User Guide

TDC

Department:	Aalborg Wireless Center
Creation Date:	12 January, 2002
Last Modified:	22 April, 2003 by Carsten Schmidt
ID and Version:	8434.510.02.014
Status:	Submitted

0 Document Control

0.1 Document History

ID	Author	Date	Status
8434.510.02.001	CSH	12 January, 2002	Being Processed
8434.510.02.002	CSH	27 January, 2002	Submitted
8434.510.02.003	CSH	21 March, 2002	Submitted
8434.510.02.004	CSH	10 April, 2002	Submitted
8434.510.02.005	CSH	17 May, 2002	Submitted
8434.510.02.006	KSP	9 September, 2002	Being Processed
8434.510.02.007	CSH	29 October, 2002	Being Processed
8434.510.02.008	CSH	30 October, 2002	Being Processed
8434.510.02.009	CSH	18 November, 2002	Being Processed
8434.510.02.010	CSH	12 December, 2002	Being Processed
8434.510.02.011	CSH	14 January, 2003	Being Processed
8434.510.02.012	JHO	5 March, 2003	Being Processed
8434.510.02.013	CSH	25 March, 2003	Submitted
8434.510.02.014	CSH	22 April, 2003	Submitted

0.2 References, Abbreviations, Terms

[TI 7010.801] 7010.801, References and Vocabulary, Texas Instruments

Table of Contents

1 Introduction

This document contains the user guide for the TDC test front end. The TDC test front end consists basically of a language syntax called TDC (Test Description Code). This TDC language can be written/edited in Visual Studio 6, because it is normal c-code, and you will gain help from the features this editor provide (dot completion etc.). From this user guide you will learn about this language and how to develop new test cases and edit existing ones.

The TDC test description language provides modularity to the test descriptions. Therefore this document also states some rules, regarding development of new test cases and how to split these into cases, steps and constraints that should be obeyed. One major reason for this is, that we plan to convert to TTCN-3, when a solution is ready.

What you will **not** learn from this document is complex and technical details on the new front end or design details – after all it is only a **user** guide.

In this document several examples are needed. However to ease the overview they have all been kept very simple. In the appendix section (6.1), a complete test case is defined in TDC, so if you need to see something specific in the complete context have a look there.

1.1 TDC version

According to the text you will read in the next sections you will see a set of TDC tools. The table below lists versions of the different tools so that you can see exactly which version this document is written for:

Tool	**Applicable version**
TDC C-libraries	Version 1.1.2
TDS_TO_TDC	Version 1.1.2

Table 1 TDC tool versions

This document will be updated according to future releases of TDC tools and so will the table above. If you cannot find the information for a feature in the document, please check the version of the tool according to this. Please note that TDS_TO_TDC has it own user guide (8434_513_02_tds_to_tdc_user_guide.doc), so if your look for information on how to convert your existing test cases please look there.

1.2 Used terms

Before we get into the interesting part, you should be familiar with the terms listed in the table below. The terms conform closely to TTCN-3.

Term	**Explanation**
Instance	An occurrence of data for a type.
Dynamic part	Test code i.e. test cases and test steps.

Static part	Test data of those types inherited from SAP, MSG and ASN.1 documents.
MUT	Module Under Test.
PCO	Point of Control and Observation.
TDS	Test Description Script
TDC	Test Description Code
MSC	Message Sequence Chart
ETS	Entity Test Specification
ETR	Entity Test Report
SDU	Service Data Unit
GUMLE	A message sequence chart tool used in UMTS development at Condat Dk.
Constraints	This is another used term for the static part of a test case or test event.
Test events	SEND, AWAIT, COMMAND, TIMEOUT, TIMEOUT_WAIT, MUTE, START_TIMEOUT, WAIT_TIMEOUT.
Test step	A series of events or other test steps.
Test case	A series of test steps and/or events[1].
Test suite	A series of test cases.

Table 2 Used terms

1.3 TDC test tool chain

TDC cases can be edited in a normal C/C++ editor. The TDC file is a normal cpp-file (source code file), which uses a set of generated header files from the SAP tool chain. The tool chain can be seen in Figure 1.

[1] In the new language it is not possible to have a test case as preamble like in the old system. It must then be defined as a step. This is compliant with TTCN-3.

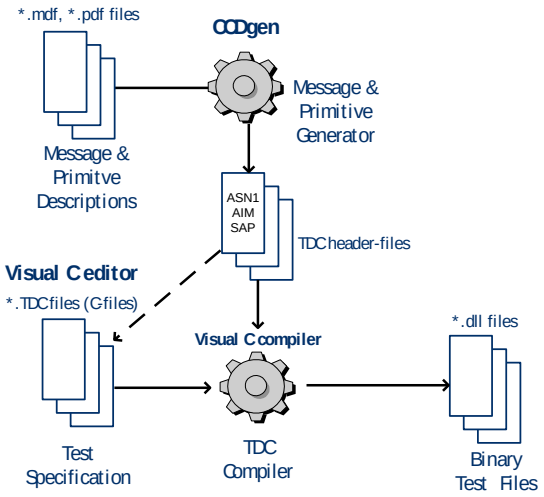


Figure 1 The TDC tool chain

When CCDgen is called (running makcdg) with the option "-gtdc" it also generates the tdc header files. Besides using the TDC-header files for generating the dll-file, they also provide type awareness so that dot completion will be available in Visual Studio 6, when specifying test cases. This dependency is indicated with the dotted arrow.

Besides the tdc header files a special C-library will provide additional member functions to the types in the header files. These member functions are used when assigning values to the constraints. This is done by adding a little C++ functionality to the generated TDC files. This results in advantages like use of templates, operator overloading, scope and modularisation. This library also called tdcinc.lib is generated when calling makcdg with a special parameter:

makcdg alr gprs **tdclib**

or

macdg alr gprs umts **tdclib**

Besides the tdcinc.lib an additional lib (tdc.lib, with and without debug information) is added to the tdc project. This lib contains stuff for interfacing to TAP and CCD.

2 TDC file structure.

This section explains how we recommend that you structure you test case – e.g. where you should put the different parts of a test case. It also list's a short list of rules, which should be obeyed. Please refer to Table 2 for an explanation of used terms.

2.1 TDC Files

It is recommend that the following files will be created as minimum for an entity.

File name	Contents
ENTITY_test.dsp	Visual studio project file.
ENTITY_cases.cpp	Contains only leaf[2] test cases.
ENTITY_steps.cpp	Contains only test steps.
ENTITY_steps.h	Contains only test step prototypes.
ENTITY_constraints.cpp	Contains only constraints.
ENTITY_constraints.h	Contains only constraints prototypes.
ENTITY_ENTITY2_constraints.cpp	Contains only interface constraints – e.g. constraints used by both entities.
ENTITY_ENTITY2_constraints.h	Contains only interface constraints prototypes.

Table 3 TDC Files

The interface constraints files might be a number of different files according to the number of entities the MUT share constraints with. If you are a little confused for what belongs in each file, an example has been made in the appendix in section 6.1 at page 44.

All the TDC files should be placed under:

UMTS\Condat\ms\src\ENTITY\ENTITY_test*.*

or

g23m\Condat\ms\src\ENTITY\ENTITY_test*.*

For integration test cases we suggest that additional directories are created – e.g. AS, NAS and UMTS. It is off course possible to have more folders and files for each entity – for example it is possible to have RRCMEAS and RRCCOMP files and folders for the RRC entity. E.g. this will give following directory:

UMTS\Condat\ms\src\rrc\rrcmeas_test*.*

2.2 Documentation files

With TDC is not possible to have functional MSCs, e.g. MSCs that result in a number of events, in the c-files. However if you convert TDS files (existing test cases from word documents) the textual MSCs are inserted as comments for each test case and test step. These MSCs can be maintained. When you create a new test case you can also add a MSC as comment, if you feel that this makes the test case a bit more explaining.

[2] Leaf test cases are final test cases – e.g. non "preamble" test cases.

The MS-Word document is serving as documentation. The document shall contain the purpose of the test cases and what to be tested with the test cases.

2.3 TDC interface and include files.

The TDC header files are included from the tdcinc directory. This directory contains a set of converted header files from cdginc. Using these h-files gives you the interface to TDC. It is from these files you obtain the types, you use in your test specifications. This means that you will have to include the interface files you need in the specification of your constraints. An example of the files to be included in ENTITY_constraints.h is listed here:

```
/*Interface files from AIMs */
#include “m_cc.h”
#include “m_umts_as_asn1.h”
/*Interface files from SAPs */
#include “p_rr.h”
#include “p_mm.h”
……
#include “tdc.h”
```

The “#include tdc.h” should be below all the interface files. It is also important that you first include the AIM interface files and the SAP interface files.

2.4 TDC Libraries

The section only applies for umts developers.

As mentioned before you need some libraries - tdcinc.lib and m_umts_as_asn1_inc.lib in your visual studio project before you can generate the test cases to be executed. Please note that the m_umts_as_asn1_inc.lib only is for umts testcases. These libraries are interface and tool dependent. It is possible to generate them via a makcdg.mak. However the build process is very slow (only for the umts project, because of the ASN1 message description file), so every time an interface release occurs these libraries are generated for you and placed at:

\\dags11\precompiled\”IF_release”\tdc_libs\

You can copy these libraries to

\g23m\condat\ms\tdclib\

If you change a SAP you need to build the tdcinc.lib again. This can be done with another visual studio project, which can be loaded into your tdc workspace. The project you should load is:

\g23m\condat\int\msdev\makcdg\makcdg_force_rebuild_cdginc_tdclib.dsp

If you build this project the tdcinc.lib is build.

2.5 Generated output file

When you want to generate test case you open you Visual Studio dsp file (assuming that your test case all have been converted to TDC). When you choose build, Visual studio generates a dll-file with all your executable test cases. A makefile can also be created, so that building of the executable test cases can be

done without starting Visual Studio. This user guide will not inform about these makefile topics. The generated test case file (only one dll is generated for each test case file) can be found at:

\g23m\condat\ms\test\test_ENTITY\ENTITY.dll

This file can be loaded into the tapcaller (for regression tests) or you can execute your test cases from the 4NT prompt with the already known batch jobs.

3 TDC Syntax

This section will provide a detailed description of the TDC syntax. It will guide you through a description and specification of all available types in the TDC format. For a complete BNF[3] for the TDC language please look in appendix section 6.2

When you specify a static part in TDC, the code will be nearly identical to the stack code, except that valid-flags (v_…), array-counters (c_…), union-controllers (ctrl_…) and all kinds of memory allocation/deallocation are handled implicit – e.g. you don't need to worry about that.

In TDC it is also possible to do normal visual studio debugging a single step trough a test case, but the result cannot be guaranteed, since the frame and the TAP application work asynchronously.

