T1 Integration

Project-Specific Preparation

User Guide – Version 1.18

www.gliwa.com

GLIWA embedded systems GmbH & Co. KG
Pollingerstr. 1
82362 Weilheim i.OB.
GERMANY

fon +49 - 881 - 13 85 22 - 0
fax +49 - 881 - 13 85 22 - 99
info@gliwa.com
gliwa.com

Document ID: 6-T1target-30-20-15-10

This document describes the steps necessary prior to a ***T1*** integration into an embedded application. It also explains some functional aspects of ***T1*** to make clear what modifications are required to an existing application in order to integrate ***T1***. This release is up-to-date with respect to T1-TARGET-SW V3.5.0.0.

Release	*Date*	*Author*	*Comment*
1.8	2016-01-05	Andreas Knickenberg	Update with latest document template and add document history. Add information about including T1_RTA.h. Update information on T1 integration template files. Re-work instructions about building with ***T1*** (see Section 7.3). Update checklists. Update naming of diagnostic interface functions. Restructure section 'Prerequisites for porting T1 to a new target and for support'.
1.9	2016-08-05	Andreas Knickenberg & Nicholas Merriam	Update with latest V850/RH850 ***T1*** exception handler name. Update Figure 11 on folder structure. Clarify privileges required for T1_Init call. Add Ethernet interface.
1.10	2016-09-01	Andreas Knickenberg	Add library order for single-pass linker process. Formatting changes.
1.11	2017-01-04	Nicholas Merriam & Andreas Knickenberg	Update T1.flex exception handler description for ARM Cortex processors. Update Figure 11. Highlight some key requirements. Add 'T1_configGen.c'.
1.12	2018-03-09	Andreas Knickenberg	Update 'Integration Preparation Checklist'. Add CAN FD. Add details on timer requirements. Add details on section allocation and alignment, TriCore sections and on memory protection requirements. Add details on disabling T1.flex

1.13	2019-05-14	Nicholas Merriam, Alexandre Baufumé, Alexander Stassis , Andreas Knickenberg	Restructure document and update description of communication scheme. Add Perl version requirement. Update code example 'Enabling/Disabling T1.flex at Startup'.
1.14	2020-06-10	Alexandre Baufumé, Andreas Knickenberg, Nicholas Merriam	Update synchronization timer requirement. Add to 6.6.2 that also write access to the trace buffer is required. Add section .T1_bssCoreComms. Add remarks on aligning TriCore SP (A10) when used in T1.flex Trampoline. Add description of .T1_constCore0. Update function names in Section 5. Add T1_InitCommunications to Section 4.1.1. Add .T1_clear section to Section 6.2.
1.15	2020-09-18	Alexandre Baufumé, Christian Herget, Hwanjoon Kang, Andreas Knickenberg	Add information about foreground T1.cont. Create OS-specific variants of this document. Add information about Lauterbach TRACE32 Interface.
1.16	2021-06-22	Nicholas Merriam, Alexandre Baufumé, Christian Herget	Clearing `.T1_bss` loses information at soft reset. Add information about 'Use Compact IDs' for RTA-OS. Add information about emulator space feature for TC1.6.2. Add further trace interface enabling option for Elektrobit tresos Safety OS.
1.17	2022-05-20	Christian Herget, Hwanjoon Kang, Andreas Knickenberg	Reworked guide for the Elektrobit tresos AutoCore OS. Renamed Section 5 *Calling **T1** Services From User Mode* to *CPU Privilege Levels* and added ISA specific information. Added information on T1_AppReadData(). Reworked section on UDS. Updated alignment requirements and T1.flex integration for Arm Cortex-M.
1.18	2022-10-21	Nicholas Merriam, Christian Herget	Document `.T1_bssChecked` section.

Table 1: Document History

Contents

1 Introduction

This document is made up of three main parts.

1. Sections 2, 3, 4, 5, 6, 7 describe in some detail the tasks to be achieved.

2. Section 9 lists the subset of tasks to be achieved by the customer *before the start* of the integration workshop.

3. Section 10 lists the remaining tasks to be achieved during the integration workshop by GLIWA and the customer working together.

If you have received this document so that you can prepare for a ***T1*** integration then please read the whole text but only perform the tasks listed in Section 9. If you experience any difficulties, please contact GLIWA for support.

1.1 *T1* Overview

The timing suite ***T1*** consists of the T1-HOST-SW which is executed on a PC and the T1-TARGET-SW which is embedded in the system under test. The T1-TARGET-SW has several interfaces: it hooks on the OS to capture certain scheduling related events, communicates via the application with the PC part of ***T1*** and comes with a few runnables which have to be called at run-time.

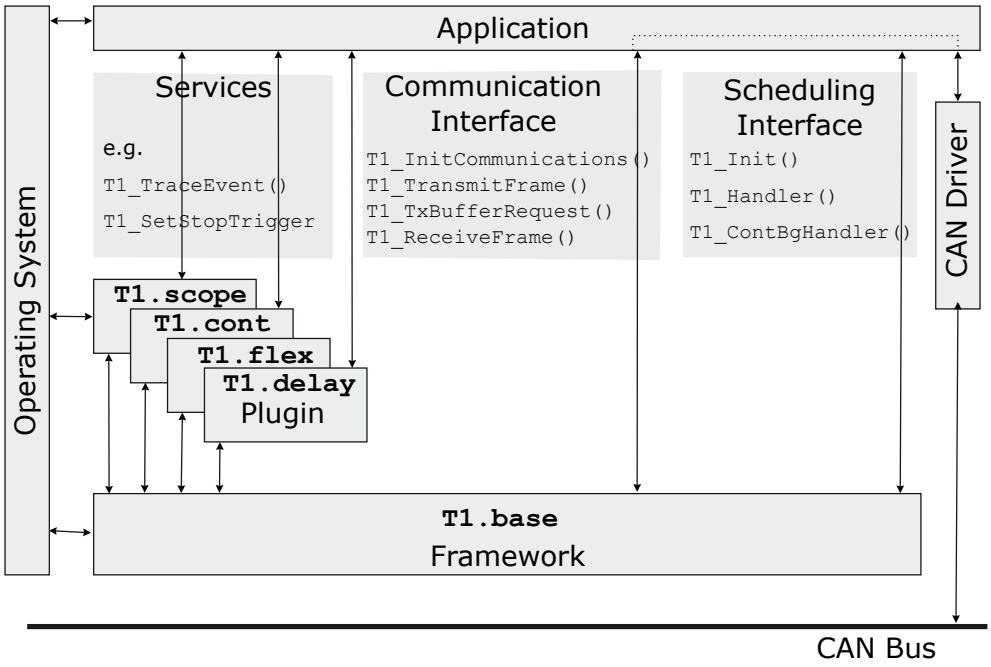

Figure 1: Interfaces of the T1-TARGET-SW (Using a CAN Interface)

2 Elektrobit tresos AutoCore OS

2.1 Enabling and Using the Trace API

'OsTrace' needs to be enabled in the Elektrobit tresos GUI as shown in Figure 2.

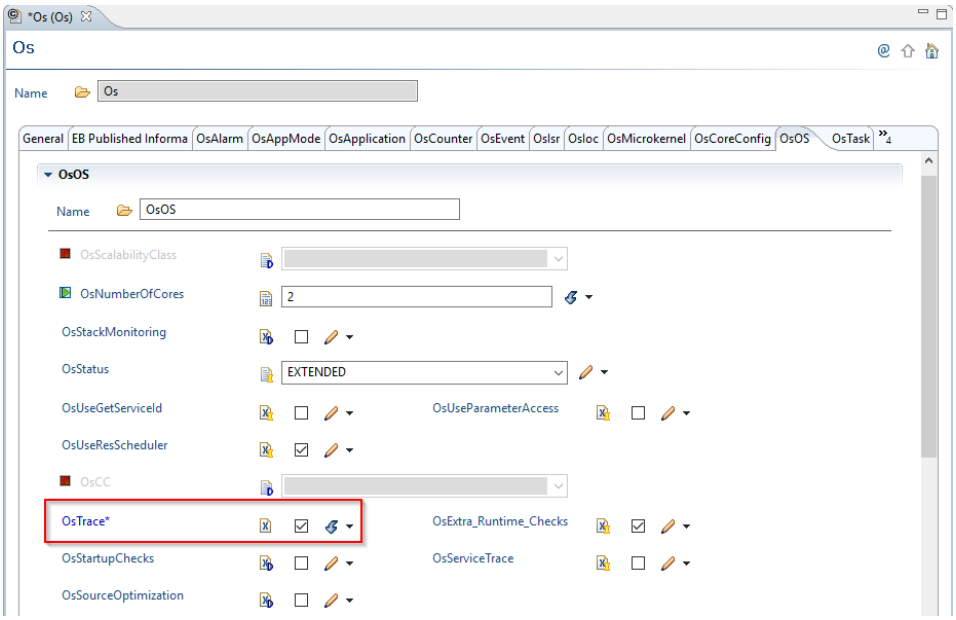

Figure 2: Enabling 'OsTrace' in Elektrobit tresos Studio

As a result, the OS generator applies macros at several points of interest whereas we are only interested in the state changes of all tasks and Category 2 ISRs.

In order to satisfy the compiler, the newly included header file *Dbg.h* (**never rename!**) is provided by GLIWA which also implements the required macros for an interface to ***T1***. These are:

`OS_TRACE_STATE_TASK( )` being called every time a task state changes. Please refer to Section 2.1.1 for implementation details.

`OS_TRACE_STATE_ISR( )` being called every time a Category 2 ISR state changes. This hook is implemented as two macros to lower the runtime overhead from tracing.

Please note, that this is a deviation from the requirements stated in the *EB tresos AutoCore OS safety application guide*, as

- symbols that start with `Os_` are reserved by the OS (**[Api.ReservedWords]**), and
- undocumented APIs are used (**[Api.Undocumented]**).

2.1.1 OS_TRACE_STATE_TASK( )

It is possible to further configure the trace interface with the following two C macros in the file *T1_TresosTraceInterface.h* and the C macro `T1_WAIT_RESUME`

which is usually defined (or not) depending on the configuration parameter `-waitResume=<true/false>`.

```
#   define T1_MAP_LITERAL_NEW_VALUES_IN_TRACE_INTERFACE (1)
#   define T1_USE_STOP_START                            (1)
```

T1_MAP_LITERAL_NEW_VALUES_IN_TRACE_INTERFACE

- If defined the C macro `OS_TRACE_STATE_TASK( )` will be mapped to different optimized C functions to lower the runtime overhead from tracing.
- If not defined the C macro `OS_TRACE_STATE_TASK( )` will be mapped to the C function `TraceStateChangeTask( )` which implements the logic to map the OS events to ***T1*** events (*i.e.* `T1_TraceEvent( )` API).

