d6 [rtosdemo_pic_thumb_ii_v7] SieveDemo()
        0x200017e0 [rtosdemo_pic_thumb_ii_v7] prvTaskExitError()
      QueueCons (Suspended)
      IDLE (Suspended)
      StackEater (Suspended)
```

Since October 2015, TRACE32 also operates as a TCF agent. This makes it possible to use the Wind River Workbench or the Eclipse debugger as an IDE and a TRACE32 debugger as a debugging back end.

TCF

The *Target Communication Framework* (TCF) was developed by the Eclipse Foundation as a protocol framework with the goal of defining a uniform debugging communication protocol between an IDE and a target system.

The TCF communication model is based on the idea of services. A service is a group of related commands, events, and the corresponding semantics. The *Memory Service*, for example, defines a group of commands and events for reading and writing to memory.

TCF defines a series of standard services. At the same time, the framework is open for the definition of proprietary services.

TRACE32 TCF

After the TRACE32 software is started as a TCF agent, it provides its services to the Wind River Workbench or the Eclipse debugger via TCP/IP.

Requested services are operated by TRACE32 with the help of the debugger connected to the target. It does not matter if this is a hardware-based debugger connected via JTAG to the target system or a pure software debugger communicating with a virtual target. At the present time, TRACE32 offers all traditional services relevant to debugging. Depending on the feedback and requests from customers, development of special TRACE32 services for extended functions is planned for the future. Lauterbach offers a TRACE32 TCF plug-in for customers who want to configure and start TRACE32 from the Wind River Workbench or the Eclipse debugger.

Wind River Workbench

Until this year, Lauterbach has not offered an integrated debugging solution for developers who prefer to work with the Wind River Workbench. This limitation has now been resolved.

Eclipse Debugger

The previous GDB-based connection was restricted to the processor architectures and compilers supported by GDB. Using its TCF services, TRACE32 can now provide an open communication interface for debugging with Eclipse for all processor architectures and compilers supported by TRACE32.

TRACE32 Support for Wind River

VxWorks 5/6/7

VxWorks 653 2.x

VxWorks 653 3.x

Wind River Linux

Wind River Hypervisor 2.x

VxWorks Microkernel Profile

TRACE32 as a Front End for UndoDB

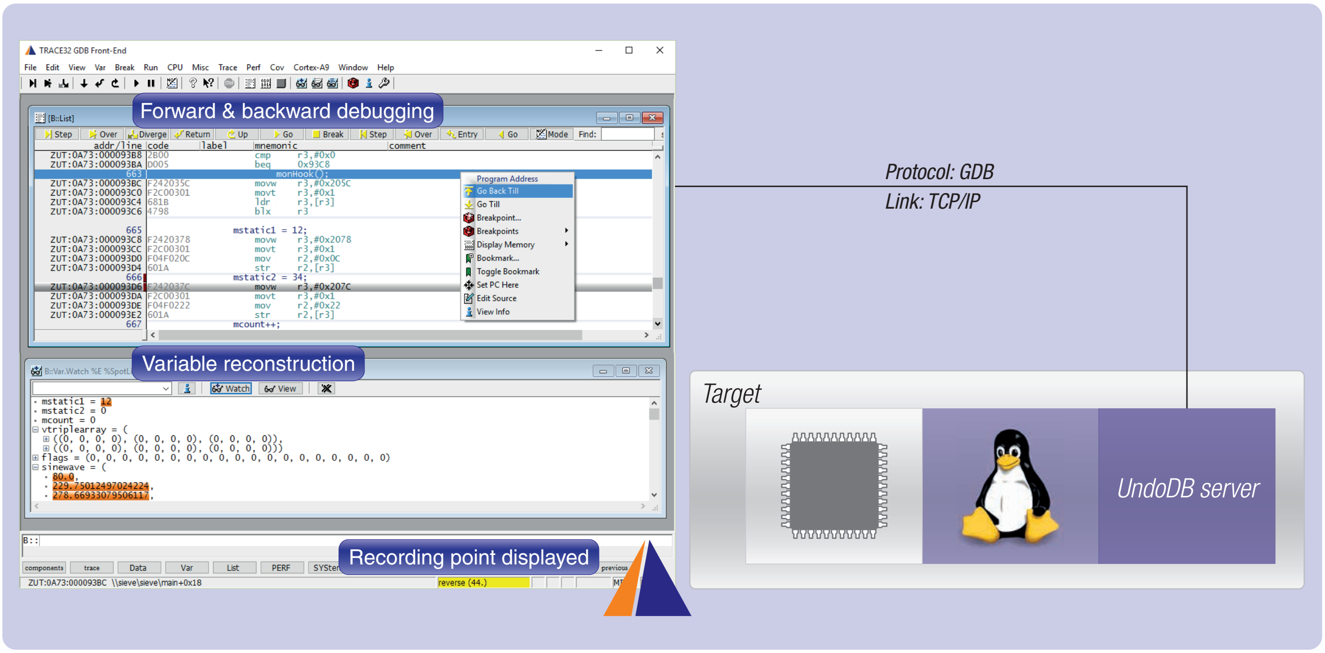
TRACE32 can be used as a front end for the UndoDB reversible debugger since the middle of 2015. The ARM/Cortex as well as Intel® x86/x64 architectures are supported.

With the UndoDB target server, Linux developers have a tool available that allows them to debug an application process as well as to record details of its execution.

In addition to controlling the debugging process, the TRACE32 front end also takes over the task of displaying the recorded data of the UndoDB target server in the TRACE32 GUI. Like a trace recording, the developer has the ability to debug the application going through the code both forward and backward ("reverse debugging"). Using this function, errors in the application process can be located quickly and easily.

To provide a visual emphasis that debugging of the recorded data has been activated, the time of recording displayed in the TRACE32 status line is referenced in reverse. In addition, the debug buttons in the source listing are displayed in yellow. The TRACE32 GUI switches automatically to the following display mode:

- In the source listing, the instruction pointer is reset to its value at the time of recording displayed.
- The contents of memory and variables at the time of recording are displayed.