3.1 TDC structure

The TDC description language has the following file content layout:

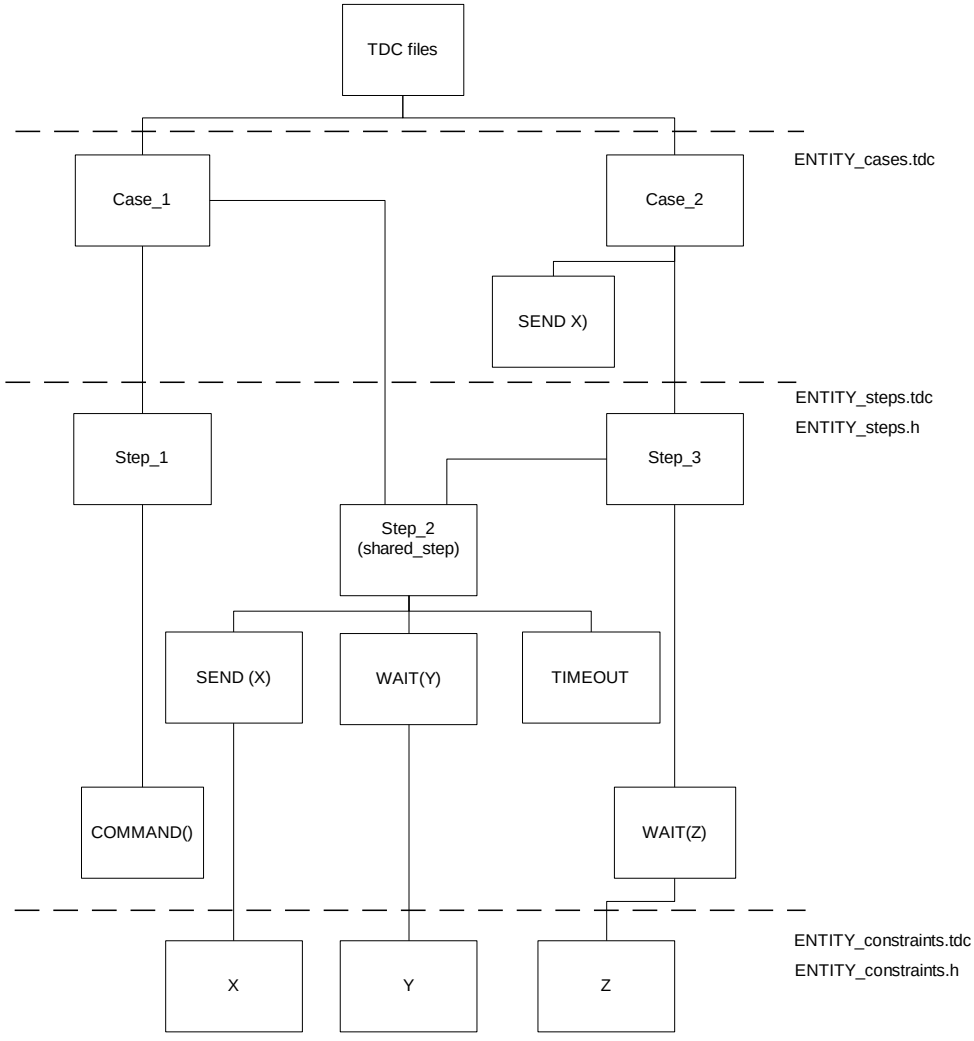


Figure 2 TDC file layout

[3] BNF is an acronym for "Backus Naur Form". This is a form that describes the syntax of a language.

From the figure it should be clear what is allowed on test case level and test step level. The above is compliant with TTCN-3, so we recommend that these rules be obeyed. The old TAP Test suites cannot be handled by TDC, but what you can do with suites is actual possible with TDC test cases.

As you can see the test cases can only contains steps and events not other test cases. This means that all test cases in TDC are leaf test cases – e.g. final test cases, and not preamble test case.

3.2 Test verdict operations

A verdict is what you expect as outcome of a test case. When you specify a test case or test step you have the possibility to assign the verdicts contained in this section. Please note that the default verdict is always PASS() for the different operations – unless otherwise specified. This means that you don't need to apply the PASS() verdict – it is default.

3.2.1 FAIL()

The fail verdict exits with failure from a test step or a test case.

3.2.2 PASS()

The pass verdict returns a pass from a test step or test case.

3.3 Test cases

Only test cases can be executed to a result ("passed" or "failed"). A test case is created with the type T_CASE. This should be followed by inclosing "{}". Inside the T_CASE inclosing "{}" a macro shall be used called BEGIN_CASE("…") also followed by inclosing "{}". BEGIN_CASE take a string as parameter. This string is the title used for each test case.

An example of a test case creation is shown here:

```
T_CASE UMTS200()
{
	BEGIN_CASE ("MO_CS_Emergency_Call") /*The title inside this macro is used in the
tapcaller.*/
	{

/* Steps and/or events should go here*/
	setup_routing();

……
}

}
```

3.3.1 Test case variant

You are probably familiar with test case variants. In TDC test cases variants are done as steps or constraints with arguments. Then create multiple test cases, which use this step with one or more arguments or you might simply use a "for-loop". It is recommended that you create an enum for the different variation possibilities – this will ease the understanding, when you read the test case. A simple example (no enum is created here) is stated below.

```
T_CASE UMTS200A()
{
	BEGIN_CASE ("MO_CS_Emergency_Call_A")

{

/* Steps and/or events*/
	setup_routing('A'); /*Calling step with an argument*/

}
```

```
}


T_CASE UMTS200B()
{
        BEGIN_CASE ("MO_CS_Emergency_Call_B")
{
……
/* Steps and/or events */
        setup_routing('B'); /*Calling step with an argument*/
}
}
```

3.4 Test steps

A test step is created with the type T_STEP. This should be followed by inclosing "{}" and inside a macro shall be used called BEGIN_STEP("…") also followed by inclosing "{}". BEGIN_STEP takes a string as parameter, just like BEGIN_CASE, and the string is traced out at the beginning of a test step. An example of a test step is shown below:

```
T_STEP setup_routing()
{
BEGIN_STEP("Setup_routing"); /*Provides tracing*/
{
/*Events with specified constraints or other steps goes here*/
        SEND(any_primitive());
        AWAIT(any_other_primitive());
                ……
}
}
```

It is not possible to execute a test step – it has to be a part of a test case. Below you will find a test step, which takes an argument – e.g. it is a variant:

```
T_STEP setup_routing(char variant)
{
BEGIN_STEP("Setup_routing"); /*Provides tracing*/
{
        /*Events with specified constraints or other steps goes here*/
SEND(any_primitive(variant));
if (variant == 'A') AWAIT(any_other_primitive_1());
if (variant == 'B') AWAIT(any_other_primitive_2());
……
}
}
```

3.5 Events

This section explains all possible events, which can be used in a test case or test step.

3.5.1 SEND (T_PRIMITIVE_UNION)

This event (macro) is used to send a primitive constraint. SEND() takes the type T_PRIMITVE_UNION as argument. An example of a send is listed below:

```
T_PRIMITIVE_UNION rrc_prim1()
{
        T_PRIMITIVE_UNION rrc_prim1;
rrc_prim1->RRC->RRC_SOMETHING_REQ.my_integer = 10;
……
return rrc_prim1;
}
SEND(rrc_prim1());
```

In SEND events all constraints should be functions – e.g. they should return a T_PRIMITIVE_UNION type or a primitive type from a SAP.

SEND events can be prefixed with a port specification, which determine the destination entity, see 3.6.1 for more info on ports.

3.5.2 AWAIT (T_PRIMITIVE_UNION)

This event (macro) is used to await a primitive constraint and compare the received primitive with the specified values. It takes the type T_PRIMITIVE as argument. AWAIT is used just like SEND()

```
T_PRIMITIVE_UNION rrc_prim2()
{
T_PRIMITIVE_UNION rrc_prim2;
rrc_prim2->RRC->RRC_SOMETHING_IND.my_integer = 10;
……
return rrc_prim2;
}
AWAIT(rrc_prim2());
```

In await events all constraints should be functions – e.g. they should return a T_PRIMITIVE_UNION type or a primitive type from a SAP.

The order of await events are significant unless otherwise specified. The following example illustrates the behaviour:

```
AWAIT(rrc_prim2());
AWAIT(rrc_prim3());
```

Is different from

```
AWAIT(rrc_prim3());
AWAIT(rrc_prim2());
```

AWAIT events can be prefixed with a port specification that specifies which entity(s) is allowed as source(s)

and/or which entities is allowed as original destination(s), see 3.6.1 for more info on ports.

3.5.3 COMMAND (char*)

This command is used to redirect, duplicate and set configuration parameters for the MUT. The possible commands are reset, redirect and duplicate. Below you will se a command example:

COMMAND ("RESET RRC");	/*Reset all previous redirect and duplicate settings for the RRC entity */
COMMAND ("RRC DUPLICATE RCM PCO);	/*Duplicates all primitives send from RRC to RCM to the PCO – this enables binary tracing of primitives in the pco_viewer*/
COMMAND ("RRC REDIRECT RCM TAP");	/*Redirects all primitives sent from RRC to RCM to the TAP */
COMMAND ("TAP REDIRECT TAP <*MUT*>");	/*This command must be the last one in a series of commands. This redirects all the stuff sent from the TAP to MUT*/

All the duplicate commands must be before the redirect commands. It is also possible to use "ALL" as parameter – e.g.

COMMAND ("RRC DUPLICATE ALL PCO");	/*Duplicates all primitives send from RRC to the PCO this enables tracing of primitives in the pco_viewer*/
COMMAND ("TAP DUPLICATE ALL PCO");	/*Duplicates all primitives send from TAP to the PCO – this enables tracing of primitives in the pco_viewer*/

If the test case use several stack entities as MUT – e.g. you are doing integration test, then the last command

COMMAND ("TAP REDIRECT TAP <MUT>"), must be replaced by several commands of the format

COMMAND ("TAP REDIRECT TAP **bbbbbb******** <MUT>")

In this line, bbbbbb is the binary coding of the Service Access Point identifier (each b is one binary digit 0 or 1) and <MUT> is the entity providing the Service Access Point. The tested entities use a line for every Service Access Point. For more information regarding the Service Access Point identifier see [C 8434.XXX.01] gpf\ doc\sap_numbering_scheme.doc. Here you will also find a description of the meaning of the bits in an operation code, so that you can search out the right bits.

In the example below, it is assumed that RR has the Service Access Point 11 (0x0B).

COMMAND ("TAP REDIRECT TAP **001011******** RR")

For all UMTS entities we use 32 bit operation codes. Therefore the binary coding of an UMTS SAP is a bit different than the case of 16 bit. Below you find an example of 32-bit operation code, for the RLC entity which has the CRLC sap and the RLC sap:

COMMAND ("TAP REDIRECT TAP **********10000100 RLC")

COMMAND (“TAP REDIRECT TAP **********10000101 RLC”)

3.5.3.1 COMMAND (“TAP EXCLUDE”)

COMMAND (“TAP EXCLUDE”) is used to temporary exclude a test-case from the set of active test cases, this is typical done for test cases that do not currently pass due to postponed features.

3.5.4 TIMEOUT (time)

TIMEOUT is used before, between or after a SEND or AWAIT event. TIMEOUT is used to suspend the TAP for the given time. Time is a non-negative integer measured in milliseconds. If placed before the expected receiving of a primitive, the instruction adds time to the default waiting time (10 s) for a primitive. A test will fail if MUT doesn’t send a primitive during the default waiting time plus time.

3.5.5 MUTE (time)

MUTE is used before, between or after a SEND or AWAIT event. A test will fail if the MUT sends a primitive during the given time. The instruction may be used to check whether a timer is stopped or if there are unexpected primitives sent from MUT. Time is a non-negative integer. The time is measured in milliseconds.

3.5.6 START_TIMEOUT (time)

START_TIMEOUT starts a timer. Time is a non-negative integer and it is measured in milliseconds.

3.5.7 WAIT_TIMEOUT ()

Wait time_out suspends the TAP until the timer started with START_TIMEOUT is expired. If WAIT_TIMEOUT is called and the timer already has expired the testcase will fail.

3.6 Advanced features

This section covers some advanced features provide by the TDC language.

3.6.1 Source and destination of primitives: T_PORT

A port can be used to specify source and destination of SEND and AWAIT events.

```
T_PORT cc_mm(“CC<->MM”,”MNCC”);
DEFAULT_ PORT = cc_mm;
T_CASE CC001(){
        setup_routing(); //[4]
        cc_mm.SEND(prim1); // explicit port
        AWAIT(prim2); // implicit port
}
```

The T_PORT constructor takes 1 or 2 string arguments: src_and_dst_list and optional a sap_list if the sap_list is present then the port only apply to primitives belonging to the SAPs in the sap_list.

```
T_PORT::T_PORT(char * src_and_dst_list); // can be applied to any primitive.
```

[4] Ports do not affect duplicate and redirect commands (they are still needed).

T_PORT::T_PORT(char * src_and_dst_list, char* sap_list); //can be applied to primitives from a SAP in sap_list.

If no PORT is specified then the global variable DEFAULT_PORT is used. DEFAULT_PORT have the initial value T_PORT("<->") which give the same behaviour to events as before ports was introduced. It is not possible to define a new global DEFAULT_PORT. Instead the default port should be declared locally. An appropriate place to do this could be in the function that set up the routing used in the test cases.

A T_PORT variable can also hold a list of ports, when such a variable is applied to an event the list is searched for an src_and_dst_list for which the sap_list contain the SAP that the primitive belongs to. In such lists ports with no sap_list acts as defaults. T_PORT list are created with the "+" operator e.g.

T_PORT cc_xx = T_PORT("CC<->MM","MNCC") + T_PORT("CC<->SS","MNSS")

If a SEND port has no source or destination then TAP is used.

