Elektrobit

EB tresos® AutoCore OS architecture notes WINDOWS

product release 6.0

Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 Erlangen, Germany
Phone: +49 9131 7701 0
Fax: +49 9131 7701 6333
Email: info.automotive@elektrobit.com

Technical support

Europe	Japan	USA
Phone: +49 9131 7701 6060	Phone: +81 3 5577 6110	Phone: +1 888 346 3813

Support URL

https://www.elektrobit.com/support

Legal notice

Confidential and proprietary information

ALL RIGHTS RESERVED. No part of this publication may be copied in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of Elektrobit Automotive GmbH.

ProOSEK®, tresos®, and street director® are registered trademarks of Elektrobit Automotive GmbH.

All brand names, trademarks and registered trademarks are property of their rightful owners and are used only for description.
Copyright 2020, Elektrobit Automotive GmbH.

Table of Contents

1. Overview

1.1. Overview of the Architecture

The EB tresos AutoCore OS for Microsoft Windows allows the user to create and test AUTOSAR applications easily on a PC without the need for extra hardware. This is achieved by not simulating a certain hardware architecture but by employing the services of the underlying host OS, i.e. Microsoft Windows.

For example, host OS threads are used as a runtime vehicle for EB tresos AutoCore OS tasks and ISRs. Timers used by those entities are based on the time services offered by the host OS which hence define their accuracy. Likewise, all synchronization mechanisms of EB tresos AutoCore OS tasks and ISRs are based on those of the host OS.

Further details about the implementation are provided in later chapters. This chapter provides a basic overview of what is offered and required by this software.

1.2. Supported Run-time Environment

See the release notes that came with this software.

1.3. Glossary

Abbreviations and technical terms used in this document

ISR
: Interrupt Service Routine

OIL
: OSEK Implementation Language

API
: Application Programming Interface

2. Implementation Details

2.1. EB tresos AutoCore OS Integration into Host OS

From the perspective of EB tresos AutoCore OS the host OS *Microsoft Windows* is considered as *hardware* and hence its services are comparable to those offered by peripherals of microcontrollers, which are otherwise used to execute EB tresos AutoCore OS.

For this reason, EB tresos AutoCore OS uses the services of the host OS to establish an AUTOSAR environment. AUTOSAR tasks and ISRs get each a dedicated Microsoft Windows thread to run the user-supplied function. These threads are created when EB tresos AutoCore OS is started and linger on until the whole application terminates. The consequence is, that the number of threads doesn't change once `StartOS()` was successfully called.

This set of threads is managed by further *core threads*. There is one such thread per configured core, i.e. there are `OsOS/OsNumberOfCores` many. The core thread of the *bootstrap core* it identical to the main application thread created by Microsoft Windows when the application starts. Any additional core thread is started by `StartCore()`. Usually, a call to `StartCore()` leads to a call of `StartOS()` on the started core. Therefore, the total number of host OS threads remains constant until application termination once EB tresos AutoCore OS is started on all configured cores.

The purpose of those core threads is to orchestrate the execution of the host OS threads associated with EB tresos AutoCore OS tasks and ISRs. This is done by selectively suspending and resuming such threads so that the correct entity is running at each point in time. This also has an effect on how user-code can use host OS services. See Section 2.5, “Using Microsoft Windows Services” for further details.

2.2. Initialization of Execution Environment

The execution environment must be initialized before any other code (e.g. AUTOSAR or application-specific) may be run. To accomplish this the function `void OS_WINDOWSPreStartOs(os_int_t argc, os_char_t* argv[])` must be called right at the beginning of `void main(int argc, char* argv[])`, before any other code is executed. Passing the command-line parameters to `OS_WINDOWSPreStartOs()` is not mandatory, but offers the advantadge of being able to manipulate different parameters without re-compiling the whole application. See also Section 2.2.1, “Command-line Arguments” for further details. Simply pass `argc = 0` and `argv = NULL` when calling `OS_WINDOWSPreStartOs`, if this functionality is not desired.

