Detailed Specification

Common Timer Base

Department:	Aalborg Wireless Center
Creation Date:	7 February, 2003
Last Modified:	24 March, 2003 by Carsten Schmidt
ID and Version:	8434.516.02.005
Status:	Accepted

0 Document Control

0.1 Document History

ID	Author	Date	Status
8434.516.02.001	CSH	27 May, 2002	Being Processed
8434.516.02.002	CSH	25 June, 2002	Being Processed
8434.516.02.003	CSH	19 February, 2003	Being Processed
8434.516.02.004	CSH	7 March, 2003	Being Processed
8434.516.02.005	CSH	24 March, 2003	Accepted

0.2 References, Abbreviations, Terms

[TI 8010.801] 8010.801, References and Vocabulary, Texas Instruments

Table of Contents

1 Introduction

This document resumes the discussion/analysis for a common timer base solution made in "GPF timer control" [TI 8434.514] and the analysis "timer control" [TI 7010_975]. Besides the analysis the document also contains a high level part, identifying needed primitives. The document also describes the necessary changes in the tools that need changes for supporting CTB. Finally the document also contains a little design specification for the TAP and FRAME, because of missing DTS documents for these GPF components.

Common for all described scenarios and the presented solution are that it is only meant for a host testing solution.

1.1 Different test scenarios

This subsection will shortly describe the different test scenarios, for which CTB is needed. Common for all scenarios are that they are not "real time" test cases. Figure 1 shows a testing session with the tap2 application. The tap2 runs on the Condat Multitasking System (CMS) with a frame. It communicates with the stack through a simulated USART (tst.exe). The TST.exe contains the test interface (TIF) to the protocol stack. On the stack side there's a task called TST and RCV which sends and receives information.

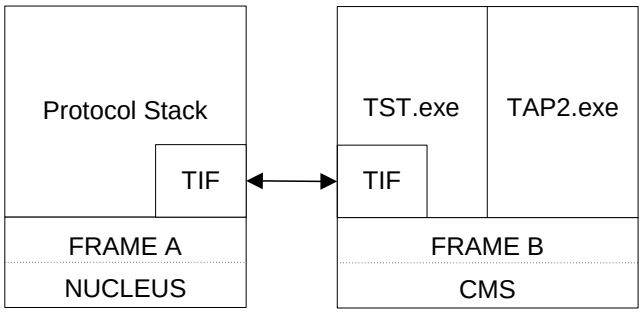

Figure 1 Testing with tap2

When testing the protocol stack, tap2 can be closed and restarted without closing the stack.

When running system integration test with the Anritsu system tester the scenario are as in Figure 2. PAL2 is a glue layer between the TI PHY/CPHY SAPs and the Anritsu layer 1 interface. Communication between PHY Stub and PAL2 is via the TIF interface, and communication between PAL2 and Anritsu Layer 1 is via a TCP/IP socket interface.

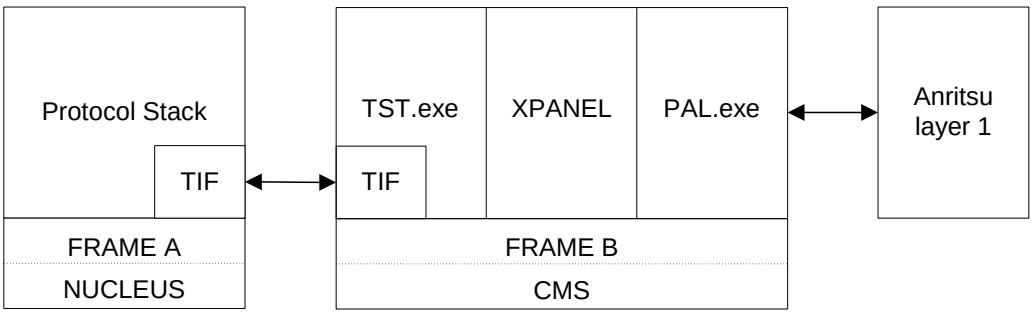

Figure 2 Testing with Anritsu

The pal application can be closed and started again without closing the protocol stack.

Besides these two scenarios there is a possible third scenario that must be taken into account although it is not seen today. The scenario is depicted in Figure 3. The scenario describes testing with Anritsu as well as with tap2 and it is actually a combined testing from the previous two scenarios.

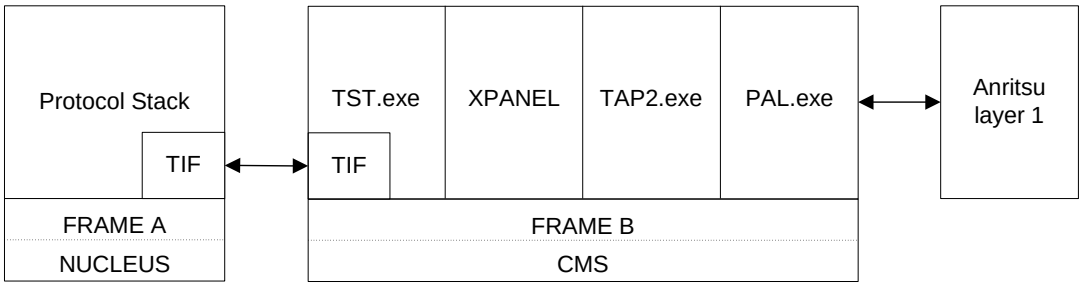

Figure 3 Testing with Anritsu and TAP2

This scenario is relevant – if for example some Anritsu Test case will bring the protocol stack into a special state and then we stimulate the stack with some module or UMTS integration test case from the tap application. Nothing special should be done here, since the PAL2 should be enabling CTB and the TAP2 should be started as normal.

1.2 Identifying the problem

This section shortly identifies the problem when testing in the previous mentioned test scenarios. For more detailed information please read [TI 7010.975].

1.2.1 Testing with TAP2

When running module or integration test cases with the tap2 application most often the Protocol Stack are stopped because of debugging. This results in a timeout in the tap2 causing the test case to fail.

With TDC (test description code) it is possible to setting breakpoints in test cases from Visual Studio (more easily than in TDS). This means that the tap2 application can be stopped and cause unexpected behaviour in the protocol stack, because of timeouts of previously started timers.