T1_USE_STOP_START

- If defined ***T1*** will defer tracing the *Stop* (task termination) event of a tasks until the resumption or start of another task. This will result in a more accurate measurement of the tasks CET at the expense of a higher overhead for tracing the tasks. Please note, that this requires an endless loop background task on each code, see Section 2.3.1.
- If not defined ***T1*** will instrument the task state transitions as they are reported by the OS. This will lead to *gaps* in the traces that might cause a slight overestimation of the Core Execution Time (CET) of lower priority tasks.

T1_WAIT_RESUME

- If defined ***T1*** will instrument *ECC* tasks using the the events *Release*, *Resume* and *Wait*.
- If not defined ***T1*** will use normal *Start* and *Stop* events to instrument *ECC* tasks. This will make *ECC* tasks appear like *BCC* tasks in ***T1***.

2.1.2 OS_TRACE_STATE_ISR( )

The function like macro `OS_TRACE_STATE_ISR( )` is mapped to two more function like macros:

OS_TRACE_STATE_ISR_OS_TRACE_ISR_SUSPENDED_OS_TRACE_ISR_RUNNING
which is used to trace the Category 2 ISRs state transition from *Suspended* to *Running*.

OS_TRACE_STATE_ISR_OS_TRACE_ISR_RUNNING_OS_TRACE_ISR_SUSPENDED
which is used to trace the Category 2 ISRs state transition from *Running* to *Suspended*.

2.1.3 TraceInit( )

The C function `TraceInit( )` is used to initialize some variables used in `TraceStateChangeTask( )` (see Section 2.1.1). This function should be called as part of `T1_AppInit( )`. A sample implementation is shown in Listing 1.

```
void T1_CODE T1_AppInit( void )
{
#if defined T1_ENABLE
    T1_uint8Least_t const coreId = T1_GetCoreIdOffset( );
    T1_errorCount[coreId] = 0; /* Clear error count */

    if( OS_CORE_ID_MASTER == coreId )
    {
        TraceInit( );
    }
    /* ... */
#endif /* defined T1_ENABLE */
}
```

Listing 1: Call of `TraceInit( )` in `T1_AppInit( )`

2.2 Reading the OS-Configuration

For Elektrobit tresos AutoCore OS reading the OS configuration is very straight forward. In *T1_OsCfg.inv* the following parameters are to be specified:

- `-osUserHeader=..\projectFolder\Os_user.h` ; name and path of *Os_user.h*
- `-osConfigHeader=..\projectFolder\Os_config.h` ; name and path of *Os_config.h*

The behavior of the OS configuration parser can be configured using the following parameters:

-appendElementType=<*true/false*> Depending on the system element type (OS object) one of the following annreviations will be appended to the name shown in the T1-HOST-SW:

ECC for Extended Tasks

CAT2 for Category 2 ISRs

CAT1 for Category 1 ISRs

CATK for OS Kernel ISRs

-mcuCore=<*MPC5xxx/V850/TMS570/ARM/TC2xx/TC3xx*> Choose the matching target architecture

-parseOsStackCfg=<*true/false*> Parse OS Stack Configuration for online stack monitoring with T1.stack

2.3 OS Tasks

2.3.1 Background Task

The OS itself has an idle loop (`OS_Idle( )`) which cannot be used to schedule the T1.cont background handler (`T1_ContBgHandler( )`) in. Instead a separate background task (priority *0*) should be configured on each core if required. Please refer to Section 4.2.3 for more information about the T1.cont background handler.

2.3.2 Task Activations

The OS only calls the trace hook if the state of a task has been changed.

This means that it will not indicate an activation event if the task is already in running or preempted state. In this case the OS will queue the task activation (multiple activation of task) and the queued activation will only get traced just before the task terminates. This behavior is visible in Figure 4 on *Core 1* `C1_QM_20ms_Multi_Act_Task`.

The OS uses cross-core interrupts (inter-core requests) to signal cross-core activations. The activation is traced on the target core, therefore there is no need to enable cross-core activations in the T1-TARGET-SW (`-crossCoreAct=false`).

2.3.3 ECC Tasks

The trace hook of the OS is not called in case the event the `WaitEvent( )` API is asked to wait for is already set. This means that ***T1*** does not get informed about this situation and does not reset the task's CET. This is different from how the OS's timing protection handles this situation. Please refer to the description of the configuration parameter `OsTaskExecutionBudget`:

> Waiting for an event that is already pending also restarts the execution timer from the beginning.

This circumstance can only be mitigated by changing the OS's source code, the relevant part of the code is shown in Listing 2. Please note that this behavior is a consequence of how the OS implements the hooks, as it only emits events if the state of a task has been changed, which is not the case here. Please see also Section 2.3.2.

```
else
{
  /*
   * We need to restart the execution timing here because we'll most likely
   * just return to the task and not enter the dispatcher.
   *
   * !LINKSTO Kernel.Autosar.Protection.TimingProtection.ExecutionTime.Measurement.Task, 1
  */
  OS_InitTaskEtb(ts);
  OS_STARTTASKEXECTIMING(ts);
}
```

Listing 2: `OS_KernWaitEvent( )` Event Already set *else* Clause

2.4 Interrupts

2.4.1 Category 2 and Counter ISR

Category 2 and Counter ISRs are instrumented using the OS supplied hook `OS_TRACE_STATE_ISR( )`.

2.4.2 Category 1, Cross Core Notification and Timing Protection ISR

These types of interrupts do not have an OS hook that can be used for tracing. Thus tracing these requires additional effort.

2.4.2.1 Direct Instrumentation

Category 1 interrupts can simply be instrumented in the the user supplied handlers themselves, as shown in Listing 3.

```
#if STM0IR1_ISR_CATEGORY == 1
ISR1(STM0IR1)
{
    T1_TraceStartNoActNoSuspPC( 0u, T1_STM0IR1_ID );

    /* ... */

    T1_TraceStopNoSuspPC( 0u, T1_STM0IR1_ID );
}
#endif /* STM0IR1_ISR_CATEGORY == 1 */
```

Listing 3: Content of *Os_config_patches.h*

Doing it like this has the drawback that the start event (`T1_START`) is traced relatively late after the arrival of the interrupt and the stop event (`T1_STOP`) is traced relatively early before leaving the interrupt. This will result in a decreased measurement accuracy. The next section describes an alternate method which can increase the measurement accuracy.

2.4.2.2 Wrapping the Dispatcher

The OS uses the following interrupt handlers as dispatchers for these types of interrupts:

CAT1 ISR `OS_Cat1Entry( )`

CATK (Kernel) ISR `OS_CatKEntry( )`, used for example for Timing Protection ISR

Cross Core Notification no wrapper is used but the handler (`OS_CrossCoreNotifyIsr( )`) is called directly

The OS has a mechanism for including a special header file called *Os_config_patches.h* for patching its configuration by globally defining the macro `OS_INCLUDE_PATCHES`. This mechanism can be exploited for wrapping the dispatchers and handler listed above. An alternative to this is to use the Linker to define wrapper functions, but this feature is not available for all toolchains. The GNU Linker can create wrappers by using the `-wrap` command line option, *e.g.*:

```
LINK_OPT += --wrap OS_CrossCoreNotifyIsr
LINK_OPT += --wrap OS_Cat1Entry
LINK_OPT += --wrap OS_CatKEntry
```

Please note, that this is a deviation from the requirements stated in the *EB tresos AutoCore OS safety application guide*, as

- symbols that start with `Os_` are reserved by the OS (**[Api.ReservedWords]**),
- undocumented APIs are used (**[Api.Undocumented]**), and
- the source code of the OS is changed although not directly, but by means of using the C preprocessor or Linker.

The code shown in Listing 4 has been tested with Infineon TriCore where the OS uses code written in assembly language to call the dispatchers and handler listed above.

```
#if defined T1_ENABLE
#   if defined OS_ASM
#       define OS_CrossCoreNotifyIsr \
            __wrap_OS_CrossCoreNotifyIsr
#       define OS_Cat1Entry \
            __wrap_OS_Cat1Entry
#       define OS_CatKEntry \
            __wrap_OS_CatKEntry
#   endif /* ! defined OS_ASM */
#endif /* defined T1_ENABLE */
```

Listing 4: Content of *Os_config_patches.h*

Listing 5 shows the code for the wrappers. Some of these might need to be adjusted based on the specific OS version. `__wrap_OS_CrossCoreNotifyIsr( )` assumes that the trace IDs for the cross core notifications are consecutive.

```
#   include <memmap/Os_mm_code_begin.h>
os_uint8_t __wrap_OS_CrossCoreNotifyIsr( os_isrid_t const iid )
{
    T1_uint8Least_t  const coreId  = T1_GetCoreIdOffset( );
    /* The OS uses the same iid value for all cores.
     * It is therefore required to adjust the ID
     */
    T1_uint16Least_t const traceId = (OS_NTASKS) + (CrossCoreNotify0) + coreId;
    os_uint8_t retVal;

    T1_TraceStartNoActPC( coreId, traceId );

    retVal = OS_CrossCoreNotifyIsr( iid );

    T1_TraceStopPC( coreId, traceId );

    return retVal;
}
#   include <memmap/Os_mm_code_end.h>

#   if OS_NCAT1_INTERRUPTS > 0
#       include <memmap/Os_mm_code_begin.h>
os_uint8_t __wrap_OS_Cat1Entry( os_isrid_t const iid )
{
    T1_uint8Least_t  const coreId  = T1_GetCoreIdOffset( );
    T1_uint16Least_t const traceId = (OS_NTASKS) + iid;
    os_uint8_t retVal;

    T1_TraceStartNoActNoSuspPC( coreId, traceId );

    retVal = OS_Cat1Entry( iid );

    T1_TraceStopNoSuspPC( coreId, traceId );

    return retVal;
}
#       include <memmap/Os_mm_code_end.h>
#   endif /* OS_NCAT1_INTERRUPTS > 0 */

#   if OS_NCATK_INTERRUPTS > 0
#       include <memmap/Os_mm_code_begin.h>
os_uint8_t __wrap_OS_CatKEntry( os_isrid_t const iid )
```

```
{
    T1_uint8Least_t  const coreId  = T1_GetCoreIdOffset( );
    T1_uint16Least_t const traceId = (OS_NTASKS) + iid;
    os_uint8_t retVal;

    T1_TraceStartNoActNoSuspPC( coreId, traceId );

    retVal = OS_CatKEntry( iid );

    T1_TraceStopNoSuspPC( coreId, traceId );

    return retVal;
}
#       include <memmap/Os_mm_code_end.h>
#   endif /* OS_NCATK_INTERRUPTS > 0 */
```

Listing 5: Interrupts Wrappers

This will result in the call tree shown in Figure 3.