If an AWAIT port have no source or destination then any source or destination is valid.

3.6.1.1 T_PORT constructor parameters

The 1st parameter src_and_dst_list is a string consisting of 3 parts:

Part	For SEND	For AWAIT	Comment
MUT entity list	Destination entity (only the first is used if more than one is specified)	Source entity(s)	Semicolon (;) separated list of entity names
Default	TAP	Not checked	
Send or await spec	"<-" or "<->"	"->" or "<->"	
Context entity list	Source entity (unused by stack)	Destination entity(s)	Semicolon (;) separated list of entity names
Default	TAP	Not checked	

The 2nd parameter is a semicolon (;) separated list of sap names.

3.6.2 Primitive parking

Parking of primitives can be used if the AWAIT order of primitives is unknown. Parking means that the test case will not fail if another primitive is received than the one awaited – e.g. await order is ignored. Parking is per default disabled in TDC.

There are two kinds of parking available, namely a short-term parking and a long-term parking. The short-term parking allows different orders of primitives to be received until the next SEND event. If primitives are parked executing the SEND event, the primitives will be discarded. A trace warning will be send. The long-term parking accepts parking of primitives until "the end" of the test case.

Parking can be turned on and off while the execution takes place. However it is only allowed in test steps or test cases – not in constraints. To switch the parking state use this function:

```
PARKING (SHORT_TERM);      /*Enables short term parking – returns the old state */
PARKING (LONG_TERM);       /*Enables parking – returns the old state */
```

```
PARKING (DISABLED);        /*Disable parking – e.g. back to normal behaviour (back to default
                           behaviour) – returns the old state */
```

For handling of the return values use the enum:

T_TDC_PARKING_ENUM.

This enum contains the defined values DISABLED, SHORT_TERM and LONG_TERM.

Please be careful with the use of parking. In general you should always have the same parking "state" when you return from a scope as when you enter the scope – see the example below:

```
Any_scope
{
        T_TDC_PARKING_ENUM old_parking_state;
        old_parking_state = PARKING (DISABLED);
        AWAIT(A);
        AWAIT(B);
        ….
        PARKING (old_parking_state);
}
```

Please note that if you turn parking off while primitives are parked your test will fail.

3.6.2.1 How parking work

When parking are on AWAIT will first check the already parked primitives if none match it will wait for a new primitive, if the new primitive do not mach it will be parked, and AWAIT will wait for the next primitive, this sequence continues until a matching primitive is received or no new primitive is received within the AWAIT timeout period[5]

When AWAIT checks a primitive for parking it do so based on the opcode of the primitive, it is not possible to park a primitive base on the data in the body of the primitive. But if the AWAIT is prefixed with a PORT then the source and destination is checked as well as the opcode.

[5] The AWAIT timout period can be specified as e.g. a command line option to TAP2

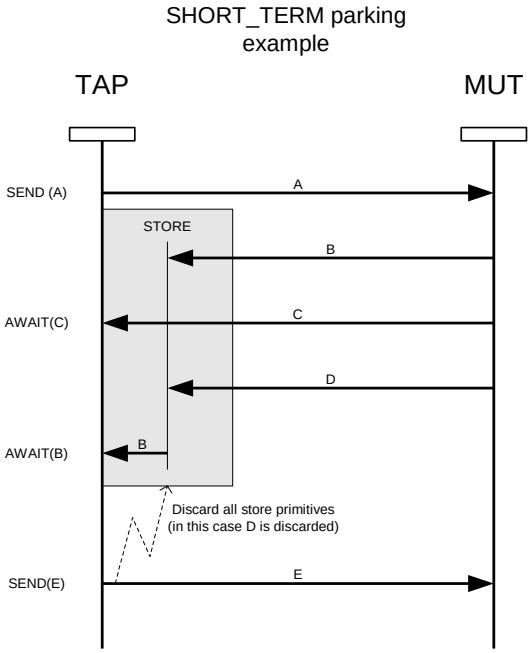


Figure 3: Example of how parking works. Note that stored primitives are discarded when the parking period is over.

3.6.3 ALT { … } (Alternative mail sequence)

These features will first be supported in TDC version 2.

If you want to have an alternative mail sequence you can use the alt {…} operation. With the ALT operations you have additional operations (ON() and OTHERWISE()) that can be used. Together they work like a switch/case with comparison on case parts. An ALT{…} operation is specified like this:

```
ALT {
  ON (AWAIT(…)) …..;
  ON (AWAIT(…)) {…};
  OTHERWISE () …;
}
```

In the table below you see an example with pseudo code (A, B and C are defined primitive constraints):

TDC code	**Pseudo code**

ALT
{
ON(AWAIT(A()))
{
/*A code*/
}
ON(AWAIT(B()))
{
	Switch (incoming_message())
	{
	case A:
	{
	/*A code*/
	break; }
	case B:
	{
	/*B code*/
	break;}
	default:
	{

/*B code*/;
}
OTHERWISE()
{
SEND(C()));
}
}
	send C
	break;}
	}

Table 4 ALT example

If the OTHERWISE is missing and parking is disabled, the ALT {} operation returns FAIL if something is received, that wasn't specified with the ON () operation. If parking is enabled execution will remain in the ALT operation until you receive something specified with the ON () operation or a timeout occurs. Using OTHERWISE is simply like a ON(AWAIT(*any_primitive*)).

Inside an ALT operation, only the "if" statement is allowed in front of an ON statement, not an "if" "else" statement. All other basic statements (see section 3.7) should be avoided inside ALT {} operations.

Please note that if PASS is used in an ALT {…} feature it returns to the surrounding of ALT {…} or STEP("blah") {…}.

3.6.3.1 ON (AWAIT(…)) {…}

This operation is to be used in an ALT () operation. The ON() operation returns PASS as the default verdict – e.g. you don't need to specify it.

3.6.3.2 OTHERWISE () {…};

OTHERWISE can be used as a function similar to the default in a switch sentence in your c-code. Having no OTHERWISE is the same as having OTHERWISE() FAIL();.

3.6.4 TRAP {…} and ONFAIL {…}

TRAP and ONFAIL works in a similar way like "try" and "catch" in C++. If fail occurs inside a TRAP statement – e.g. a parameter in a primitive is wrong or a wrong primitive is received[6], execution will proceed with the ONFAIL events and continues after the end of the TRAP/ONFAIL combination, as if no fail occurred. Having no ONFAIL is the same as having an empty ONFAIL{};.

An example is shown here (A, B and C are defined primitive constraints):

TDC code	**Pseudo code**

TRAP
{
AWAIT(A());
AWAIT(B());
}
ONFAIL
{
SEND(C());
	try
	{
	receive (A);
	receive (B);
	}
	catch
	{
	send (C);

[6] Parking is disabled.

}	}

Table 5 TRAP example

3.6.5 Common Timer Base (CTB)

From TAP version TAP_2.6.X and FRAME_2.10.X the feature Common Timer Base (CTB) is implemented. CTB only works for host testing. It cannot be used for target testing.

3.6.5.1 What is CTB?

CTB is, as the name says, a way to have a common timer base, between the test tools and the protocol stack. Enabling CTB causes the PS time to be controlled from the test application (TAP). The time resolution on host is 50ms. This means that timers only can be tested on host with an accuracy of 50ms. The consequence is that specifying a START_TIMEOUT(4001) would result in a timer value of 4000 ms. The value 3999 results in a timer value of 3950 ms. The timer values will always be rounded down in case of non multiple of 50ms.

3.6.5.2 How does CTB work?

In short CTB disables the Nucleus simulation of hardware interrupts for timer ticks. This means that if CTB is enabled the time will stand still inside the stack, unless the stack is asked to "spent" some time. When running entity tests with the TAP, the TAP tells the stack to "start" spending time for a well-defined amount of time. This happens automatically, each time the TAP is in idle mode (e.g. when the TAP awaits a primitive, on TIMEOUTs, on MUTEs or on WAIT_TIMEOUTs). The spending of time only occurs when all entities in the stack are in idle mode – e.g. no entities are scheduled. The time spent inside the stack will happen much faster than real-time. When debugging test cases or the protocol stack, it can be an advantage to use CTB, because timeouts of the tap will be avoided.

Figure 4 depicts a simple TAP2 test scenario with CTB enabled. All necessary CTB system primitives are not displayed. The default timeout in the TAP2 is assumed to be 10000ms (normal behaviour). It is assumed that SOME_PRIMTIVE_REQ starts a task timer inside the entity. The value of the timer is 5000ms. When the timer expires, the primitive awaited by the TAP2, is sent (SOME_PRIMITIVE_IND).

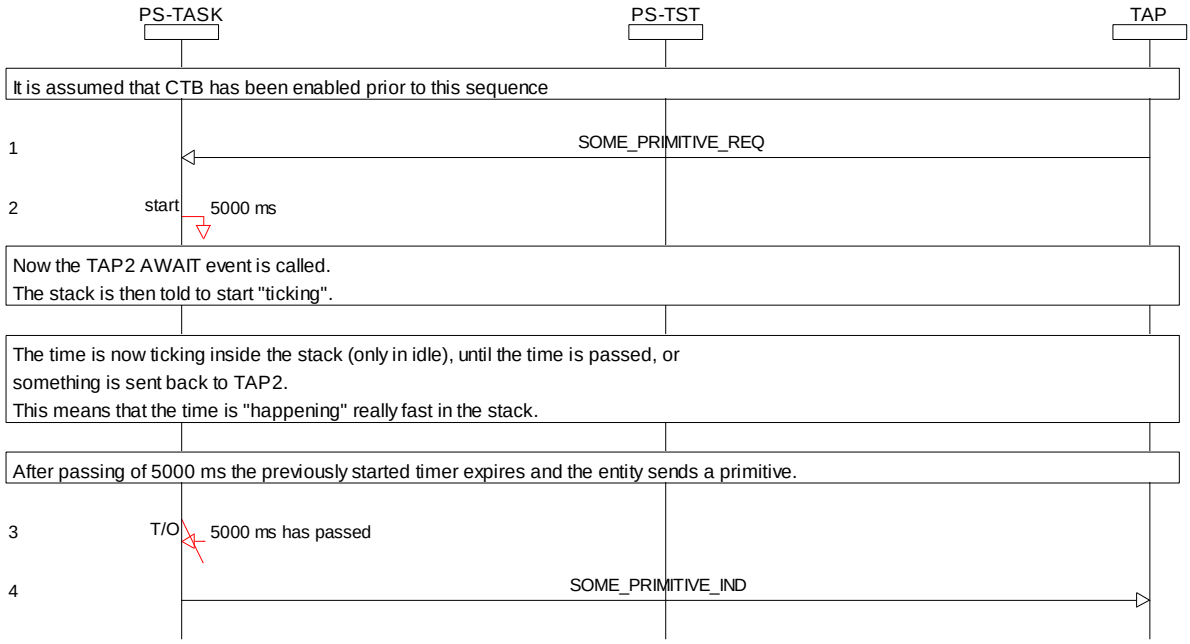


Figure 4 Example of TAP2 test with CTB – pass example

In Figure 5 the same scenario is shown, except that timer value is 12000ms. The default timeout in TAP2 is still 10000 ms.

