TASKING®

TASKING EMBEDDED DEBUGGER PRODUCT OVERVIEW

A Feature-Rich and Cost-Efficient Debug Solution for Tricore/AURIX

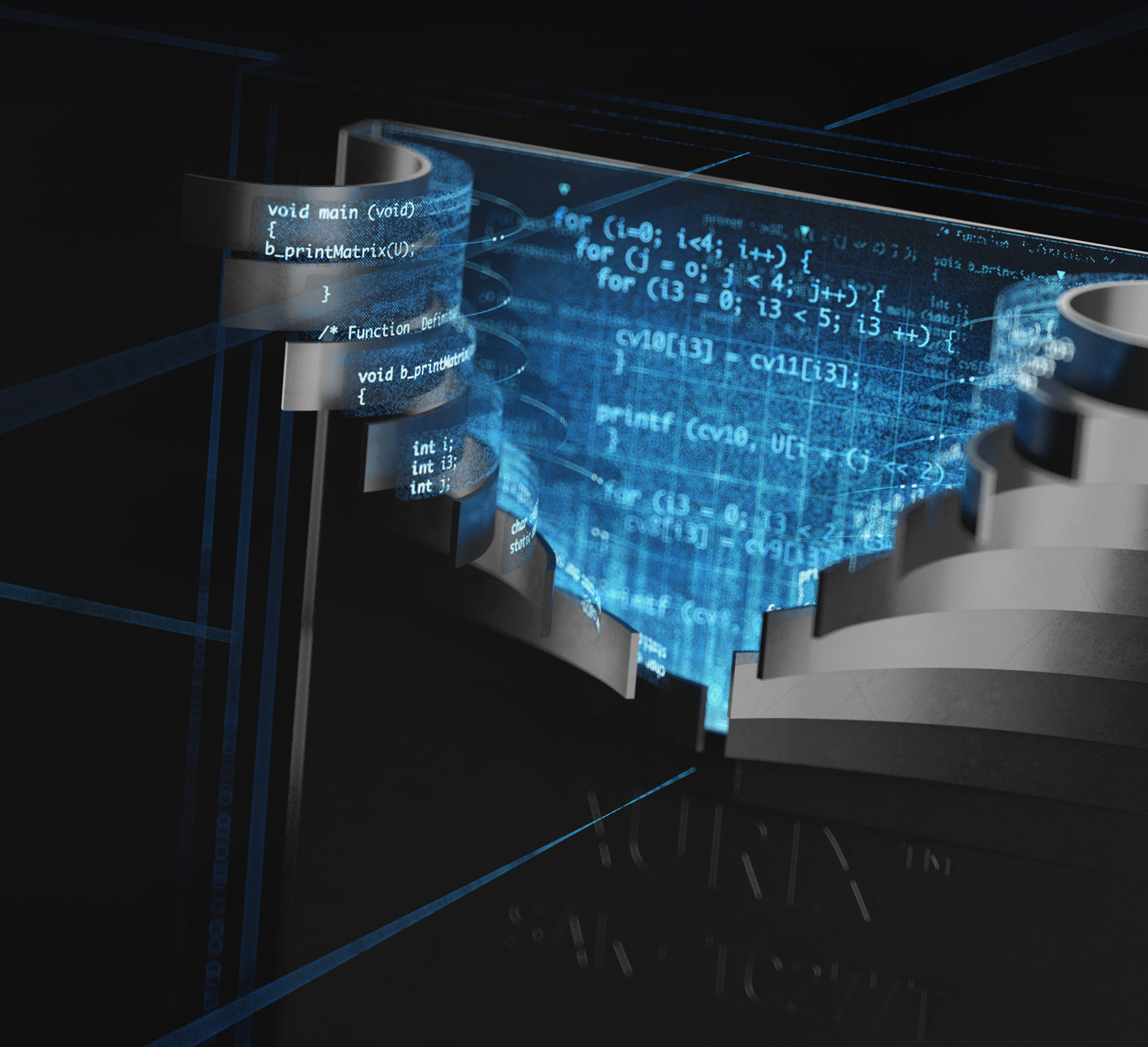


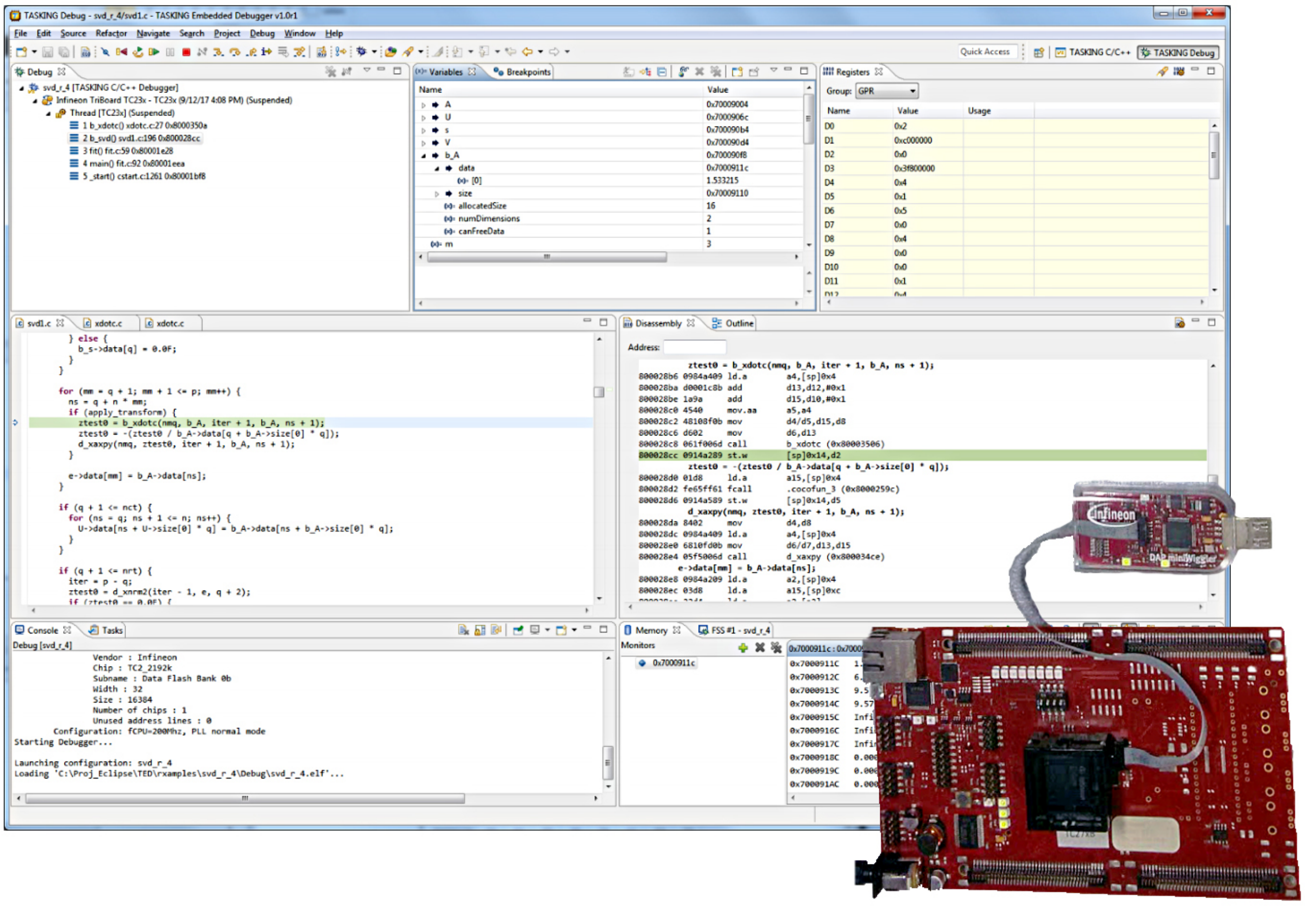
TABLE OF CONTENTS

EMBEDDED DEBUGGER - HIGH-LEVEL OVERVIEW

TASKING has been providing debuggers for Infineon TriCore™ microcontrollers for over 20 years. Until recently these debuggers were released as a component of the TASKING TriCore VX Toolset only, and were not available as stand-alone tool. Many customers observed that this debugger provided all features to handle the majority of their debug tasks and have asked to make it available as a stand-alone tool for their software developers independent from the compiler toolset. This has led to the release of the TASKING Embedded Debugger which offers streamlined single and multicore debugging for up to six TriCore cores, and debugging of auxiliary controllers such as GTM/MCS, SCR, HSM and PCP.

The debugger is an ECLIPSE based stand-alone tool, or can be integrated as a plugin into existing Eclipse Mars environments. All features that streamline the debug process are available such as loading a program to RAM and/or FLASH memory, C/C++ and Assembly level symbolic debugging -- also of highly optimized binaries --, code breakpoints and data watchpoints, concurrent debugging of all cores, run-control with multi-core start/stop synchronization, Autosar-OS aware debugging, full symbolic access to all device control registers (SFRs), and more. Developer productivity is further enhanced by out-of-the box board support, wizard-guided project setup and task execution, extensive online help and user documentation, and responsive customer support.

The TASKING Embedded Debugger provides high-speed access to the target device via the JTAG or DAP protocol using a low-cost probe, the Infineon DAP miniwiggler. Features that require expensive hardware probes such as elaborate tracing and complex timing analysis are not supported. If no physical hardware is available the debugger can connect to the instruction-set-simulators that are included in the package.



```
svd_r_4 [TASKING C/C++ Debugger]