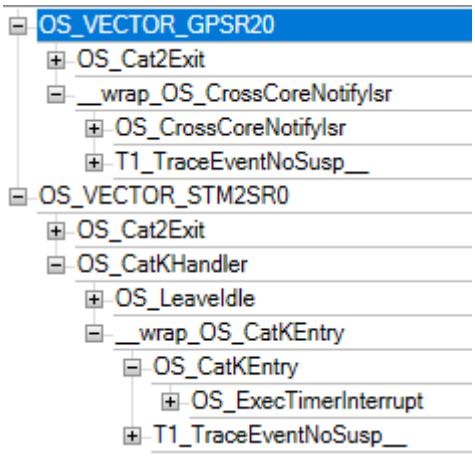

Figure 3: Call Tree With Wrappers

Figure 4 shows a T1.scope trace that contains both, a Cross Core Notification (`CrossCoreNotify1`) and a Timing Protection ISR (`OS_Execution_STM2_T1`) on *Core1*. Both are instrumented using the wrappers shown in Listing 5.

3 Trace Timer

Normally, ***T1*** reads an on-target, timer register to generate timestamps. This timer must have at least 16 bits and must run continuously. ***T1*** can be configured to work with either an up-counting or down-counting timer. The speed of the timer determines the resolution, and therefore accuracy, of all time measurements, so the configured timer must be fast enough to meet your measurement requirements. Typically, a trace timer tick is between 1 and 8 times the duration of CPU clock tick. It might be necessary to configure a new timer[1] for a ***T1*** integration, if the timers currently setup in the system do not fulfill the users requirements on the resolution or the technical requirements described later in this chapter.

T1.scope uses the lower 16 bits of the trace timer to form trace event timestamps. In order that ***T1*** recognizes each 16-bit overflow of this part of the trace timer, the maximum permitted interval between two events is 65535 ticks. If foreground T1.cont

[1] An unused PWM module or similar peripheral may provide an additional timer.

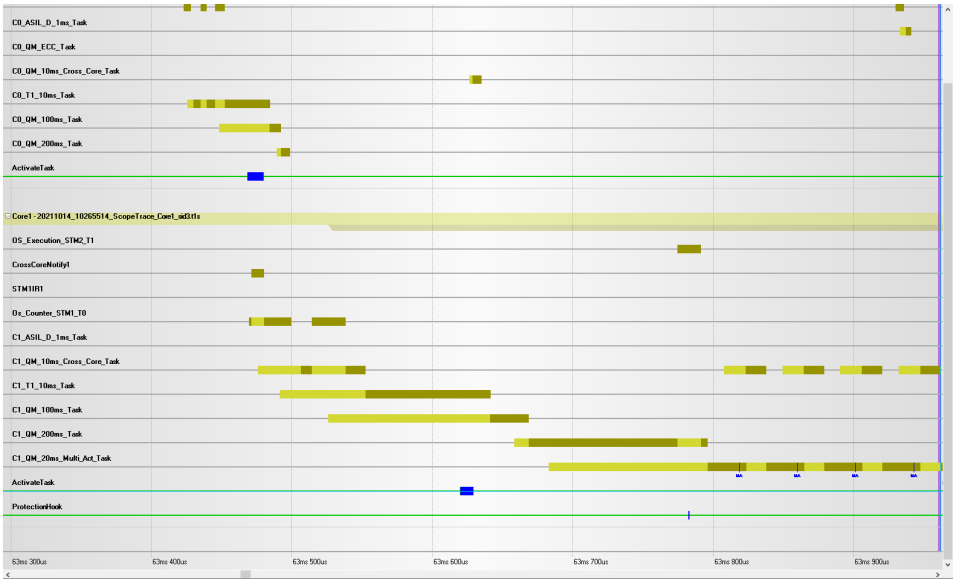

Figure 4: Trace showing Cross Core Notification and Timing Protection ISR

is used (see Section 11.1) then it is able to handle larger intervals on its own. The periodic task or interrupt with the *shortest* period can, therefore, be used to determine the minimum tick duration. Assuming 1 ms as the shortest period, the minimum tick duration is given as follows:

$$\mathrm{td}_{min} = \frac{1\mathrm{ms}}{65535} \approx 15.26\mathrm{ns} \tag{1}$$

If there is a difficulty to find or setup a suitable timer, then alternative solutions like a software-based timer shift can be implemented during the ***T1*** integration with the support of GLIWA.

On a multi-core processor, all cores can use the same trace timer, which guarantees that all traced events can be perfectly synchronized. However it is more important that each core uses a sufficiently fast timer, synchronization being possible when different cores use different timers, as long as one synchronization timer exists that can be referenced from every core. This synchronization timer should, ideally, have 32 bits, for T1 to records either 28 or 32 bits. With less than 32 bits, upper bits of the synchronization timer are computed by T1_Handler. The synchronization timer must have sufficient counting bits so that the range of the timer spans the maximum interval between successive calls to T1_Handler

4 *T1* Application Interface

4.1 Communication

Currently, the T1-HOST-SW supports CAN and Ethernet-based communication using the GLIWA Communication Protocol (GCP). It can be used either directly (pure T1 over classic CAN, CAN FD or Ethernet) or in conjunction with Unified Diagnostic

Services (UDS) via the CAN-TP transport protocol (ISO 15765-2 aka ISO-TP). UDS permits the use of a FlexRay to CAN "gateway" to connect the CAN communications from the T1-HOST-SW to a FlexRay bus of the target.

The smallest possible maximum frame size is 8 bytes, *i.e.* it must be possible to transmit 8 bytes in a single frame.

Experience has shown that it is always better to configure the payload as bytes instead of **short** or **long** types, to avoid endian-ness and byte-swapping issues.

To configure the GCP layer, two macros specify the maximum transmit and receive frame sizes in bytes, T1_GCP_MAX_TX_FRAME_SIZE and T1_GCP_MAX_RX_FRAME_SIZE. They are generated by the T1_projGen.pl Perl script into T1_config.h based on the settings in the invocation files. Those macros are used to initialize two variables which are defined in T1_AppInterface.c:

```
#   define T1_START_SEC_CONST_16
#   include "T1_MemMap.h"
const T1_frameSize_t T1_maxRxFrameBytes T1_SEC_CONST_16
    = T1_GCP_MAX_RX_FRAME_SIZE;
const T1_frameSize_t T1_maxTxFrameBytes T1_SEC_CONST_16
    = T1_GCP_MAX_TX_FRAME_SIZE;
#   define T1_STOP_SEC_CONST_16
#   include "T1_MemMap.h"
```

Note that, on a multi-core processor, the target side of the target interface is serviced by just one core, the communication core. This means that T1_ReceiveFrame must be called only on that communication core and that ***T1*** will call T1_TransmitFrame only on the communication core. By default, core 0 is used as communication core. If required, ***T1*** can be reconfigured so that any other core is used as communication core.

There are two approaches for the communication interface depending whether the target can actively send data or if a polling mechanism has to be use for the communication from the target to the host. Both approaches are described in the following sections.

4.1.1 Communication Interface Event-Driven

In this case the target is able to actively send data when required by the T1-TARGET-SW.

An overview of the function calls and dependencies in a ***T1*** integration in this case is shown in Figure 5. The function names on the 'Application' side are example names only which will be different for each integration.

Four template routines exist in T1_AppInterface.c which are used to construct this communication interface with the application, see Figure 6.

T1_InitCommunications() Is called from a T1-TARGET-SW library. It is used to initialize and reset the communication (*e.g.* free buffers).

T1_TxBufferRequest() is called from a T1-TARGET-SW library. It is used to check if a TX buffer is available and if so mark it as allocated.

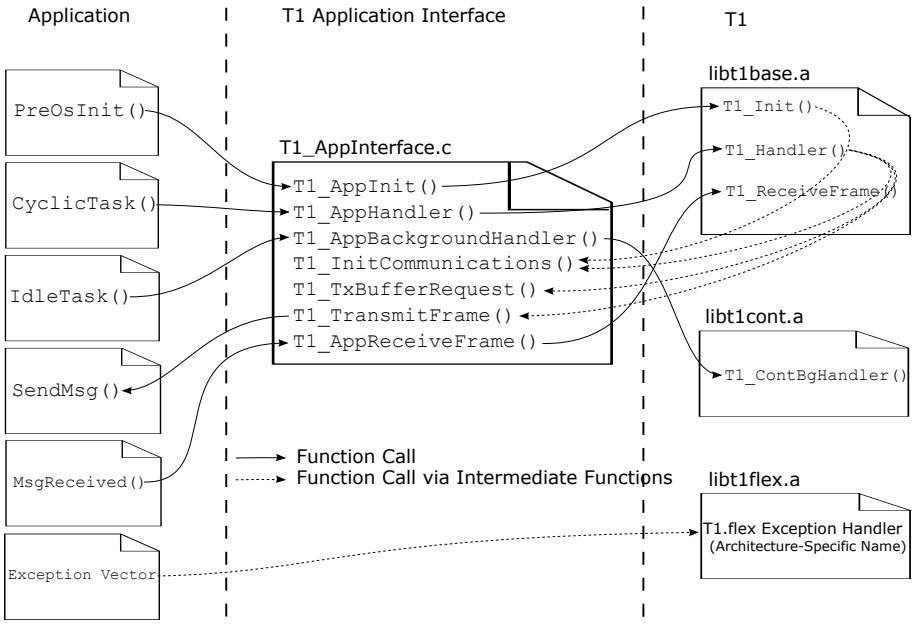

Figure 5: ***T1*** Integration Interfaces

T1_TransmitFrame() is called from a T1-TARGET-SW library. It has to be implemented by the customer to transmit `nOfBytes` bytes of data (event-driven) on the bus. Attention has to be paid regarding the return value. In case the message was successfully transferred to the communications driver, T1_OK has to be returned. In the event that that data was not queued for transmission, T1_FAILED has to be returned so that ***T1*** will re-try the transmission.

T1_AppReceiveFrame() forwards the reference to the received payload to the routine T1_ReceiveFrame which is implemented in a T1-TARGET-SW library. The referenced data will not actually be read until the next T1_Handler, so the data must buffered. In Figure 6, a software buffer is used.

4.1.1.1 CAN / CAN FD Two CAN IDs are required in order to setup the communication between the T1-HOST-SW and the T1-TARGET-SW. These can be extended or normal identifiers. The messages from the T1-TARGET-SW to the T1-HOST-SW have to be configured as event-driven messages and **must not** be transmitted periodically by the applications CAN handler.

4.1.1.2 Ethernet The T1-HOST-SW also supports Ethernet as target interface. UDP is used, the IP address and port can be configured. The Ethernet stack and TX and RX functions need to be prepared by the customer on the target side.

