Integration for eXDI2 on Windows CE Platform Builder

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ....

3rd-Party Tool Integrations ....

Integration for eXDI2 on Windows CE Platform Builder 1

Overview 2

Concept of hardware-assisted debugging 3

How hardware-assisted debugging modifies eXDI Architecture? 4

Driver installation and configuration 6

Getting necessary files 9

Creating OS Design 10

Downloading Windows CE image to target and booting system 15

Adding example application to Windows CE image 18

Debugging Windows CE 21

Loading EXE/DLL modules symbols in TRACE32 21

Preparing Windows CE image 22

Driver configuration 22

Debugging session 23

Debugging hardware bring-up 29

Hardware-assisted debugging and KITL 31

Using TRACE32 FDX for KITL Kernel Transport 32

FDX Overview 33

Architecture of KITL over FDX 33

Enabling KITL over FDX 34

Download service 37

Debugging timings 38

Memory caching 38

Troubleshooting 39

©1989-2020 Lauterbach GmbH

Integration for eXDI2 on Windows CE Platform Builder

Version 18-Nov-2020

Overview

Microsoft Platform Builder for Windows CE contains an interface that allows the Platform Builder (PB) internal debugger to drive external hardware debuggers. This interface is called eXdi2. Lauterbach developed an eXdi2 driver that allows the PB internal debugger to use TRACE32 as a hardware backend to the target.

NOTE:	This integration uses internally the **TRACE32 Remote API**. The Remote API has **restrictions** if TRACE32 runs in demo mode. Please see there for further details.

©1989-2020 Lauterbach GmbH

Concept of hardware-assisted debugging

Platform Builder enables you to use Extended Debugging Interface (eXDI) for hardware-assisted debugging to control the execution of a target device and to examine and modify the state of the device.

Hardware-assisted debugging enables debugging not supported by the default debugger in Platform Builder. For example, you can use hardware-assisted debugging to debug code used in hardware bring-up, boot loading and to debug execution that occurs prior to the start of the kernel.

Lauterbach provides the required hardware and software to perform hardware-assisted debugging in Platform Builder.

Hardware-assisted debugging extends the debugging capabilities of Platform Builder beyond the potential of traditional software debuggers. Because hardware-assisted debugging is independent of the OS, you can, for example, isolate low-level problems that may arise in drivers and OAL code.

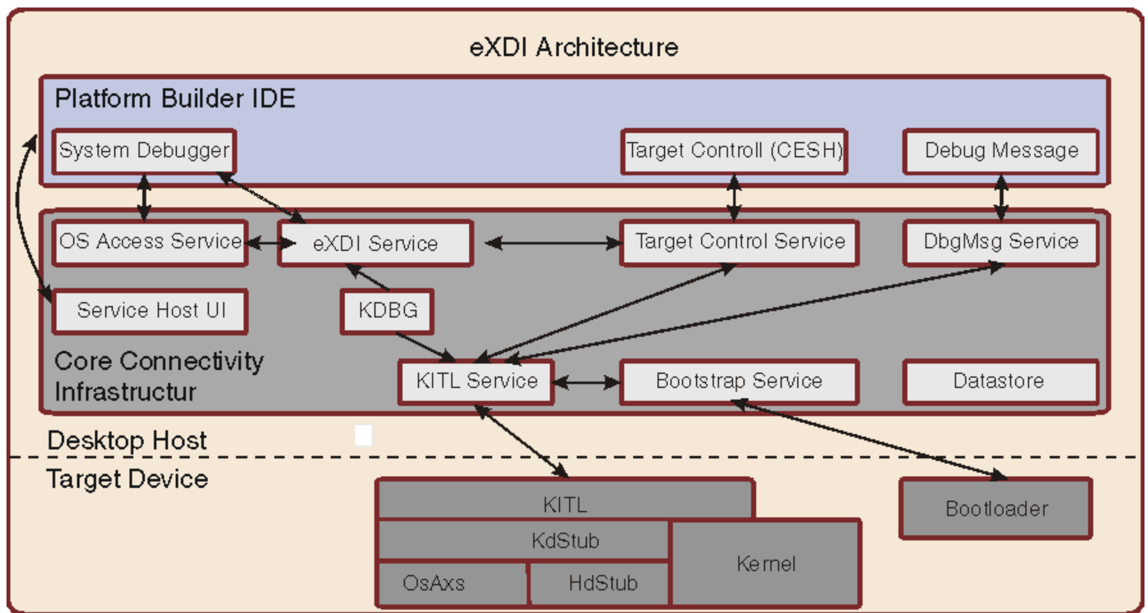
You can also use hardware-assisted debugging to debug other kinds of code, such as drivers and applications with the same ease of use as with the Platform Builder kernel debugger, the software probe solution for system-wide debugging or with native application debuggers.

For detailed information about benefits of eXDI hardware-assisted debugging, please refer to following location: http://msdn2.microsoft.com/en-us/library/aa935533.aspx

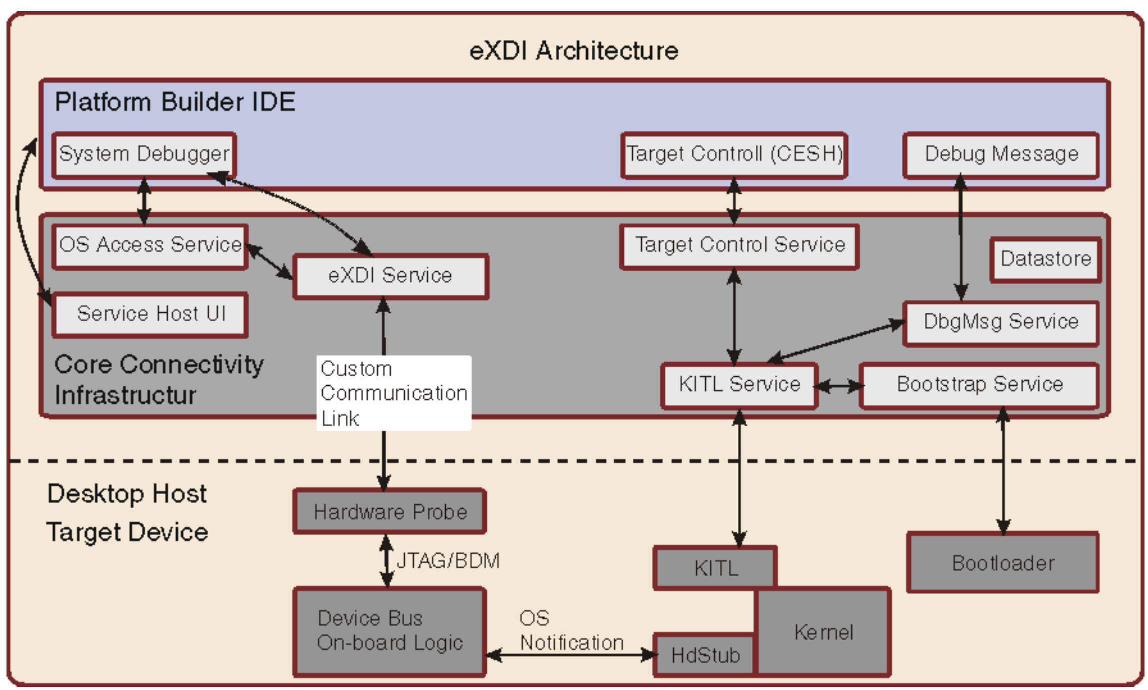
©1989-2020 Lauterbach GmbH

How hardware-assisted debugging modifies eXDI Architecture?

Standard eXDI architecture with Kernel Debugger (KdStub):



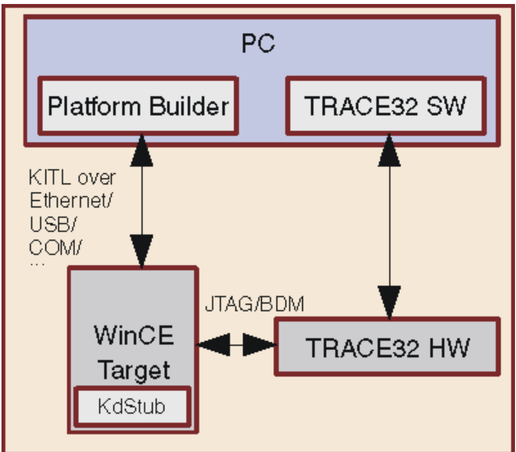
Hardware-assisted debugging (without Kernel Debugger):



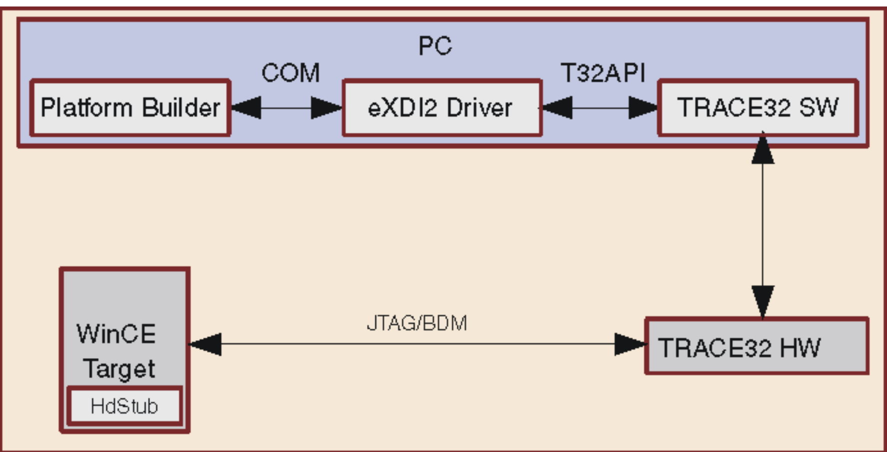
©1989-2020 Lauterbach GmbH

Above diagrams show that hardware-assisted debugging changes the way of “EXDI Service” communication with Target Device. Below pictures show the general difference:

Standard eXDI architecture with Kernel Debugger (KdStub):



Hardware-assisted debugging (without Kernel Debugger):



Using hardware-assisted debugging allows to remove KITL (Kernel Independent Transport Layer) connection from the WinCE target. Debugging features such as Break, Go, all kinds of steps, memory dumps, watches and breakpoints are still available as with KITL. Additionally, it is possible to debug all kind of code that is not available in standard debugging using “Kernel Debugger”. This includes hardware bring-up, boot loading, and code execution that occurs prior to the start of the kernel.

It is also possible to keep KITL connection. In this case KITL can be used for launching applications, downloading files, etc.

©1989-2020 Lauterbach GmbH

Driver installation and configuration

Follow these steps to install the eXdi2 intergration driver:

1. Close Platform Builder.

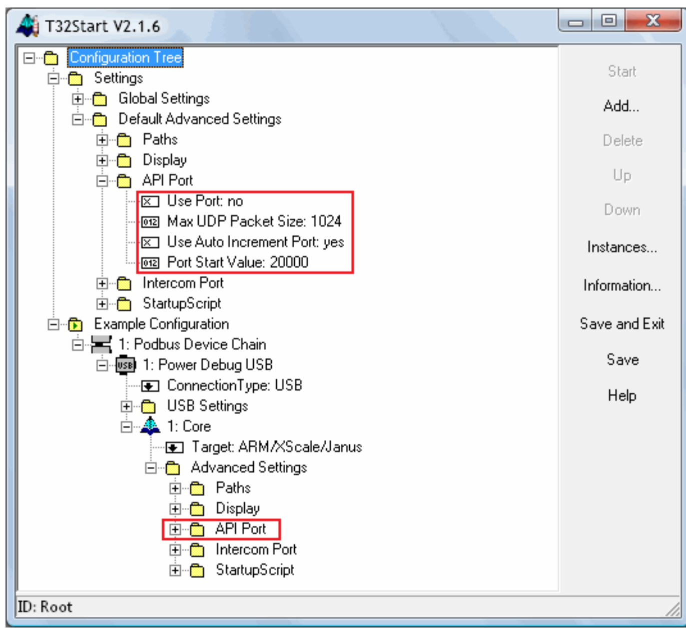
2. Close TRACE32.