  Infineon TriBoard TC23x - TC23x (9/12/17 4:08 PM) (Suspended)
    Thread [TC23x] (Suspended)
      1 b_xdotc() xdotc.c:27 0x8000350a
      2 b_svd() svd1.c:196 0x800028cc
      3 fit() fit.c:59 0x80001e28
      4 main() fit.c:92 0x80001eea
      5 _start() cstart.c:1261 0x80001bf8
```

Name	Value
A	0x70009004
U	0x7000906c
s	0x700090b4
V	0x700090d4
b_A	0x700090f8
data	0x7000911c
[0]	1.533215
size	0x70009110
allocatedSize	16
numDimensions	2
canFreeData	1
m	3

Group: GPR

Name	Value	Usage
D0	0x2	
D1	0xc000000	
D2	0x0	
D3	0x3f800000	
D4	0x4	
D5	0x1	
D6	0x5	
D7	0x0	
D8	0x4	
D9	0x0	
D10	0x0	
D11	0x1	

```c
        } else {
          b_s->data[q] = 0.0F;
        }
      }

      for (mm = q + 1; mm + 1 <= p; mm++) {
        ns = q + n * mm;
        if (apply_transform) {
          ztest0 = b_xdotc(nmq, b_A, iter + 1, b_A, ns + 1);
          ztest0 = -(ztest0 / b_A->data[q + b_A->size[0] * q]);
          d_xaxpy(nmq, ztest0, iter + 1, b_A, ns + 1);
        }

        e->data[mm] = b_A->data[ns];
      }

      if (q + 1 <= nct) {
        for (ns = q; ns + 1 <= n; ns++) {
          U->data[ns + U->size[0] * q] = b_A->data[ns + b_A->size[0] * q];
        }
      }

      if (q + 1 <= nrt) {
        iter = p - q;
        ztest0 = d_xnrm2(iter - 1, e, q + 2);
```

```
           ztest0 = b_xdotc(nmq, b_A, iter + 1, b_A, ns + 1);
800028b6 0984a409 ld.a       a4,[sp]0x4
800028ba d0001c8b add        d13,d12,#0x1
800028be 1a9a     add        d15,d10,#0x1
800028c0 4540     mov.aa     a5,a4
800028c2 48108f0b mov        d4/d5,d15,d8
800028c6 d602     mov        d6,d13
800028c8 061f006d call       b_xdotc (0x80003506)
800028cc 0914a289 st.w       [sp]0x14,d2
           ztest0 = -(ztest0 / b_A->data[q + b_A->size[0] * q]);
800028d0 01d8     ld.a       a15,[sp]0x4
800028d2 fe65ff61 fcall      .cocofun_3 (0x8000259c)
800028d6 0914a589 st.w       [sp]0x14,d5
           d_xaxpy(nmq, ztest0, iter + 1, b_A, ns + 1);
800028da 8402     mov        d4,d8
800028dc 0984a409 ld.a       a4,[sp]0x4
800028e0 6810fd0b mov        d6/d7,d13,d15
800028e4 05f5006d call       d_xaxpy (0x800034ce)
         e->data[mm] = b_A->data[ns];
800028e8 0984a209 ld.a       a2,[sp]0x4
800028ec 03d8     ld.a       a15,[sp]0xc
```