Another issue is when the both frames are in idle mode (nothing to do) for an amount of time. Naturally it is not preferable to wait that time – instead it should be skipped and the testing should continue.

1.2.2 Testing with Anritsu VST

Testing with Anritsu is not real-time. The Anritsu can be slow meaning that time “progresses” faster in the PS than on the Anritsu. This can cause time outs of previously started timers in the PS.

When the PAL2 is being debugged the same problem can occur. When testing today some timers for the entity CC are even disabled in the protocol Stack to secure the behaviour of the protocol stack so that the test cases passes.

1.2.3 The problem

After listing the problems we can now identify the cause of the problem. Basically it is because the two frames aren’t aware of each other. In case of testing of with PAL2 the problem is also that the Anritsu is

asynchronous to both frames.

They frames run asynchronous. This means that for instance FRAME A can be stopped and FRAME B will continue because it has no knowledge about FRAME A. We therefore need to have implemented something into the two frames – so that they can be aware of each other. This requires changes in the tools (PAL2 and TAP2) and the frame, so that it can be configured to run in a mode, that avoids the mentioned problems.

1.3 Frame, timers and nucleus information

This subsection contains some relevant information found during the development of this document.

During start up of the stack, when initialising all the tasks, the frame call os_SuspendTask several times for most of the entities. This function depends on timer ticks, so that suspend time can expire and the processing can continue. We need to have a timer tick on start up of the frame – e.g. use the Windows timer tick. Later it should then switch to another configuration – initiated by the test environment. Inside Nucleus the timer tick interrupt is simulated by starting a normal Windows timer. This timer runs in it's own thread. The started Windows timer expires after 50ms. This means every time the callback function in nucleus is called 50ms of time has passed.

2 CTB Concept

This section contains a short description of the CTB concept. CTB is only required for host testing, together with the TAP2 or together with PAL2[1].

To ease understanding of the following concept we distinguish between, an "internal" frame tick configuration (normal mode), with timer ticks generated from Windows, and an "external" frame tick configuration, where the ticks only occur on requests from an external entity (CTB). The basic features of this concept have been implemented and tested successfully.

The basic idea is to control the time passed inside the stack from the test tools (PAL2 / TAP2). This means that the interrupt simulation of the nucleus timer ticks should be disabled, so that the entire protocol stack environment only "passes" time, when the test environment tells it to do so. In the Nucleus (host version) the timer interrupts are simulated through a Windows timer that fires every 50ms. The timer process calls a "timer expire" function, that increments the time in Nucleus. This Windows controlled timer should be disabled, when CTB is enabled. The function called, when the timer expires, should be called from the PS FRAME/TST on requests. This way the complete timing behaviour in the stack would be the same, as if the ticks were occurring periodically.

The ticking should only happen when the stack is in idle mode. This requires an idle task, which can tell the stack, when there's nothing to do.

On the tool side the frame should run normally – e.g. the timer ticking is not disabled here. This way changes are avoided in the tool frame (CMS).

When testing with the Anritsu VST, the stack should be controlled from the ticks generated from the tester. Every tick generated from Anritsu corresponds to 10ms. Since the stack timer "resolution" is 50ms, PHY has to receive 5 tick requests, before telling the stack to tick.

When testing with TAP2, the stack should be asked to "spend" time, every time the TAP2 is in "idle" mode. The TAP2 is considered to be in idle in following states:

- AWAIT events (Idle time is the default timeout specified when starting the TAP2)
- MUTE events (Idle for the time given as parameter)
- WAIT_TIMEOUT events (Idle for the remaining time given as parameter to START_TIMEOUT)
- TIMEOUT (Idle for the time given as parameter + the default timeout parameter)

Every time the TAP2 is coming into one of these states the TAP2 request the stack to "execute" the idle TAP2 time. The stack executes time until something is sent back to TAP2 or until the requested time has been executed (what ever comes first).

Doing this will make it possible to test expiring of "big" timers from test cases, since a TIMEOUT for several hours can be made. In such a case the stack is told to "tick" the amount of time until something is sent back to the TAP2 or the time has passed.

2.1 CTB examples

In order to explain the concept, some examples are depicted in following.

Figure 4 depicts a simple TAP2 test scenario with CTB enabled. All necessary CTB system primitives are not displayed. In the following two scenarios, the default timeout in the TAP2 is assumed to be 10000ms. It

[1] At a kick off meeting, it was identified, that CTB should just cover a host solution. Testing with Anritsu VST on target is very unlikely, since we have acquired the Anritsu PST. Module testing on target with TAP2 could be relevant, but this would make the CTB solution more complex. It also not clear what would happen to other involved parts on target (ESF and Riviera), if the nucleus timer tick is disabled. Therefore this is out of scope for this solution.

is assumed that SOME_PRIMTIVE_REQ starts a task timer inside the entity. The time is 5000ms. When the timer expires, the primitive awaited by TAP2, is sent (SOME_PRIMITIVE_IND).