Also note, that the function `OS_WINDOWSPreStartOs()` may be called serveral times. Only the first invocation has an effect. This may happen when more than one core is started by `StartCore()`, because each starts execution in the function `main()`.

2.2.1. Command-line Arguments

By default applications containing AUTOSAR entities when ported to Microsoft Windows support a minimum set of command-line arguments. To find out which ones, just invoke such an application with the argument `-h` or `--help`. The output tells you all supported arguments and their meanings alongside with information about the PC-platform, e.g. if a high-performance counter is available and the maximum timer resolution.

See Section 2.2, “Initialization of Execution Environment” to know, how to properly initialize the execution environment to make command-line arguments usable.

2.3. Exception Handling

This version of EB tresos AutoCore OS does not support exception handling. This means, that any illegal activity, e.g. divide by zero or out-of-bounds memory access, is reported by a host OS mechanism. No AUTOSAR error or protection hook function will be called in this case.

2.4. Interrupt Handling

EB tresos AutoCore OS for Microsoft Windows supports both category 1 and category 2 interrupts. User-defined interrupts can be automatically enabled at startup by setting the `OsEnable_On_Startup` attribute of the ISR to TRUE.

All interrupt service routines are written as depicted in the following code excerpt.

```
#include <Os.h>
ISR(isr_name)
{
    /* handling of interrupt */

    return;
}
```

The value `isr_name` is chosen by the user to name the ISR and may be used to refer it in other parts of the code, e.g. to trigger it. The code block of the `ISR()` statement can be treated as that of a C language function.

Furthermore, the nesting of interrupts is supported. This means, that ISRs can interrupt each other according to their configured priorities.

2.4.1. Triggering Interrupts

To trigger EB tresos AutoCore OS ISRs the following two functions are provided. Include `Os.h` to make their prototypes available.

`void OS_WINDOWSTriggerInterrupt(os_isrid_t isrId)`
: Triggers the interrupt with the specified ID. The same value, which is used as argument to `ISR()` may be used. This is because the OS generator defines a macro with a name identical to that value to identify the correspondig ISR. Hence, it is possible to write `OS_WINDOWSTriggerInterrupt(isr_name);` to trigger the ISR defined at the beginning of Section 2.4, “Interrupt Handling”.

`void OS_WINDOWSClearInterrupt(os_isrid_t isrId)`
: Undoes the effect of OS_WINDOWSTriggerInterrupt(). Note, that once processing of a triggered interrupt has started, it cannot be canceled.

2.5. Using Microsoft Windows Services

To use host OS services the respective code section must be bracketed by `OS_WINDOWSGoingToUseHostOsService()` and `OS_WINDOWSFinishedUsingHostOsService()`. Failing to do so might result in undefined behavior. The reason for this is that host OS threads are suspended and resumed by the core threads within the same application. To prevent suspended threads from occupying system resources, which other running threads need to acquire, they must first notify EB tresos AutoCore OS to prevent their suspension for the time they use host OS services.

These functions are declared as follows.

`os_intstatus_t OS_WINDOWSGoingToUseHostOsService(void)`
: This function must be used to announce the use of host OS services. The return value must be passed to a supsequent call of `OS_WINDOWSFinishedUsingHostOsService()`.

`void OS_WINDOWSFinishedUsingHostOsService(os_intstatus_t is)`
: Terminates the code block, in which host OS services are used. The argument `is` is the return value of the most recent call of `OS_WINDOWSGoingToUseHostOsService()`.

The example Example 2.1, “Using Microsoft Windows services from AUTOSAR software.” demonstrates the use of these functions. There, to use Microsoft Windows servcies, a new C-language block is opened in which the variable `is` is defined. Its value is the return value of `OS_WINDOWSGoingToUseHostOsService()`, which must be treated as opaque value. After this line host OS services may be used, i.e. it marks the start of a block

in which such services may be used. This block ends when the function `OS_WINDOWSFinishedUsingHostOsService()` is called. It must be passed the value previously retuned by `OS_WINDOWSGoingToUseHostOsService()`. Once outside this block again, it is not safe to use any Microsoft Windows services.