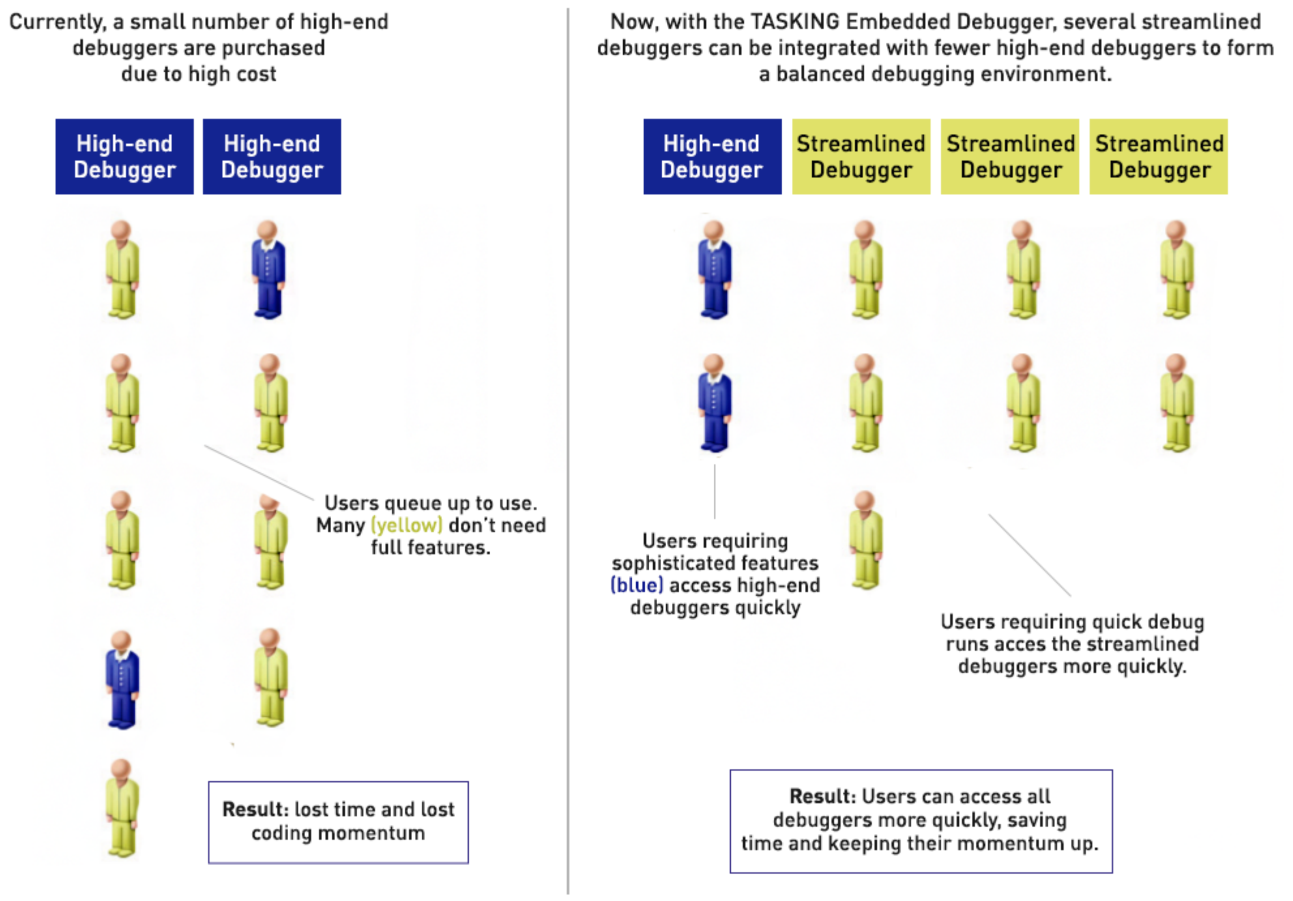
```
Debug [svd_r_4]
              Vendor : Infineon
              Chip : TC2_2192k
              Subname : Data Flash Bank 0b
              Width : 32
              Size : 16384
              Number of chips : 1
              Unused address lines : 0
       Configuration: fCPU=200Mhz, PLL normal mode
Starting Debugger...

Launching configuration: svd_r_4
Loading 'C:\Proj_Eclipse\TED\rxamples\svd_r_4\Debug\svd_r_4.elf'...
```

The embedded debugger is shown with the DAP miniwiggler and the TriCore target board

Communication between the debugger and probe is implemented through the Infineon Device Access Server (DAS) protocol. This assures interoperability between the debugger and other tools that require device access such as calibration tools, and offers remote access to the target device enabling developers to connect from their office desk to a target device in the lab. This way, the device can also be time-shared among multiple developers.

Finish your projects faster, spend less, reduce schedule risk, and eliminate developer down time. Most of the debugging time spent by developers is not with complex issues, but rather simple code verification. Rather than purchase a few high-end debuggers that developers have to share, a better solution is fewer expensive debuggers along with several streamlined debuggers that can analyze the majority of issues. Interrupting the developers edit-compile-debug cycle to wait for a debugger license is not only inefficient, but developers usually develop a "momentum" when coding...a string of contiguous thought about how to solve the problem. If this is interrupted, it can take time to regain that train of thought. Enough TASKING Embedded Debuggers can be deployed to keep coding efficient, maintain coding momentum, and stay within budget.


Currently, a small number of high-end debuggers are purchased due to high cost
High-end Debugger
High-end Debugger
Users queue up to use. Many (yellow) don't need full features.
Result: lost time and lost coding momentum
Now, with the TASKING Embedded Debugger, several streamlined debuggers can be integrated with fewer high-end debuggers to form a balanced debugging environment.
High-end Debugger
Streamlined Debugger
Streamlined Debugger
Streamlined Debugger
Users requiring sophisticated features (blue) access high-end debuggers quickly
Users requiring quick debug runs acces the streamlined debuggers more quickly.
Result: Users can access all debuggers more quickly, saving time and keeping their momentum up.