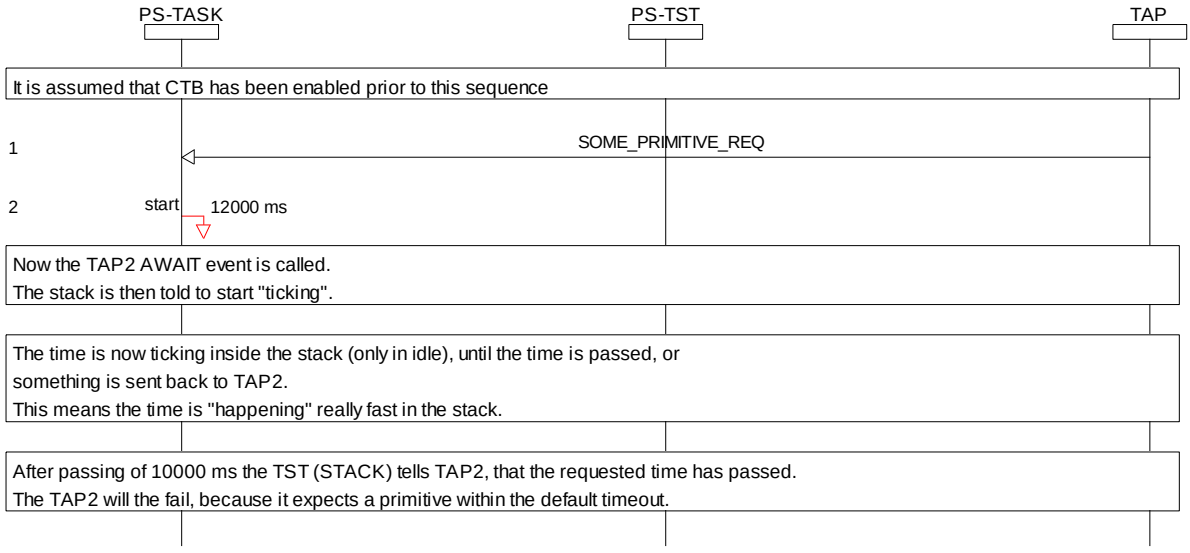


Figure 5 Example of TAP2 test with CTB - fail example

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

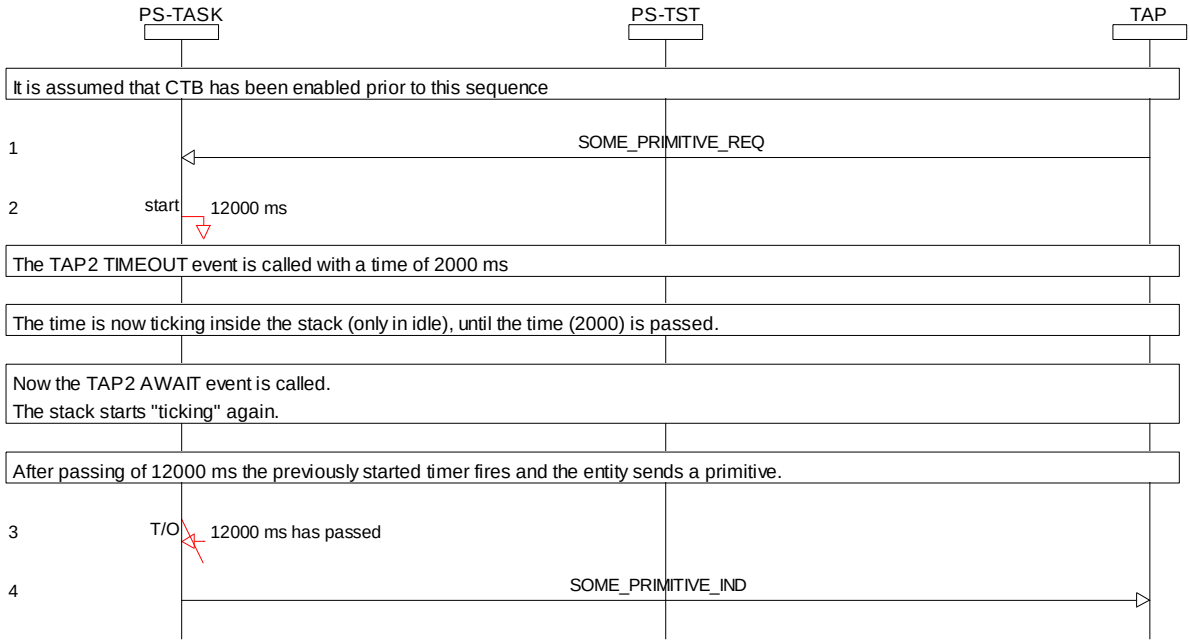


Figure 6 Example of TAP2 testing with CTB – using a TIMEOUT

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

3.6.5.3 How to use CTB?

CTB can be enabled in two ways. The TAP can be added an option:

```
-ctb
```

This can be done from the 4NT prompt using “runtc.bat” or from the tapcaller in the “other options” under “configuration-> settings -> general options”. The TAP automatically disables CTB on exit, no matter if the test case fails or passes.

Another way to enable / disable CTB is from test cases. This can done like this:

```
COMMAND("TST EXT_TICK_MODE_REQ"); //Enables CTB
COMMAND("TST INT_TICK_MODE_REQ"); //Disables CTB
```

The COMMANDs should always be sent to TST. If CTB is enabled inside a test case, TAP automatically disables CTB on exit.

A combination of the two “starting” ways can be used. For instance, if you know that one test case cannot work with CTB, you can just start the case with disabling of CTB, like shown above.

NB: In case of enabling/disabling CTB from test cases this duplicate command will cause a warning in the TAP/SYST if this duplicate command is used inside the test case:

COMMAND ("TAP DUPLICATE ALL PCO");

Instead the primitives sent from TAP should be duplicated, like this (in this example the TAP sends primitives to the entities RRC and RCM):

COMMAND ("TAP DUPLICATE RRC PCO");
COMMAND ("TAP DUPLICATE RCM PCO");

When running with CTB, the timestamps of the traces sent from the tools (traces with "~" in front), can be misleading, because the time on the tool side is running, although it only runs for a well defined periods (the amount of depending on the test case) on the stack side. Therefore use the trace number in the pco_viewer instead.

3.7 Basic C statements

The table below contains all basic control statements that can be used in test cases and test steps in TDC.

Statement	Explanation
if-else	Can be used to get test case variants – e.g. conditional mail sequences.
For	Can be used to have repeated mail sequences.
While	Can be used to have a conditional repeated mail sequence.
Do-while	Can be used to have a conditional repeated mail sequence.

Table 6 Basic statements for steps and cases.

3.8 Constraints

In this section you will learn how to handle and specify the different constraint types.

Additional code is possible when you specify the constraints - e.g. filling of arrays or arithmetic statements. However doing this might result in problems when converting to TTCN-3. This means that you should expect some work, when converting to TTCN-3.

To help you specifying the constraints for your test case several additional types are offered. They are listed in the table below. They are all accessible from Visual Studio via the drop down list.

Type name	Explanation
T_PRIMITIVE_UNION	This type contains a list of all existing primitives listed on SAP level.
T_SDU	This type is used when specifying SDUs in primitives. It is a union like type, which consists of two types – a T_MESSAGE_UNION (aim) and a "normal" sdu (raw) with an o_buf (buffer offset), a l_buf (buffer length) and a buf (buffer with the bitstream). The "normal" sdu are used when you want to specify a bitstream manually. T_MESSAGE_UNION are used when you want CCD to coding / decoding of the specified AIM into / from a bitstream.
T_MESSAGE_UNION	This type contains a list of all air interface messages listed AIM document level. ASN1 is considered one document. Besides this it also contains a TI (transaction identifier) and a TIE (extended transaction identifier). These are used when specifying AIM for GSM and NAS. For ASN1 they should be ignored. Please note that there's no PD (protocol discriminator) or message type. They are implicit when you specify the constraint.

T_ARRAY<array_type>	This type can be used to declare arrays – e.g. array of structs. This is also to be used when having arrays as parameters in functions.

Table 7 Additional constraint types

T_PRIMITIVE_UNION can be type casted to any SAP primitive and T_MESSAGE_UNION can be type casted to any AIM from either ASN1 or an AIM document. Examples with the different types are found in the next sections.

Please note that all types, no matter if they come from your SAP or MSG files or if it's one of them mentioned in Table 7, can be return types of functions. This means that you for instance can specify a SDU (T_SDU) in a function, and when you want to use this in a primitive that has a sdu, you just call this function.

3.8.1 Instance navigation

When you navigate through the instances, you need to know how to access the drop down list (or dot completion list). In TDC it will nearly be compliant with what you are used to in Visual Studio. The difference is that all user defined instances are treaded as pseudo pointers. This means that you have to access the drop down list with the "->" operator at the main level instead of the "." operator. The rest is like you are used to. The examples below show the details:

```
T_CC_CALL_SETUP call_setup;
call_setup->struct_1.struct_member = 10;
```

As you can see you must use -> at the main level otherwise is should be like you used to. The additional types, mentioned in the previous section, also require special handling. In the example below you also see that after selecting the AIM (in this case "CC") you have to use the -> operator.

```
T_PRIMITIVE_UNION mmcc_data_req()
{
   T_MMCC_DATA_REQ prim;
   prim->sdu.aim.entity.CC->U_CALL_CONF.call_ctrl_cap = call_ctrl_cap_1();
   return prim;
}
```

You don't have to remember this – when you edit your test case and you can't get the drop down list to appear, it is properly because you're using the wrong operator. The dropdown list will have "operator ->" if you use "." where you are supposed to use "->". If you see this just use the other operator.

3.8.2 Standard member functions

The constraint types are provided with member functions, so that for example the skip action easily can be assigned to a struct. These member functions appear in the drop down list for each element. The standard member functions that are available for all types are:

Member function	Explanation
_skip	Element can be present. If it's a SEND() event it is only applicable on optional

	elements, otherwise a runtime error occurs. On AWAIT() events _skip is also allowed on mandatory elements.
_show	Element can be present. On a SEND() event it is the same as skip and therefore it is only applicable at optional elements. On AWAIT() events the element and it's values will be traced if present.
_forbid	Element must NOT be present. _forbid is only a applicable on optional elements in AWAIT contexts. Please note that it is not possible to assign _forbid directly to primitive or root elements of the primitive.
_require	Element must be present, but the value is not checked. Applicable in AWAIT contexts and SEND context, but for SEND only for elements that only have v_… flags. Please note that it is not possible to assign _require directly to primitives (T_PRIMITIVE_UNION) or root elements of the primitive. In TDC version 1 _require is not supported on messages elements (T_MESSAGE_UNION). It can only be used to await empty messages (see section 3.8.4.2).

Table 8 Standard member functions

Besides the standard member functions, the types might have additional member functions (depending on the type). These will be mentioned as we go through the types in the following sections. Most of the member functions have names beginning with "_". This is done because of the drop down list in Visual Studio. Here the members are sorted alphabetically, so we force our member functions to be the first ones in the list.

The reason for all assignments are functions, are that the only way code can be executed easily, is trough a function. No one of the member functions take arguments.

3.8.3 Primitive constraints

When you want to specify a primitive constraint you might find the type T_PRIMITIVE_UNION very handy. Starting with this gives you the possibility to choose the SAP from where you want primitive. Figure 7 shows a drop down list with possible SAPs

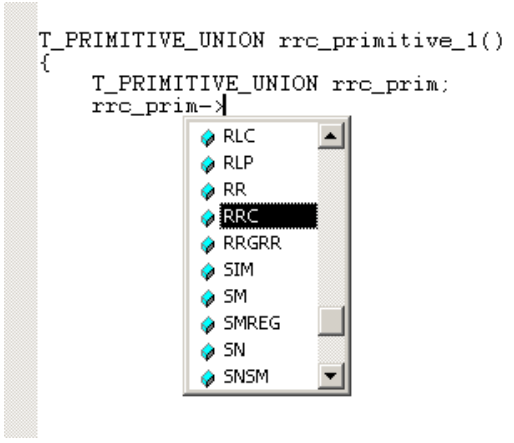


Figure 7 SAP dropdown list

After choosing the sap is you can get a drop down list with all primitives – like in Figure 8.