3. Start installation program of the eXdi2 driver - t32exdi2_setup_CEn00.msi. This name can vary for different eXdi2 drivers provided for different Windows CE Platform Builder versions.

   Please make sure that you have installed MS Platform Builder first.

4. Driver communicates with TRACE32 through T32API over UDP local port.

   Please choose free port (for example 20000) and set it as below.



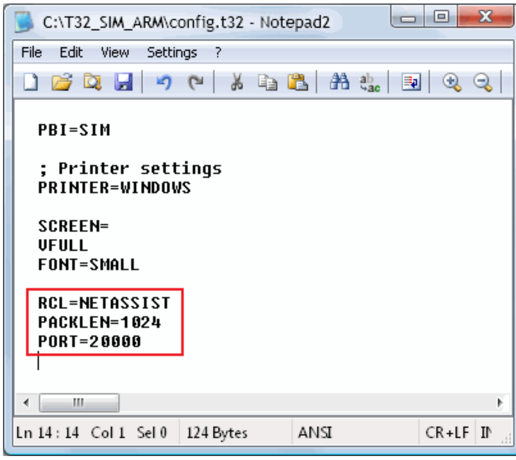
In case of using more than one TRACE32 simultaneously, please remember to set different API port numbers for each configuration.

If, for some reason, T32Start is not used, API port number can be set in “config.t32” configuration file. This file can be found in local TRACE32 directory. Please add following lines to this file:

```
RCL=NETASSIST
PACKLEN=1024
PORT=20000
```

Due to internal issues, please place empty lines between added section and other sections (entries) in this file.

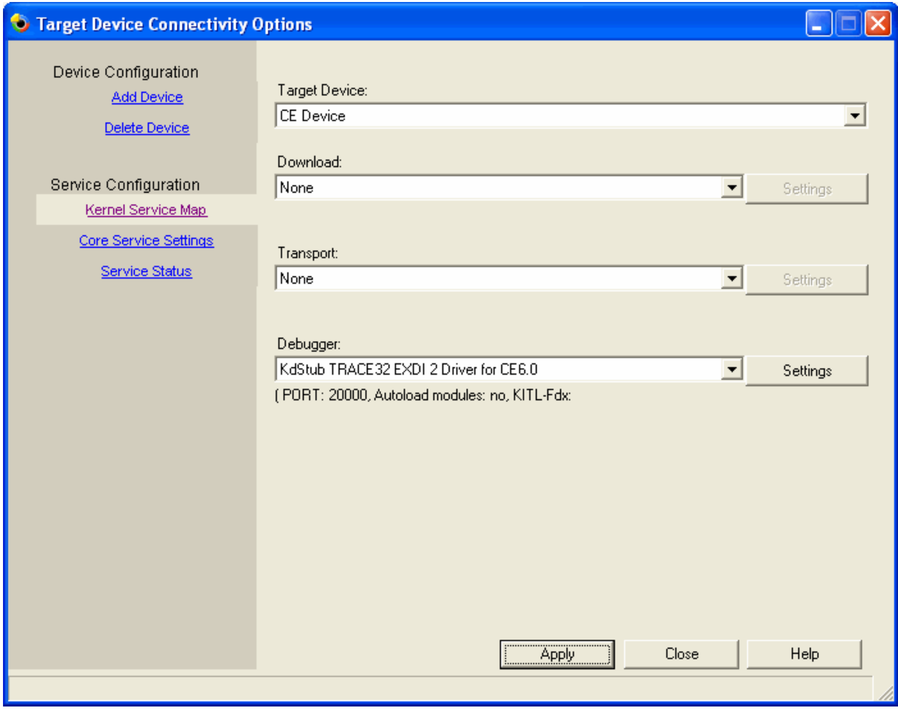
©1989-2020 Lauterbach GmbH



5. Start Platform Builder.

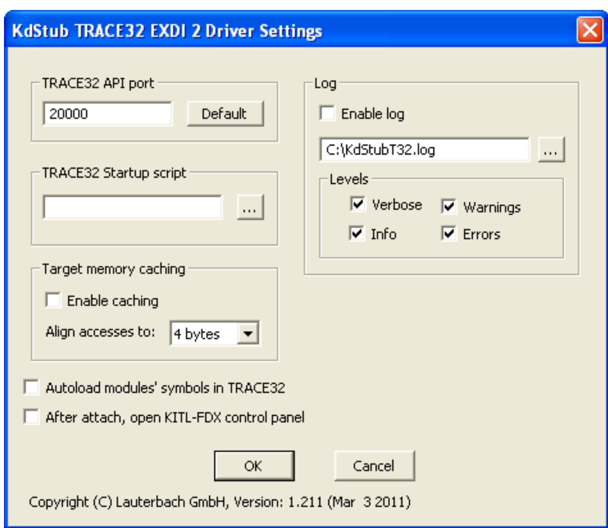
6. Select “Connectivity Options” command from “Target” menu to open “Target Device Connectivity options” dialog box.

7. Select “Kernel Service Map” option in the “Service Configuration” section in the control panel on the left side of the dialog box. Set “Kernel Download”, “Kernel Transport” and “Kernel Debugger” services to the values shown on below screenshot.



8. Click “Settings” button to configure “Kernel Debugger” service:

©1989-2020 Lauterbach GmbH



Set or change “TRACE32 API port” (exactly like in T32Start or ’config.t32’ file). By pressing “Default” button, you will always restore the default port number 20000.

Other options will be explained later. Please, leave them unchanged.

Press “OK” to save changes.

9. Press “Apply” in “Target Device Connectivity options” dialog box to save changes.

©1989-2020 Lauterbach GmbH

Getting necessary files

This documentation uses TRACE32SIMARM BSP for TRACE32 ARM Instruction Set Simulator, created by Lauterbach to show the capabilities of Windows CE debugging.

The download script described in chapter **“Downloading Windows CE image to target and booting system”** (int_exDI2.pdf) uses several files additionally (virtual hardware library, autoloader script and WinCE awareness files).

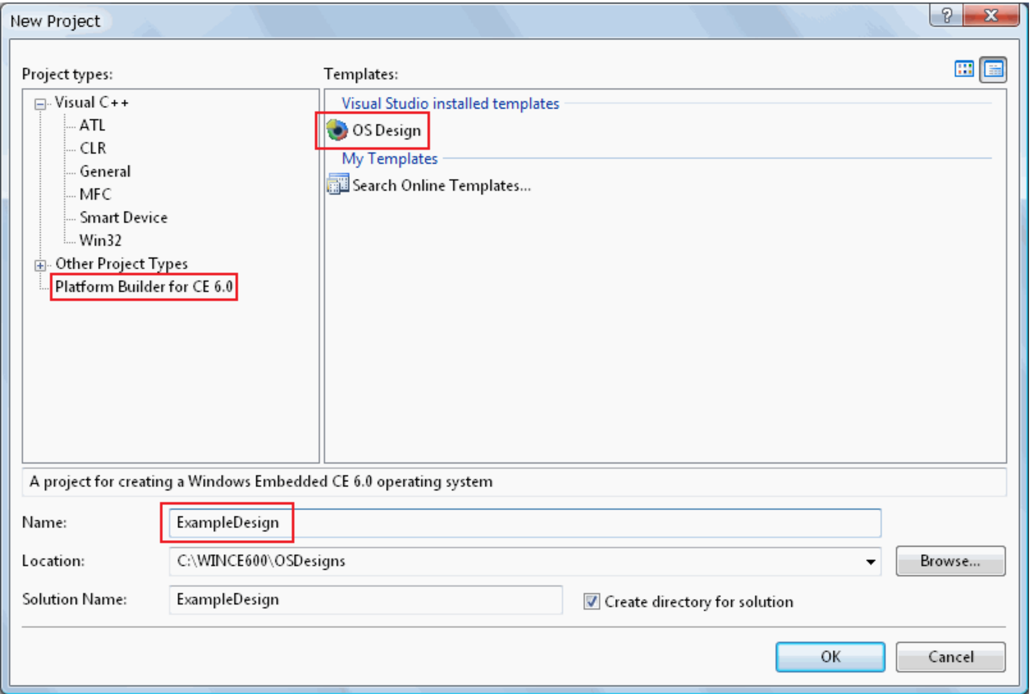
To get access to the TRACE32 Simulator BSP and the example files used in this guide, send an email with your specific request to: rtoswince-support@lauterbach.com

©1989-2020 Lauterbach GmbH

Creating OS Design

1. Start Platform Builder (for Windows CE6, Platform Builder is a plug-in of Visual Studio 2005).

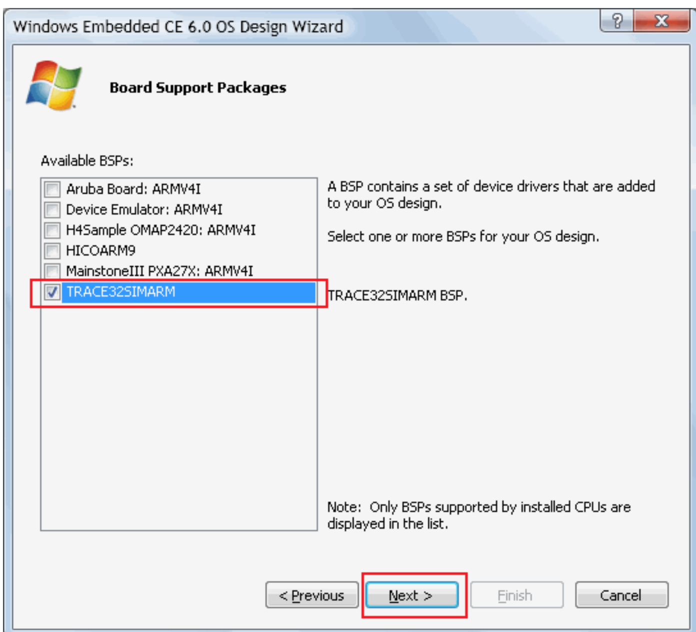
2. From menu, select File->New->Project. Select project type “Platform Builder for CE 6.0”, template “OS Design” and name “ExampleDesign”. Click OK.



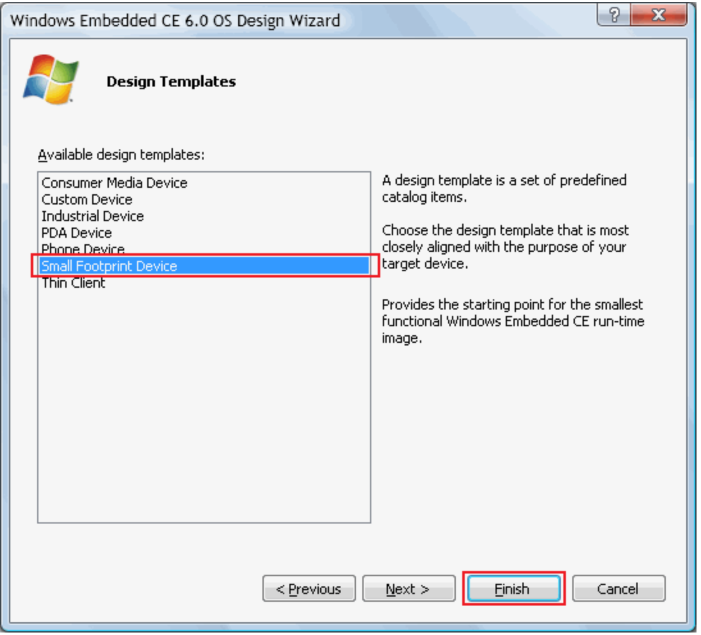
3. In welcome window of OS Design Wizard click “Next”.