A debugger on every desk

EMBEDDED DEBUGGER - FUNCTIONS AND FEATURES

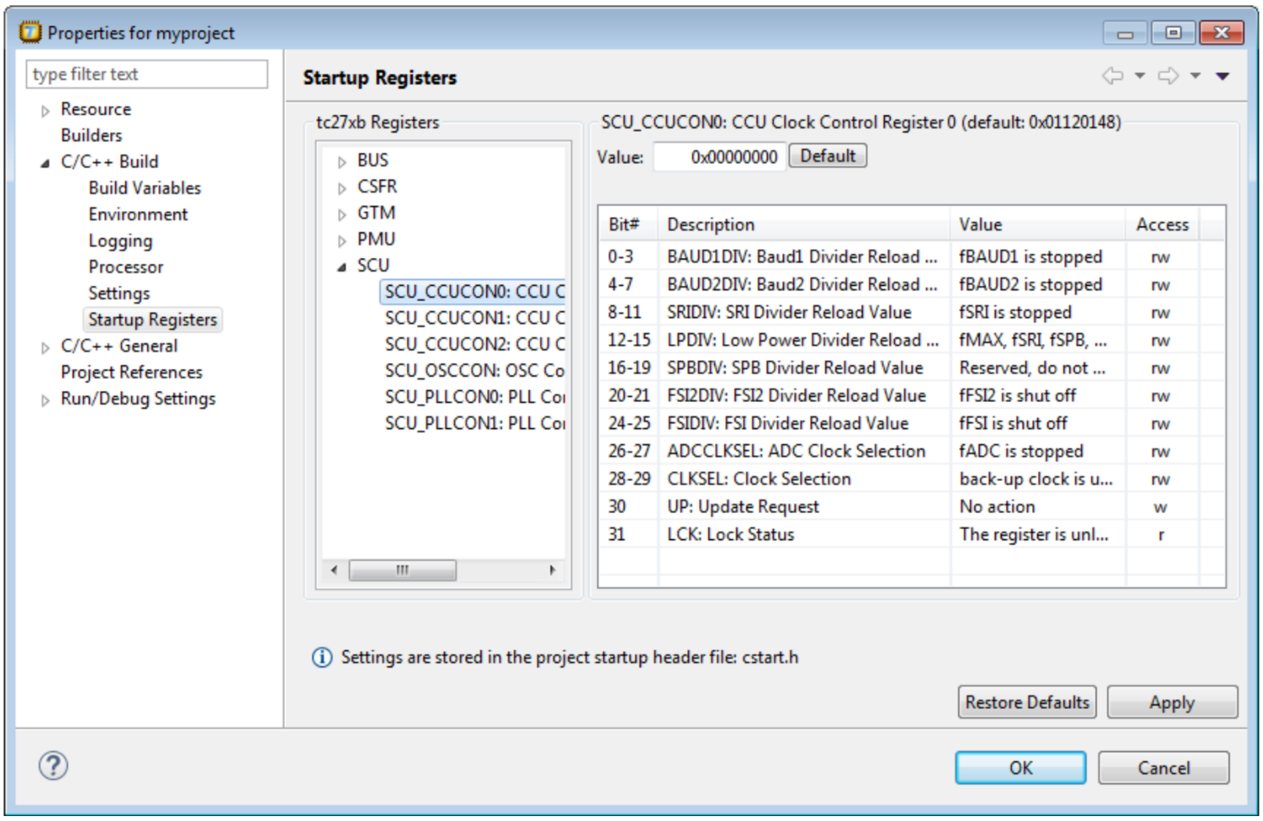
ECLIPSE based User Interface

The TASKING Embedded Debugger is integrated in the industry standard Eclipse IDE. By default the debugger is installed in a stand-alone TASKING Eclipse environment, but can also be integrated as a plugin into customer specific Eclipse Mars environments.

Eclipse offers software developers a standardized and familiar work environment that reduces the learning curve of a new tool. All productivity features of Eclipse are fully exploited by the Embedded Debugger. TASKING Embedded Debugger provides wizards to ease the often complex and tedious task of configuring the target system. Out-of-the box board support is available for development boards from Infineon and several 3rd parties. Elaborate user documentation including on-line context sensitive help and example projects are available to get you going.

All data that is needed to efficiently debug a program — optionally executing on multiple cores — such as active threads, program locations, stacks, value of variables and registers are shown simultaneously in multiple panes which are updated once a thread halts execution. Default window layouts, called perspectives, are provided but you also have great freedom to create layouts that cover your specific needs, which can be saved as an Eclipse perspective for later reuse.

Debugger commands can be entered via the mouse or through keyboard shortcuts. Using shortcuts is usually preferable as you can perform actions much faster. The debugger provides keyboard shortcuts for the most common actions.



SCU_CCUCON0: CCU Clock Control Register 0 (default: 0x01120148)

Value: 0x00000000

Bit#	Description	Value	Access
0-3	BAUD1DIV: Baud1 Divider Reload ...	fBAUD1 is stopped	rw
4-7	BAUD2DIV: Baud2 Divider Reload ...	fBAUD2 is stopped	rw
8-11	SRIDIV: SRI Divider Reload Value	fSRI is stopped	rw
12-15	LPDIV: Low Power Divider Reload ...	fMAX, fSRI, fSPB, ...	rw
16-19	SPBDIV: SPB Divider Reload Value	Reserved, do not ...	rw
20-21	FSI2DIV: FSI2 Divider Reload Value	fFSI2 is shut off	rw
24-25	FSIDIV: FSI Divider Reload Value	fFSI is shut off	rw
26-27	ADCCLKSEL: ADC Clock Selection	fADC is stopped	rw
28-29	CLKSEL: Clock Selection	back-up clock is u...	rw
30	UP: Update Request	No action	w
31	LCK: Lock Status	The register is unl...	r

Settings are stored in the project startup header file: cstart.h

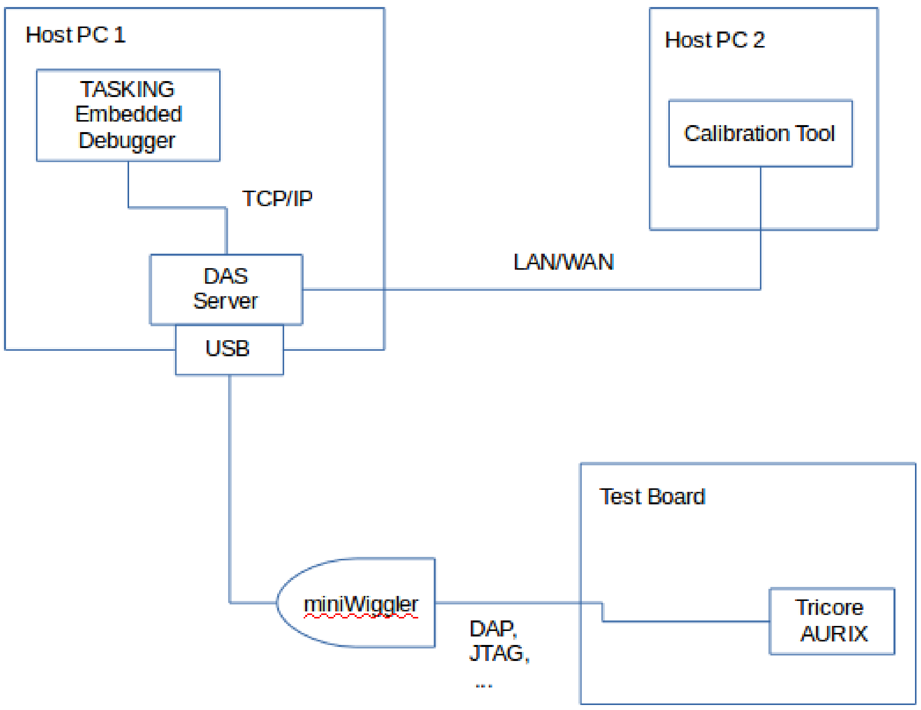
Configuration wizard showing values of device registers in symbolic format

Target Device Access and Tool Interoperability

The TASKING Embedded Debugger uses the on-chip debug features of the physical target, or connects to single core instruction-set simulators which are described in a separate section.