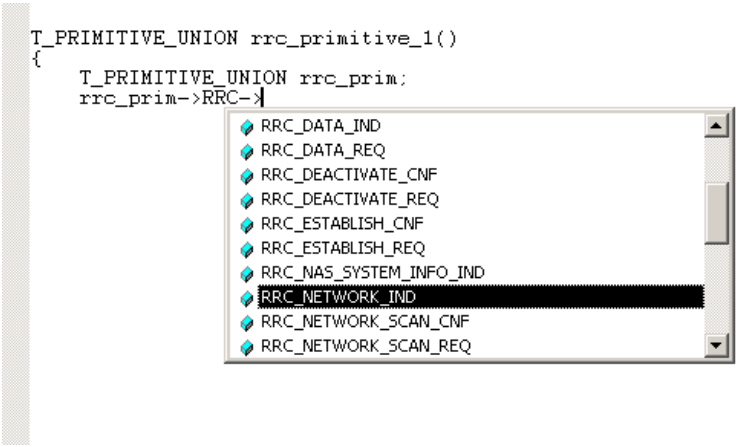


Figure 8 Primitive dropdown list

After this you should just specify the contents of the primitive, and end the specification with a return of the specified primitive. Don't worry about the return type T_PRIMITIVE_UNION, as stated earlier this is automatic converted if it is needed. When you want to use your constraint in a step or case your just do like this:

SEND(rrc_primitive_1());

3.8.4 AIM constraints

When you want to send or await an AIM it is done by specifying primitives that contains a SDU. You can choose to let CCD code/decode the message or specify the bit stream manually according to previous statements. If your primitive contains the type T_SDU you will get following options in the drop down list:

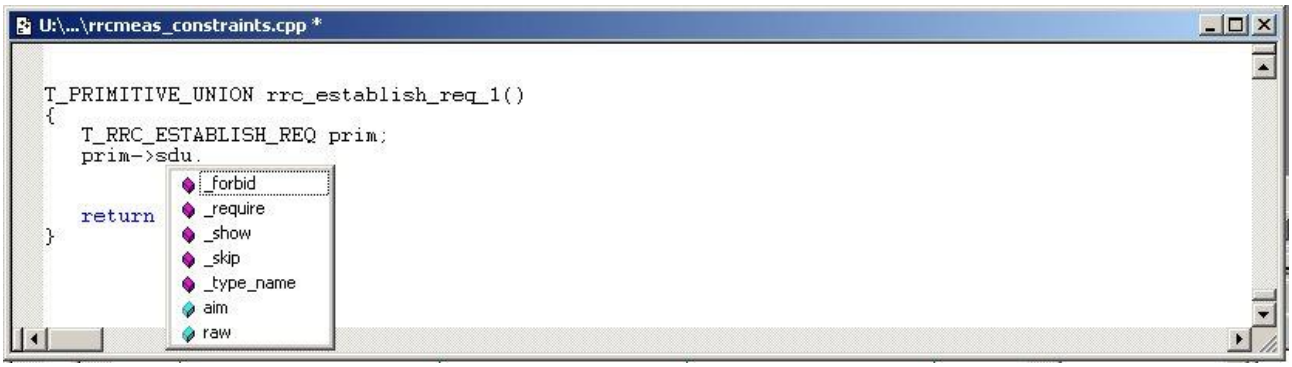


Figure 9 sdu/aim constraint

Select "aim" if you want an air interface message from an AIM document. Select "raw" if you want to specify it manually. Below you find examples for each type.

3.8.4.1 Manual specification of bitstream

When you want to specify the bitstream you should select the "buf" in the dropdown list for sdu's. You should specify the offset (o_buf), the buffer length (l_buf) and the content of the buffer (buf). Figure 10 shows an example of the drop down list when you specify a SDU manually.

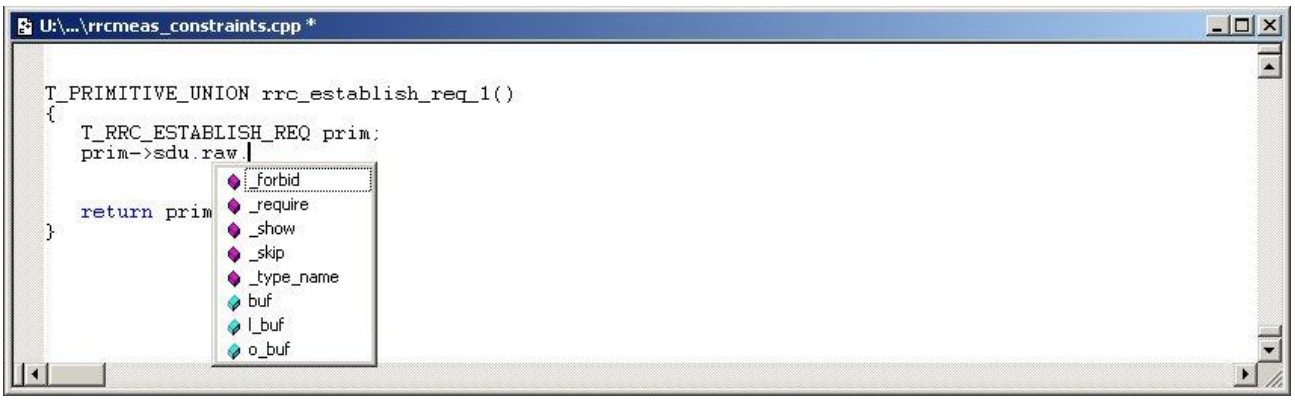


Figure 10 Manual specification of a bitstream in a SDU

3.8.4.2 GSM, GPRS and UMTS NAS (non ASN1)

When you want to specify a GSM, GPRS or a UMTS NAS message, which should be coded (on SEND) or decoded (on AWAIT) in CCD, the type "aim" should be used. The "aim" contains the TI and TIE.

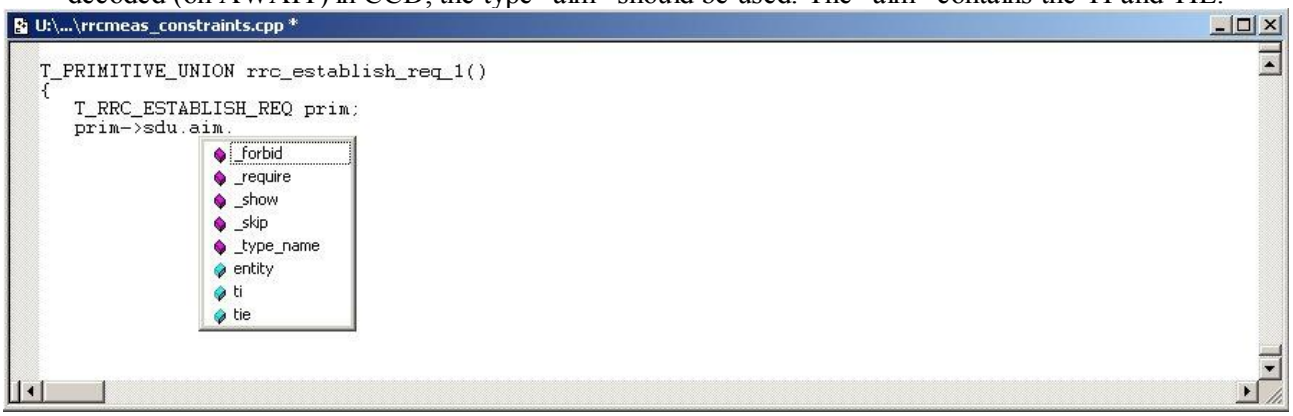


Figure 11 Selecting the type "aim"

As minimum you have to specify the TI – the TIE is optional (See figure Figure 11).

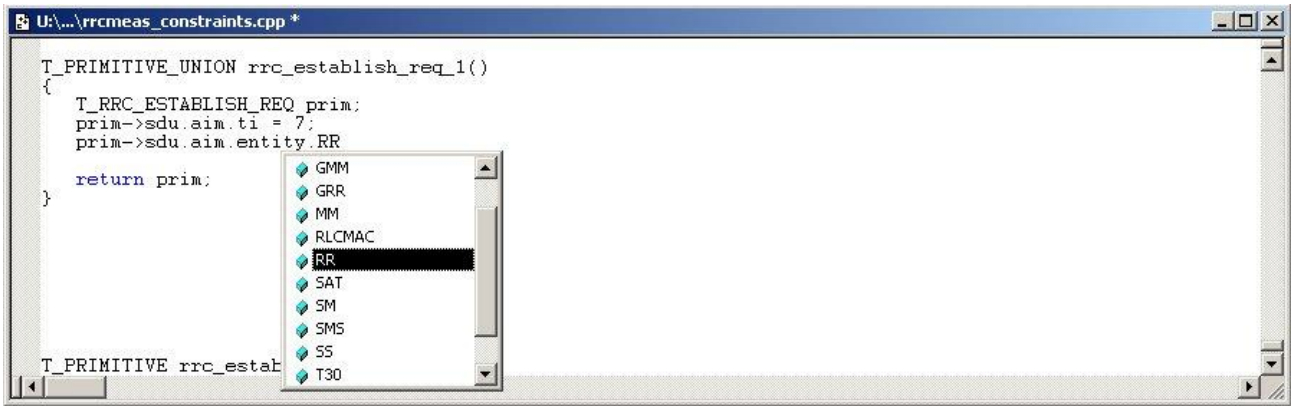


Figure 12 Specification of an entity

There's no need to specify the protocol discriminator[7], direction and message type[8]. This is implicit in the

[7] e.g. CC or MM. In TDS the Protocol discriminator was implicit as well, but the message type was abbreviated as PD.

specification of an aim.

However if the air interface message is empty you need to use the "_require() member function. Figure 13 shows an example, where the "msg" has been chosen. After this you select the AIM document, where the AIM you want to send is defined.

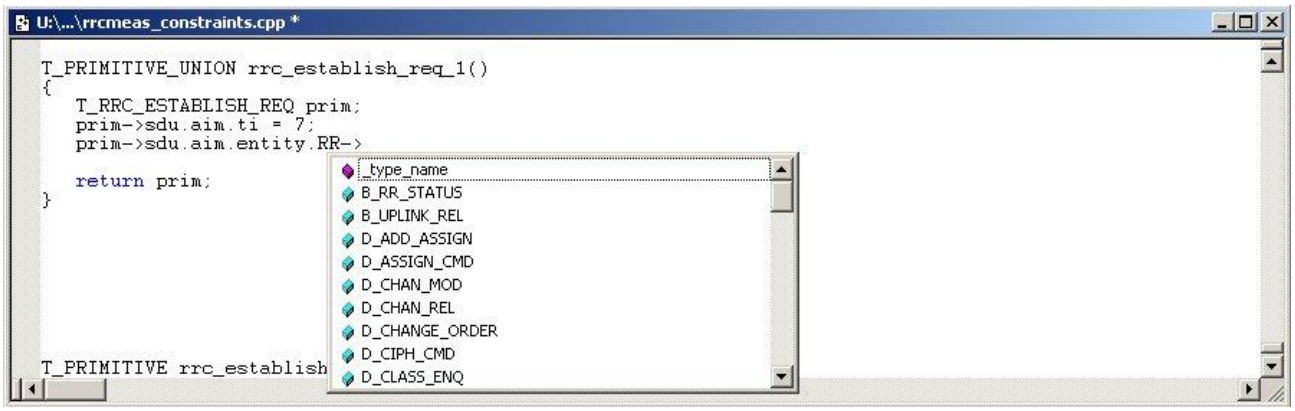


Figure 13 UMTS NAS specification of an SDU

After you have chosen the AIM you want to send, you specify the content of this like all other constraints.

3.8.4.3 ASN1 air interface messages

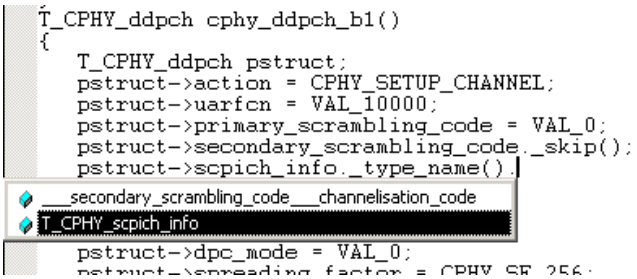
For UMTS AS AIMs the procedure is pretty much the same as UMTS NAS, but the TI and TIE is not specified. As AIM document ASN1 should be selected, so that you get a dropdown list with all the possible ASN1 AIMs. However for UMTS memhandles are used. This is not supported in the current tool chain, so unless the type "SDU" are used in the primitives it is not possible to specify AIMs in TDC testcases.

Handling of SIB's and MIB's is not yet supported, because they require special handling – e.g. they have to be coded / decoded twice. In one of the next releases of TDC it will be possible to encode single SIB's or MIBS, but without fully segmentation support.

3.8.5 Struct and union constraints

When specifying a struct or a union constraint and you use the drop down list you will get a list with member functions. Besides all the default member function all structs and unions will have those members specified in the SAP or MSG files.

As you might have discovered there are an additional choice in the dropdown list "_type_name". The **_type_name** is used to get information of the type of the member.



[8] E.g. CALL_SETUP_REQ or CALL_SETUP_IND

Figure 14 _type_name example

3.8.6 Array constraints

This section covers all types of arrays e.g. array of structs, array of numbers etc. The possible member functions for an array can also be selected from the drop down list. The member function [] operator applies a value to any given element in the array. Please note that it is not possible to assign, _require, _skip, _forbid or _show to array elements in TDC version 1.