©1989-2020 Lauterbach GmbH

4. Select BSP and click next.



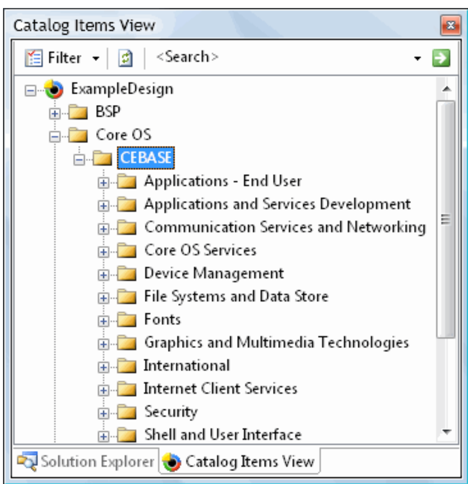
5. As design template select “Small footprint device” and click “Finish”. OS Design will be created.



6. From menu select View->Other Windows->Catalog Items View.

   From item “Core OS -> CEBASE” select below components:

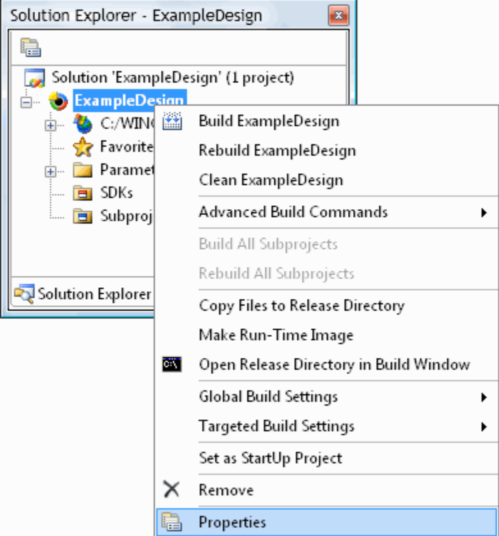
©1989-2020 Lauterbach GmbH



©1989-2020 Lauterbach GmbH

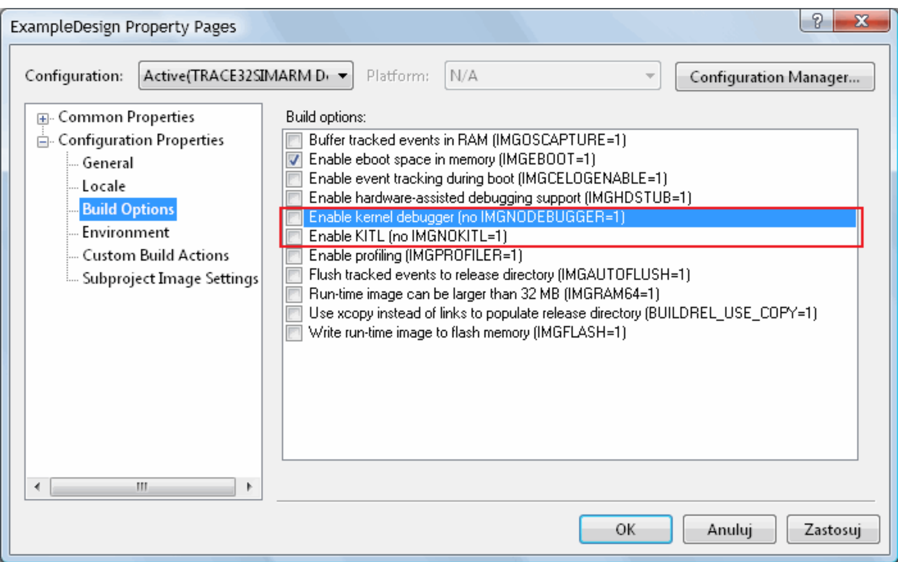
Core OS Services / Display Support
Core OS Services / Kernel Functionality / Target Control Support
File Systems and Data Store / File and Database Replication / Bit-based
File Systems and Data Store / File System - Internal / RAM and ROM File System
File Systems and Data Store / Registry Storage / Hive-based Registry
Graphics and Multimedia Technologies / Graphics / Gradient Fill Support
Shell and User Interface / Shell / AYGShell API Set
Shell and User Interface / Shell / Command Shell / Console Window
Shell and User Interface / Shell / Graphical Shell / Standard Shell
Shell and User Interface / User Interface / Overlapping Menus

7. In “Solution Explorer” right-click “ExampleDesign” project and select “Properties” from context menu.

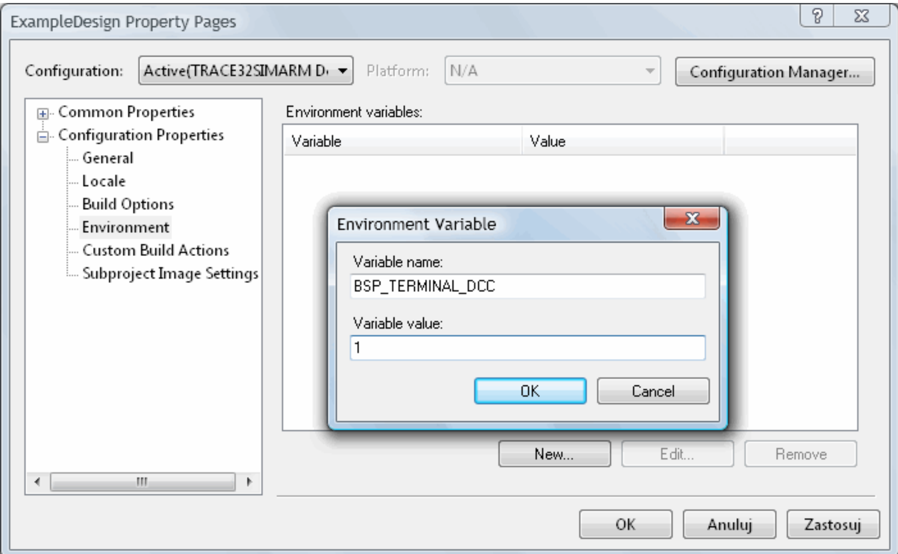


©1989-2020 Lauterbach GmbH

8. In “Build Options” tab of ExampleDesign properties dialog disable Kernel Debugger and KITL.



9. In “Environment” tab of ExampleDesign properties dialog add variable BSP_TERMINAL_DCC=1. This variable turns on TRACE32 Terminal support and is specific for used BSP.



10. At this point OS Design is ready to perform Sysgen. From menu select “Build -> Advanced Build Commands -> Sysgen”.

©1989-2020 Lauterbach GmbH

Downloading Windows CE image to target and booting system

In previous chapter OS was sysgened and Windows CE image was created at the and of sysgen process. This chapter describes how to download OS image to target without using download service provided by Platform Builder.

To download and boot Windows CE image, a PRACTICE script needs to be created. PRACTICE is a scripting language used by TRACE32. Please refer to documentation of TRACE32 for detailed information about PRACTICE commands.

Below script (wince.cmm) downloads Windows CE image to target (in this case, TRACE32 Instruction Set Simulator) and boots OS until it reaches OEMIdle() function. Target is stopped at OEMIdle and further debugging, using TRACE32 interface, can be performed.

```
; Set build directory localization and physical/virtual addresses of OS image


&build_directory="C:\WINCE600\OSDesigns\ExampleDesign\ExampleDesign\RelDir\TRACE32SIMARM_ARMV4I_Debug"

    &physical=0x20000000
    &virtual=0x84000000

; Debugger Reset

    screen.always
    winpage.reset
    area.reset
    WINPOS 0. 25. 84. 8. 0. 0.
    area

    print "resetting..."

    RESet
    SIM.UNLOAD

; setup of Debugger

    print "initializing..."

    SYStem.CPU ARM926EJ

    SIM.LOAD virtual_hardware.dll 0xFF000000 520. 300. 1 2 3 0

    SYStem.Option DACR ON          ; give Debugger global write permissions
    TrOnchip.Set DABORT OFF        ; used by wince for page miss!
    TrOnchip.Set PABORT OFF        ; used by wince for page miss!
    TrOnchip.Set UNDEF OFF         ; used to detect not present FPU
    SYStem.Option MMUSPACES ON     ; enable space IDs to virtual addresses
    SETUP.IMASKASM ON              ; lock interrupts while single stepping

    SYStem.Up

    SIM.CACHE.ON
    SIM.CACHE.SETS DC 0
    SIM.CACHE.SETS IC 0

; Target Setup: initialize DRAM controller and peripherals

    print "target setup..."

    ; set CP15 registers
    PER.SET C15:0x0    %LONG 0x41069263    // identity code
    PER.SET C15:0x100  %LONG 0x1D112152    // cache type
    PER.SET C15:0x1    %LONG 0x5727E       // cache control (round robin)
```