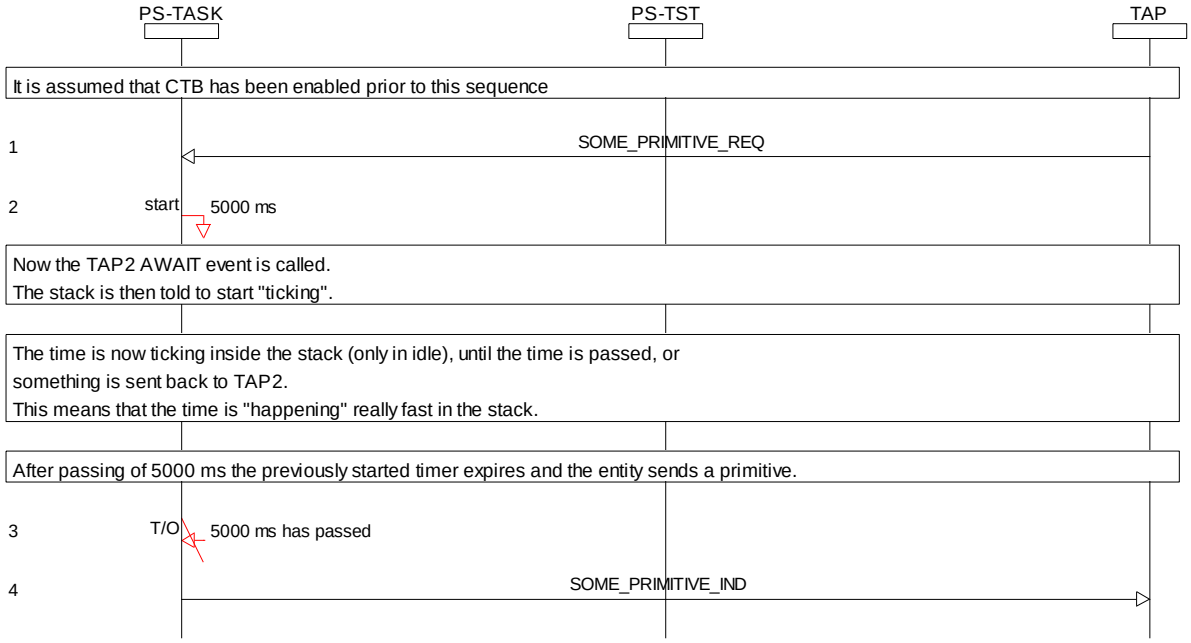

Figure 4 Example of TAP2 test with CTB – pass example

In Figure 5 the same scenario is shown, except that timer value is 12000ms. The default timeout in TAP is still 10000 ms.

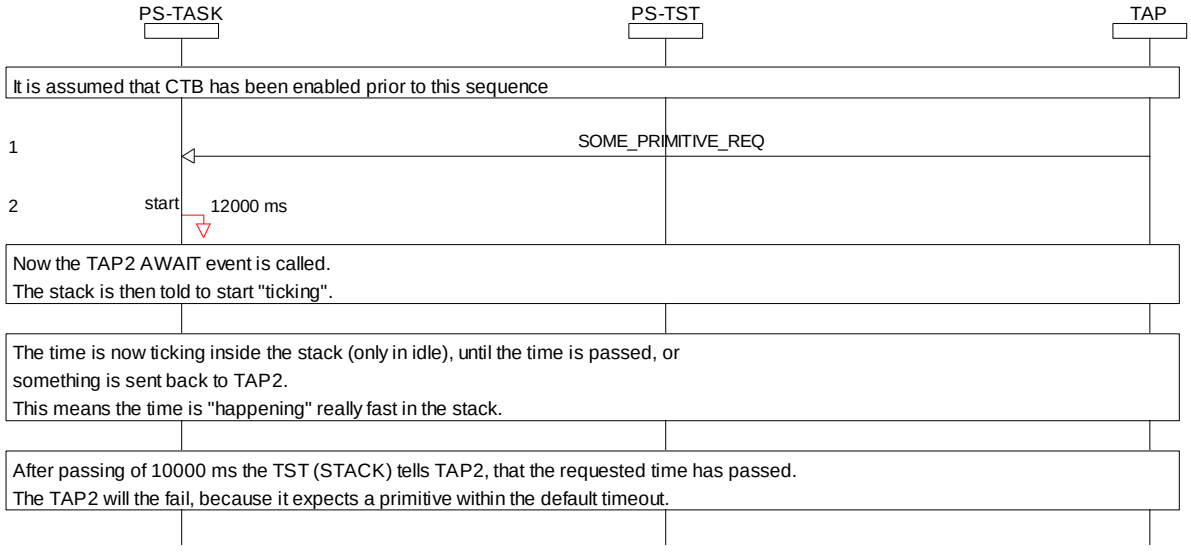

Figure 5 Example of TAP2 test with CTB - fail example

This time the TAP2 will report an error, because nothing was received within the default TIMEOUT. In order to get passed as verdict for the test scenario, a TIMEOUT () with 11950 ms could be used before awaiting the SOME_PRIMTIVE_IND. The scenario is depicted in Figure 6:

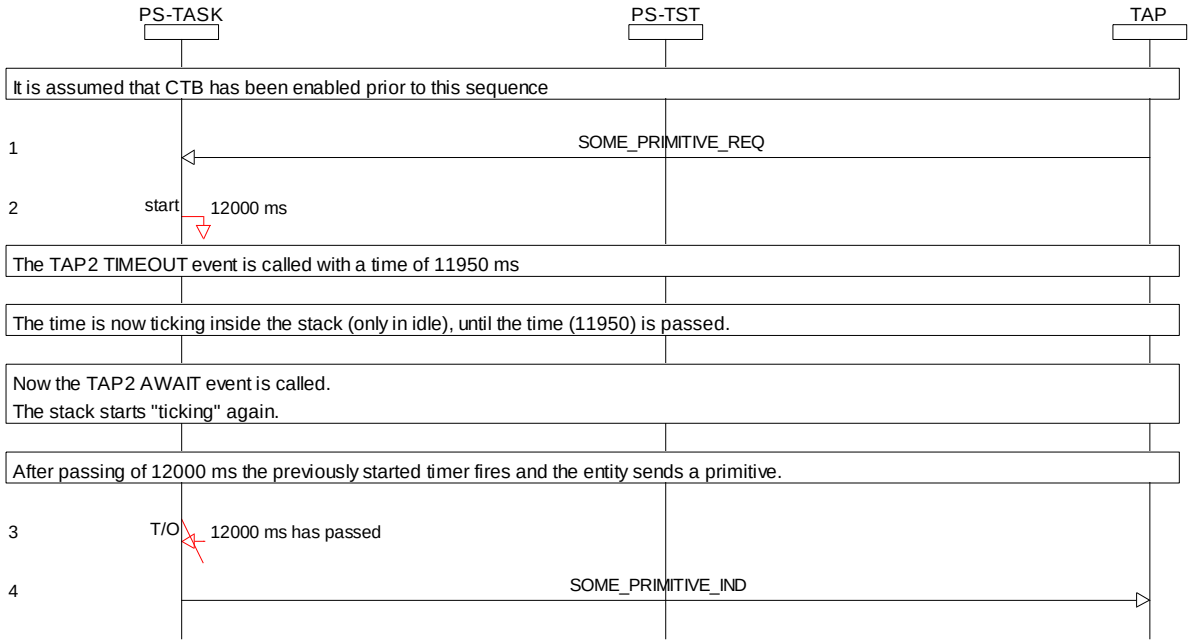

Figure 6 Example of TAP2 testing with CTB – using a TIMEOUT

The behaviour would be the same if MUTE or START_TIMEOUT/WAIT_TIMEOUT were used.