The connection to the physical target device is made via the Infineon miniWiggler, a device provided by Infineon to access the TriCore, AUDO Future, AUDO MAX, and AURIX™ devices, as well as upcoming microcontrollers. It supports the Device Access Port DAP2 protocol which allows debug communication with higher transmission rates than existing JTAG based communication channels.

The logical connection to the target is implemented through Infineon's DAS protocol, which is a proven technology broadly used by Infineon and their tool partners. DAS assures interoperability between the debugger and other tools that require device assess such as calibration tools. Furthermore it offers remote access enabling developers to connect from their office desk to a target device in the lab as shown in the figure below. This way, the device can also be time-shared among multiple developers.



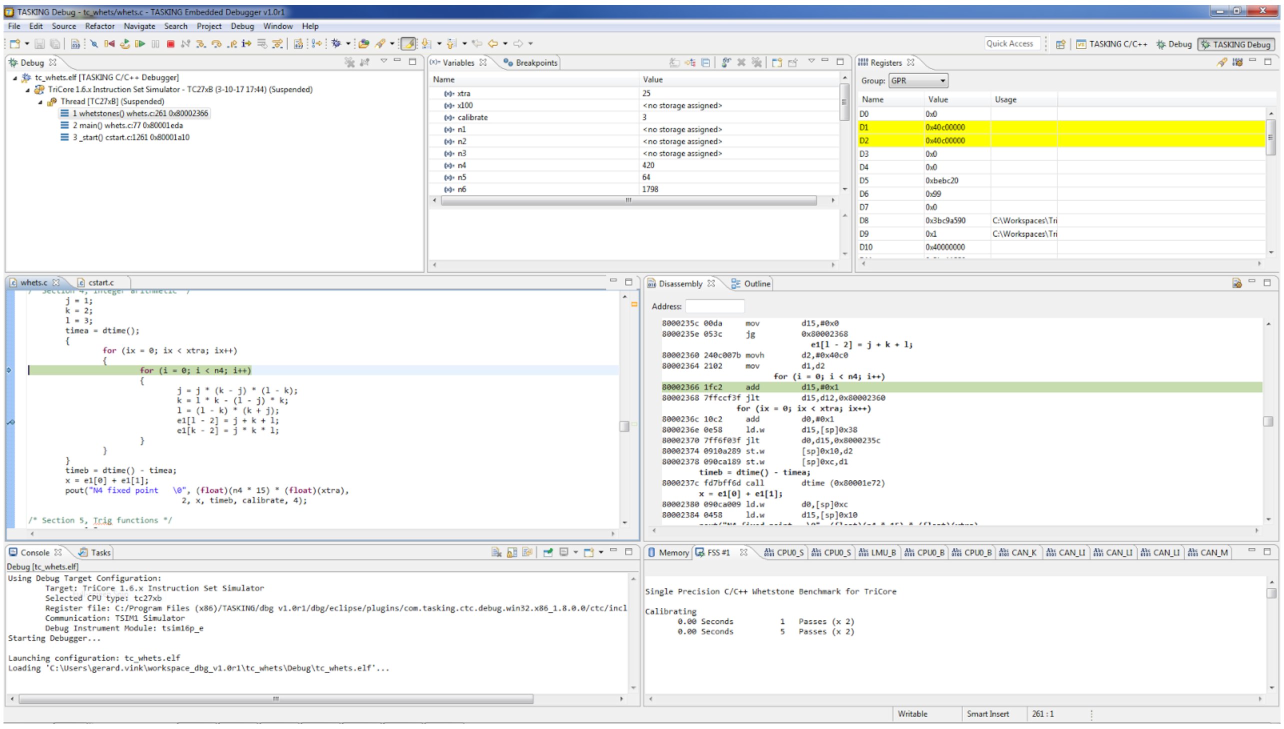
Remote multi-tool operation

C/C++ and Assembly Symbolic Debugging

Debugging can be done in C/C++ language, assembler, or in a mixture of both. High-level language can be intermixed with disassembly in one window, or can be shown in separate windows. All variable types specific to the high-level language can be displayed, modified, and used in expressions, including function calls. Addresses can be symbolic, absolute, or line number based. Symbolic C/C++ debugging of highly optimized code is also feasible.

The default debugger perspective (shown on page 2) provides access to the following symbolic information:

- The Debug pane provides access to the active cores, threads being executed, and their stack, and allows you to jump to the code associated with a function and set breakpoints on function return addresses.
- The Variables pane allows you to inspect and modify local and global variables -- also variables that are temporarily stored in one or multiple registers -- and set data breakpoints (aka watchpoints) on the memory location that a variable is stored in.
- The Breakpoints pane lets you add, inspect, and modify code and data breakpoints.
- The Register pane lets you inspect and modify the registers of the target device. This includes address and general purpose registers as well as device configuration registers also known as special function registers (SFRs). Values that have changed since last update are highlighted. Symbolic access of device configuration registers is supported and saves the developer from consulting the processors user's guide to translate bit patterns into something meaningful, which is error prone if done manually.
- The Source pane shows the current location within the high-level source code, provides quick inspection of the content of variables, and allows you to set breakpoints on source lines.
- The Disassembly pane shows the disassembled memory image optionally intermixed with the source code.
- The Memory pane lets you inspect and modify the content of the target's memory. The debugger can also list the differences between the content of the downloaded program file and the current memory image, providing you with the ability to e.g. detect self-modifying code.
- The Heap pane (not shown in default perspective) shows the content of the heap and lest you quickly debug memory allocation issues.



```
tc_whets.elf [TASKING C/C++ Debugger]
  TriCore 1.6.x Instruction Set Simulator - TC27xB (3-10-17 17:44) (Suspended)
    Thread [TC27xB] (Suspended)
      1 whetstones() whets.c:261 0x80002366
      2 main() whets.c:77 0x80001eda
      3 _start() cstart.c:1261 0x80001a10
```

Name	Value
xtra	25
x100	<no storage assigned>
calibrate	3
n1	<no storage assigned>
n2	<no storage assigned>
n3	<no storage assigned>
n4	420
n5	64
n6	1798