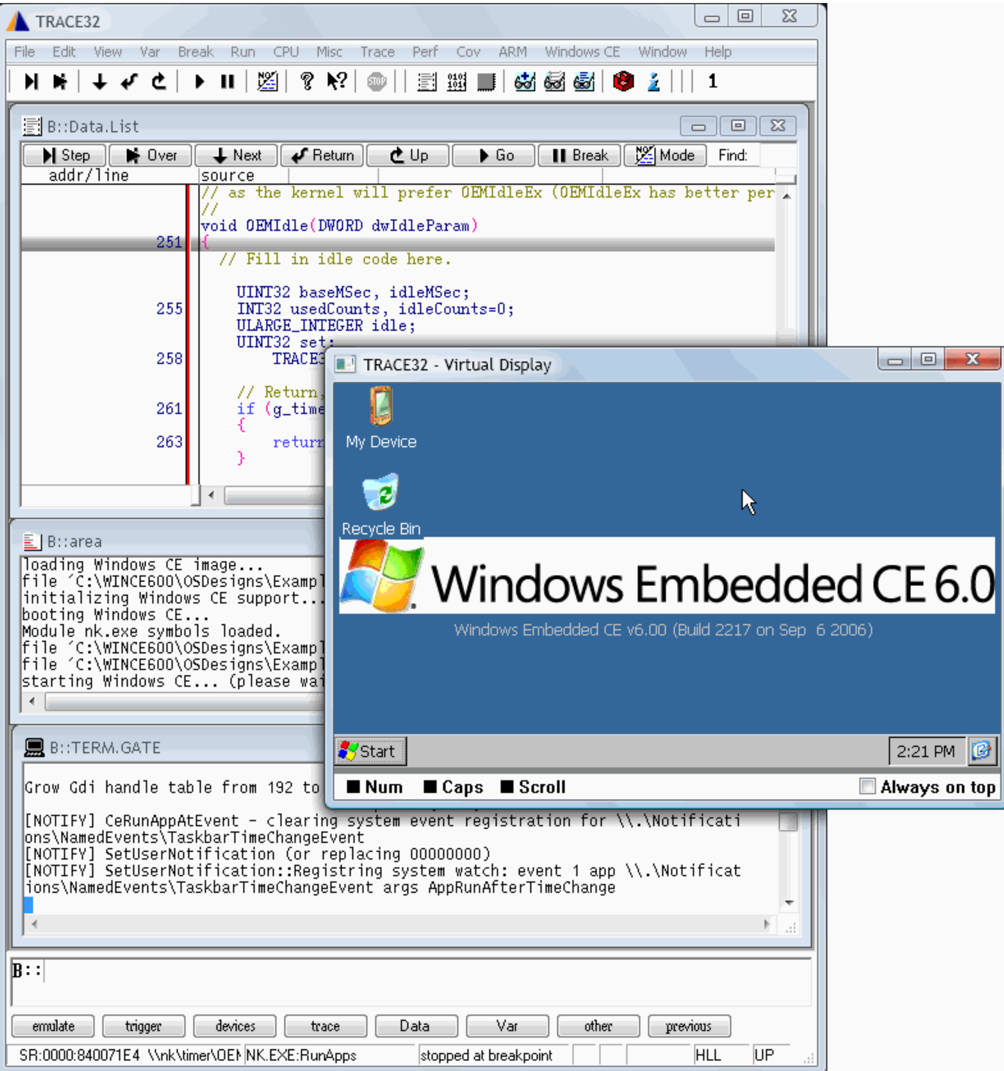
©1989-2020 Lauterbach GmbH

```
; Load the Windows CE image

    print "loading Windows CE image..."

    &offset=0x1000

    ; download the image to physical address
    Data.LOAD.EXE &build_directory\nk.bin &physical-&virtual

    ; set PC to physical start address
    Register.Set pc &physical+&offset

    ; We'd like to see something, open a code window.
    WINPOS 0. 0. 84. 19. 20. 1.
    Data.List


; Declare the MMU format to the debugger

    ; table format is "WINCE6"
    ; skip root table (0)
    ; declare default translation for kernel
    MMU.FORMAT WINCE6 0 &virtual++0x07ffffff &physical

    ; ROM DLL, shared heap and kernel addresses are common to all processes
    MMU.COMMON 0x40000000--0x5fffffff||0x70000000--0xffffffff

    ; debugger uses a table walk to decode virtual addresses
    MMU.TableWalk ON

    ; switch on debugger(!) address translation
    MMU.ON

; Initialize RTOS Support

    print "initializing Windows CE support..."
    TASK.CONFIG wince6             ; loads WinCE awareness (wince6.t32)
    MENU.ReProgram wince6          ; loads WinCE menu (wince6.men)
    HELP.FILTER.Add rtoswince      ; add WinCE awareness manual to help

    ; switch on autoloader and add path to symbol files to source path list
    sYmbol.AutoLOAD.CHECKWINCE "do "+OS.PresentPracticeDirectory+"/autoload "
    sYmbol.SourcePATH &build_directory     ; for symbol files (dll/pdb)
    sYmbol.SourcePATH C:\WINCE600          ; for source files (c/cpp)

    ; Group kernel area to be displayed with red bar
    GROUP.Create "winceos" 0x80000000--0xffffffff /RED

; Open debug output terminal

    TERM.METHOD DCC3
    TERM.Mode ASCII
    TERM.SIZE 80. 1000.
    TERM.SCROLL ON
    WINPOS 0.5 38. 84. 9. 0. 0. debugterm
    TERM.GATE

; Boot Windows CE

    Go
    print "booting Windows CE..."
    wait 1.s
    Break

; Now let's start Windows CE!

    TASK.sYmbol.LoadRM "nk.exe"    ; load OAL symbols
    Go OEMIdle
    print "starting Windows CE... (please wait)"
    wait !run()

; Change current TRACE32 directory to &build_directory

    cd &build_directory

enddo
```

To start PRACTICE script in TRACE32, execute below command:

```
CD.DO wince.cmm
```

Below screenshot shows TRACE32 after wince.cmm script finished execution. Target is stopped at OEMIdle and Virtual Display is showing booted Windows CE desktop.



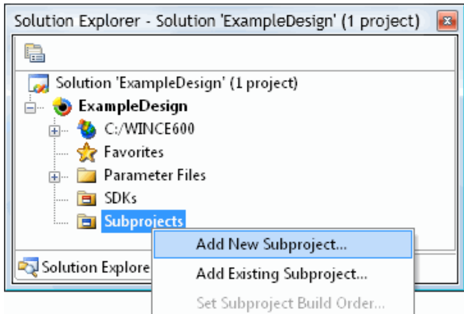
This example script shows general procedure needed to download and boot Windows CE.

©1989-2020 Lauterbach GmbH

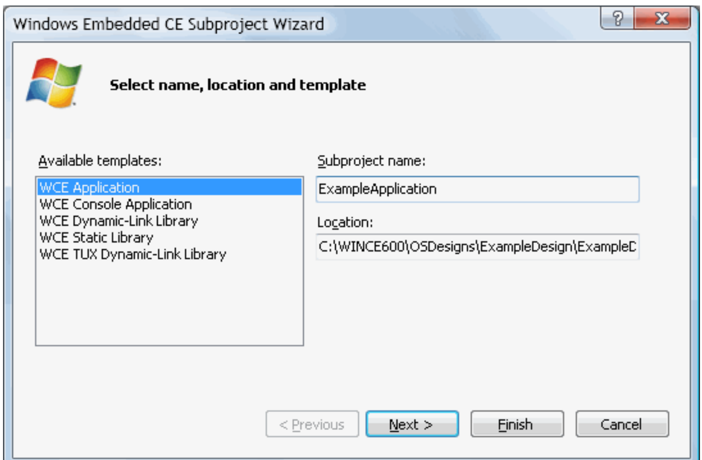
Adding example application to Windows CE image

This chapter describes how to create and add example application to Windows CE image. This application will be used in later chapters to show debugging feautures.

1. In Solution Explorer right-click on Subprojects and select “Add New Subproject”.

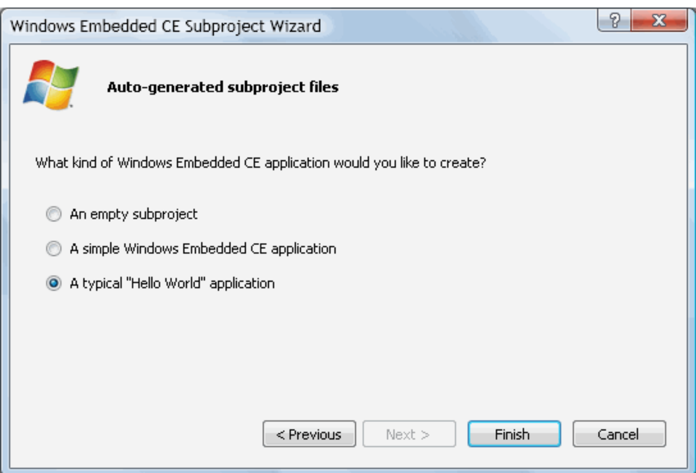


2. In Subproject Wizard select template “WCE Application” and change Subproject name to “ExampleApplication”. Click “Next”.

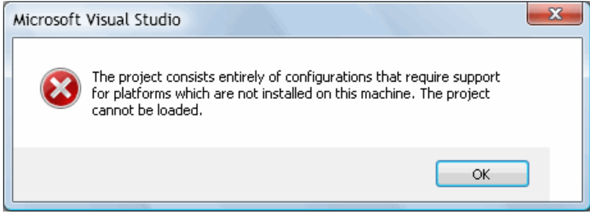


©1989-2020 Lauterbach GmbH

3. Select “A typical ‘Hello World’ application” and click Finish.



4. If below dialog will appear, click OK and reload OS Design project, by closing and opening entire solution. Further dialog boxes of this type can be ignored.



5. Build ExampleApplication. In Solution Explorer right-click on ExampleApplication in Subprojects tree and select “Build”.

6. Create a new text file on your desktop machine (in this case in “C:\” location) and name it “ExampleApplication.txt”.

7. Edit created file by opening in some editor and include text from below frame.

```
31#\Windows\ExampleApplication.exe
```

8. Change extension of the file to “.lnk”.

9. Open platform.bib from “Solution Explorer” and add the following line in the FILES section of the file:

```
ExampleApplication.LNK C:\ExampleApplication.LNK NK
```

©1989-2020 Lauterbach GmbH

10. Open platform.dat from “Solution Explorer” and add the following line:

```
Directory("\Windows\Desktop"):-File("ExampleApplication.lnk","\Windows\ExampleApplication.lnk")
```

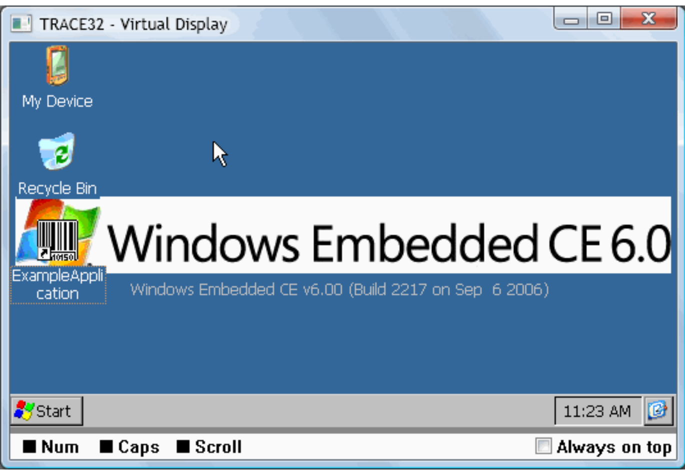
11. In “Solution Explorer” click “Sysgen” in context menu of tree element:

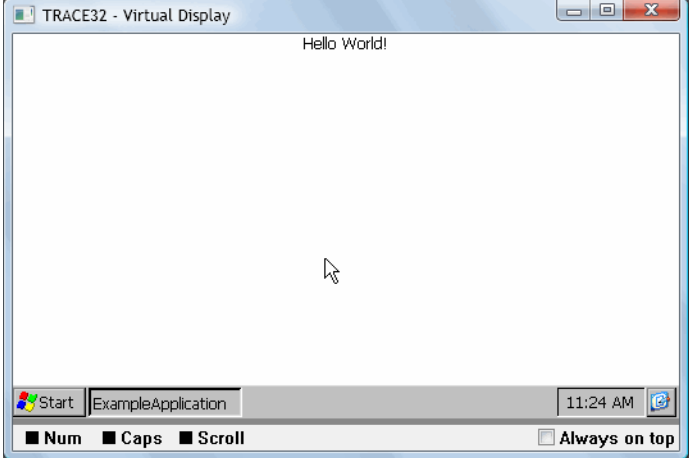
    ExampleDesign->C:/WINCE600->PLATFORM->TRACE32SIMARM

12. From Platform Builder menu select “Build->Copy Files to Release Directory”.

13. From Platform Builder menu select “Build->Make Run-Time Image”.

14. Download and boot Windows CE as in **“Downloading Windows CE image to target and booting system”** (int_exdi2.pdf). After system is up, ExampleApplication can be launched by clicking shortcut.





©1989-2020 Lauterbach GmbH

Debugging Windows CE

This chapter describes how to debug Windows CE from Platform Builder with hardware-assisted debugger.

Loading EXE/DLL modules symbols in TRACE32

For purpose of debugging, “Autoload modules symbols in TRACE32” feature of eXDI2 driver need to be used. Enabling this functionality causes driver to automatically load symbols in TRACE32 environment for EXE/DLL modules that are loaded and executed in Windows CE.