Example 2.1. Using Microsoft Windows services from AUTOSAR software.

```
void MyFunction(void)
{
  [...]
  {
    os_intstatus_t const is = OS_WINDOWSGoingToUseHostOsService();
    {
      /* Use Microsoft Windows services, e.g. printf(). */
      [...]
    }
    OS_WINDOWSFinishedUsingHostOsService(is);
  }
  [...]
}
```

NOTE **What is a Microsoft Windows Service?**

A Microsoft Windows service is everything used by AUTOSAR software, which ships with Microsoft Windows itself or is part of the programming environment. For example, the function `printf()` of the C library is a host OS service.

2.6. Time Warp Factor

The *time warp factor* is an instrument to warp the wall clock time into the AUTOSAR time. The effect is that the AUTOSAR time passes more slowly as compared to the wall clock time. A warp factor of 1 means no warping at all, hence AUTOSAR time and wall clock time are identical. A larger warp factor slows down the AUTOSAR-time proportionally. It can be specified via the command-line argument `--time-warp-factor` and this means, that applications don't have to be re-compiled to adapt their timing characteristics. It is also possible to set the warp factor via the configuration parameter `OsOS/Os_WINDOWSTimeWarpFactor`. The factor set via the command-line takes precedence, though.

The purpose of time warping is to avoid having to change source code of AUTOSAR applications, when they are ported to Microsoft Windows. The timing requirements of such applications can be quite demanding, when compared to what Microsoft Windows offers. By increasing the time warp factor it might be possible to adapt the timing requirements to what is feasible on Microsoft Windows. Unfortunately, there is no golden rule for picking a warp factor, hence a trial-and-error method must be used.

The image Figure 2.1, “Time warping.” illustrates how the wall clock time is warped into the AUTOSAR time. A factor of 1 makes AUTOSAR and wall clock time pass equally fast. Larger factors make the time in the AUTOSAR world elapsing more slowly. For example, let an AUTOSAR timer expire every 100 microseconds. With a warp factor of 100 the 100 microseconds period then takes 10 milliseconds (on the wall clock) and this would be a timing requirement feasible on Microsoft Windows.

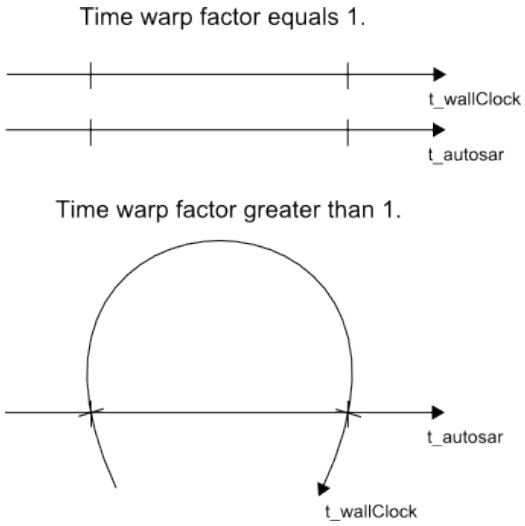

Figure 2.1. Time warping.

2.7. Logging

The command-line option `--log-level` controls basic logging capabilities by setting the current log level. These allow insight into the operations of EB tresos AutoCore OS on Microsoft Windows.

Each log message has a severity and is only displayed, if that value is higher than the current log level. The following log levels are supported (in ascending severity).

- DEBUG
- INFO
- WARNING
- ERROR
- DISABLE

A log message has the following format.

```
thread-id-hex '=' thread-id-dec core-id int-disable-lvl \
  filename '[' line-number ']:' log-message
```

It starts with the id of the Microsoft Windows thread emitting the log message in hexadecimal and decimal format. It follows the EB tresos AutoCore OS core id and the interrupt disable level of that core. Next comes the filename and line number where the log message is generated and finally the log message itself.

Note, that a log message comprises one line on the screen and has been broken up in two above to fit the page size. This is indicated by \ at the end of a line.