Group: GPR

Name	Value	Usage
D0	0x0	
D1	0x40c00000	
D2	0x40c00000	
D3	0x0	
D4	0x0	
D5	0xbebc20	
D6	0x99	
D7	0x0	
D8	0x3bc9a590	C:\Workspaces\Tri
D9	0x1	C:\Workspaces\Tri
D10	0x40000000	

```
        j = 1;
        k = 2;
        l = 3;
        timea = dtime();
        {
                for (ix = 0; ix < xtra; ix++)
                {
                        for (i = 0; i < n4; i++)
                        {
                                j = j * (k - j) * (l - k);
                                k = l * k - (l - j) * k;
                                l = (l - k) * (k + j);
                                e1[l - 2] = j + k + l;
                                e1[k - 2] = j * k * l;
                        }
                }
        }
        timeb = dtime() - timea;
        x = e1[0] + e1[1];
        pout("N4 fixed point   \0", (float)(n4 * 15) * (float)(xtra),
                        2, x, timeb, calibrate, 4);

/* Section 5, Trig functions */
```

Address:

```
8000235c 00da     mov        d15,#0x0
8000235e 053c     jg         0x80002368
                               e1[l - 2] = j + k + l;
80002360 240c007b movh       d2,#0x40c0
80002364 2102     mov        d1,d2
                     for (i = 0; i < n4; i++)
80002366 1fc2     add        d15,#0x1
80002368 7ffccf3f jlt        d15,d12,0x80002360
                for (ix = 0; ix < xtra; ix++)
8000236c 10c2     add        d0,#0x1
8000236e 0e58     ld.w       d15,[sp]0x38
80002370 7ff6f03f jlt        d0,d15,0x8000235c
80002374 0910a289 st.w       [sp]0x10,d2
80002378 090ca189 st.w       [sp]0xc,d1
        timeb = dtime() - timea;
8000237c fd7bff6d call       dtime (0x80001e72)
        x = e1[0] + e1[1];
80002380 090ca009 ld.w       d0,[sp]0xc
80002384 0458     ld.w       d15,[sp]0x10
```

```
Debug [tc_whets.elf]
Using Debug Target Configuration:
        Target: TriCore 1.6.x Instruction Set Simulator
        Selected CPU type: tc27xb
        Register file: C:/Program Files (x86)/TASKING/dbg v1.0r1/dbg/eclipse/plugins/com.tasking.ctc.debug.win32.x86_1.8.0.0/ctc/incl
        Communication: TSIM1 Simulator
        Debug Instrument Module: tsim16p_e
Starting Debugger...

Launching configuration: tc_whets.elf
Loading 'C:\Users\gerard.vink\workspace_dbg_v1.0r1\tc_whets\Debug\tc_whets.elf'...
```

```
Single Precision C/C++ Whetstone Benchmark for TriCore

Calibrating
       0.00 Seconds          1   Passes (x 2)
       0.00 Seconds          5   Passes (x 2)
```

Default debug perspective

Fast Program Download into RAM and FLASH

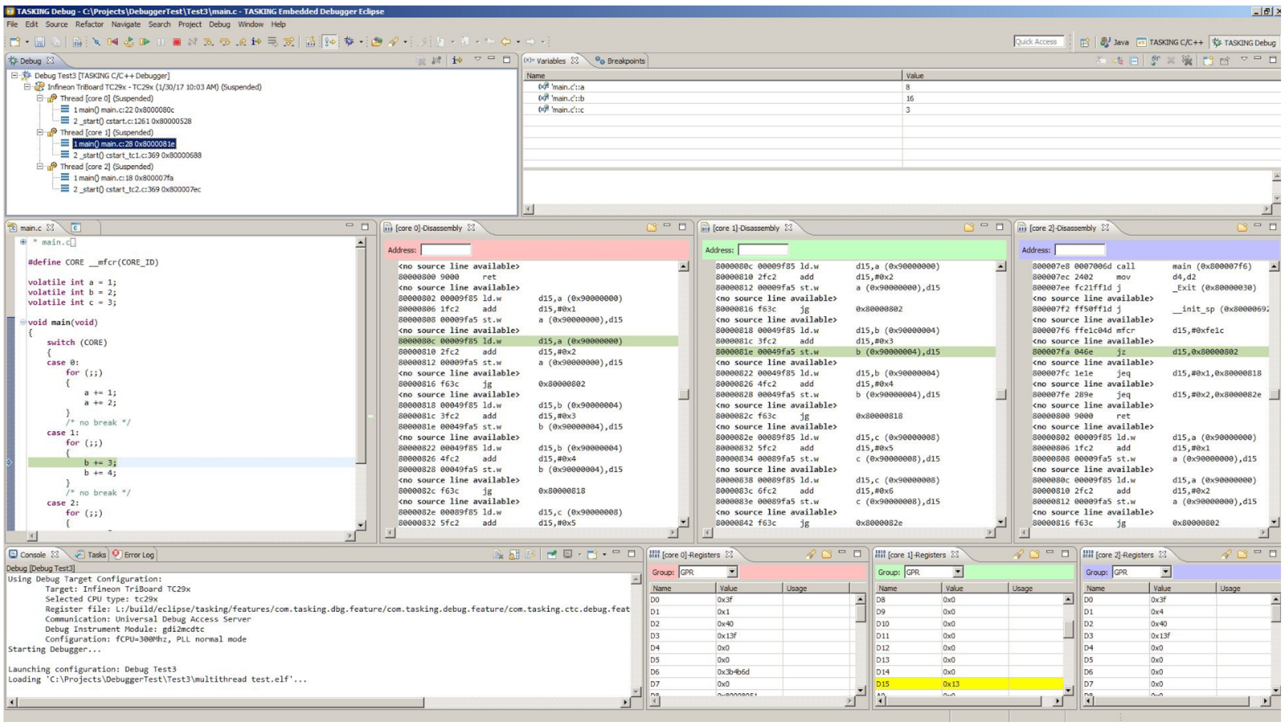
Program code and data can be downloaded into RAM and/or internal FLASH memory. The programming of the FLASH is controlled by the debugger. Customers value the speed at which the FLASH is programmed and that multiple files can be downloaded in one operation.

The downloaded code can be executed by any core, or combination of cores, on the device. Of course the code must comply with the initialization requirements of the device.

Multi-Core Debugging GUI

Multicore debugging for up to six TriCore cores is supported. The figure below shows a window pane layout that displays detailed information about one core/thread, and disassembly intermixed with source code of the other cores/threads.