4.1.2 Communication Interface Using Polling

If the target cannot actively send data over the communication interface used for ***T1***, then transmit data has to be queued and polled out by some other mechanism, such as

```
void T1_CODE
T1_InitCommunications( void )
{
    T1_txFrameBufferAllocated = T1_FALSE;
}

T1_uint8_t T1_FARPTR T1_CODE
T1_TxBufferRequest( T1_UNUSED( T1_frameSize_t nOfBytes ) )
{
    if( T1_txFrameBufferAllocated )
    {
        /* Wait until buffer is free */
        return T1_NULL;
    }

    T1_txFrameBufferAllocated = T1_TRUE;
    return T1_txFrameBuffer.u8;
}

T1_status_t T1_CODE
T1_TransmitFrame( T1_uint8_t const T1_FARPTR pData,
                  T1_frameSize_t             nOfBytes )
{
    if( TRANSMIT_SUCCESSFUL( pData, nOfBytes ) )
    {
        txFrameBufferAllocated = T1_FALSE;
        return T1_OK;
    }
    return T1_FAILED; /* Demand re-try of this data. */
}

void T1_CODE
T1_AppReceiveFrame( T1_uint8_t const T1_FARPTR pData,
                    T1_frameSize_t             nOfBytes )
{
    T1_uint16Least_t i;
    for( i = 0u; i < nOfBytes; ++i )
    {
        T1_rxFrameBuffer.u8[i] = pData[i];
    }
    if( T1_OK != T1_ReceiveFrame( T1_rxFrameBuffer.u8, nOfBytes ) )
    {
        INC_T1_ERROR_COUNT( T1_commsCoreOffset );
    }
}
```

Figure 6: ***T1*** Communication Interface Functions

the standard diagnostic interface. Figure 7 shows the ***T1*** integration interfaces for that case. The relevant functions are also implemented as templates in T1_AppInterface.c and need to be adapted according to the customer's application.

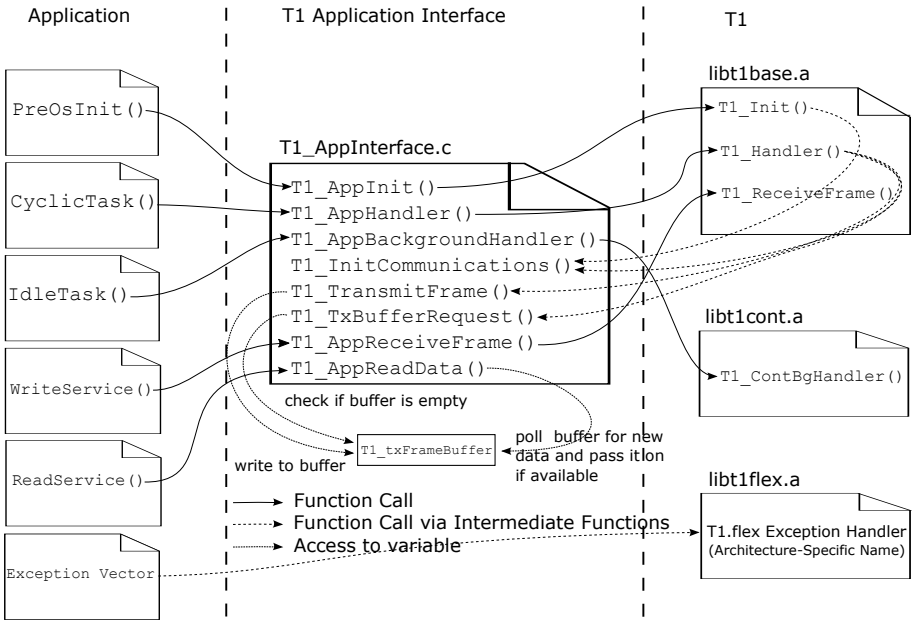

Figure 7: ***T1*** Integration Interfaces Using Polling

The following section illustrates this polling model in the context of the diagnostic interface. In this illustration, the key elements are not specific to the diagnostic interface and consist of a single-element transmit buffer plus two Boolean conditions:

1. The transmit buffer is occupied and not available for writing if and only if `T1_txFrameBufferAllocated` is non-zero

2. The transmit buffer is ready for reading if and only if `T1_txFrameNOfBytes` is non-zero.

Obviously, it is possible to encode alternative hand-shaking protocols and to support more elements in the transmit buffer but this code should provide a good basis for virtually all integrations with polled transmit.

4.1.2.1 Unified Diagnostic Services (UDS)

The T1-HOST-SW currently only supports Unified Diagnostic Services on Controller Area Networks (UDSonCAN, ISO 14229-3). It is possible to use a gateway to translate between UDSonCAN and for example *UDS on FlexRay* (UDSonFR) if the ECU does not support UDSonCAN.

With UDS, two different services are used for the communication between the T1-TARGET-SW and T1-HOST-SW:

WriteDataByIdentifier (0x2E) is used to send ***T1*** commands from the T1-HOST-SW to the T1-TARGET-SW. `T1_AppReceiveFrame( )` shall be called in response to this service request.

ReadDataByIdentifier (0x22) is used to receive ***T1*** commands from the T1-TARGET-SW by polling from the T1-HOST-SW. `T1_AppReadData( )` shall be called in response to this service request.

- If there is data pending from the T1-TARGET-SW to the T1-HOST-SW (`*pNOfBytes != 0u`) then the *Positive Response Code* along with the data (`*ppData != T1_NULL`) should be returned
- If there is no data pending from the T1-TARGET-SW to the T1-HOST-SW (`*pNOfBytes == 0u` and `*ppData == T1_NULL`) then the *Negative Response Code Conditions Not Correct* should be returned

Note: The T1-HOST-SW currently requires to have the same *Data Identifier* (DID) configured for both services (*ReadDataByIdentifier* and *WriteDataByIdentifier*). With regard to the huge ID-space this rarely poses any real problem.

The frame size is configurable.[2] The only restriction is that the maximum frame sizes must be at least 8 bytes, and that the existing diagnostic manager supports the chosen frame size.

Using a larger frame length increases the ratio of data to diagnostic packet overhead and can thus increase the speed of data transfers, particularly when downloading a ***T1*** trace and when receiving frequent "focus" updates.

To avoid an unnecessary reliance on dynamic memory allocation, our example template allocates one global transmit buffer, large enough for the maximum transmit frame bytes. Also, a variable contains the length in bytes of the actual data content. When this length variable is set to 0, it denotes an empty buffer available for (re-)allocation, see Figure 8.

`T1_AppReadData( )` writes to the output parameter `ppData` and `pNOfBytes` the pointer to the transmit buffer and the number of bytes to be transmitted, respectively. For Diagnostic interface, this function needs to be called when *ReadDataByIdentifier* is requested. Since `T1_txFrameBuffer` will be written by `T1_Handler( )` after seeing `T1_txFrameNOfBytes` and `T1_txFrameBufferAllocated` clear, it needs to be ensured that `T1_txFrameBuffer` is read out before `T1_Handler` is called. Alternatively, it can be handled by clearing `T1_txFrameNOfBytes` and `T1_txFrameBufferAllocated` not here in `T1_AppReadData( )` but in a callback that indicates the data has been transmitted. If `T1_NULL` is returned for `ppData`, negative acknowledgment needs to be transmitted to T1-HOST-SW.

4.1.2.2 Lauterbach TRACE32 Interface

Lauterbach TRACE32 can be used as ***T1*** communication interface under the following conditions

1. The TRACE32 debugger is open with the remote controlled API enabled. The remote controlled API can be enabled by adding the lines as shown in Figure 9 to the file 'config.t32'. See also https://www2.lauterbach.com/pdf/api_remote.pdf.
2. DUALPORT option is enabled in TRACE32 System Settings

[2] The frame size can even be varied at run-time but this should only be done under advice from GLIWA.

```
void T1_CODE T1_InitCommunications( void )
{
    T1_txFrameBufferAllocated = T1_FALSE;
}
T1_uint8_t T1_FARPTR T1_CODE
T1_TxBufferRequest( T1_UNUSED( T1_frameSize_t nOfBytes ) )
{
    if( T1_txFrameBufferAllocated )
    {
        /* Wait until buffer is free */
        return T1_NULL;
    }
    T1_txFrameBufferAllocated = T1_TRUE;
    T1_txFrameNOfBytes = 0u;
    return T1_txFrameBuffer.u8;
}
T1_status_t T1_CODE T1_TransmitFrame( T1_uint8_t const
               T1_FARPTR pData, T1_frameSize_t nOfBytes )
{
    if( 0u != txFrameNOfBytes ) return T1_FAILED;
    if( pData != txFrameBuffer.u8 )
    {
        /* Data not yet copied to txFrameBuffer, do it now */
        T1_uint16Least_t i;
        txFrameBufferAllocated = T1_TRUE;
        for( i = 0u; i < nOfBytes; ++i )
        {
            txFrameBuffer.u8[i] = pData[i];
        }
    }
    txFrameNOfBytes = (T1_uint16_t)nOfBytes; /* Data ready */
    return T1_OK;
}
void T1_CODE T1_AppReadData( T1_uint8_t const **ppData,
                T1_frameSize_t *pNOfBytes )
{
    *ppData = T1_NULL;
    *pNOfBytes = 0u;
    if( 0u != T1_txFrameNOfBytes )
    {
        *ppData = T1_txFrameBuffer.u8;
        *pNOfBytes = T1_txFrameNOfBytes;
        T1_txFrameNOfBytes = 0u;
        T1_txFrameBufferAllocated = T1_FALSE;
    }
}
```

Figure 8: Transmit Code for Polling Transmit

```
                    <- mandatory blank line
RCL=NETASSIST
PACKLEN=1024
PORT=20000
                    <- mandatory blank line
```

Figure 9: config.t32 for TRACE32 interface

4.2 Runnables

In addition to the function `T1_ReceiveFrame()` or the diagnostic interface routines respectively, three or four other functions must be called by the application on each core to correctly operate ***T1***. Wrapper functions are defined in the file 'T1_AppInterface.c' (see the templates) for each ***T1*** function described below. Call these at the appropriate locations in the application code.

4.2.1 T1 Initialization

To initialize ***T1***, the **timer being used must be running** and then the function

```
T1_status_t T1_Init(T1_pluginTable_t pluginTable)
```

has to be called by the application's initialization routine **on each core** that is being traced **before the OS is started**. The prototype of this function is declared in the header 'T1_baseInterface.h'.

The parameter `pluginTable` is a pointer to a table of all enabled ***T1*** plugins and not of interest for preparation.

Typically `T1_Init` is called via the "wrapper" function `T1_AppInit` in 'T1_AppInterface.c' to include error handling and any additional initialization code. Whilst other ***T1*** services can be configured to run in user mode, `T1_Init` **must always run with privileged rights**, see Section 5.