2.2 Assumptions/solved issues

In case of having test cases (TAP2 testing), which cause heavy load of the PS, can result in “no” scheduling of the IDLE entity. Such cases should not use CTB.

This new concept avoids high loading of the test interface. Timer tick requests are only send in case of TAP2 “idle” or when the PAL2 wants the frame to trig (for PAL it was always like this).

On the host configuration it is assumed that no drivers / entities are depending on real time timers. Any way this was never real-time on the host, so it should still be able to work with this CTB concept.

A common test scenario, where PAL2 and TAP2 are used together, does not require special handling. PAL2 is acting a master and completely determines when ticking should happen.

The option to link tap into the PS is not possible, since the solution also should work for Anritsu testing as well.

No changes are needed on TOOL frame/TST. The time is “ticking” as normal here, only the stack ticks are disabled in the CTB configuration.

3 Interface for CTB

Because of missing Service Access Points in the frame configuration we need to define the interface "manually". This section contains the interface specification between the involved parts. In addition to these primitives a header file with external type declarations for nucleus will be made.

Therefore all these definitions including the types below will be added into one common header file (ctb_interface.h).

3.1 EXT_TICK_MODE_REQ

Description:

This system primitive should be sent to PS frame to configure CTB.

Definition:

Short name	ID	Direction
EXT_TICK_MODE_REQ	-	TAP2 / PAL2 -> PS TST

3.2 EXT_TICK_MODE_CNF

Description:

This system primitive should be send from TST to TAP2, when CTB is enabled. The time_stamp of this primitive is used as base time stamp inside the tap. Beside this the process id of the protocol stack executable is included in this primitive, so the tap can if the stack is still running.

Definition:

Short name	ID	Direction
EXT_TICK_MODE_CNF	-	PS TST -> TAP2

Elements:

Long name	Short name	Ref	Type	Description
Windows process id of the protocol stack	process_id	-	U32	

3.3 INT_TICK_MODE_REQ

Description:

This system primitive should be sent to PS-TST to switch of CTB. The sending can either be done from the tool (in case of errors) or from a test case.

Definition:

Short name	ID	Direction
INT_TICK_MODE_REQ	-	TAP2 / PAL2 -> PS TST

3.4 INT_TICK_MODE_CNF

Description:

This system primitive should be sent to the TAP2 from TST when CTB is disabled.

Definition:

Short name	ID	Direction
INT_TICK_MODE_CNF	-	PS TST -> TAP2

3.5 TIMER_TICK_REQ

Description:

This system primitive should be sent to the PS-TST requiring spending of time. The time parameter is the amount of time, the TOOL request the frame to tick in. The time parameter is in milliseconds. Since the resolution in Nucleus is 50ms, the time parameter is converted into a multiple of 50 ms, before sending from the TAP2.

In principle the time stamp in the TST header will be used for carrying the time.

Definition:

Short name	ID	Direction
TIMER_TICK_REQ	-	TAP2 / PHY -> PS-TST

Elements:

Long name	Short name	Ref	Type	Description
Maximum time	Time	-	U32	

3.6 TIMER_TICK_CNF

Description:

This system primitive should be sent to the initiator of the TIMER_TICK_REQ as a confirm of passed time.

In principle the time stamp in the TST header will be used for carrying the “done_time”.

Definition:

Short name	ID	Direction
TIMER_TICK_CNF	-	PS-TST -> TAP2 / PHY

Elements:

Long name	Short name	Ref	Type	Description
Done time	done_time	-	U32	-

3.7 IDLE_REQ

Description:

When the TST should skip time in the PS stack an IDLE-REQ is sent to the idle entity. This primitive is only sent as a signal. Since this primitive is sent as a signal it requires an ID.

Definition:

Short name	ID	Direction
IDLE_REQ	0x00000010	TST -> IDLE

3.8 IDLE_CNF

Description:

When the idle entity is scheduled and it previously has received an IDLE_REQ it sends an IDLE_CNF to TST. This primitive is only sent as a signal. Since this primitive is sent as a signal it requires an ID.

Definition:

Short name	ID	Direction
IDLE_CNF	0x00010010	IDLE -> TST

4 Message Sequence charts

This section explains the message flow for the different scenarios:

4.1 Initialisation / closing of CTB

Following figure indicates the initialisation procedure.