When Windows CE loads an EXE/DLL module, the hardware-assisted debugger that is a part of Windows CE, notifies Platform Builder about this event. The driver uses these notifications and causes TRACE32 to load the apropriate *.PDB (Program Database) file specific for the module currently being loaded. It allows the user to see source code in TRACE32 as well as in Platform Builder.

NOTE:	Alternative method for loading EXE/DLL symbols is using Autoloader that is a part of TRACE32 Windows CE Awareness. In that case, the last line of the download script from chapter **“Downloading Windows CE image to target and booting system” (int_exdi2.pdf)** that changes TRACE32 current directory, is not necessary. For more information about Autoloader, please refer to chapter “Symbol Autoloader“ of Windows CE5 Awareness documentation (rtos_windows_ce.pdf) or Windows CE6 Awareness documentation (rtos_windows_ce6.pdf)

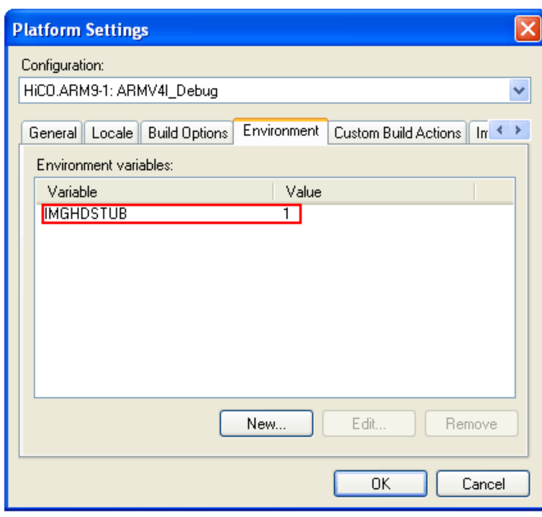
©1989-2020 Lauterbach GmbH

Preparing Windows CE image

1. In “Solution Explorer” right-click “ExampleDesign” project and select “Properties” from context menu.

2. In “Build Options” tab of ExampleDesign properties dialog enable hardware-assisted debugging support.

NOTE	Platform Builder for Windows CE 5.0 doesn’t have special option to enable hardware-assisted debugger. Instead, environment variable need to be defined. Select Platform->Settings from Platform Builder menu. Go to “Environment” tab and add new variable “IMGHDSTUB” with value “1”.



3. From Platform Builder menu select “Build->Make Run-Time Image”.

Driver configuration

1. Select “Connectivity Options” command from “Target” menu to open “Target Device Connectivity options” dialog box.

2. Select “Kernel Service Map” option in the “Service Configuration” section in the control panel on the left side of the dialog box. Set Kernel Debugger service to “KdStub TRACE32 EXDI 2 Driver for CE6.0”

3. Click “Settings” button to configure Kernel Debugger. Check option “Autoload modules symbols in TRACE32”. Click OK.

4. If needed, “TRACE32 Startup script“ can be used to specify PRACTICE cmm script (other than download script). This script is intended for general-purpose use, and is executed at the beginning of Attach.

©1989-2020 Lauterbach GmbH

Debugging session

1. Download and boot Windows CE (see **“Downloading Windows CE image to target and booting system”** (int_exdi2.pdf)).

2. Assuming that target is halted, Platform Builder can be attached. From menu of Platform Builder select “Target -> Attach Device”.

   On successfull attach “Output” window in Platform Builder should output log from “Windows CE Debug” similar to this example:

```
PB Debugger The Kernel Debugger is waiting to connect with target.
PB Debugger The Kernel Debugger connection has been established (Target CPU is ARM).

PB Debugger Target name: CE Device
PB Debugger Probe name: KdStubT32
PB Debugger Kernel debugger connected.
PB Debugger Binary Image should be loaded at 0x84001000 / Data relocated at 0x84d36000
PB Debugger Loaded symbols for 'C:\WINCE600\...\TRACE32SIMARM_ARMV4I_DEBUG\NK.EXE'
PB Debugger Loaded symbols for 'C:\WINCE600\...\TRACE32SIMARM_ARMV4I_DEBUG\UDEVICE.EXE'
PB Debugger Loaded symbols for 'C:\WINCE600\...\TRACE32SIMARM_ARMV4I_DEBUG\EXPLORER.EXE'
PB Debugger Loaded symbols for 'C:\WINCE600\...\TRACE32SIMARM_ARMV4I_DEBUG\SERVICESD.EXE'
PB Debugger Loaded symbols for 'C:\WINCE600\...\TRACE32SIMARM_ARMV4I_DEBUG\COREDLL.DLL'
PB Debugger Loaded symbols for 'C:\WINCE600\...\TRACE32SIMARM_ARMV4I_DEBUG\CESHELL.DLL'
PB Debugger Loaded symbols for 'C:\WINCE600\...\TRACE32SIMARM_ARMV4I_DEBUG\TIMESVC.DLL'

......

PB Debugger Loaded symbols for 'C:\WINCE600\...\TRACE32SIMARM_ARMV4I_DEBUG\KERNEL.DLL'
PB Debugger Loaded symbols for 'C:\WINCE600\...\TRACE32SIMARM_ARMV4I_DEBUG\K.COREDLL.DLL'
PB Debugger Loaded symbols for 'C:\WINCE600\...\TRACE32SIMARM_ARMV4I_DEBUG\DEVMGR.DLL'
```

©1989-2020 Lauterbach GmbH

Please read below notes and see screenshots that show Platform Builder and TRACE32 after attach.

NOTE	In this integration Platform Builder is a master debugger. That is why TRACE32 cannot be simultaneously used with Platform Builder to debug code. “Go” command, all types of Step commands and setting breakpoints manually in TRACE32 are not monitored by driver, and can cause unpredictable behavior of “Platform Builder - Driver - TRACE32” connection.

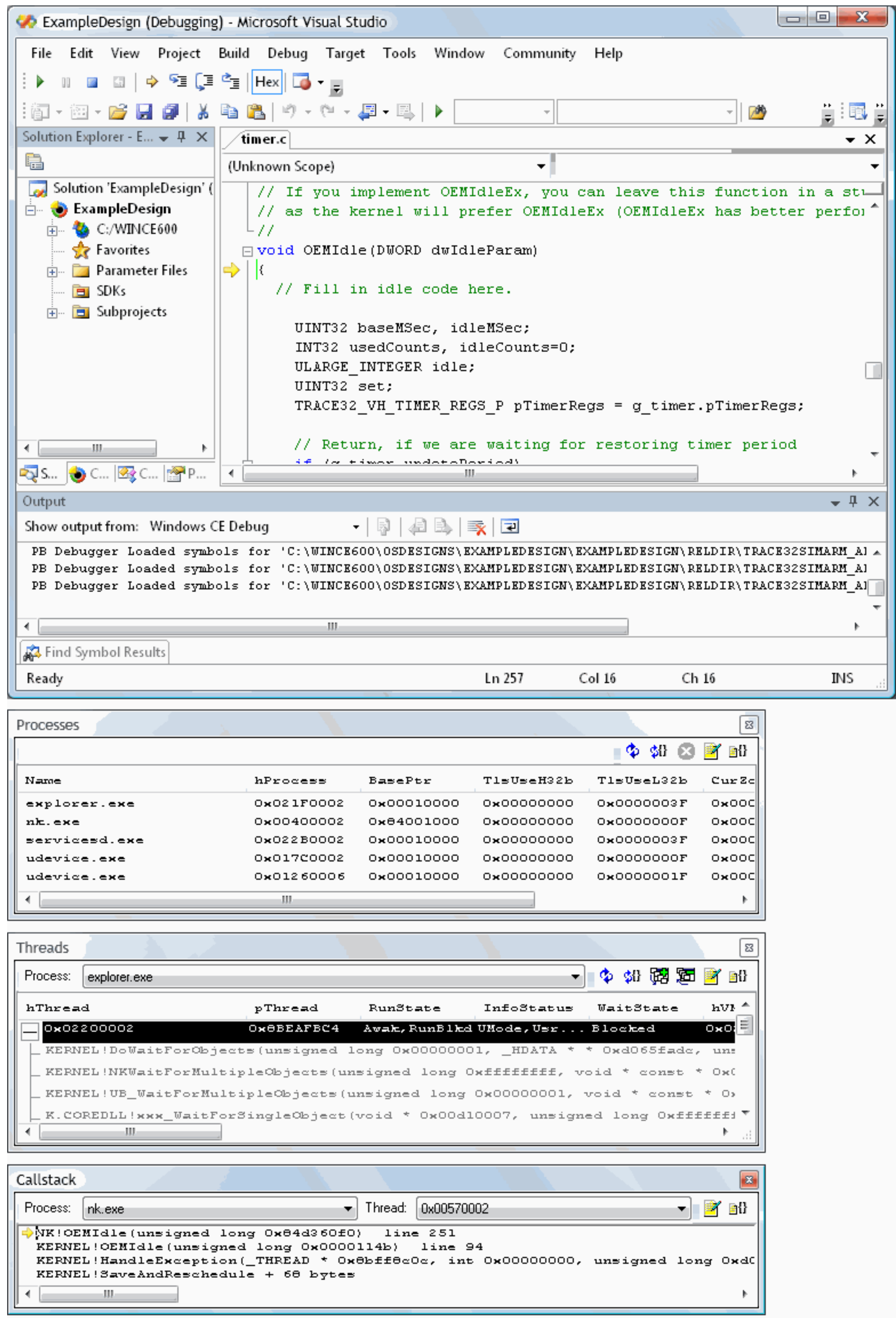
NOTE	If after Attach or Break commands Current Statement Pointer (yellow arrow) is invisible, right-click any source code and select “Show Next Statement”. If the same situation occurs in disassembly window (menu Debug->Windows->Disassembly) right-click on assembler listing and select “Show current statement“.

NOTE	Screenshot from TRACE32 shows in breakpoints list window that after attach Platform Builder sets breakpoint at location OsAxsHwTrap+0x14. This breakpoint is used by hardware-assisted debugger for OS notifications, such modules loading/ unloading, exceptions in target, etc. When, for example, module is loaded, target executes OsAxsHwTrap function and breakpoint is hit. Platform Builder detects this condition and reads all needed information from target. When all data is read, execution is resumed. This is transparent for user, and user should not resume execution by pressing “GO” in TRACE32, regardless of amount of time the target stays at this breakpoint.