4.2.2 Cyclic *T1* Handler

To run ***T1***, the function

```
T1_status_t T1_Handler(void)
```

has to be called by the application periodically **on each core** that is being traced. The prototype of this function is declared in the header 'T1_baseInterface.h'.

If there is ***T1*** data to transmit, `T1_Handler` on the communications core calls the function `T1_TransmitFrame`, described in 4.1.1. Because at most one ***T1*** message is transmitted per call to `T1_Handler`, the period with which `T1_Handler` is called on the communications core determines the maximum communication bandwidth used by T1-TARGET-SW.

Typically `T1_Handler` is called via the "wrapper" function `T1_AppHandler` in 'T1_AppInterface.c' to include error handling and any additional periodic handler code.

On a multi-core processor, `T1_Handler` is permitted to have a different period on each core.

4.2.3 T1.cont Background Handler

If background T1.cont is used (see Section 11.1), the function

```
T1_status_t T1_ContBgHandler(void)
```

has to be called from within the application's idle task (not idle loop!) **on each core** that is being traced. The prototype of this function is declared in the header 'T1_contInterface.h'.

The simplest case is when the idle task only runs `T1_ContBgHandler`. If there is other functionality to be performed in the idle task, please notify GLIWA before the start of the integration, since the amount of data processed, and therefore the time used, by each call to `T1_ContBgHandler` is configurable.

Typically `T1_ContBgHandler` is called via the "wrapper" function `T1_AppBackgroundHandler` in 'T1_AppInterface.c' to include any additional background handler code.

4.2.4 T1.flex Exception Handler

T1.flex uses an exception handler that must be bound directly to the exception vector. It is *not* a C function that can be called from an OS interrupt handler. In AUTOSAR OS terms, it is a Category 1 interrupt and not a Category 2 interrupt.

Since the vector table is at the core of the application and its configuration is highly application-specific, *it is the responsibility of the customer to make appropriate modifications to the vector table* to allow T1.flex integration. GLIWA will provide all necessary information about the T1.flex exception handler for a specific target.

Important: If a temporary stub implementation of the T1.flex exception handler is created for testing prior to ***T1*** integration, it is the responsibility of the customer to remove that stub again during the integration workshop. If the application is providing the relevant symbol (`T1_OuterExceptionHandler` or similar), the ***T1*** library symbol will never be linked into the application.

NXP MPC5xxx / ST SPC5 implementations of T1.flex require that the debug exception vector (IVOR 15 or offset 0x90 for newer cores) be made available for *direct* handling by the ***T1*** exception handler `T1_OuterExceptionHandler` (or its core-specific versions). It is not sufficient for an OS or application handler to attempt to call the ***T1*** handler as an ordinary function.

Renesas RH850 implementations of T1.flex require that the debug exception vector (0xB0) be made available for *direct* handling by the ***T1*** exception handler `T1_OuterExceptionHandler` (or its core-specific versions). It is not sufficient for an OS or application handler to attempt to call the ***T1*** handler as an ordinary function.

Infineon TriCore (including AURIX) implementations of T1.flex do not require any additional exception vector to be made available.

The DBGTCR.DTA bit must not be changed by application software after calling `T1_Init`.

With the V1.3.1 TriCore architecture (*e.g.* TC1797, TC1387), T1.flex requires the memory protection registers. The application will be unable to use the DPR0 and CPR0 memory protection registers when T1.flex is operating. No such limitations apply to the V1.6 and V1.6.x TriCore architectures.

Please see also Section 6.4 for further details on the possible impact on the memory protection scheme of the application.

ARM Cortex M implementations of T1.flex require two additional exception vectors to be made available for *direct* handling. `T1_HardFaultHandler` has to be inserted into the vector table at exception number 3 (HardFault) and `T1_DebugMonitorHandler` has to be inserted at exception number 12 (DebugMonitor). As the hard fault vector might be used by the customer application as well, the ***T1*** library code forwards the exception to the application HardFault handler, if the corresponding exception was taken as a result of a non-***T1*** event. The configuration of this forwarding is part of the integration workshop. The symbolic name (or address, if there is no symbol) of the application HardFault handler must be noted in preparation for the integration workshop.

ARM Cortex R implementations of T1.flex require two additional exception vectors to be made available for *direct* handling. `T1_PrefetchAbortHandlerCore<X>` and `T1_DataAbortHandlerCore<X>` have to be inserted into the vector table at entries 0x0C and 0x10, respectively. The usage of variants of those handlers can be discussed during the integration workshop. As these vectors might be used by the customer application as well, the ***T1*** library offers the possibility to forward the exception to application-specific handlers, if the corresponding exception was taken as a result of a non-***T1*** event. The configuration of this forwarding is part of the integration workshop.

5 CPU Privilege Levels

All instruction set architectures (ISA) supported by the T1-TARGET-SW do implement some sort of managing execution rights. For simplicity we only consider two levels of execution rights in this document:

Privileged a mode that allows access to all hardware (special-purpose) registers which need to be accessed by the T1-TARGET-SW including manipulating those which are needed to disable and enable CPU interrupt lines.

Unprivileged a mode which does allow accessing all hardware registers.

`T1_Init( )` and `T1_InitExtra1/2/3( )` for the certified libraries are the only ***T1*** services that must always be called with privileged execution rights. Please carefully read the following paragraphs if other ***T1*** services shall be called with unprivileged execution rights.

Some services do disable and enable interrupts, these might require privileged rights depending on the ISA. Table 2 summarizes the CPU modes that allow for disabling/enabling interrupts. Please refer to Section 5.3 for more specific information.

5.1 Handling Mutual Exclusion of *T1* Services

A number of ***T1*** services require mutual exclusion with other services. The default behavior of the ***T1*** library code is to achieve this mutual exclusion by directly disabling interrupts, which would mean that ***T1*** services have to be called with in a CPU mode that allows doing this (see Table 2). If other ***T1*** services shall be called from a CPU mode that does not allow to disable and enable interrupts then the implementation of the following functions needs to be adapted in 'T1_config.c'.

`T1_SuspendAllInterruptsPC( )` Inhibit all interrupts that use ***T1*** services, typically all interrupts. The default implementation calls the ***T1*** library function `T1_SuspendAllInterruptsPC_( )`.

`T1_ResumeAllInterruptsPC( )` Restore the interrupt status again. The default implementation calls the ***T1*** library function `T1_ResumeAllInterruptsPC_( )`.

`T1_DisableT1HandlerPC( )` Commence mutual exclusion with (the code calling) `T1_Handler( )`. The default version simply calls `T1_SuspendAllInterruptsPC( )`.

`T1_EnableT1HandlerPC( )` Terminate the mutual exclusion with `T1_Handler( )`. The default version simply calls `T1_ResumeAllInterruptsPC( )`.

Please refer to Section 5.3 for details about the two ***T1*** library functions `T1_SuspendAllInterruptsPC_( )` and `T1_ResumeAllInterruptsPC_( )`.

If ***T1*** services are to be used with unprivileged execution rights when using an *AUTOSAR OS*, `T1_SuspendAllInterruptsPC( )` can normally be defined to call `SuspendAllInterrupts( )` and `T1_ResumeAllInterruptsPC( )` can be defined to call `ResumeAllInterrupts( )`. This may well be sufficient, with no need to change `T1_DisableT1HandlerPC( )` and `T1_EnableT1HandlerPC( )`.

[1] Depends on configuration, please refer to Section 5.3.4 for more details.
[2] Depends on configuration, please refer to Section 5.3.5.1 for more details.

ISA	Allowed	Not Allowed
ARMv7-M	Handler and Thread mode Thread mode with `CONTROL.nPRIV == 0` only	Thread mode with `CONTROL.nPRIV == 1`
ARMv7-R ARMv8-R (AArch32)	All modes except for user mode (PL1)	User (PL0)
Power ISA (e200)	Supervisor	User
RH850 (G3 and G4)	◦ Supervisor ◦ User[1]	User[1]
TriCore 1.6.x	◦ Supervisor ◦ User-1[2]	◦ User-1[2] ◦ User-0

Table 2: CPU Modes That Allow Disabling / Enabling Interrupts or not per ISA

5.2 Tracing Code With Unprivileged Execution Rights

The `T1_TraceEvent( )` services require sufficient right to disable and enable interrupts. The CPU modes which do allow this are ISA dependent, please refer to Table 2 and Section 5.3 for ISA specific information. Some ISAs or MPU configurations might also prevent read access to the configured ***T1*** trace timer (Section 3) with unprivileged rights.

If it is necessary to trace code with unprivileged rights, the service `T1_TraceEventNoSusp( )` can safely be used provided that no (traced) interrupts can preempt that code and provided that the trace timer can be read. Otherwise, there are three possible approaches.

1. Suspend interrupts, call `T1_TraceEventNoSusp( )` and resume interrupts again, if the trace timer can be read from user mode.

2. Switch to a mode with privileged rights, call `T1_TraceEvent( )` or `T1_TraceEventFast( )` and switch back to the previous mode.

3. Create a new trap mechanism to suspend interrupts or switch a mode with privileged rights on calls to `T1_TraceEvent( )` or `T1_TraceEventFast( )` without explicit code to do so. This will normally require the support of a GLIWA engineer.

Only if T1.flex is enabled, you can trace code with unprivileged rights using T1.flex by calling `T1_FlexTraceEvent( )`. Of course this is only suitable if you can be sure that T1.flex will always be enabled, or if the events traced this way are sensible safe to omit when T1.flex is disabled. When T1.flex is disabled, `T1_FlexTraceEvent( )` has no effect.

5.3 Implementation Details for the Supported ISAs

The following sections describe how the querying of the interrupt state, disabling and respective enabling of interrupts are done for the different CPU cores supported by the T1-TARGET-SW:

- Section 5.3.1 ARMv7-M Architectures
- Section 5.3.2 ARMv7-R and ARMv8-R (AArch32) Architectures
- Section 5.3.3 Power ISA (NXP/Freescale e200 Core Family)
- Section 5.3.4 Renesas RH850 (G3 and G4)
- Section 5.3.5 Infineon TriCore 1.6.x

5.3.1 ARMv7-M Architectures

CPUs based on this architecture allow to query the interrupt state by reading the *PRIMASK* register. The T1-TARGET-SW uses the `MRS` instruction for reading from the *PRIMASK* register. Interrupts can be disabled and enabled by using the `CPSID` and

CPSIE instructions or by writing to the *PRIMASK* register using the MSR instruction, both methods are equivalent. Setting the *PRIMASK* register to *1* will disable all exceptions and interrupts with configurable priority. Please note, that exceptions with configurable priority might get escalated to a Hard Fault exception.