```
          addr/line code      label   mnemonic                comment
  ZUT:0A73:000093B8 2B00              cmp      r3,#0x0
  ZUT:0A73:000093BA D005              beq      0x93C8
                663                       monHook();
  ZUT:0A73:000093BC F242035C          movw     r3,#0x205C
  ZUT:0A73:000093C0 F2C00301          movt     r3,#0x1
  ZUT:0A73:000093C4 681B              ldr      r3,[r3]
  ZUT:0A73:000093C6 4798              blx      r3

                665             mstatic1 = 12;
  ZUT:0A73:000093C8 F2420378          movw     r3,#0x2078
  ZUT:0A73:000093CC F2C00301          movt     r3,#0x1
  ZUT:0A73:000093D0 F04F020C          mov      r2,#0x0C
  ZUT:0A73:000093D4 601A              str      r2,[r3]
                666             mstatic2 = 34;
  ZUT:0A73:000093D6 F242037C          movw     r3,#0x207C
  ZUT:0A73:000093DA F2C00301          movt     r3,#0x1
  ZUT:0A73:000093DE F04F0222          mov      r2,#0x22
  ZUT:0A73:000093E2 601A              str      r2,[r3]
                667             mcount++;
```

```
· mstatic1 = 12
· mstatic2 = 0
· mcount = 0
⊟ vtriplearray = (
  ⊞ ((0, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0)),
  ⊞ ((0, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0)))
⊞ flags = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
⊟ sinewave = (
  · 80.0,
  · 229.75012497024224,
  · 278.66933079506117,
```

```
B::
ZUT:0A73:000093BC  \\sieve\sieve\main+0x18          reverse (44.)
```

If your address has changed or you do not want to receive a newsletter from us any more, please send a brief email to the following address:
mailing@lauterbach.com

***LEADING** through Technology*