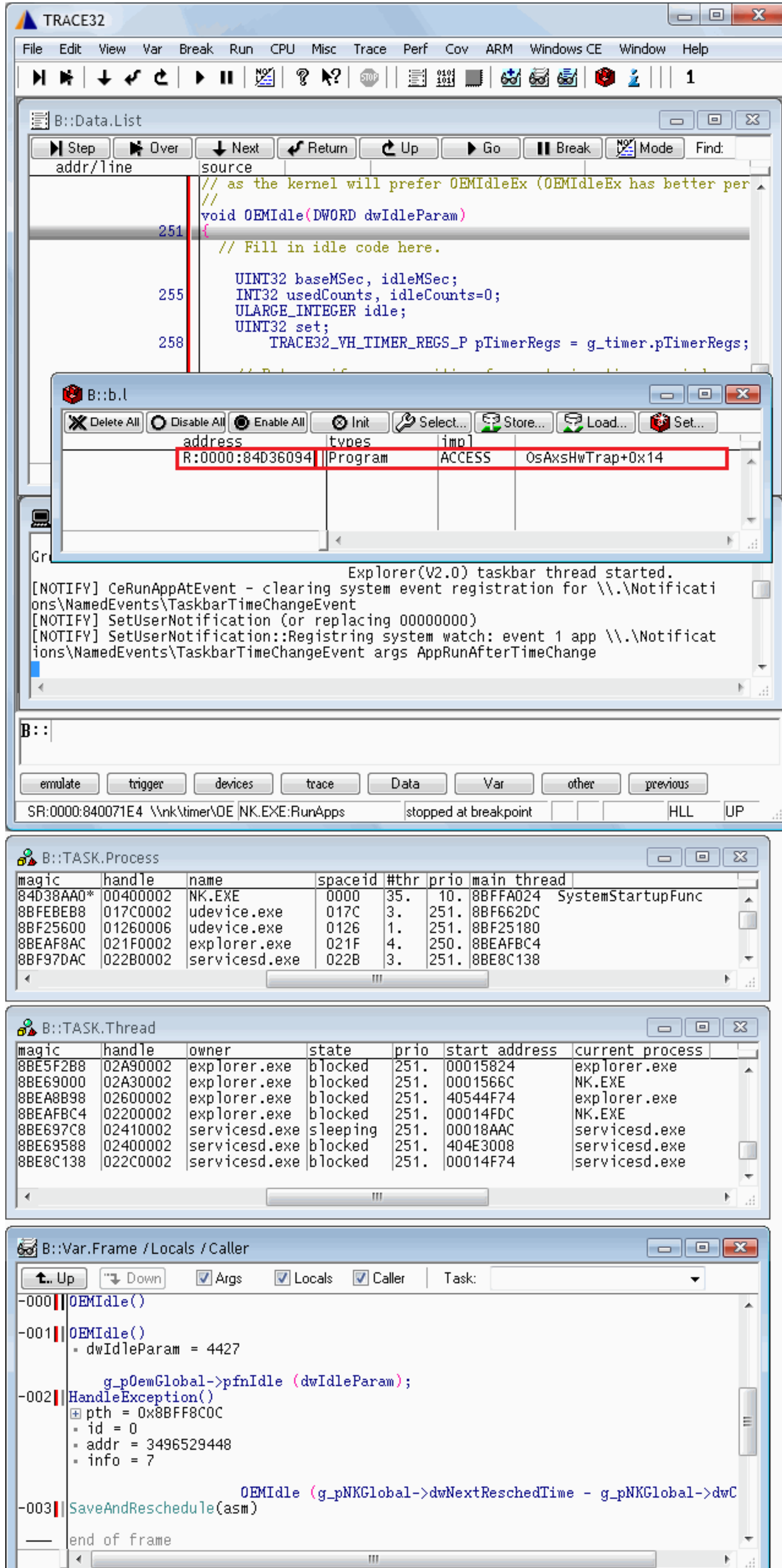
NOTE	It is possible to attach to target from Platform Builder while target is running. In this case, after attach user has to break execution from Platform Builder. When user breaks execution, Platform Builder reads target state (loaded modules, processes list, threads list) and sets OS notifications breakpoint in OsAxsHwTrap function, exactly like when attaching to already halted target.

NOTE	It may happen that during during debugging Windows CE5 or Windows Mobile 5/6, Platform Builder will try to read/write memory at an address that is not properly handled by TRACE32. This is due to the fact that the debugger translation table needs to be refreshed befure such memory accesses. In that case the driver will use the command TASK.MMU.SCAN to refresh the debugger translation table. This command is a part of TRACE32 Windows CE Awareness, which needs to be activated as given in the example scripts and described in the Windows CE5 Awareness documentation (rtos_windows_ce.pdf)

©1989-2020 Lauterbach GmbH



ExampleDesign (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Target Tools Window Community Help
Hex
Solution Explorer - E...
Solution 'ExampleDesign' (
ExampleDesign
C:/WINCE600
Favorites
Parameter Files
SDKs
Subprojects
timer.c
(Unknown Scope)
// If you implement OEMIdleEx, you can leave this function in a st
// as the kernel will prefer OEMIdleEx (OEMIdleEx has better perfo
//
void OEMIdle(DWORD dwIdleParam)
{
// Fill in idle code here.
UINT32 baseMSec, idleMSec;
INT32 usedCounts, idleCounts=0;
ULARGE_INTEGER idle;
UINT32 set;
TRACE32_VH_TIMER_REGS_P pTimerRegs = g_timer.pTimerRegs;
// Return, if we are waiting for restoring timer period
Output
Show output from: Windows CE Debug
PB Debugger Loaded symbols for 'C:\WINCE600\OSDESIGNS\EXAMPLEDESIGN\EXAMPLEDESIGN\RELDIR\TRACE32SIMARM_AI
PB Debugger Loaded symbols for 'C:\WINCE600\OSDESIGNS\EXAMPLEDESIGN\EXAMPLEDESIGN\RELDIR\TRACE32SIMARM_AI
PB Debugger Loaded symbols for 'C:\WINCE600\OSDESIGNS\EXAMPLEDESIGN\EXAMPLEDESIGN\RELDIR\TRACE32SIMARM_AI
Find Symbol Results
Ready
Ln 257
Col 16
Ch 16
INS
Processes
Name hProcess BasePtr TlsUseH32b TlsUseL32b CurZc
explorer.exe 0x021F0002 0x00010000 0x00000000 0x0000003F 0x000
nk.exe 0x00400002 0x84001000 0x00000000 0x0000000F 0x000
servicesd.exe 0x022B0002 0x00010000 0x00000000 0x0000003F 0x000
udevice.exe 0x017C0002 0x00010000 0x00000000 0x0000000F 0x000
udevice.exe 0x01260006 0x00010000 0x00000000 0x0000001F 0x000
Threads
Process: explorer.exe
hThread pThread RunState InfoStatus WaitState hVI
0x02200002 0x8BEAFBC4 Awak,RunBlkd UMode,Usr... Blocked
KERNEL!DoWaitForObjects(unsigned long 0x00000001, _HDATA * * 0xd065fadc,
KERNEL!NKWaitForMultipleObjects(unsigned long 0xffffffff, void * const *
KERNEL!UB_WaitForMultipleObjects(unsigned long 0x00000001, void * const *
K.COREDLL!xxx_WaitForSingleObject(void * 0x00d10007, unsigned long 0xffffffff
Callstack
Process: nk.exe
Thread: 0x00570002
NK!OEMIdle(unsigned long 0x84d360f0) line 251
KERNEL!OEMIdle(unsigned long 0x0000114b) line 94
KERNEL!HandleException(_THREAD * 0x8bff8c0c, int 0x00000000, unsigned long 0xd0
KERNEL!SaveAndReschedule + 68 bytes



©1989-2020 Lauterbach GmbH

3. Resume target execution by pressing go in Platform Builder.

4. In Platform Builder open ExampleApplication.cpp source code and set breakpoint at WinMain function.

NOTE When breakpoint is set in Platform Builder, it takes one of two possible states: not instantiated (set in Platform Builder, but not set in target) and instantiated (set in both Platform Builder and target). Below picture shows these both types of breakpoints. First one is instantiated and second is not instantiated breakpoint.

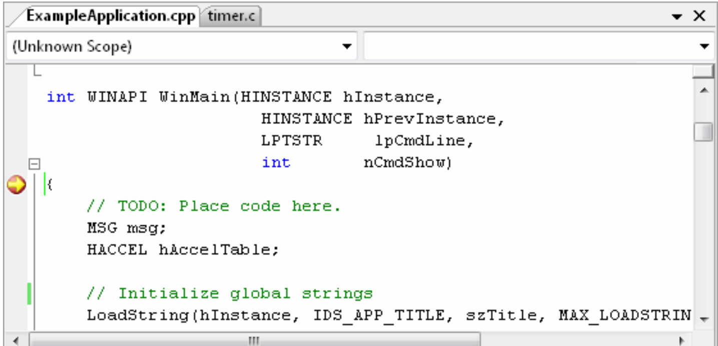
```
// Get current system timer counter
baseMSec = CurMSec;

// Compute the remaining idle time
idleMSec = dwReschedTime - baseMSec;
```

When software debugger is used, breakpoints set at code belonging to loaded module are instantiated (set in target) immediately through KITL connection.

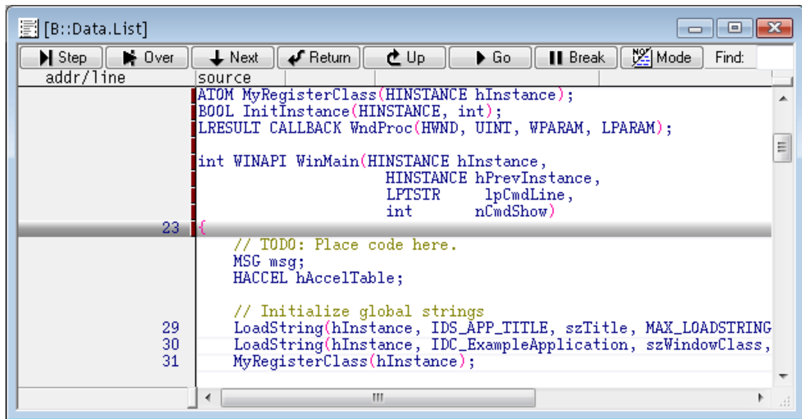
When hardware-assisted debugger is used, breakpoints cannot be instantiated when target is running. Setting breakpoint at loaded module requires halted target to give hardware-assisted debugger opportunity to instantiate that breakpoint. After that, execution can be resumed, and breakpoint will behave in usual way.

When target hits OS notification breakpoint, Platform Builder instantiates all breakpoints that belong to loaded modules. This is the only case when breakpoints instantiation is transparent to user.

5. Start ExampleApplication by clicking shortcut on Windows CE Desktop. Target will hit breakpoint set at WinMain function.



©1989-2020 Lauterbach GmbH



6. In contrast to software debugger, when breakpoint is hit when debugging with hardware-assisted debugger, target is completely stopped. Despite of that, Platform Builder is able to show all information about target: processes, threads, loaded modules, memory and registers content. It gives possibility to debug all kind of code that cannot be debugged by software debugger: hardware bring-up before kernel start, interrupt handlers, drivers, KITL, etc. To show this functionality open interrupt handler source code from below location and set breakpoint at OEMInterruptHandler. After breakpoint is set, resume execution by pressing GO.

```
C:\WINCE600\PLATFORM\TRACE32SIMARM\src\oal\oallib\intr.c
```

Target hits breakpoint at interrupt handler:

```
intr.c | ExampleApplication.cpp | timer.c
(Unknown Scope)
        return 0;
    }

DWORD OEMInterruptHandler(DWORD dwEPC)
{                                          dwEPC = 0xffffc7e0
    UINT32 irq = OAL_INTR_IRQ_UNDEFINED;
    UINT32 sysIntr = SYSINTR_NOP;

    irq = INREG32(&g_pIntcRegs->ISTAT);

    OUTREG32(&(g_pIntcRegs->IDIS),1ul << irq);
```

©1989-2020 Lauterbach GmbH

Debugging hardware bring-up

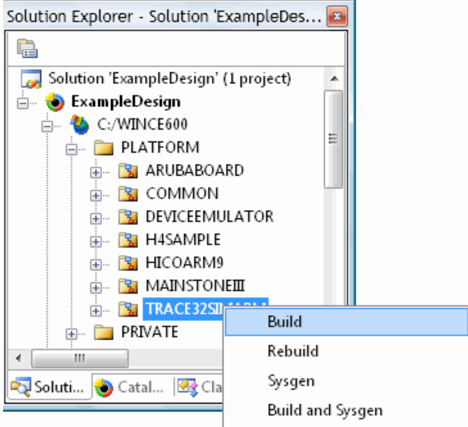
As hardware-assisted debugging doesn't need KITL connection, it is possible to debug system boot from very early phase.