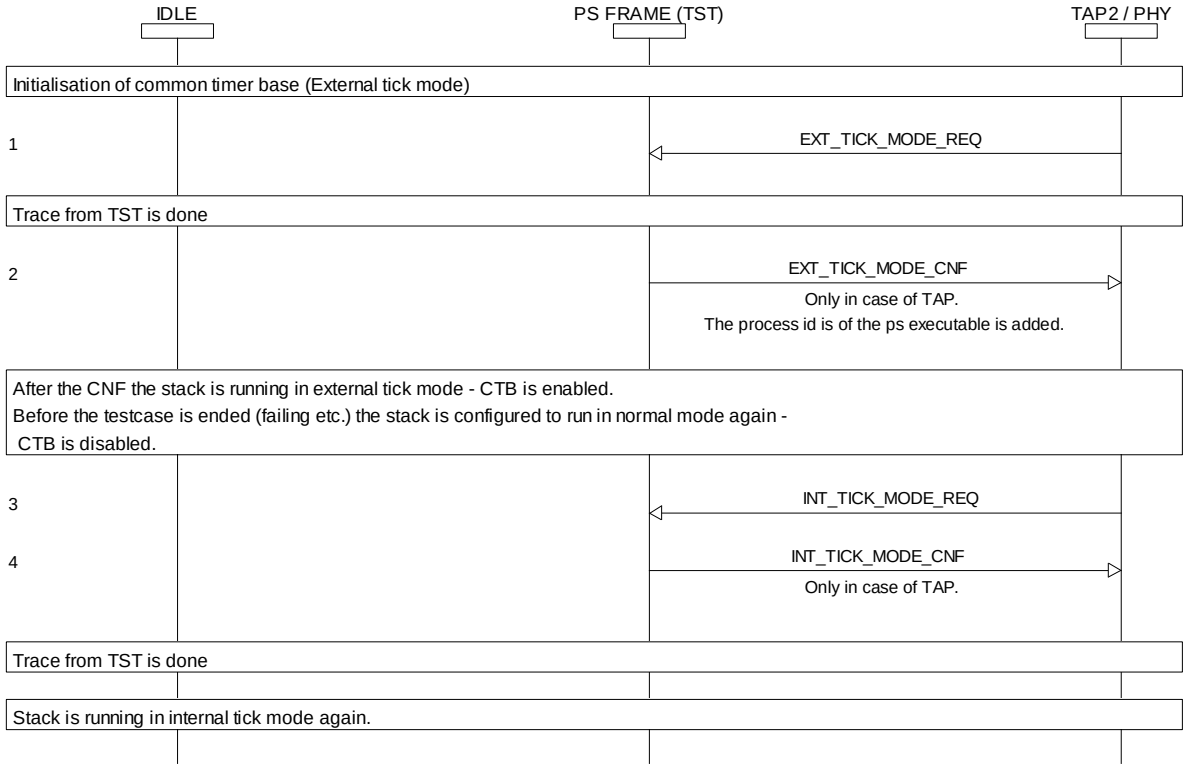

Figure 7 Initialisation of CTB

In case of errors (test case failure) the TAP2 sends INT_TICK_MODE_REQ, to request the stack to run normal again. Upon exit the TAP2 and PAL2 should also set the PS back in internally software ticks.

If the stack crashes during tests, no reinitialising is done. The test case has to be restarted from the beginning.

In case of killing a tool – the stack will not be reconfigured to run in normal mode again. However if the test case is restarted again the stack is reconfigured to run in CTB mode and therefore an EXT_TICK_MODE_CNF will still be sent back.

In case of testing with Anritsu VST, no confirms on enabling/disabling CTB are necessary. When doing TAP2 test, they are used to enable/disable CTB internally in the TAP2, when CTB is invoked by sending a COMMAND to TST on the stack side. When starting CTB from the TAP2 an EXT_TICK_MODE_CNF is sent back. Through this system primitive, the timestamp and process id from the stack is transferred to the TAP2. The process id should be used to see if the stack executable still is running. Otherwise the TAP2 should fail.

It should be possible to enable/disable CTB from test cases.

4.2 Testing with TAP2

The figure below shows the mail flow for TAP2 testing.

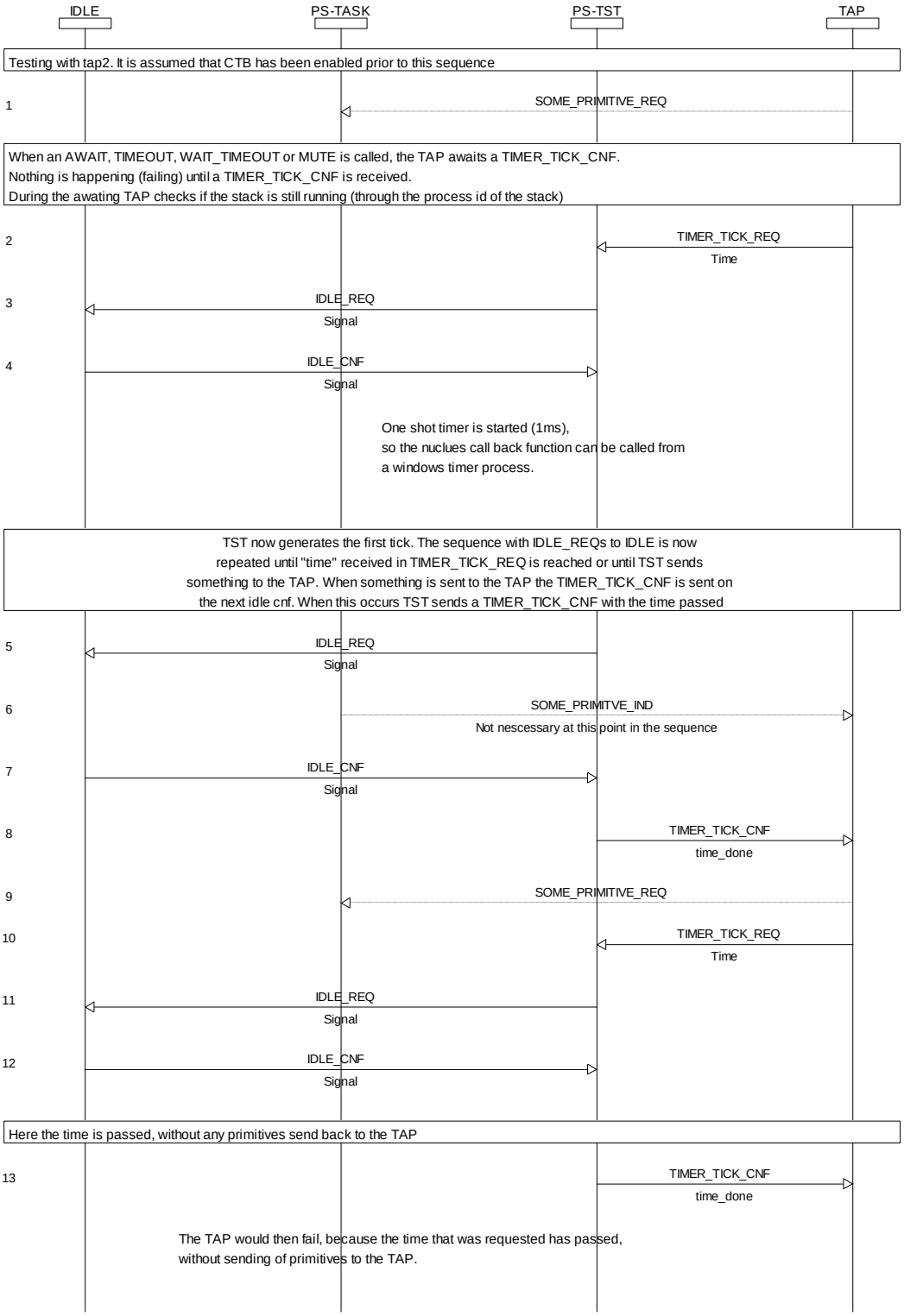

Figure 8 TAP2 testing

Since the TIMER_TICK_REQ can occur later than SOME_PRIMITIVE_IND, due to higher prioritised tasks inside the stack, a check has to be performed before TIMER_TICK_REQ is sent in the TAP2. The queue in the TAP2 should be checked before sending it, because SOME_PRIMITIVE_IND could already have been sent.