The Debug pane shows the stacks of all threads in the system. Once the user selects a thread or function within a thread, all windows are automatically updated to show the state associated with that thread and the core on which it executes. Combined with the multi-core start/stop features described in the next section you have ample features to find and fix bugs caused by inter-core interactions.



```
Debug Test3 [TASKING C/C++ Debugger]
  Infineon TriBoard TC29x - TC29x (1/30/17 10:03 AM) (Suspended)
    Thread [core 0] (Suspended)
      1 main() main.c:22 0x8000080c
      2 _start() cstart.c:1261 0x80000528
    Thread [core 1] (Suspended)
      1 main() main.c:28 0x8000081e
      2 _start() cstart_tc1.c:369 0x80000688
    Thread [core 2] (Suspended)
      1 main() main.c:18 0x800007fa
      2 _start() cstart_tc2.c:369 0x800007ec
```

Name	Value
'main.c'::a	8
'main.c'::b	16
'main.c'::c	3

```c
 * main.c

#define CORE __mfcr(CORE_ID)

volatile int a = 1;
volatile int b = 2;
volatile int c = 3;

void main(void)
{
    switch (CORE)
    {
    case 0:
        for (;;)
        {
            a += 1;
            a += 2;
        }
        /* no break */
    case 1:
        for (;;)
        {
            b += 3;
            b += 4;
        }
        /* no break */
    case 2:
        for (;;)
        {
```

[core 0]-Disassembly

```
<no source line available>
80000800 9000     ret
<no source line available>
80000802 00009f85 ld.w        d15,a (0x90000000)
80000806 1fc2     add         d15,#0x1
80000808 00009fa5 st.w        a (0x90000000),d15
<no source line available>
8000080c 00009f85 ld.w        d15,a (0x90000000)
80000810 2fc2     add         d15,#0x2
80000812 00009fa5 st.w        a (0x90000000),d15
<no source line available>
80000816 f63c     jg          0x80000802
<no source line available>
80000818 00049f85 ld.w        d15,b (0x90000004)
8000081c 3fc2     add         d15,#0x3
8000081e 00049fa5 st.w        b (0x90000004),d15
<no source line available>
80000822 00049f85 ld.w        d15,b (0x90000004)
80000826 4fc2     add         d15,#0x4
80000828 00049fa5 st.w        b (0x90000004),d15
<no source line available>
8000082c f63c     jg          0x80000818
<no source line available>
8000082e 00089f85 ld.w        d15,c (0x90000008)
80000832 5fc2     add         d15,#0x5
```

[core 1]-Disassembly

```
8000080c 00009f85 ld.w        d15,a (0x90000000)
80000810 2fc2     add         d15,#0x2
80000812 00009fa5 st.w        a (0x90000000),d15
<no source line available>
80000816 f63c     jg          0x80000802
<no source line available>
80000818 00049f85 ld.w        d15,b (0x90000004)
8000081c 3fc2     add         d15,#0x3
8000081e 00049fa5 st.w        b (0x90000004),d15
<no source line available>
80000822 00049f85 ld.w        d15,b (0x90000004)
80000826 4fc2     add         d15,#0x4
80000828 00049fa5 st.w        b (0x90000004),d15
<no source line available>
8000082c f63c     jg          0x80000818
<no source line available>
8000082e 00089f85 ld.w        d15,c (0x90000008)
80000832 5fc2     add         d15,#0x5
80000834 00089fa5 st.w        c (0x90000008),d15
<no source line available>
80000838 00089f85 ld.w        d15,c (0x90000008)
8000083c 6fc2     add         d15,#0x6
8000083e 00089fa5 st.w        c (0x90000008),d15
<no source line available>
80000842 f63c     jg          0x8000082e
```

[core 2]-Disassembly

```
800007e8 0007006d call        main (0x800007f6)
800007ec 2402     mov         d4,d2
800007ee fc21ff1d j           _Exit (0x80000030)
<no source line available>
800007f2 ff50ff1d j           __init_sp (0x80000692
<no source line available>
800007f6 ffe1c04d mfcr        d15,#0xfe1c
<no source line available>
800007fa 046e     jz          d15,0x80000802
<no source line available>
800007fc 1e1e     jeq         d15,#0x1,0x80000818
<no source line available>
800007fe 289e     jeq         d15,#0x2,0x8000082e
<no source line available>
80000800 9000     ret
<no source line available>
80000802 00009f85 ld.w        d15,a (0x90000000)
80000806 1fc2     add         d15,#0x1
80000808 00009fa5 st.w        a (0x90000000),d15
<no source line available>
8000080c 00009f85 ld.w        d15,a (0x90000000)
80000810 2fc2     add         d15,#0x2
80000812 00009fa5 st.w        a (0x90000000),d15
<no source line available>
80000816 f63c     jg          0x80000802
```

Debug [Debug Test3]

```
Using Debug Target Configuration:
        Target: Infineon TriBoard TC29x
        Selected CPU type: tc29x
        Register file: L:/build/eclipse/tasking/features/com.tasking.dbg.feature/com.tasking.debug.feature/com.tasking.ctc.debug.feat
        Communication: Universal Debug Access Server
        Debug Instrument Module: gdi2mcdtc
        Configuration: fCPU=300Mhz, PLL normal mode
Starting Debugger...

Launching configuration: Debug Test3
Loading 'C:\Projects\DebuggerTest\Test3\multithread test.elf'...
```

[core 0]-Registers, Group: GPR

Name	Value	Usage
D0	0x3f	
D1	0x1	
D2	0x40	
D3	0x13f	
D4	0x0	
D5	0x0	
D6	0x3b4b6d	
D7	0x0	

[core 1]-Registers, Group: GPR

Name	Value	Usage
D8	0x0	
D9	0x0	
D10	0x0	
D11	0x0	
D12	0x0	
D13	0x0	
D14	0x0	
D15	0x13	

[core 2]-Registers, Group: GPR

Name	Value	Usage
D0	0x3f	
D1	0x4	
D2	0x40	
D3	0x13f	
D4	0x0	
D5	0x0	
D6	0x0	
D7	0x0	

Multi-Core perspective showing state of cores 0, 1 and 2

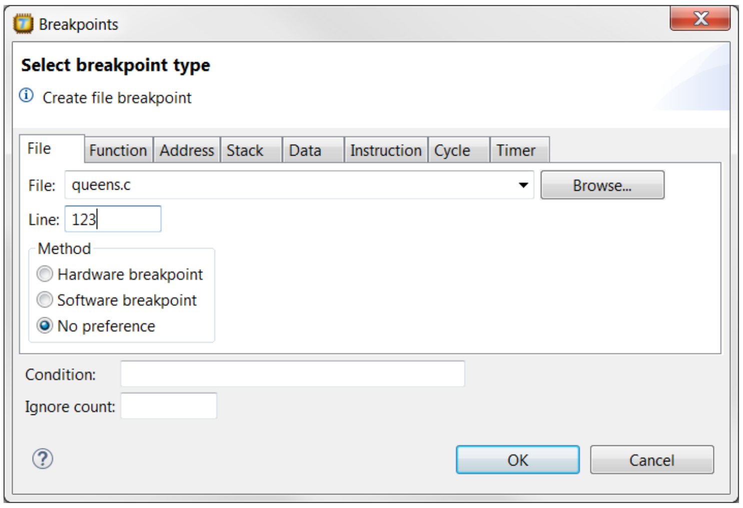
Elaborate Run-Control with Multi-Core Start/Stop Synchronization

Elaborate run-control features are available and include: reset & run, restart, resume, C/C++ as well as instruction-level stepping, step into, step over, return from current function, and interrupt aware stepping, whereby the user can specify whether a run-control command affects all cores or one specific core only.