Let's assume that debugging of OEMInit function is needed. Please follow below procedure to perform debugging session. This is one of possible scenarios.

1. Platform Builder can properly attach to target only when OS jumps to Kernel Space (addresses 0x80000000--0xFFFFFFFF) and after minimal initialization is made. This happens before OEMInit. To make target to stop at OEMInit function below change in OEMInit is needed (C:\WINCE600\PLATFORM\TRACE32SIMARM\src\oal\oallib\init.c):

```
BEFORE:
void OEMInit(void)
{
    RemapOALGlobalFunctions();
    SetOALGlobalVariables();
    ....
}
AFTER:
#ifdef DEBUG
static int OEMInit_stop = 1;
#endif

void OEMInit(void)
{
#ifdef DEBUG
    while (OEMInit_stop);
#endif
    RemapOALGlobalFunctions();
    SetOALGlobalVariables();
    ....
}
```

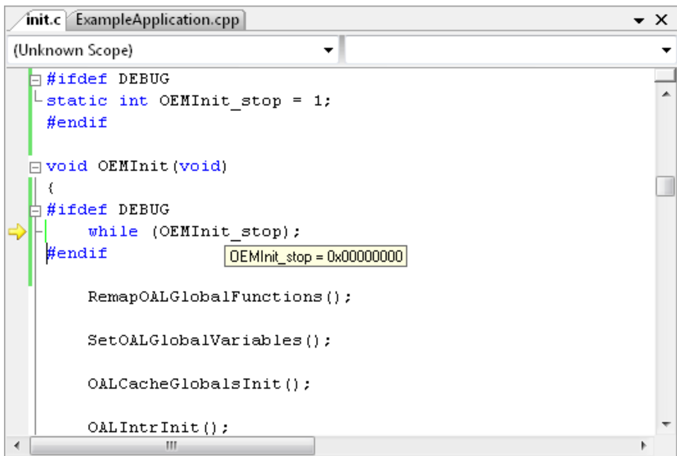
2. Change wince.cmm download script (see **"Downloading Windows CE image to target and booting system"** (int_exdi2.pdf)) at the end of file:



©1989-2020 Lauterbach GmbH

```
.....

; Now let's start Windows CE!

    TASK.sYmbol.LoadRM "nk.exe"    ; load OAL symbols
    //Go OEMIdle
    //print "starting Windows CE... (please wait)"
    //wait !run()
    var.set OEMInit_stop=0

; Change current TRACE32 directory to &build_directory

    cd &build_directory

enddo
```

3. Execute wince.cmm script in TRACE32 and attach Platform Builder to target (Target -> Attach Device).



Hardware-assisted debugging and KITL

KITL connection between Platform Builder and target can work together with hardware-assisted debugger. In this case Platform Builder has access to processes list and threads list in real-time. Additionaly, remote tools such as “File Viewer” or “Heap Walker” can be used. To perform debugging session with KITL follow below steps. Because simulator, on which this example is running on, does not have physical connection such as USB or Ethernet, the below steps are just demonstrative. To use KITL without communication channel (USB, Ethernet, Serial) on simulator or debugger connected to real target see **“Using TRACE32 FDX for KITL Kernel Transport”** (int_exdi2.pdf).

1. In “Solution Explorer” right-click “ExampleDesign” project and select “Properties” from context menu. In “Build Options” tab of ExampleDesign properties dialog select “Enable KITL”. From menu select “Build -> Make Run-Time Image”.

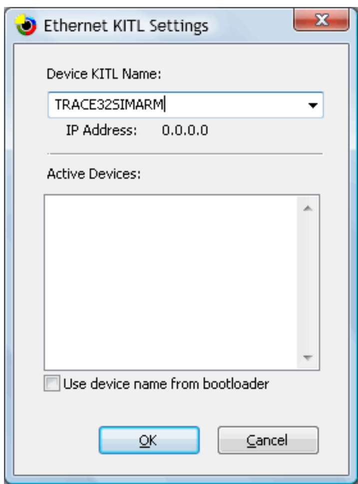
2. In download script from chapter **“Downloading Windows CE image to target and booting system”** (int_exdi2.pdf) comment below lines at the end of file:

```
; Now let's start Windows CE!

    TASK.sYmbol.LoadRM "nk.exe"    ; load OAL symbols
    //Go OEMIdle
    //print "starting Windows CE... (please wait)"
    //wait !run()

; Change current TRACE32 directory to &build_directory

    cd &build_directory

enddo
```

3. Select “Connectivity Options” command from “Target” menu to open “Target Device Connectivity options” dialog box.

4. Select “Kernel Service Map” option in the “Service Configuration” section in the control panel on the left side of the dialog box. Set “Kernel Transport” service to “Ethernet”.

5. In Settings dialog of Ethernet Kernel Transport set “Device KITL Name” to “TRACE32SIMARM”.



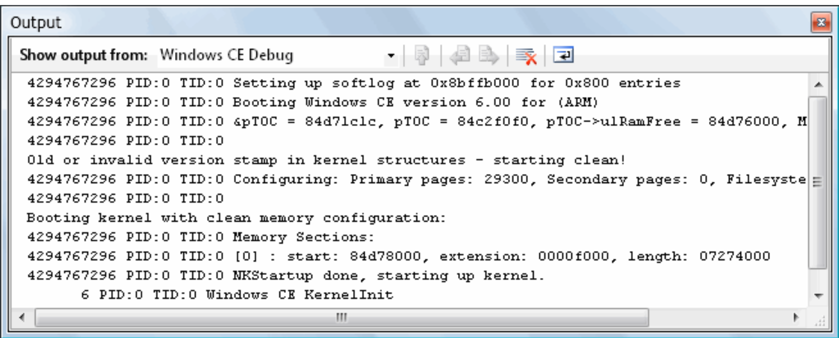
6. Execute wince.cmm download script in TRACE32.

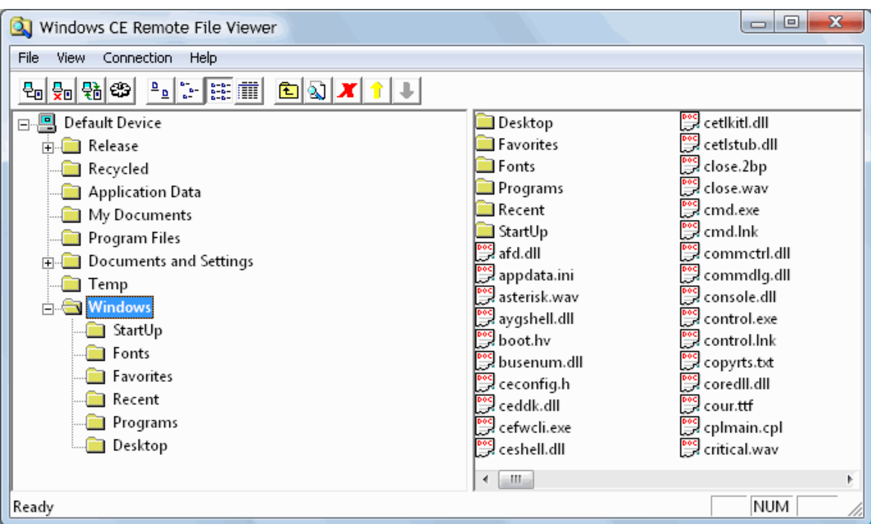
©1989-2020 Lauterbach GmbH

7. From menu of Platform Builder select “Target -> Attach Device”.

8. Press “Go” in Platform Builder to resume target execution.

9. “Output” window in Platform Builder should display Debug Messages from target. Remote tools are also available.





Using TRACE32 FDX for KITL Kernel Transport

eXDI2 driver can be used to implement KITL over Fast Data Exchange (FDX) mechanism. It doesn’t require any communication hardware such as Ethernet, Serial or USB. All KITL transfers are performed through JTAG/BDM connection with target.

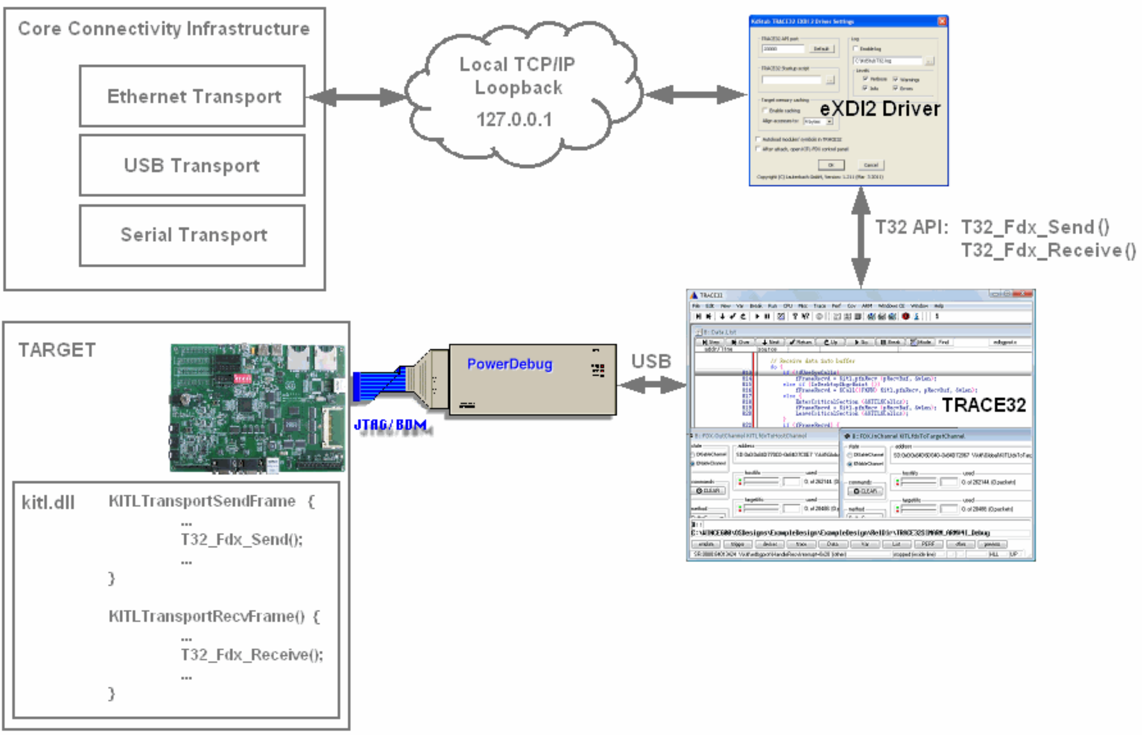
©1989-2020 Lauterbach GmbH

FDX Overview

The Fast Data Exchange (FDX) enables transfering universal data between the target and the host. The protocol implementation on target side is included in the target application. The source code (C) is provided by Lauterbach. On the host side the transmitted data can be processed by a user application communicating with the T32 Application Interface or through Named Pipes.

The basic packet transport method differs dependent on the target. T32 supports memory mapped buffered transfer through dualport memory access or normal access at breakpoints or spot breakpoints. Some target devices support a Debug Communication Channel (DCC), which can be used to transfer FDX data in real time.