Someone could say that if the TAP2 sends more primitives before an idle state, the behaviour inside the stack would be non deterministic. This is not true, because TST and RCV are the entities with lowest priority except the IDLE and EXTR. Therefore the first primitive will always be processed first, when sent from TST to the queue of another task, so the behaviour will always be deterministic.

4.3 Testing with PAL2

The mail flow for testing against the Anritsu system tester is shown in Figure 9.

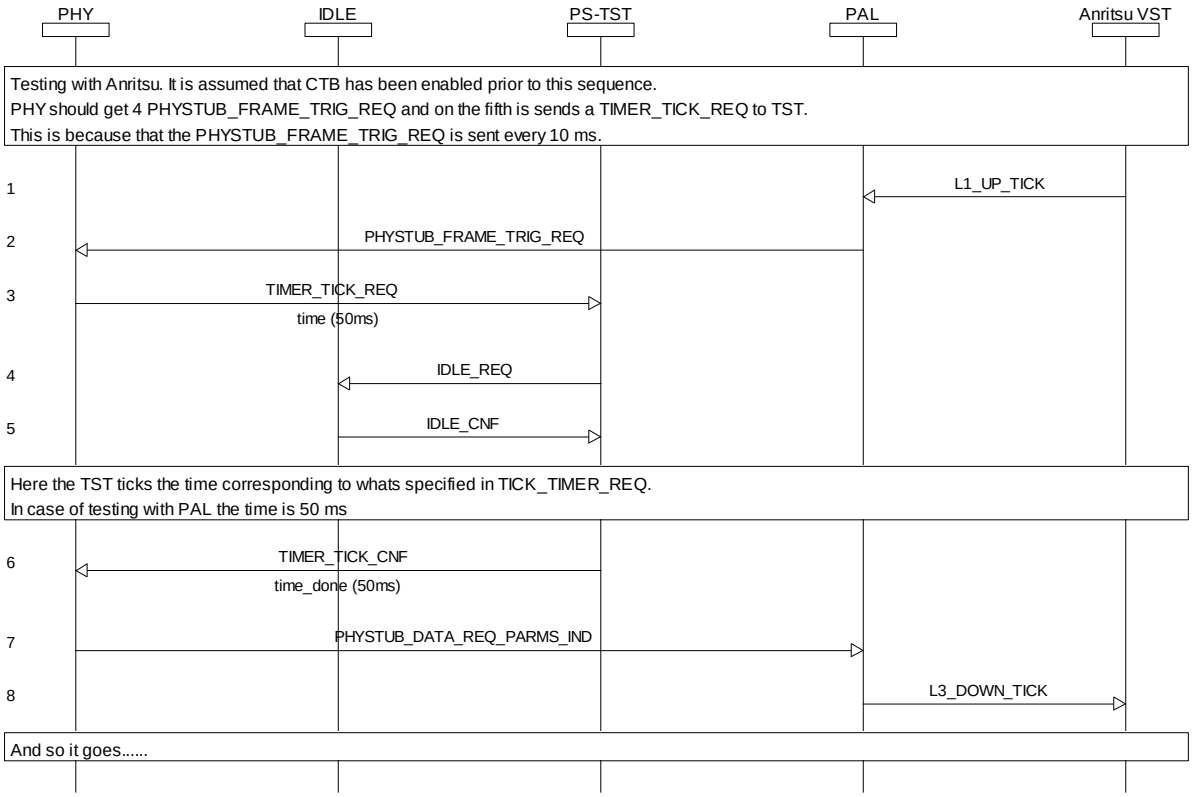

Figure 9 PAL2 testing

The stack will perform a tick, when TST receives a TIMER_TICK_REQ with 50ms and the stack is IDLE. This TIMER_TICK_REQ, should be sent from PHY. TST answers back to PHY with a CNF. As mentioned earlier the L1_UP_TICK indicates a frame trig of 10 ms. to ensure the same time frame in the stack. Pal2 has to await 5 of these before sending the PHYSTUB_FRAME_TRIG_REQ.

5 Changes in the involved parts.

5.1 Nucleus

Nothing has to be changed here, since all the needed variables/functions from Nucleus are global.

5.2 PS-Frame

Changing of priorities of TST and RCV, so an idle entity can be added with the lowest priority. EXTR has lower priority than IDLE task – it seems as if the priority is changed for EXTR, the stack cannot start. It is assumed that this has no influence on the stack behaviour.

The idle task has to be added to the start list of entities inside CONFIG and STUBS (for UMTS).

The time_sliced tasks on host testing should be disabled. Otherwise we will not have task switching in case of running with CTB (when the time isn't ticking). This can cause deadlocks. This is done by replacing following code with 0:

```
#ifndef _TARGET_
        10,
#else
        0,
#endif
```

in os_CreateTask (gpf\frame\nuc\os_proc.c). The value should be 0, even though the frame and Nucleus are running on PC.

In vsi.h a new macro for P_TIME() should be created to access the time_stamp in the TST header.