2.8. Deviations and Peculiarities

For the application of EB tresos AutoCore OS on Microsoft Windows you should take notice of the following list. It contains deviations and peculiarities which affect its operation.

Kernel Interruptibility

The EB tresos AutoCore OS kernel is not interruptible. This means, that once the kernel has started execution in the context of one of the core threads (see Section 2.1, “EB tresos AutoCore OS Integration into Host OS”, it doesn't respond to any further syscall or interrupt request. The execution of EB tresos AutoCore OS tasks and ISRs on the respective core is suspended during this time. This is in contrast to the usual case on other hardware platforms, where the EB tresos AutoCore OS kernel may be interrupted during most of its activities.

Stack Management

All stacks of EB tresos AutoCore OS tasks and ISRs are managed by the host OS. This is a direct consequence of the fact, that such entities are directly mapped to host OS threads. This also means, that EB tresos AutoCore OS is not able to do its stack checks as normal and report accurate statistics. Therefore, the function `OS_UserGetStackInfo()` yields inaccurate results. Furthermore, the specified stack sizes of EB tresos AutoCore OS tasks and ISRs are ignored and are just present to support existing configurations.

Hardware Counter

Counters of hardware type do not stop, when the AUTOSAR application is stopped by a debugger. This can lead to timing problems when the application is continued after a while and there are active AUTOSAR alarms or schedule tables. Whether this poses a problem depends on the requirements of the AUTOSAR application under test.

3. Extended OIL Attributes

The EB tresos AutoCore OS generator uses the WINDOWS-specific OIL attributes listed in the following table. The attributes are decribed fully in the subsequent sections.

3.1. OS Attributes

The OS object has the following WINDOWS-specific attributes:

MICROCONTROLLER
: This is where the specific derivative to be used is selected. Here all derivatives that are currently supported by the EB tresos AutoCore OS can be found.

3.2. ISR Attributes

ISR objects have the following WINDOWS-specific attributes:

WINDOWS_VECTOR
: This attribute specifies the vector to which the ISR will be attached. It is only supported for compatibility reasons and has otherwise no effect. Hence, its concrete value is not important.

3.3. COUNTER Attributes

COUNTER objects have the following WINDOWS-specific attributes:

WINDOWS_TIMER
: This attribute specifies the timer that the counter will use.

4. Board Support

4.1. Directory Structure

The board directories are located under the `demos\AutoCore_OS\os_multicore_demo_WIN32X86_<version>\source\boards\WIN32X86` directory in your EB tresos Studio installation. Each board directory typically contains the following files:

- board.h
- board.c
- WIN32X86.mak

4.2. Header File: `board.h`

The EB tresos AutoCore OS kernel configuration files comprise the `board.h` header file. It contains specific parameters for a platform. In the current case, it is of limited use, because there is no real hardware. Nevertheless, the following macros might be of interest for a user.

`OS_BoardNsToTicks(nS)`
: This macro provides a way to convert nanoseconds (ns) to ticks of the “system clock”. It must compute entirely in the preprocessor because it is used to initialize constant tables. It must perform the conversion without overflow and with minimal rounding errors. It is assumed that the time resolution is 1 ms, i.e. one tick lasts one millisecond. To know the supported timer resolution on your platform invoke the AUTOSAR application with the command-line argument `-h` or `--help`. At the end of the help text there are statistics about the capabilities of the time services of the host OS. These timers form the basis for AUTOSAR timers.

`OS_BoardTicksToNs(tik)`
: This macro converts ticks (tik) to nanoseconds. It is the inverse of `OS_BoardNsToTicks(nS)`.

4.3. Source File: `board.c`

Contains some internal helper functions for the OS demo. That code is mainly about controlling the console.

4.4. Make File: `WIN32X86.mak`

This file provides the integration into the build system.

Appendix A. Error Codes

A.1. Generator Error Codes

WINDOWS-specific error codes are listed here. For architecture-independent error codes see the EB tresos AutoCore OS User's Guide.

A.2. Kernel Error Codes

Please refer to the AUTOSAR OS Users Guide.