Architecture of KITL over FDX



KITL over FDX uses standard Ethernet Transport provided by Platform Builder Core Connectivity Infrastructure. On the target side KITL is implemented as calls to T32_Fdx_Send and T32_Fdx_Receive functions provided by Lauterbach. These functions send and receive packets from TRACE32 software through communication channels. eXDI2 integration driver is a bridge between target and Ethernet transport.

©1989-2020 Lauterbach GmbH

Enabling KITL over FDX

1. In “Solution Explorer” right-click “ExampleDesign” project and select “Properties” from context menu. In “Build Options” tab of ExampleDesign properties dialog select “Enable KITL”.

2. In “Environment” tab of the ExampleDesign properties dialog, delete variable BSP_TERMINAL_DCC that was added in step **“Creating OS Design”** (int_exdi2.pdf). The reason is that terminal uses the same communication channel that we want to use for FDX.

3. In “Environment” tab of the ExampleDesign properties dialog, add one of the below mentioned variables. For TRACE32SIMARM BSP select **BSP_KITL_FDX_DCC_ARM9**.

BSP_KITL_FDX_MEMBUFF=1	This variable enables FDX in memory buffers mode.
BSP_KITL_FDX_DCC_ARM7=1	This variable enables FDX in DCC mode for ARM7 target. BSP_TERMINAL_DCC=1 is not allowed in this mode (see **“Creating OS Design”** (int_exdi2.pdf)).
BSP_KITL_FDX_DCC_ARM9=1	This variable enables FDX in DCC mode for ARM9 target. BSP_TERMINAL_DCC=1 is not allowed in this mode (see **“Creating OS Design”** (int_exdi2.pdf)).
BSP_KITL_FDX_DCC_ARM11=1	This variable enables FDX in DCC mode for ARM11 target. BSP_TERMINAL_DCC=1 is not allowed in this mode (see **“Creating OS Design”** (int_exdi2.pdf)).
BSP_KITL_FDX_DCC_XSCALE=1	This variable enables FDX in DCC mode in XSCALE target. BSP_TERMINAL_DCC=1 is not allowed in this mode (see **“Creating OS Design”** (int_exdi2.pdf)).

4. Rebuild TRACE32SIMARM BSP by selecting “Rebuild” in context menu in Solution Explorer.

5. Modify download script from chapter **“Downloading Windows CE image to target and booting system”** (int_exdi2.pdf) as below:

©1989-2020 Lauterbach GmbH

For BSP_KITL_FDX_MEMBUFF:

```
.....

; Now let's start Windows CE!

    TASK.sYmbol.LoadRM "nk.exe"    ; load OAL symbols
    //Go OEMIdle
    //print "starting Windows CE... (please wait)"
    //wait !run()

    FDX.DISABLE
    FDX.RESET

    FDX.METHOD BUFFERC V.VALUE(KITLSynchronizeFDX)

    FDX.OutChannel KITLfdxToHostChannel
    FDX.InChannel KITLfdxToTargetChannel

    FDX.CLEAR KITLfdxToHostChannel
    FDX.CLEAR KITLfdxToTargetChannel

    BREAK.SET KITLSynchronizeFDX

; Change current TRACE32 directory to &build_directory

    cd &build_directory

enddo
```

For BSP_KITL_FDX_DCC_*:

```
; Open debug output terminal

    //TERM.METHOD DCC3
    //TERM.Mode ASCII
    //TERM.SIZE 80. 1000.
    //TERM.SCROLL ON
    //WINPOS 0.28571 37.308 84. 9. 0. 0. debugterm
    //TERM.GATE

.....

; Now let's start Windows CE!

    TASK.sYmbol.LoadRM "nk.exe"    ; load OAL symbols
    //Go OEMIdle
    //print "starting Windows CE... (please wait)"
    //wait !run()

    FDX.DISABLE
    FDX.RESET

    FDX.METHOD DCC

    FDX.OutChannel
    FDX.InChannel

    FDX.CLEAR

; Change current TRACE32 directory to &build_directory

    cd &build_directory

enddo
```

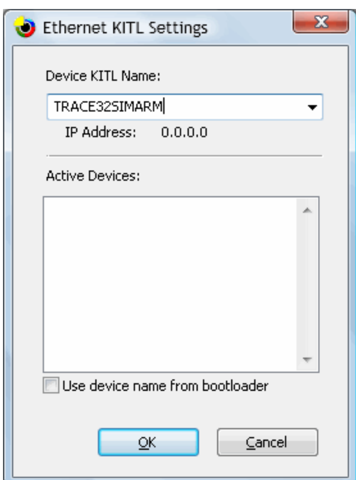
6. Execute wince.cmm download script in TRACE32.

7. Select “Connectivity Options” command from “Target” menu to open “Target Device Connectivity options” dialog box.

8. Select “Kernel Service Map” option in the “Service Configuration” section in the control panel on

the left side of the dialog box. Set “Kernel Transport” service to “Ethernet”.

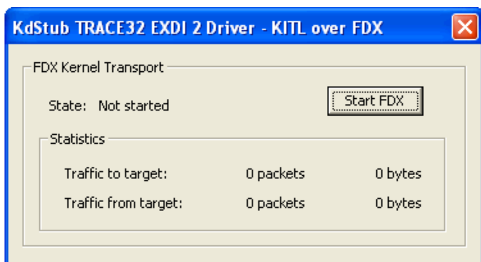
9. In Settings dialog of Ethernet Kernel Transport set “Device KITL Name” to “TRACE32SIMARM”.



10. In Settings dialog of debugger service “KdStub TRACE32 EXDI 2 Driver for CE6.0“ enable option “After attach, open KITL-FDX control panel“.

11. From menu of Platform Builder select “Target -> Attach Device”.

12. In KITL-FDX control panel that appears after attach, press “Start FDX” button.



13. Press “Go” in Platform Builder to resume target execution.

14. “Output” window in Platform Builder should display Debug Messages from target. Remote tools are also available.

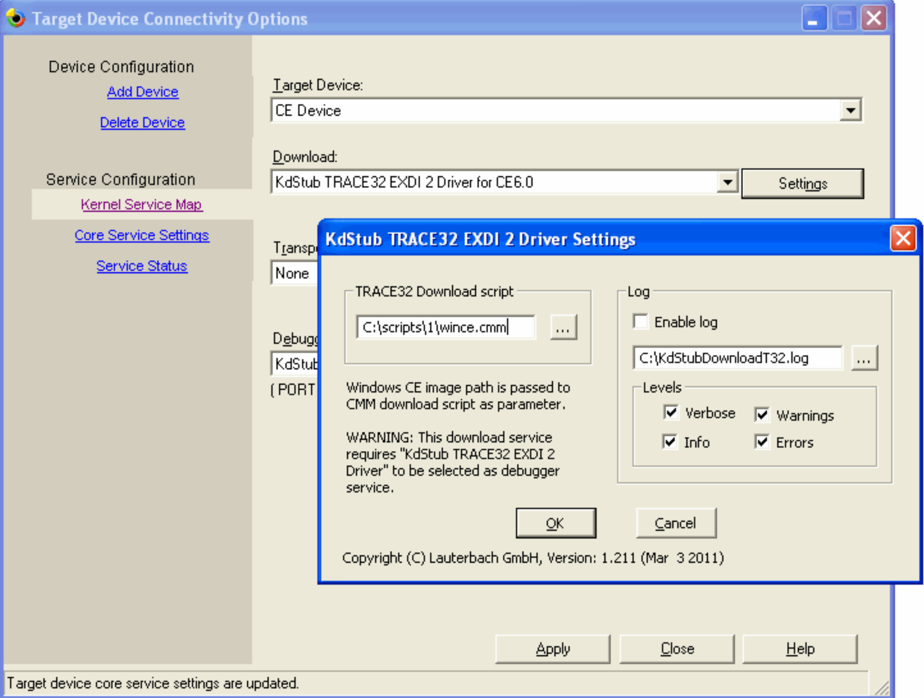
©1989-2020 Lauterbach GmbH

Download service

Integration driver contains Download Service - KdStub TRACE32 EXDI 2 Driver for CE6.0 - that can start wince.cmm download script execution during Attach. Refer to **"Downloading Windows CE image to target and booting system"** (int_exdi2.pdf) for detailed description of creating download script.

When Platform Builder attaches to target, download script is started and driver waits untill wince.cmm script execution is finished. After that, Kernel Debugger connects to target in usual way.

NOTE	The download service requires "KdStub TRACE32 EXDI 2 Driver" to be selected as debugger service.



Debugging timings

Platform Builder performs a lot of operations (memory reads, setting breakpoints, etc …), so commands like Break, Go, Step Over, Step Into, Step Out and other used during debugging can take some time.

Commands execution time can vary from less than a second up to over a dozen of seconds. Time depends on used architecture, JTAG speed and host system load. If Platform Builder reads a lot of memory, enabling memory caching (see **“Memory caching”** (int_exdi2.pdf)) can speed-up commands execution, too.

Memory caching

Integration driver contains target’s memory caching mechanism that can speed-up memory reads performed by Platform Builder. To enable this feature click “Enable caching” in settings dialog of debugger service “KdStub TRACE32 EXDI 2 Driver for CE6.0“.

Each memory read address requested by Platform Builder is aligned by cache to 4, 8 or 16 bytes boundary. Amount of bytes that cache reads from target to create entry is a multiply of 4, 8 or 16. Each cache entry contains aligned address and data (of size that is multiply of 4, 8 or 16 bytes). When Platform Builder reads memory from some address, driver returns data from cache only if this address (aligned) matches address of entry in cache and all data is available in this entry. If address doesn’t match any entry or not all requested data is available in entry, such entry is deleted from cache, and cache reads data from target. Because of nature of this algorithm it is important to observe in log how Platform Builder reads data from target. If, for example Platform Builder reads sequentially 4-bytes data chunks from continuous addresses, it is better to set align to 8 or 16 bytes, because only one “TRACE32 <-> target” memory read transaction will be performed instead of 2 or 4 transactions.

Cache is invalidated each time execution is resumed.

©1989-2020 Lauterbach GmbH

Troubleshooting

Because of some reasons, installation of Core Connectivity infrastructure or KdStub TRACE32 EXDI 2 Driver can be corrupted. Following list contains possible symptoms of corruption:

1. “Settings” button in “Connectivity Options” for driver is inactive.
2. “Connectivity Options” window does not appear.
3. Message “Could not get ICcService interface from OsAccess!!!” appears when performing Attach command in Platform Builder.

To repair installation, please close Platform Builder and reinstall KdStub TRACE32 EXDI 2 Driver. If symptoms still exists, please follow this procedure:

1. Close Platform Builder.
2. Make sure that process list (in Task Manager) doesn't contain process CESVCH~1.EXE and CEPB.EXE. If any of these processes exists, please end it.
3. Uninstall KdStub TRACE32 EXDI 2 Driver.
4. Remove directory:

   ```
   C:\Documents and Settings\Your_Profile_Name\Local Settings\Application Data\Microsoft\CoreCon
   ```

   Please notice that some directories in above path can be hidden. Additionally some names are dependent on Your MS Windows language version.

   This directory stores Your personal settings of other drivers installed. Those settings will be lost.

5. Go to directory:

   ```
   C:\Program Files\Windows CE Platform Builder\5.00\CORECON\SCRIPTS
   ```

   Run below scripts with parameters:

   ```
   unregister.bat "C:\Program Files\Windows CE Platform Builder\5.00"
   register.bat "C:\Program Files\Windows CE Platform Builder\5.00"
   ```

6. Install KdStub TRACE32 EXDI 2 Driver.

©1989-2020 Lauterbach GmbH