5.2.1 Nucleus interface

In order to be able to start/stop the Windows timer, that maintains the ticking in Nucleus, an interface file with extern declarations of the Nucleus timer types is required. This file will be added to the os_abstraction layer. The types needed are as follows:

```
#define TARGET_RESOLUTION       1 //Used when calling timeSetEvent.
extern DWORD tm_tick // The timer value (50 milliseconds)
extern UINT mntim_id   //The timer id returned by timeSetEvent()
extern DWORD tm_hdl  //Timer handle in Nucleus
extern void __stdcall tm_exp(UINT,UINT,DWORD,DWORD,DWORD); //Callback function when timer expires.
```

These extern declarations will be located in os_ctb.h under gpf/frame/nuc/.

Besides the declarations a corresponding c file (os_ctb.c), containing functions for starting and stopping the timer and for doing one tick, is required. There should also be a function for getting the process id to the executable.

These functions will get following names:

os_stop_ticking() //Stop the Windows timer

os_start_ticking() //Restarts/starts the Windows timer

os_tick() //Do one tick, by starting a Windows one shot timer and use the nucleus callback function.

os_get_process_id() //Returns the Windows process id of the executable.

Prototypes will also be added to gpf\inc\os.h, which is included by tst_pei.c

Because of starting / stopping of Windows timers, the Windows frame library needs to be linked with the winmm.lib (Multimedia library).

All functionality will be wrapped into a compiler flag (CTB).

5.2.2 os_stop_ticking()

This function will stop the timer (mntim_id) started by Nucleus, by calling the Windows function timeKillEvent(mntim_id). After the stopping a Sleep will be made in order to ensure that the ticking has been stopped.

5.2.3 os_start_ticking()

This function should restarts the Windows timer (mntim_id), by calling timeSetEvent(), with the exact same parameters as in Nucleus. Before this a short Sleep() should be made, in order to ensure that the ticking has been performed.

5.2.4 os_tick()

This function will perform one tick. This should be done from a Windows timer process, like in Nuclues. To ensure the right behaviour, when calling the callback function in Nucleus, a Windows timer is started. This timer should immediately expire (1ms), so that the callback function is called from the timer process scope. Before continuing a sleep of 1ms, should be performed, to ensure that the tick has occurred.

5.2.5 os_get_process_id()

The process id can be fetched with a call to this function getCurrentProcessId(). The os_get_process_id should act as a “wrapper” function.

5.3 TST (protocol stack)

There are no changes required in TST on the tool side. The definition of theses primitives including the rest of the CTB system primitives will be added in a common header file, that will be used by IDLE and TST. The header file will be called “tst_primitives.h”.

Following subsections will describe the behaviour of TST, when receiving the primitives.

All functionality will be implemented in tst_pei.c, wrapped in #ifndef _WIN32_ .

5.3.1 EXT_TICK_MODE_REQ

When receiving this system primitive CTB will be enabled, by calling the os_stop_ticking(). A trace will be made, stating that CTB is being enabled. If CTB already is enabled another trace is done. In case of TAP as sender an EXT_TICK_MODE_CNF is sent back, with the process_id fetched from os_get_process_id.

5.3.2 INT_TICK_MODE_REQ

When this system primitive is received CTB will be disabled. The function os_start_ticking() will be called. If the sender is TAP, an INT_TICK_MODE_CNF is sent back.

5.3.3 TIMER_TICK_REQ

When a TIMER_TICK_REQ is received, the time_stamp is read and stored as remaining ctb_time. The sender of TIMER_TICK_REQ is stored. An IDLE_REQ is now sent to IDLE.

5.3.4 IDLE_CNF

When an IDLE cnf is received the os_tick() function is called. The remaining ctb_time is decremented with the resolution (50ms).

If the CTB has been disabled, the remaining time is zero or if something has been sent to the TAP a TIMER_TICK_CNF is sent back to sender of TIMER_TICK_REQ. Otherwise an IDLE_REQ is sent to IDLE.

5.4 IDLE task

This task will have the lowest priority (except EXTR). The task should have a signalling interface through TST. After receiving an IDLE_REQ the task should answer back to TST with IDLE_CNF. The idle entity consists of two files, idle_pei.c and idle.h. Only basic functionality should be implemented – e.g. IDLE should only support the IDLE_REQ signals.

5.5 TAP2

This section covers TAP2 changes for the CTB solution.

In order to support CTB in TAP2 two new files will be created for tap2_base. These files will be named tap_ctb.h and tap_ctb.c. These files will contain all functions that are related to CTB.

CTB requires special handling of the internal TAP2 timer that can be started from the test case (probably timers in the future). Besides this a special function should be called every time the TAP waits something (vsi_c_await) in order to handle the reception of TIMER_TICK_CNF.

5.5.1 Enabling/disabling of CTB

In order to determine whether the TAP2 is running in CTB mode or not a “ctb” type is added to T_TAP_OPT. Two functions will be added two support reading / setting this:

tap_get_ctb(), for returning the state (TRUE if CTB is enabled, FALSE if CTB is disabled)

tap_set_ctb(int mode), for enabling / disabling CTB.

These two functions will be added tap_opt.c.

Giving the tap an extra parameter on the command line should enable the extern timer configuration. The command should be “-ctb”. If the TAP2 is started with –ctb, a function for enabling CTB is called. The handling of this extra option will be placed together will all other options – e.g. tap_opt.c. In case of no reception of an EXT_TICK_MODE_CNF the TAP2 will fail.

When CTB is enabled from a test case, which is done by sending the EXT_TICK_MODE_REQ command to TST on the stack side, the TAP2 does not know about it. Therefore it cannot fail in case of no EXT_TICK_MODE_CNF.

In general it should therefore always be checked if the system primitives received in the TAP is an EXT_TICK_MODE_CNF or an INT_TICK_MODE_CNF. If so, CTB should be enabled or disabled internally in the TAP2, by calling the corresponding functions below.

5.5.1.1 tap_ctb_enable()

This function clears the special timer simulation parameters (ctb_timer[timer_index]) needed for simulating the internal TAP2 timer(s) in case of running with CTB. Besides this the function should take the remaining timer values of TAP2 timers (only if started) and store them in ctb_timer array. The started timers should also be stopped. The function also calls tap_set_ctb(TRUE);

5.5.1.2 tap_ctb_disable ()

This function restarts the TAP2 timers if there is any values > 0 in ctb_timer[timer_index] and it calls tap_set_ctb(FALSE).