3.8.6.1 Skipping of elements in an array

If you want to assign only the first elements in an array and skip the rest use the feature called SKIP_TO_END. Following example explains the use.

Let's imagine that we have an U8 array with 100 elements, but we only want to check the first 3 elements with values 0,1 and 2 and the skip the rest. First you should declare a normal array:

```
U8 array_3_elements[3] = { 0x00, 0x01, 0x02 };
```

After this use T_ARRAY:

```
T_ARRAY<U8> array_3(array_3_elements, SKIP_TO_END);
```

Assign array_3 to an array with more elements than 3 results in skipping of the rest of the elements.

3.8.6.2 Creating empty arrays

In C and C++ it is not possible to create arrays with zero elements. For some reason this feature has been introduced in TDS, so that it is possible to receive and send empty arrays. This is therefore also made possible in TDC. This example shows how this can be achieved:

First declare the array like this

```
T_ARRAY<U8> empty_array;
```

Then this instance can be used for assignment of empty arrays to a U8 array constraint.

3.8.6.3 Advanced array features

Following will first be available in TDC version 2.

When defining arrays the count will automatically be set to the number of defined elements in the array – e.g. according to the number for elements assigned to _skip(), _forbid() etc.

If an element in the array is set to forbid, you have implicitly set all the elements after this to forbid – e.g. it is enough to "forbid" one element.

If an element is set to _require all previous forbidden elements with lower index are changed to _require.

If an element is set to _skip all previous forbidden elements with lower index are changed to _skip.

3.8.7 Integer and enum constraints

In addition to the mentioned member functions, integer and enum classes also have a member function for each named constant. Invoking one of these named constant member functions has the same affect as assigning the value of the named constant. They are only added to have a convenient entry in the dot-complete dropdown list. See Figure 15 for an example.

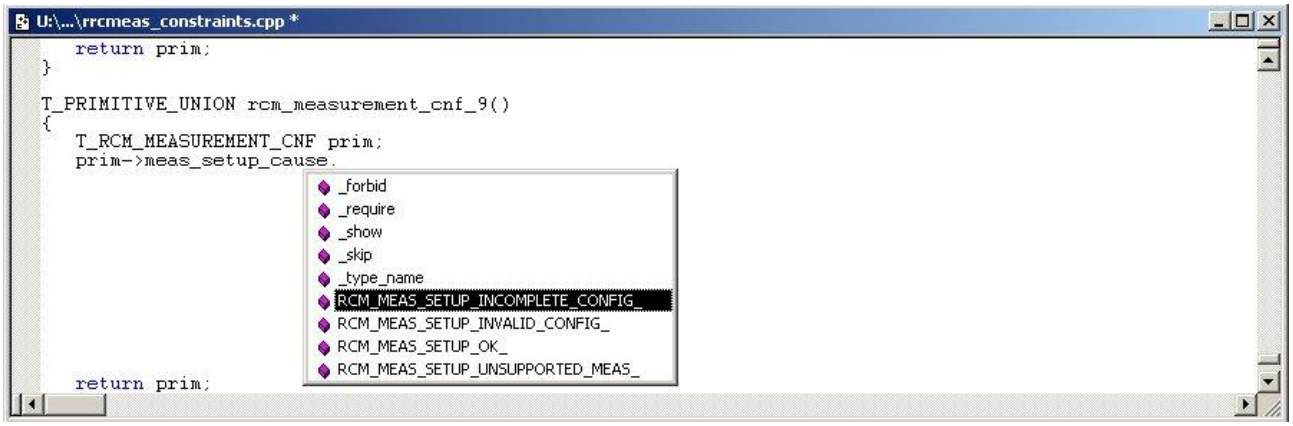


Figure 15 VC6 dot completion example on an enumeration

3.8.7.1 Bit constants for integers

A special set of constants exists for assigning a bit pattern to an integer:

`B0, B1`

`B00, B01, B10, B11`

`B000, B001, …, B111`

…

The maximum pattern length is 16 bit.

3.8.8 Bit and basic type array constraints

For bit array constraints a sequence of values can be assigned, using special value strings (see section 3.8.8.1). Accessing a single bit in a bit string can be achieved using the [] operator. This operator is overloaded in this case.

When initialising arrays of other basic types with a bit sequence, the number of bits must be a multiply of the number of bits in the basic type.

Please note that TDC release 1.0 does not support arithmetic operations on bit arrays.

3.8.8.1 Value strings

Value strings are on one of the forms:

BIN("…") where … are a sequence of one of '0', '1', or '?', optional ending with '*'.

HEX("…") where … are a sequence of hex digits or '?', optional ending with '*'. HEX"?" == BIN"????"

CHR("…") where … are a standard c-character sequence

The '?' acts in the same way as _required(). '*' is the same as _skip() till the end of the array. Please note that in TDC release 1.0 '?' and '*' can't be used. Value strings can be concatenated using the concat() function (2 to 10 arguments). The concat() function returns the concatenated string. Examples for using the

different value strings are listed below:

```
T_x x;
x->y.my_U8_array1 = HEX (“EF FE 09”);          /*The three first elements in the array set. The
                                                counter is set Implicit.*/
x->y.my_U8_array2 = BIN (“01010101 10101010”)  /*The two first bytes are set */


T_ARRAY<U8> string1;
T_ARRAY<U8> string2;
T_ARRAY<U8> string3;

string1 = CHR(”This is a string”);
string2 = CHR(“ This is another string”);
string3 = concat(string1, string2);            /*Result is: “This is a string This is another string”*/
```

As you see from the examples the values are separated by spaces. This is not necessary, but it provides a better readability. However you should be consequent in each use of value strings – e.g. don’t mix spaces with non spaces.

3.9 Specifying MUT

Please note that until version 2 is released this has to been with COMMAND (XXX).

4 TDC and Visual Studio 6

There are several Visual Studio macros available to easy the use of TDC. The first macro, StartTouchExternalDependencyfiles, is to easy the use of dot completion by touching the .h files in the external dependencies. Furthermore there are two tdc visual studio macros available for expanding the to a set of lines for each member and they can create a local name of a type to simplify long lines.

4.1 Loading TDC Visual Studio 6 macros

Before you can use these macros you need to have service pack 5 or higher, further more you need to load the "tdc_macros.dsm" and "kbdmac.dll" files under "tools->customize->add-ins and Macro files". The file can be found at:

\gpf\template\vc6

after this you should have the add-ins listed in Figure 16 Add-ins.

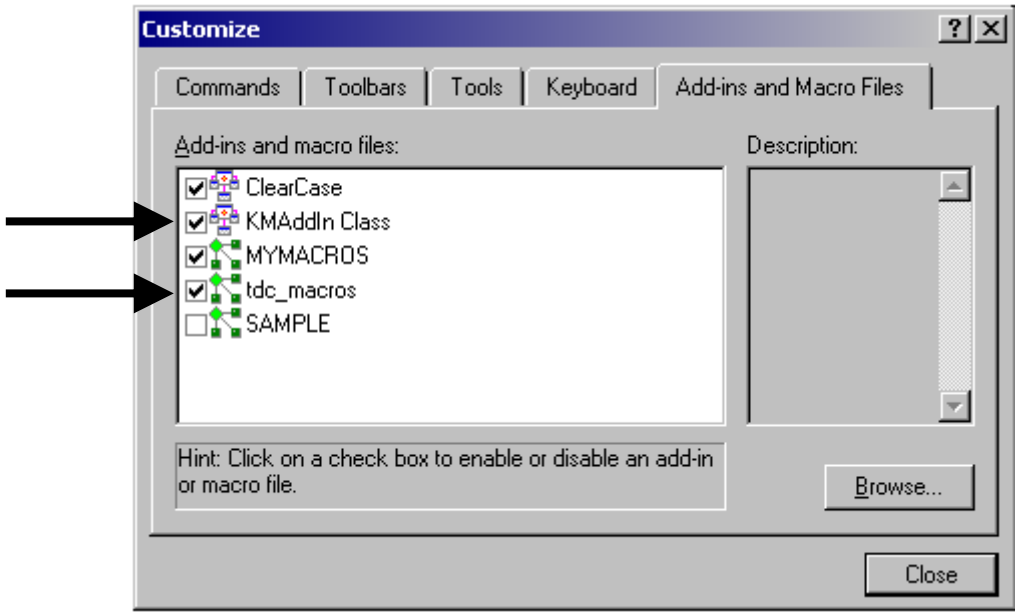


Figure 16 Add-ins

After loading the macro file you should run a macro called "tdckeybindings". This is done from "tools->macros" – see Figure 17 TDC macros.

Figure 17 TDC macros

After this you now have access to the 4 other macros listed in Figure 17 with short cuts. The short cuts are:

"CTRL & Q" = TdcMemberList

"CTRL & SHIFT & Q" = TdcTypeName

"CTRL+ALT+E" = "StartTouchExternalDependencyfiles"

("CTRL+ALT+Q" = "TouchExternalDependencyfiles")[9]

Please note that dot-completion should work before the macros will work, since the macros uses the dot-completion facility.

4.1.1 Known problems with TDC Visual Studio 6 macros

The first execution of the Visual Studio 6 macros can goes wrong for a number of reasons, usual it will work if one tries to execute them again. But to avoid the need for repeating the action of adding the add-ins to the load list it is recommended that you exit and re-run Visual Studio 6 after adding the add-ins to the load list. If you have any problems executing the macros.

Alternatively you can "touch" a .h file manually. You can do that by disabling the "Check out source file(s) when edited" in Tools->options->Source Control and afterwards open the .h files with the given type definitions. Make a space in the end of the .h file and press ctrl-z for undo afterwards. Close the file again without saving. Now the .h file has been touched and the "."completion will work for the types defined in that particular .h file. When no more .h files should be touched, you should enable the "Check out ..." option again.

4.2 Macro to touch .h files

This macro exports the make file and uses the <project-name>.dep file to get a list of all the .h files. The h files are opened sequentially and "touched". The opened files are not saved upon closing.

[9] Used internal by StartTouchExternalDependencyfiles

1. Compile/build in order to get the dependency list in the file view.

2. Run macro, and all the .h files should now be touched one by one.

The original macro has been expanded in order to work properly in conjunction with Clearcase. Running StartTouchExternalDependencyFiles starts this new macro. This will automatically start the TouchExternalDependencyFiles macro. The expansion consist of unchecking/checking an option in the "tools|option|"Source Control" tab at the start and end of the original macro. This way the message box popping up asking to check out files should be eliminated.

In order to use the new macro it is necessary to install the keyboard add-in KbdMac.dll. This is done in MS Visual C++ by selecting the tools|customize|"Add-ins and Macro files" tab. Use the browse function to find the file. The KbdMac.dll is located at \gpf\template\vc6. Furthermore it is necessary to install the latest service pack for Visual Studio 6 (service pack 5).

4.3 TdcMemberList

This macro is used to expand the struct to a set of lines one for each of the struct members sub members. Before running the macro:

```
T_CPHY_ddpch cphy_ddpch_b1()
{
   T_CPHY_ddpch pstruct;
   pstruct->action = CPHY_SETUP_CHANNEL;
   pstruct->uarfcn = VAL_10000;
   pstruct->primary_scrambling_code = VAL_0;
   pstruct->secondary_scrambling_code._skip();
   pstruct->scpich_info.
```

Figure 18 Before macro execution

As you can see from the figure you should enter a "." after the type you want to expand (scpich_info). Then press CTRL and Q and you will get the result:

```
T_CPHY_ddpch cphy_ddpch_b1()
{
   T_CPHY_ddpch pstruct;
   pstruct->action = CPHY_SETUP_CHANNEL;
   pstruct->uarfcn = VAL_10000;
   pstruct->primary_scrambling_code = VAL_0;
   pstruct->secondary_scrambling_code._skip();
   pstruct->scpich_info.secondary_scrambling_code;
   pstruct->scpich_info.channelisation_code;
```

Figure 19 Memberlist example

4.4 Typename example

The type name makes a variable of the type to avoid long lines.

```
T_CPHY_ddpch cphy_ddpch_b1()
{
   T_CPHY_ddpch pstruct;
   pstruct->action = CPHY_SETUP_CHANNEL;
   pstruct->uarfcn = VAL_10000;
   pstruct->primary_scrambling_code = VAL_0;
   pstruct->secondary_scrambling_code._skip();
   T_CPHY_scpich_info scpich_info;
   scpich_info->
   pstruct->scpich_info = scpich_info;
```

Figure 20 typename example

As you can see in Figure 20 a local variable "scpich_info" has been made and assigned to pstruct->scpich_info. Especially if you have very long lines this macro can be useful.