For *unprivileged* accesses the *PRIMASK* register is *read as zero* and writes are ignored. Therefore executing ***T1*** services that try to disable/enable interrupts with *unprivileged* rights will result in unpredictable behavior of the T1-TARGET-SW.

5.3.2 ARMv7-R and ARMv8-R (AArch32) Architectures

CPUs based on this architecture allow to query the interrupt state by reading the *CPSR*. The T1-TARGET-SW uses the MRS instruction for reading from the *CPSR*. This must be executed in a *privileged* mode, the result will be *UNKNOWN* if executed in *user* mode and the behavior of the T1-TARGET-SW will therefore be unpredictable.

Interrupts can be disabled and enabled by writing to, the CPSR. The T1-TARGET-SW however uses the dedicated CPSID i and CPSIE i instructions to disable and enable interrupts. This must be done in a *privileged* mode, as these instructions are treated as NOP if executed in *user* mode and therefore have no effect.

Please note, that these do only disable and enable the *IRQ*, these do not disable *FIQ* and *SError*.

5.3.3 Power ISA (NXP/Freescale e200 Core Family)

CPUs based on this architecture allow to query the interrupt state by reading the *MSR*. The T1-TARGET-SW uses the mfmsr instruction for reading from the *MSR*.

Interrupts can be disabled and enabled by writing to the *MSR*. The T1-TARGET-SW however uses the dedicated wrteei instruction to disable and enable interrupts. All these instructions must be executed from *supervisor* mode, any attempt to execute these from *user* mode will cause a *privileged instruction exception*.

Please note, that the wrteei instruction will only manipulate the *EE* bit of the MSR and therefore only disable certain types of interrupts.

5.3.4 Renesas RH850 (G3 and G4)

CPUs based on this architecture allow to query the interrupt state by reading the PSW. The T1-TARGET-SW uses the stsr instruction for reading from the PSW. Reading this register is allowed in all modes.

Interrupts can be disabled and enabled by writing to the PSW. The T1-TARGET-SW however uses the dedicated DI and EI instructions to disable and enable interrupts. It depends on the state of the *MCTL.UIC* bit if these are *supervisor-level* only instructions (*MCTL.UIC* == 0) or if these are also allowed in *user* mode (*MCTL.UIC* == 1). If the core is configured so that these are *supervisor-level* only instructions, executing these in *user* mode will result in a *Privilege Instruction Exception* (PIE).

5.3.5 Infineon TriCore 1.6.x

CPUs based on this architecture allow to query the interrupt state by reading the *ICR*. The T1-TARGET-SW uses the MFCR instruction for reading from the *ICR*. The MFCR

instruction can be used in any mode.

Interrupts can be disabled and enabled by writing to the *ICR*. This must be done in *supervisor* mode to avoid an exception. However, the dedicated instructions `DISABLE`, `ENABLE` and `RESTORE` allow (only) the *IE* bit of the *ICR* to be changed and these can be executed in either *supervisor* or *user-1* mode. The T1-TARGET-SW therefore uses these dedicated instructions.

Please note, that clearing the *IE* bit of the *ICR* does only disable interrupts. Traps, including the *Non-Maskable Interrupt* (NMI), are not affected by this.

5.3.5.1 User-1 Mode

It is configurable if it is allowed to execute the following instructions in *user-1* mode or not using the *SYSCON.U1_IDE* bit:

- `ENABLE`
- `DISABLE`
- `RESTORE`

If set to *1* trying to execute one of these instructions will cause a *Privilege Violation* (PRIV) trap. Therefore executing `T1_TraceEvent( )` is not possible in this case.

5.3.5.2 Access to Peripheral Space

The peripheral space (segments E_H and F_H) is not accessible when the TriCore CPU is in *user-0* mode. On AURIX MCUs ***T1*** integrations often use the *System Timer* modules (STM) as synchronization and/or trace timers. Accessing these peripherals is not possible in *user-0* mode. Therefore executing ***T1*** services that are trying to read the one of these timers while in *user-0* mode will cause a *Memory Protection Peripheral Access* (MPP) trap.

6 *T1* Sections and Building

6.1 Kinds of Section

T1 normally employs 5 main types of linker input sections with an optional, additional near section. Depending on the the target CPU type and the number of cores those may be split up into multiple subtypes.

Code This is used for almost all the ***T1*** code. Typically it is mapped to the largest flash code memory.

Fast code ***T1*** code that must run with predictable execution time is located in the fast code section(s). Typically it is mapped to core-local flash, core-local code RAM, or locked into code cache. To provide the predictability when using flash, it is normally necessary to align functions in fast code to flash fetch line boundaries.

Read-only data This is used for read-only data and is mostly not expected to provide fast or predictable access times. Typically it is mapped to the largest flash data memory.

- There are core-specific read-only data sections (*e.g.*`.T1_constCore0`) that *are* expected to provide fast and predictable access times. These might be mapped to write-protected RAM.

Read-write data This is used for read-write data and is expected to provide fast and approximately predictable access times. Typically it is mapped to internal RAM. ***T1*** does not rely on the C start-up code to initialize this section[3], except for the section `.T1_clear`. Any such initialization by the start-up code destroys potentially valuable information after soft reset, preventing persistence of the T1.cont results, for example.

- On targets where far addressing is relatively bulky and slow, the most time-critical parts of the read-write data are located in a separate "near" section, *e.g.*with the C166 architecture. In addition to `.T1_bss` (see Section 6.2) this additional section is called `.T1_sbss`.
- If using ***T1*** on multiple cores, the read-write data sections must not be cached.[4]

Trace buffer The trace buffer itself may be very large and requires fast and predictable write access for optimal performance. Read accesses are less time-critical. Because of these different requirements, it is not located in the same section as the other read-write data. Typically it is mapped to core-local RAM but might have to be mapped to whatever RAM has sufficient space available. To allow this, it is always accessed with far addressing. ***T1*** does not rely on the C start-up code to initialize this section. Any such initialization by the start-up code destroys potentially valuable information after soft reset, preventing "post-mortem" analysis of the trace buffer, for example.

- If using ***T1*** on multiple cores, the trace buffer sections must not be cached.

Tasking disallow the use of uninitialized (noclear) sections when the `-eabi-compliant` compiler flag is set. To preserve ***T1*** trace buffer and other data through soft reset, the linker script must overwrite the section attributes, see Figure 10.

6.2 Section Naming

The following section naming as shown in Table 3 is used for the different kinds of section described above.

The reference to "bss" in the naming of the ***T1*** read-write data section is intended to indicate uninitialized memory. Note that the standard compiler-generated section

[3] Please note, that the RAM should be initialized in accordance with the application's needs. If the RAM for example is protected by ECC it normally has to be initialized to avoid spurious ECC errors.

[4] With the Infineon AURIX processor, the default behavior of the Tasking start-up code and linker is to locate `.bss.share.T1_bss` in cached LMU addresses. Take care to avoid this.

```
section_layout :vtc:linear
{
    group _dspr1 ( run_addr = mem:mpe:dspr1, attributes=rws )
    {
        select "*.T1_traceBufferCore1";
    }
}
```

Figure 10: Tasking overwrite section attributes to preserve CPU1 trace buffer

`.bss` is initialized with zero bytes at start-up time but this initialization is *not* required with `.T1_bss`, as described above.[3] In contrast, `.T1_clear` must be cleared (to all '0') at reset and will typically be handled exactly the same way as compiler-generated `.bss` sections.

The base names shown in Table 3 column *Section name* may be then extended with suffixes to denote, for example, that a section contains only 8-bit types and is located local to core 0, where appropriate. Table 3 shows the resulting section names for a dual-core ***T1*** integration. For ***T1*** integrations with devices based on the TriCore architecture or containing Arm Cortex-R CPU cores additional suffixes exist, see Section 6.4 and 6.5.

Section Kind	Section Name	Section Name *Core 0*	Section Name *Core 1*
Code	`.T1_code`		
Fast code	`.T1_codeFast`	`.T1_codeCore0`	`.T1_codeCore1`
Read-only data	`.T1_const`	`.T1_constCore0`	`.T1_constCore1`
Read-write data	`.T1_bss`	`.T1_bssCore0`	`.T1_bssCore1`
Read-write data	`.T1_bssCoreComms`		
Read-write data	`.T1_bssChecked`	`.T1_bssCheckedCore0`	`.T1_bssCheckedCore1`
Read-write data	`.T1_clear`		
Trace buffer		`.T1_traceBufferCore0`	`.T1_traceBufferCore1`

Table 3: Standard Dual-Core Section Names

6.3 Section Allocation

Table 4 shows the ***T1*** sections and their location requirements.

For each input section the second column specifies whether the corresponding section contains *eXecutable* code (X), *Read-only data* (R) or *Read/Write data* (RW). Furthermore, it is stated whether the section is time critical which in turn leads to the recommended memory type and alignment specified in the last two columns.

6.3.1 `.T1_code`

While the `.T1_code` section does not contain time-critical code the `.T1_codeFast` and `.T1_codeCore`***X*** sections contains all overhead relevant code that should preferably be located in RAM or be locked in cache in order to provide possibly low and constant ***T1*** overheads and thus reliable timing results. Code in section `.T1_codeCore`***X***

Input Section	X/R/RW	Time Critical	Alignment	Preferred Memory Type
.T1_bss	RW	Yes	32-bit	RAM
.T1_bssChecked	RW	Yes	32-bit	RAM
.T1_bssCore*X*	RW	Yes	32-bit	Core *X* RAM or shared RAM
.T1_bssCheckedCore*X*	RW	Yes	32-bit	Core *X* RAM or shared RAM
.T1_bssCoreComms	RW	No	32-bit	Core *X* RAM or shared RAM
.T1_clear	RW	No	32-bit	Shared RAM
.T1_traceBufferCore*X*	RW	No	32-bit	Core *X* RAM or shared RAM
.T1_constCore*X*	R	Yes	64-bit	Core *X* RAM or locked cache or flash
.T1_const	R	No	32-bit	Flash
.T1_codeCore*X*	X	Yes	64-bit	Core *X* RAM or locked cache or aligned in flash
.T1_codeFast	X	Yes	64-bit	RAM or locked cache or aligned in flash
.T1_code	X	No	64-bit	Flash

Table 4: Requirements for ***T1*** Input Sections

is more time-critical than code in section .T1_codeFast. The alignment of the .T1_codeFast and .T1_codeCore*X* sections should ideally be cache-line aligned.

6.3.2 .T1_bssCore*X* and .T1_traceBufferCore*X*

These sections contain data that is accessed from core *X* and from the communications core. Data accessed only from the communications core is in section .T1_bssCoreComms. Speed and predictability are very important for .T1_bssCore*X* so this should be located in any core-specific data memory such as the TriCore DSPR but access must be permitted also from the communications core. If using ***T1*** on multiple cores, the read-write data sections *must not* be cached. ***T1*** uses only 32-bit accesses to .T1_traceBufferCore*X*.