5.5.2 Handling of "IDLE" states in tap

When the TAP2 gets into an idle state (see section 2) and CTB is enabled some special handling is required. This functionality is covered in this section.

Common for all idle states are that the vsi_c_await is called. Instead of calling this function a special tap_ctb_await_prim function should be called, which ensures the right behaviour in all cases. If CTB is disabled the tap_ctb_await_prim, just acts as a wrapper function for vsi_c_await. This means that in all TAP2 files, vsi_c_await should be replaced by tap_ctb_await_prim. The places are in tap_com.c at tap_rcvprim() and in tap_tdl.c at tcd_wait_timeout().

Someone might say that this function belongs to the frame instead of the TAP2. This is more or less correct, but since the only tool, where this function should be used (for now) is the TAP2, it will be placed inside the TAP2.

In case of CTB the diagram below depicts the functionality for tap_ctb_await_prim.

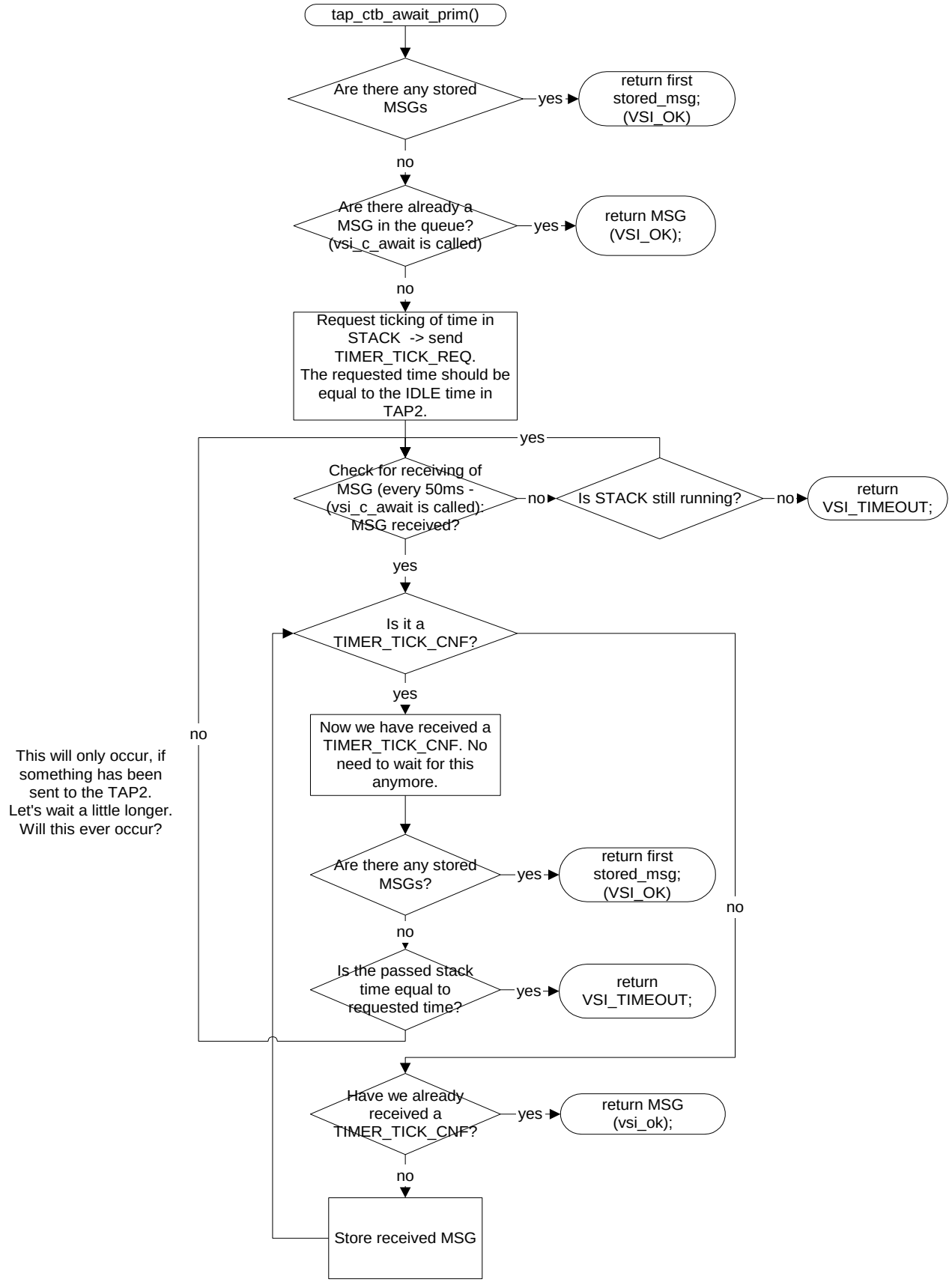

Figure 10 tap_ctb_await_prim()

Upon exit the TAP2 should always reconfigure the stack to run in internal tick mode, by sending the INT_TICK_MODE_REQ.

In order to keep track of the time passed in the stack the timestamp can be used from the primitives.

When waiting for a primitive the TAP2 should check if the stack is running, by checking if the process is still alive. This is done trough the process id, received during initialisation of CTB. The function to be called is GetExitCodeProcess. The exit code should be STILL_ACTIVE (0x103), else the TAP2 should exit with test case failed.

TBD – what it's the return code in case of debug mode of executable when calling GetExitCodeProcess?

It should be possible to store more than one message in the tap_ctb_await_prim routine. It is not possible to know how many primitives that could be received before a TIMER_TICK_CNF. Therefore a MSG store is needed inside the function.

Before sending a TIMER_TICK_REQ the timer values are rounded down to multiple of 50ms, since the resolution on the stack is 50ms.

5.5.3 TAP2 timer functions

All internally timers in TAP2 has to be treated manually in case of running with CTB. This requires wrapper functions for following timer functions. The functions are tap_timer_int(), tap_timer_start(), and tap_timer_stop().

5.6 PAL2

No changes are necessary.

5.7 PHY

The required changes in PHY are not handled in this document (The changes are very simple and does not take much time).