4.5 Debug test cases

It is possible to debug test cases in TDC format. This can be done directly from the Visual Studio project. To do so you need to add some settings to your project file.

4.5.1 Setup

Right-click on your project file and select "settings". Select the fan "debug" and set the category to general.

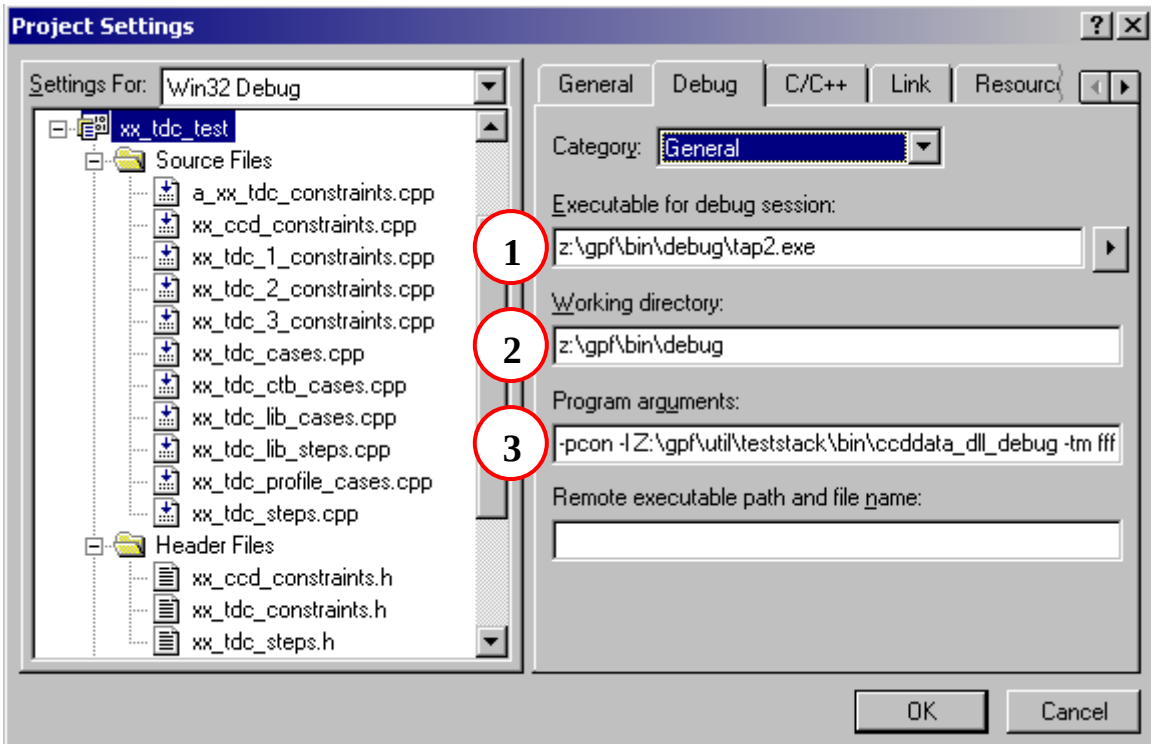


Figure 21 Project file settings

Fill in the fields (1) and (2) like depicted in Figure 21, except that Z should be replaced with the drive letter of your view.

The field (3) contains the arguments for tap2.exe. The whole line is not visible in the figure, but it is given as

"-pcon -l Z:\gpf\util\teststack\bin\ccddata_dll -tm ffff -v -t 500000 -tb xx_tdc Z:\gpf\util\teststack\bin\test_xx_tdc XX_TESTCASE482B"

where Z:\gpf\util\teststack\bin\test_xx_tdc is the test directory and XX_TESTCASE482B is the name of the test case. The other flags are more explicit described under 4.6.1. Press the OK button.

4.5.2 Run

To make sure, that parts of PCO is not "hanging" run "pco_kill" from a cmd-prompt. Optionally you can start the "pco_view.exe" from your cmd-prompt in the directory "\gpf\bin\debug\". You will now be asked if you wish to start the pco_server, which you of cause do.

Start the stack either from the cmd-prompt or from another Visual C++ project. Then just put in your break points and start debugging.[10]

4.6 Advanced Debugging

This part describes how to make more advanced debugging. It should give an overview of necessary settings in different situations, while it does not cover debugging in general.

4.6.1 Meaning of TAP2.exe arguments

In Figure 21 on the previous page the field (3) contains the arguments for TAP2.exe.

"-pcon -l Z:\gpf\util\teststack\bin\ccddata_dll -tm ffff -v -t 500000 -tb xx_tdc Z:\gpf\util\teststack\bin\test_xx_tdc XX_TESTCASE482B"

You can see a list of these arguments from your cmd-prompt by typing "tap2.exe" in the directory "gpf\bin\". But to summarize, the ones used in the example above are explained here

- -v : Writes more details to the screen.
- -pcon : This flag causes the primitives to be converted into a more compact type before they are sent to the stack.
- -tm <mask>: This sets the trace mask for the TAP.
- -l <filename>: This allow you to specify the ccddata_dll.dll file.
- -tb <filename>: Normally the a test case has the name structure "<dll_name><testcase_number><variant_letter>". If this is not the case the name of the .dll file can explicitly be stated with this flag, as seen in the example.
- -t <time (ms)>: This flag specifies the time to wait for a primitive. When debugging – remember to set this relatively high. Otherwise it will time out during single stepping. On the other hand one do not wish to wait for ages if an expected primitive is not send.

4.6.2 Tracing Stack Side Errors

When an error or warning appears while running a test case, the "Snd" column in the PCO_view shows where it has occurred e.g. SYST, ~SYST[11], ~TAP, XX...etc.

example:

Nr	Time	Snd	Name	Rcv	Content

[10] If you press F10 from Visual Studio tap_main.c will open. Now if you put a breakpoint at the last bracket of this main function, you will be able to read messages/errors in the dos-prompt windows before they are closed.
[11] "SYST" is the Stack side while "~SYST" is tool side (TAP/TDC/PCO_VIEW etc).

350	00005800 ms	SYST	<TRACE>	PCO	SYSTEM ERROR: No Partition available, entity XX, size 100004, xx_tdc_2.c(353)

If you want to trace where in the stack code the error occurred in preparation for putting in a breakpoint, the table below shows in which directory the files are located depending on the error type.

Snd	Directory
SYST	\gpf\frame
XX	Your stack code

To find out where the error exactly appeared, you can use the tool "grep" in the cmd-prompt to search through files for the error message you've seen. This is done in Figure 22.

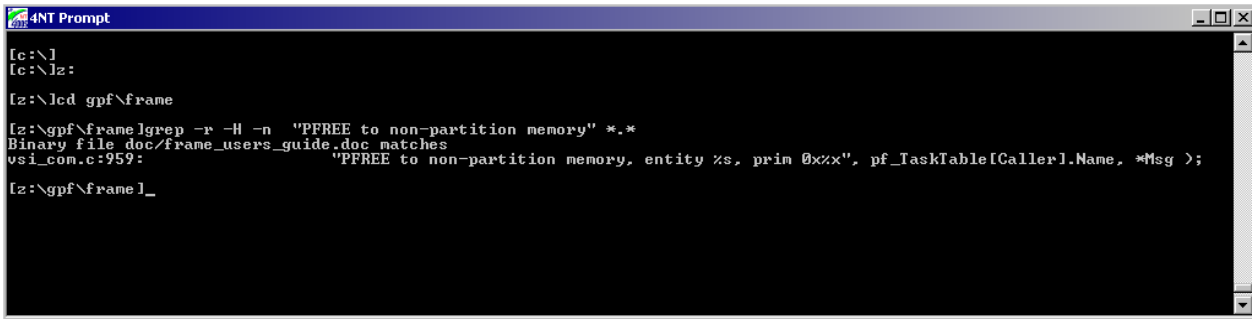


Figure 22 Finding the file and line where the error appeared using "grep"

The used flags for "grep" are seen in the table below.

Flag	Name	Description
-r	recursive	Read all files under each directory, recursively.
-H	With filename (Human)	Print the filename for each match.
-n	Line number	Prefix each line of output with the line number within its input file.

You can now open your stack project in Visual Studio and additionally open the file including the error message. In this case we open the file "vsi_com.c" and put the break point near line 959. Start the stack in debug mode by pressing F5.

4.6.3 Tracing Tool Side Errors - Additional DLLs

When a tool side error or warning appears while running a test case, the column Snd in the PCO_view shows where it has occurred e.g. ~SYST, ~TAP, ...etc.

example:

Nr	Time	Snd	Name	Rcv	Content
350	00005800 ms	~SYST	<TRACE>	PCO	-----

If you want to trace where in the tool code the error exactly occurred, you should use the tool "grep" as described in 4.6.2. The only difference is the directories to search in. These are showed in the table below. When the file and line with the error message is found you open your test project. Since the tool side is not static linked you have to add some additional .dll files to your project, to achieve the debug information. These files are also pointed out in the table below.

Snd	Directory	DLL
~SYST	\gpf\frame	frame.dll
~TAP	\gpf\util\tap\	tap_base.dll + tap_tdl.dll
~TDC	Your test case code	

Figure 23 shows how to add the additional .dll files. Right-click your project file and choose settings. Mark the Project file in the left side (1), choose the fan "debug"(2) and change the category (3) to "additional DLLs". Now you can specify the files you need. To debug the SYST error, like in the example above, you need frame.dll. Press the OK button.

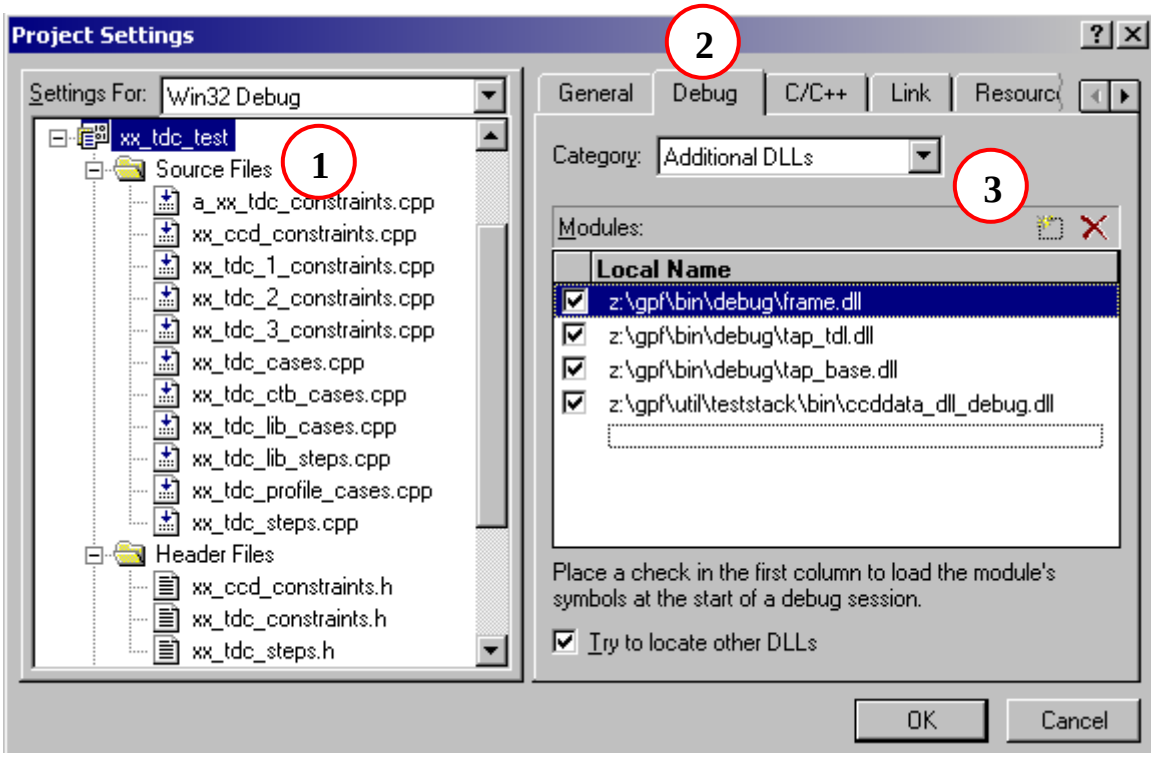


Figure 23 Adding additional DLL files

After you've started the debugger you can choose "exceptions" under the menu "debug", choose all the elements in the list, mark "stop always" and click "Change". With these settings you will be able to get information about where an error occurs even though it is encapsulated in a TRY/CATCH expression. If the

function “testentry()” in the code below assigns a NULL-pointer (This will result in an error!), you will stop by the assignment and not just end up in the exception code.

```
__try
{
    (void) testentry ();
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
 tap_trace ("Test case crashed");
 tap_set_exitcode (TAP_TC_CRASH);
}
```

4.6.4 Debugging TDC and TAP

To debug TDC and TAP you need to do the following: Choose FileView in the left side window in Visual Studio. You need to have a “tdc-lib” file including debug information. If the icon of the file only has two dashed lines, you need to replace it. This is done by deleting the one you’ve got and right-clicking your project file and choose “Add Files to Project...” in the list. The location of the file is \gpf\LIB\WIN32\debug\tdc.lib.[12]

Alternatively:

Alternatively you can have both the file with and without debug information. But then you have to exclude the one you don’t want to use from the build. This is done by right-clicking file and choose “settings”. Now in the fan “Generel” you can mark “Exclude file from build”. See Figure 24. Press the OK button and rebuild the project.