6.3.3 .T1_bssChecked and .T1_bssCheckedCore*X*

These sections have a special purpose for the T1-TARGET-SW-CERT only. It can be treated as .T1_bssCore*X* otherwise. Please refer to the Safety Manual of the T1-TARGET-SW-CERT for more details.

6.3.4 .T1_constCoreX

This contains constants that need are accessed from time critical ***T1*** APIs. Therefore it should go into cached flash or into core-local RAM. If it is located in core-local RAM

it might require write protection to ensure it is never overwritten (***T1*** will not attempt to modify it) and has to be initialized before calling `T1_Init`.

6.4 TriCore-Specific Sections

In order to ensure that at least the hardware handling of debug exceptions can be handled without a memory protection unit (MPU) violation, the TriCore memory protection is masked for the addresses around DMS (the entry address of the handler) and DCX (the address of the debug context save area). The following extract from the TriCore user manual defines the concept of "emulator space" and defines exactly the masking behavior.

> To enable the debug monitor to operate without requiring the modification of the current memory protection settings, the following protection modifications are applied in debug mode:
>
> - The 16 MByte region containing the DMS pointer (Base address == (DMS & 0xFF000000) will have MPX and peripheral space PSE traps disabled for instruction fetches in debug mode.
> - The 16 MByte region containing the DCX pointer (Base Address == DCX & 0xFF000000) will have MPR and PMW traps disabled for load and store operations in debug mode.
>
> These two memory regions are referred to as emulator space.
>
> The cacheability of emulator space depends on the memory attributes assigned to the segments in which they reside, by the PMA registers.

By default, ***T1*** integrations locate DMS in the normal code segment and DCX in the normal data segment. The result of this is that when T1.flex is enabled (DBGSR.DE == 1), memory protection violations are not detected at all.

The TC1.6.2 architecture (starting with B-step only) allows this emulator space feature to be disabled by setting `SYSCON.ESDIS`. In this event, or in the event that no memory protection would be compromised anyway, set `-useDMStrampoline=false` to bypass the work-around described below. In that case, section `.T1_codeDMSCore`***X*** is not used and need not be mapped to an output section.

T1 uses dedicated section names for the T1.flex exception handler trampoline (a single jump instruction in `.T1_codeDMSCore`***X***) and debug context save area (`.T1_bssDCXCore`***X***). By carefully locating these sections, the DMS and DCX values can be set to contain addresses that do not compromise the memory protection required by the other software.

We generally recommend locating the ***T1*** DMS sections in the uncached program flash starting at address 0xA0000000. The reset value for DMS is in segment 0xA and so locating the ***T1*** DMS in segment 0xA should minimize any new MPU issues. Another segment to consider is 0xC, which only provides access to the core-local PSPR and may, therefore, not require memory protection. With an emulation device, the DMS sections can also be located in EMEM, providing an additional alternative, albeit with potentially much poorer performance.

Although the reset value for DCX is in segment 0xA, the ***T1*** DCX sections cannot be located there because write access is required. The LMU RAM may be used for ***T1*** DCX sections, providing one alternative location to the default RAM memory. With an emulation device, the DMS sections can also be located in EMEM, providing another alternative.

Warning: Just as for any other TriCore context save area, the ***T1*** debug context save area in `.T1_bssDCXCore`*X* must be aligned to a 64-byte boundary. This is normally ensured with no adaptation because ***T1*** uses the appropriate compiler-specific keyword to achieve this alignment. However, with the HighTec compiler, it is possible to misalign `.T1_bssDCXCore`*X* by overriding the alignment in the linker file using the `ALIGN` keyword immediately after the output section name. Please ensure that such alignment is either at least 64 bytes or that the alignment is applied not to the output section but to the contained input sections by adding `ALIGN` *after* the colon following the output section name. Alignment applied to the input sections does *not* override the alignment provided in the ***T1*** C source code.

6.5 Cortex-R-Specific Sections

The ARM Cortex-R libraries contain two different kinds of T1.flex exception handler. The "SP" variant expects to be reached by a trampoline that uses (corrupts) the stack pointer (SP) register. As such, they have no special linkage requirements and are placed in standard sections just as described in Section 6.3.

If, however, the T1.flex exception handler must be reached by a relative jump from the vector table, a non-SP variant expects to be reached with all registers intact, including the stack pointer, and in ARM (not Thumb) mode. Because it needs to be located sufficiently close to the vector table to be reached with a relative jump, the non-SP variants are located in dedicated core-specific sections `.T1_codeHandlerCore`*X*. The common, not core-specific handlers which can be used as alternative are located in a common section `.T1_codeHandler`.

6.6 Memory Protection

If memory protection using an MPU is implemented in the application, this needs to be considered for the ***T1*** integration. ***T1*** uses shared global variables to exchange data within a core and in multi-core ***T1*** integrations between the cores as well. The following access permissions are required by the different ***T1*** components.

6.6.1 T1_TraceEvent

`T1_TraceEvent` requires write access to the trace buffer (`.T1_traceBufferCore`*X*) and read-write access to `.T1_bss`, `.T1_bssCore`*X* and `.T1_bssCheckedCore`*X* from core *X*.

6.6.2 T1_Handler

`T1_Handler` requires read-write access to `.T1_bss` and, for multi-core integrations, `.T1_bssCore`*X* from core *X* and read-write access to the trace buffer

(.T1_traceBuffer or .T1_traceBufferCore*X*). Additionally, on the communication core T1_Handler requires read-write access to .T1_bssCore*X* on all cores.

6.6.3 T1_ContBgHandler

T1_ContBgHandler requires read-access to the trace buffer (.T1_traceBufferCore*X*) and read-write access to .T1_bss, .T1_bssCore*X* and .T1_bssCheckedCore*X* from core *X*.

6.6.4 T1_ReceiveFrame

T1_ReceiveFrame requires read-write access to .T1_bss and T1_ReceiveFrame called on the communication core requires read-write access to .T1_bssCore*X* on all cores.

6.6.5 T1.flex Exception Handler

The T1.flex exception handler requires write access to the trace buffer (.T1_traceBufferCore*X*) and read-write access to .T1_bss, .T1_bssCore*X*, .T1_bssChecked and .T1_bssCheckedCore*X* from core *X*.

6.6.6 ARM Cortex-R Specific Settings

For T1.flex to work, it is possible to configure the address range for the ARMv7 memory-mapped debug interface as Strongly-ordered memory or Device memory. However, if this region is configured as Device memory, this should result in a reduced T1.flex overhead.

6.7 Linking Sequence

As is conventional, ***T1*** libraries should be linked after all application object files and before the C library. In particular, the libraries must be linked after the object files 'T1_AppInterface.o', 'T1_config.o' and 'T1_configGen.o'. If a single-pass linker is used then the libraries need to be in the following order:

- libt1cont.a
- libt1delay.a
- libt1mod.a
- libt1flex.a
- libt1scope.a
- libt1base.a

6.8 Build Sequence

Where a integration includes a Perl script to automatically configure ***T1*** to match the OS configuration, the Perl script must execute after the OS configuration tool and before compiling 'T1_config.c' and 'T1_configGen.c', which include a header file generated by the Perl script. If using a Makefile, the correct sequence will be guaranteed by including the correct file dependencies.

Correct build sequencing is especially important when using the build identifier (BID) to ensure consistent versioning.

6.9 Alignment Requirements

In Section 6.4 one specific alignment requirement is already described. Furthermore it needs to be ensured that the following variables, each of them having the suffix 0...N-1, are 4-byte aligned. This should be automatically ensured by the T1-TARGET-SW but shall be double checked in the resulting map-file:

- `T1_taskAct`
- `T1_swdStart`
- `T1_featureMask`
- `T1_flexGlobals`
- `T1_addrs`
- `T1_modGlobals`
- `T1_delayGlobals`
- `T1_delays`
- `T1_taskData`
- `T1_focus`
- `T1_vStpw`
- `T1_taskStack`
- `T1_resultBuffer`
- `T1_csrnData`
- `T1_stpwData`
- `T1_contGlobals`
- `T1_baseGlobals`
- `T1_traceBuffer`

7 Files and Folders

T1 is delivered as a set of C libraries with the corresponding C headers. Additionally, there is set of application specific files which hold the configuration and the interface functions described in Section 4.1. All these files have to be added to the application's build-process.

7.1 Recommended Folder Structure

Some applications come with a rather strict guideline regarding the location of third party libraries/headers and source files. ***T1*** has no requirements regarding the folder structure so that all these guidelines can be easily fulfilled. If these guidelines are more relaxed or if there are no restrictions at all, we recommend the file and folder structure as shown in Figure 11. This structure has the advantage that most projects use it which simplifies any support activities because GLIWA employees are familiar with it. Please note that not all files shown in Figure 11 are required for each ***T1*** integration.

7.2 Template Interface Files

To facilitate preparation, templates for the files 'T1_AppInterface.c' and 'T1_AppInterface.h' are provided. Those files contain templates for the wrapper functions that should be called from within the application. All template functions may be renamed with the exception of `T1_TransmitFrame()` when using the event-driven communication. For preparation, integrate the files into your project. As described above, call on each core

- `T1_AppInit()` from your initialization routine before calling any other ***T1*** services and before the OS is started.
- `T1_AppHandler()` from a cyclic task
- `T1_AppBackgroundHandler()` from your idle (background) task

For the event-driven communication, adapt the parameter of the function `T1_AppReceiveFrame` to fit the requirements of the application (for the first step this might be left unchanged).
Call this function from the communication stack or another appropriate location in your application.

If using the diagnostic interface, adapt the parameters of the two target functions `T1_AppDgn_ReadDataByIdentifier` and `T1_AppDgn_WriteDataByIdentifier` to meet the requirements of the application (for the first step these might be left unchanged). Call these functions from the diagnostic handler, or another appropriate location of your application.

7.3 Building With *T1*

Compile and link your project, initially without defining `T1_ENABLE`. It should build and run on the target, as the ***T1*** components are still disabled in this case.