The interrupt aware stepping feature is quite useful, allowing you to step through a thread, while interrupts are processed in the background, without stepping into the interrupt handler.

Code Breakpoints and Data Watchpoints

Breakpoints and watchpoints are used to halt the program at critical junctures of program execution to allow the developer to acquire knowledge about a program during its execution. A code breakpoint halts the program when an instruction is fetched from a given address. A data breakpoint, or watchpoint, halts the program if data is read from and/or written to a specific memory location. Data breakpoints allow you to easily track the use and misuse of variables without having to (single)step through the code. In addition to code and data breakpoints the TASKING Embedded Debugger also supports breakpoints associated with elapsed time expressed as clock ticks, CPU cycles, or instructions executed.


Breakpoints

Select breakpoint type

Create file breakpoint

File: queens.c

Line: 123

Method
- ( ) Hardware breakpoint
- ( ) Software breakpoint
- (●) No preference

Condition:

Ignore count:

Breakpoint configuration dialog showing multitude of breakpoint types

The number of available on-chip break- and watchpoints depends on the resources of the device used. An unlimited number of software breakpoints can be set in RAM. A condition can be linked to a break- or watchpoint to halt the program only when the expression evaluates to true.

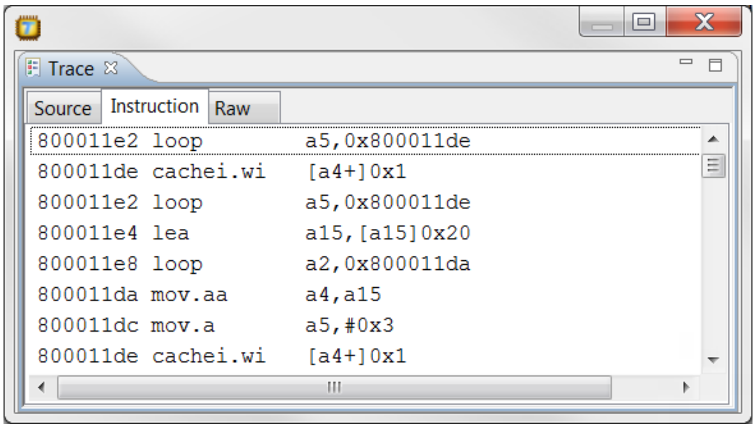
Autosar-OS Aware Debugging

The TASKING Embedded Debugger has built-in support for real-time operating systems (RTOS) aware debugging, and provides symbolic access to the operating system's internal resources like the task list and message queues. Support for OSEK / ISO 17356 compliant operating systems is available out-of-the-box and is setup for your specific OS implementation via the OSEK Run Time Interface (ORTI).

The ORTI is a specification, which enables debuggers to become OS aware, without knowing the OS itself. Most AUTOSAR/OSEK system builders are able to extract all necessary information of the OS component into a text file, called "ORTI file", which can be loaded by the debugger.

Basic Trace Support

The debugger has a separate window that displays the most recently executed C statements or machine instructions. This feature uses the execution target device's trace buffer along with symbolic information generated during compilation and is supported by AURIX emulation devices and most simulators. The trace window is updated automatically each time the target halts.



```
800011e2 loop       a5,0x800011de
800011de cachei.wi  [a4+]0x1
800011e2 loop       a5,0x800011de
800011e4 lea        a15,[a15]0x20
800011e8 loop       a2,0x800011da
800011da mov.aa     a4,a15
800011dc mov.a      a5,#0x3
800011de cachei.wi  [a4+]0x1
```

Instruction trace window showing the program's execution

File System Virtualization

The file system simulation capability offers a means to easily move data from the board to the host system for further analysis, enabling you to debug programs before the actual input and output devices are present.
This features uses functionality built into the TASKING C-library to redirect all file IO calls executed on the target to the host system. The debugger can read input data from the keyboard or a file, or can send output to a window or a file offering various ways to visualize the data.

Test and Debug Automation Using TASKING Script Debugger

Next to the Eclipse Debugger, also a command line debugger is included in the package. This Script Debugger is not an interactive debugger but merely a tool to drive test automation. Regression testing and debugging operations are sequenced via a user specified script written in an easy to lean yet powerful proprietary language. All aforementioned features such as file system virtualization are supported by the Script Debugger.

Single-Core Instruction Set Simulators

Performance optimized single-core instruction set simulators are provided for all cores located on Infineon TriCore, AUDO and AURIX devices encompassing: TriCore, Generic Timer Module (GTM/MCS), Standby controller (SCR), Hardware Security Module (HSM) and Peripheral Co-Processor (PCP) cores.

The simulators allow you to debug your program prior to obtaining target hardware. In addition to executing code, the simulators gather statistics and timing information about your code providing a basic means for hotspot analysis & instruction usage and facilitating relative performance measurement between different versions of an algorithm. The simulators also support trace functionality.

PRODUCT SPECIFICATIONS

Supported host systems:

- Windows 7 or higher

Supported TriCore Devices:

- TC11xx Family (TC1130, TC1164, TC1166, TC1167, TC1197)
- TC173x Family (TC1736)
- TC176x Family (TC1762, TC1766, TC1767)
- TC178x Family (TC1782, TC1784)
- TC179x Family (TC1791, TC1792, TC1793, TC1796, TC1797, TC1798)
- AURIX TC2xx Family (TC21X, TC22X, TC23X, TC26X, TC27X, TC29X)
- AURIX TC3xx Family

Supported Debug Probes and target Connectors:

- Infineon miniWiggler
- Automotive 20-pin JTAG on 10-pin DAP

Supported Single-core Instruction Set Simulators

- TriCore, GTM/MCS, SCR, HSM and PCP (all included in package)

SUMMARY

The TASKING Embedded Debugger is a cost-efficient, yet complete, solution for code verification in large development teams. For the price of a single high-end debugger, with functionality that is only required for a small percentage of the time, several streamlined debuggers can be purchased, enabling developers to identify logic problems and correct coding errors early on.

REFERENCES

[1] DAS Device Access Server - available via www.infineon.com/DAS

[2] MCD Multi-Core Debug API - available via www.infineon.com/DAS

[3] DAP miniWiggler - available via www.infineon.com/DAS