After that, define `T1_ENABLE` at the top of 'T1_AppInterface.h' or via a compiler option. Now the compiling and linking should sill work without errors, however the

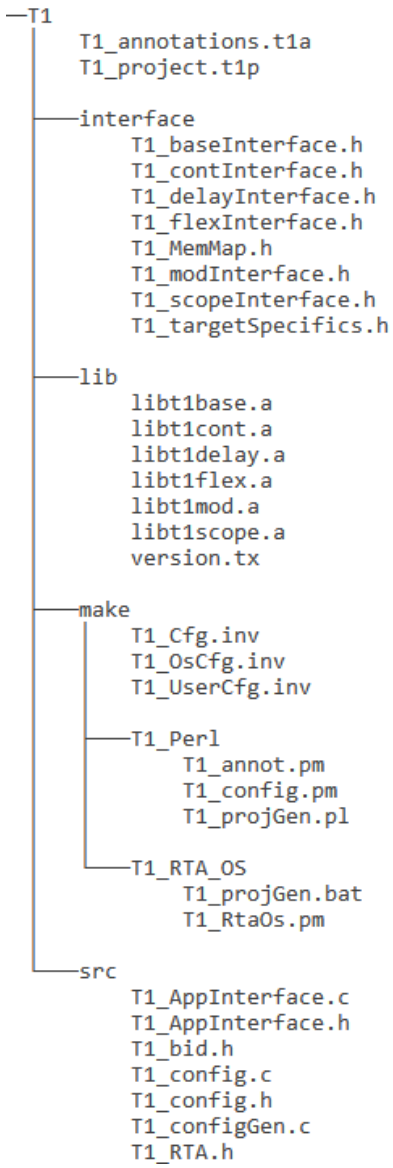

Figure 11: Recommended Folder Structure

software may not run at this stage as not all ***T1*** settings for this project have been verified. This will be done during the ***T1*** integration workshop.

7.4 Archiving Files for Support

Our experience with past integrations has shown that supporting a ***T1*** integration is greatly facilitated if all ***T1***-related, application-specific files are archived by GLIWA. Typically, these files are not IP-relevant and thus can be transferred to GLIWA even without a non-disclosure agreement. In order to be able to reconstruct all activities of the ***T1*** integration, we would like to store all files which were either modified or added in two versions: before and after the ***T1*** integration.

8 Prerequisites for Porting *T1* to a New Target

If the porting of ***T1*** to a new target has to be done, this needs to be completed at GLIWA prior to the integration workshop. For this task it is necessary to provide the following equipment to GLIWA before the porting can be started:

- Compiler and license. When providing evaluation licenses, it must be verified that the code limit will not be reached.
- A demo project that is editable, uses the same ABI and compiler as the final application and compiles without errors. The demo project does not have to be the final application code, nor provide the final OS. If a provided OS is not the final OS, it must be one already supported by GLIWA.

Additional requirements arise when T1.flex is to be ported to a new target:

- Debugger and license: Source level debugging is required
- Final hardware or demo board with the final processor core

The correct functionality of the demo project and the debugger is to be demonstrated on commissioning. If the corresponding prerequisites are not fully met and additional effort is required, for example to solve license problems, GLIWA will charge the additional work at regular engineering rates. The corresponding equipment needs to reside at GLIWA for as long as support is required, ideally permanently.

9 *T1* Integration Preparation Checklist

Before the integration workshop takes place, the following preconditions should be met:

[ ] OS and interrupt details provided to GLIWA for off-site preparation
[ ] Sufficiently fast timer enabled with 16 bits or more
[ ] Communication interface prepared and TX and RX tested
[ ] New modules ‘T1_AppInterface.c’, ‘T1_config.c’ and ‘T1_configGen.c’ added to build process
[ ] `T1_AppInit()` gets called in the initialization phase of each core
[ ] `T1_AppHandler()` gets called periodically on each core (*e.g.* every 2..10ms)
[ ] `T1_AppBackgroundHandler()` gets called in the background task of each core
[ ] If T1.flex is to be used, the required exception vector(s) are allocated
[ ] The build process includes the T1-TARGET-SW libraries
[ ] The application builds and runs without `T1_ENABLE` being defined
[ ] The application compiles and links with `T1_ENABLE` being defined
[ ] The application can be source-level debugged
[ ] The application also starts without debugger connection after reset
[ ] The build process can execute a Perl script (Perl v5.8.8 or higher) before compilation
[ ] Admin rights are available for the PC on which the T1-HOST-SW shall be installed

If these preconditions cannot be met, the ***T1*** integration workshop may need to be postponed to ensure a successful ***T1*** integration in the allocated time-frame.

10 *T1* Integration Workshop

During the on-site ***T1*** integration workshop, a GLIWA representative will complete the necessary connections between the application and ***T1*** together with the responsible contact at the customer and verify the ***T1*** integration. The specific activities normally include:

[ ] Verify initialization call to `T1_AppInit()`
[ ] Verify periodic calls to `T1_AppHandler()` and configure the actual period
[ ] Verify background calls to `T1_AppBackgroundHandler()`
[ ] If T1.flex is to be used, verify T1.flex configuration
[ ] Check mapping from OS task identifiers to ***T1*** task identifiers
[ ] Check mapping from ISRs to ***T1*** ISR identifiers
[ ] Add call to ***T1*** Perl script to build process and verify its output
[ ] Provide necessary adaption for any calls from user mode
[ ] Verify ***T1*** communication link
[ ] Determine and verify capacity for streaming (optional)
[ ] Verify correct behavior of each ***T1*** plug-in
[ ] Optimize integration for size and speed
[ ] Archive modified and newly created files after the ***T1*** integration workshop
[ ] Demonstrate ***T1*** features in the context of the integration

11 Further *T1* Integration Topics

11.1 Background vs. Foreground T1.cont

There are two fundamentally different ways how the T1.cont processing can be integrated into the customer's application code: either the T1.cont processing is done in the background during the idle time of the system or it is done in the foreground in the context of the tracing of ***T1*** events. Both approaches come with significant differences which shall be summarized here. Note, that switching between foreground T1.cont and background T1.cont can be done with a minimum of modifications at build time.

	Background T1.cont	**Foreground T1.cont**
T1.scope Overhead and Interrupt Blocking	Small and constant	Larger and varying
Processing	Done during idle time, execution not guaranteed	Done in the foreground, execution guaranteed even in case tracing is halted
Default Timing Parameters	CET(max)	CET(min/max), RT(max), IPT(max)

Intervals > 65535 Ticks	Not supported	Can be handled
Cross-Core Event Chains	Not supported	Supported
Multiple Activation of Tasks	Not supported	Supported (IPT and RT, up to 128 activations)

Table 5: Background vs. Foreground T1.cont

11.2 Disabling T1.flex

Depending on the combination of CPU core and debugger, conflicts during parallel usage of T1.flex and the debugger can arise. Therefore T1.flex may need to be disabled in certain cases.

11.2.1 Disabling T1.flex at Compile-Time

T1.flex can be disabled at compile-time by defining the macro `T1_DISABLE_T1_FLEX` at the top of 'T1_AppInterface.h'.

11.2.2 Disabling T1.flex at Run-Time

It is also possible to determine at run-time if T1.flex shall be enabled or not. The code in Figure 12 illustrates this.

In the default `T1_pluginTable`, T1.flex is enabled or disabled according to `T1_DISABLE_T1_FLEX` and, in this case, we assume that `T1_DISABLE_T1_FLEX` is not defined and that T1.flex is enabled. Additionally a second plugin table is defined here using the macro `T1_ALLOCATE_PLUGIN_TABLE_NO_T1_FLEX`. In that plugin table T1.flex is always disabled. In `T1_AppInit`, one of the two plugin tables is used to start ***T1*** depending on a condition.

This example uses the variable `startT1flex` (which could be stored in NVRAM), but any other implementation can be used as well. Alternatively an I/O line could be checked or `T1_AppInit` could use an additional parameter that determines whether or not to start T1.flex. Also the state of the debugger connection could be determined and T1.flex could be disabled when the debugger is attached.

11.3 Elektrobit tresos AutoCore OS Specific Instrumentation

11.3.1 Tracing of OS Services

The Elektrobit tresos AutoCore OS allows for tracing the

- Entry
- Exit
- Function parameters

- Return value

of OS services (OS APIs).

The OS C source file *Os_trace.h* contains all valid trace hooks. All trace hooks are implemented as C macros and are empty by default. The hooks for the OS services can for example be instrumented using ***T1*** user stopwatches or events. An code example is shown in Listing 6, this example can also be found in the file *Dbg.h*.

```
#   define OS_TRACE_ACTIVATETASK_ENTRY( taskId_ ) \
        T1_TraceEvent( T1_STOPWATCH_START, T1_SW_ActivateTask_CORE_ALL )
#   define OS_TRACE_ACTIVATETASK_EXIT( )          \
        T1_TraceEvent( T1_STOPWATCH_STOP, T1_SW_ActivateTask_CORE_ALL )

#   define OS_TRACE_CALLTRUSTEDFUNCTION_ENTRY( tfIndex_ ) \
         T1_TraceEvent( T1_STOPWATCH_START, T1_SW_CallTrustedFunction_CORE_ALL )
#   define OS_TRACE_CALLTRUSTEDFUNCTION_EXIT( )           \
         T1_TraceEvent( T1_STOPWATCH_STOP, T1_SW_CallTrustedFunction_CORE_ALL )
```

Listing 6: Example for Instrumentation of OS Services Using ***T1***

```
/* ... in T1_config.c */
#define T1_START_SEC_CONST_32
#include "T1_MemMap.h"
T1_ALLOCATE_PLUGIN_TABLE( T1_pluginTable );
T1_ALLOCATE_PLUGIN_TABLE_NO_T1_FLEX( T1_pluginTableNoT1Flex );
#define T1_STOP_SEC_CONST_32
#include "T1_MemMap.h"

/* ... in T1_AppInterface.h */
#define T1_START_SEC_CONST_32
#include "T1_MemMap.h"
T1_DeclarePluginTable(T1_pluginTable);
T1_DeclarePluginTable(T1_pluginTableNoT1Flex);
#define T1_STOP_SEC_CONST_32
#include "T1_MemMap.h"

/* ... in T1_AppInterface.c */
void T1_CODE T1_AppInit( void )
{
...
  if( startT1flex ) /* e.g. startT1flex in NVRAM */
  {
    if( T1_OK != T1_InitPC( coreId, T1_pluginTable ) )
    {
      for( ;; )
      {
        /* Infinite loop to trap init failure */
      }
    }
  }
  else
  {
    if( T1_OK != T1_InitPC( coreId, T1_pluginTableNoT1Flex ) )
    {
      for( ;; )
      {
        /* Infinite loop to trap init failure */
      }
    }
  }
...
}
```

Figure 12: Enabling/Disabling T1.flex at Startup

GLIWA
embedded systems

www.gliwa.com

GLIWA embedded systems
GmbH & Co. KG
Pollinger Str.1
82362 Weilheim i.OB
Germany

fon +49-881-138522-0
fax +49-881-138522-99
info@gliwa.com

Geschäftsführer (CEO) Peter Gliwa
V.A.T. No. DE814169157
D.U.N.S. 551053924