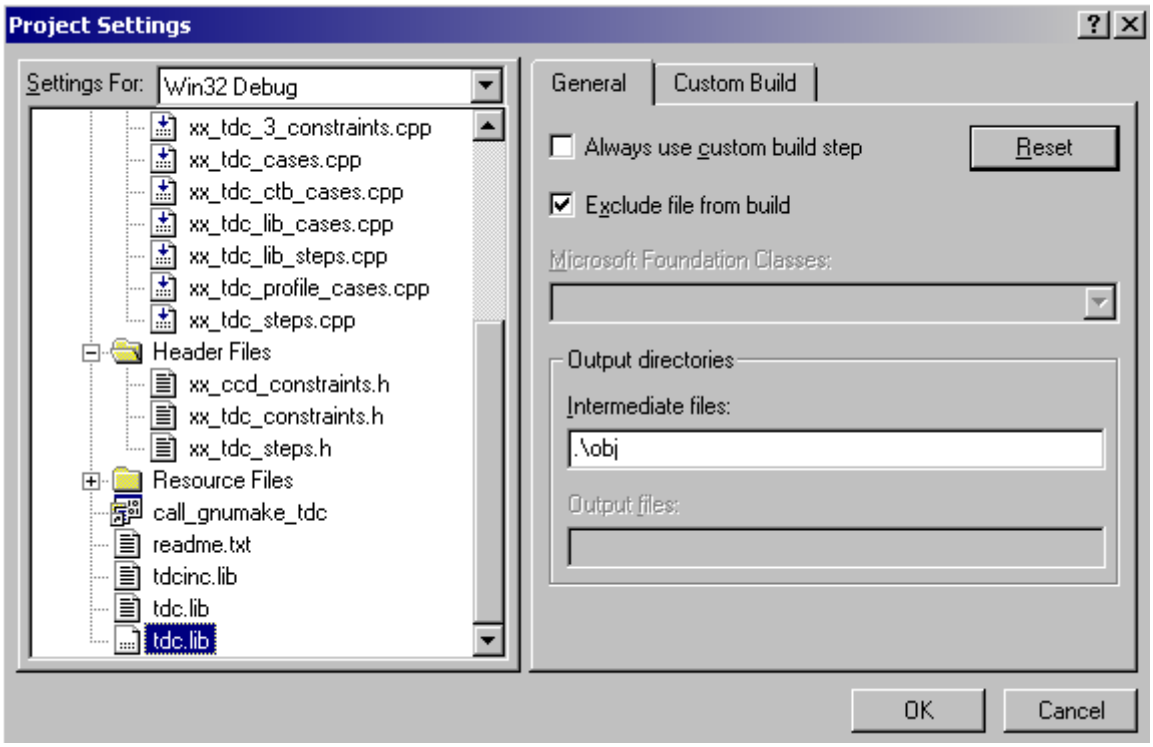


Figure 24 Choosing whether TDC and TAP are to be debugged.

[12] The locations of the file without debug information is \gpf\LIB\WIN32\tdc.lib

5 TDC Code standard

To ensure that the test cases written in TDC are compliant with TTCN-3 (for conversion later on) following rules must be followed.

If OOP (Object oriented programming) is used it is for further study how to convert that into TTCN-3 (only the use of the type T_ARRAY<> has been investigated). Whereas how to handle class, union and struct is unclear.

You should use TI standard types like U8, S8 etc.

Use of pointers (*) and the address of (&) operators are not allowed in connection with TDC types generated by CCDGEN.

Use of memcpy() on TDC types is not allowed either.

All user-defined functions should be done with lower cases.

Avoid the use of overloaded functions (function declarations that only differs in number of parameters or types of parameters).

Avoid assignments with implicit type conversion. – e.g. assignment of an char to U8 (U8 i = ‘a’), but use char to CHAR (CHAR c=’a’;). This will cause complications when converting to TTCN-3.

5.1 Test case layout

When you convert your existing test cases, they will be given this layout from the TDS_TO_TDC converter. We recommend that you try to maintain a header layout similar to the test case listed below.

```
/*--------------------Following is for test case USM204------------------------------

Description:
Pre-R98 test case only!  SM is in state PDP_INACTIVE. All available tis are already in use.
SM sends an SMREG_PDP_ACTIVATE_REJ to GMM with cause SMREG_RC_INSUF_RES.
Note: This test case will fail if extended TIs are in use, i.e. R99.
Preamble:
USM203

    ACI                         SM                          GMM
     |                           |                           |
(1)  | SMREG_PDP_ACTIVATE_REQ    |                           |
     *=========================>*                           |
(2)  |                           | GMMSM_ESTABLISH_REQ       |
     |                           *=========================>*
(3)  | SMREG_PDP_ACTIVATE_REJ    |                           |
     *<=========================*                           |
     |                           |                           |

-------------------------------------------------------------------------------*/
```

```
T_CASE (USM204, "Mobile_originated_PDP_context_activation_no_more_transaction_identifiers_available")
{
BEGIN_CASE ("Mobile_originated_PDP_context_activation_no_more_transaction_identifiers_available")
        {
   /*TEST STEPS*/
   Mobile_originated_PDP_context_activation_allocate_all_7_possible_nonext_tis_UMTS203();
           /*Events*/
          ……
        }
}
```

5.2 Test step layout

When you convert your existing test case all that are preambles will be converted to steps and they will have this layout. We recommend that you try to keep a header layout similar to the test step listed below. All calls to these steps will be inserted in test case and other steps.

```
/*-----------------------Following is for test step USM203-----------------------------------

Description:
SM is in state PDP_INACTIVE. The user tries to activate all possible 7 MS initiated contexts.
SM sends a GMMSM_ESTABLISH_REQ for each attempt.
Preamble:
USM100

 SNDCP/GACI                    SM                          GMM
     |                          |                           |
(1)  | SMREG_PDP_ACTIVATE_REQ   |                           |
     *=========================>*                           |
(2)  |                          | GMMSM_ESTABLISH_REQ       |
     |                          *=========================>*
(3)  | SMREG_PDP_ACTIVATE_REQ   |                           |
     *=========================>*                           |
(4)  |                          | GMMSM_ESTABLISH_REQ       |
     |                          *=========================>*
(5)  | SMREG_PDP_ACTIVATE_REQ   |                           |
     *=========================>*                           |
(6)  |                          | GMMSM_ESTABLISH_REQ       |
     |                          *=========================>*
(7)  | SMREG_PDP_ACTIVATE_REQ   |                           |
     *=========================>*                           |
.....
-------------------------------------------------------------------------------*/

T_STEP Mobile_originated_PDP_context_activation_allocate_all_7_possible_nonext_tis()
```

```
{
   BEGIN_STEP("Mobile_originated_PDP_context_activation_allocate_all_7_possible_nonext_tis")
   {
        /*Include other steps*/
        Setup_Routing_and_PCO_View_for_SM_Tests();
        /*Events*/
        …….
   }
}
```

6 Appendix

6.1 Test case example

For examples on tdc testcases please look at \gpf\util\teststack\spec\test\xx_tdc*.*

6.2 BNF for TDC syntax

The table below contains a BNF for the TDC syntax.

BNF	Comment
TestcaseBody ::= StatementList$_{optional}$	
TeststepBody ::= StatementList$_{optional}$	
StatementList ::= Statement StatementList$_{optional}$	
Statement ::= SendStatement | AwaitStatement | CommandStatement | TimeoutStatement | FailStatement | PassStatement | AlternativeStatement | TrapStatement | CcodeStatement | "{" StatementList$_{optional}$ "}"	
SendStatement ::= "SEND" "(" SendArgumentList ")" ";"	SendArgumentList from *_send* function prototypes[13]
AwaitStatement ::= "AWAIT" "(" AwaitArgumentList ")" ";"	AwaitArgumentList from *_await* function prototypes[14]
CommandStatement ::= "COMMAND" "(" CommandString ")" ";" //char*	CommandString is a *char**
TimeoutStatement ::= "TIMEOUT" "(" Milliseconds ")" ";" | "TIMEOUT_WAIT" "(" Milliseconds ")" ";" | "MUTE" "(" Milliseconds ")" ";" |	Milliseconds is an *int* Milliseconds is an *int* Milliseconds is an *int*
FailStatement ::= "FAIL" "(" ")" ";"	Goto end of BodyStatement[15]
PassStatement ::= "PASS" "(" ")" ";"	Goto end of BodyStatement[16]
AlternativeStatement ::= "ALT" "{" OnStatementList$_{optional}$ OtherwiseStatement "}"	
OnStatementList ::= OnStatement OnStatementList$_{optional}$	
OnStatement ::= "ON" "(" AwaitStatement ")" Statement | "if" "(" AwaitStatement ")" Statement	
OtherwiseStatement ::= OTHERWISE" "(" ")" Statement |	Statement default to pass[17]

[13] The _send function is an overloaded function thus there is more than one prototype.
[14] The _await function is an overloaded function thus there is more than one prototype.
[15] "fail();"jumps directly to the end of the end of "on(…)…;" if called inside "(…)", else it jumps directly to the end of the nearest enclosed "trap{…}" and executes the "onfail…"; part if it's present.
[16] "pass();" jumps directly to the end of the nearest enclosesing "alt{…}" or "trap{…}" an "onfail…" will be ignored.

/*fail by default*/ TrapStatement ::= "TRAP" Statement OnFailStatement CcodeStatement ::= “if” “(” “)” “{” “}” “else” “{” “}” | “for” “(” “)” “{” “}” | “while” “(” “)” “{” “}” “;” | “do” “{” “}” “while” “(” “)” “;” | “switch” “(” “)” “{ “ “case” “:” “}” “;” | *variable declaration* OnFailStatement ::= "ONFAIL" Statement CommandString ::= FromEntity “REDIRECT” Original_Dest_Entity New_Dest_Entity | “RESET” Entity | “FromEntity” “DUPLICATE” “Dest_Entity” “Dest_Entity” | “Entity” “CONFIG” “string”	Statement default to pass ENTITY could be the name of any entity, the ALL parameter or the TAP. The Dest_Entity is normally TAP, PCO or NULL. “string” is a char string.

Table 9 BNF for TDC keywords

6.3 List of member functions

The table below contains a complete list of available member functions for each generated type in the TDC header files.

name	**Explanation**	**Stuct-like**	**Union-like**	**Array-like**	**Pointer-like**	**Int-like**	**Enum-like**	**Pseudo pointer**[18]	**Local variables**[19]
_forbid()	Element must not be present.	Y	Y	Y	Y	Y	Y	Y	Y
_require()	Element must be present. Values are traced from tap.	Y	Y	Y	Y	Y	Y	Y	Y
_skip()	Element can be present.	Y	Y	Y	Y	Y	Y	Y	Y
_show()	Element can be present. Values and or elements are traced from TAP.	Y	Y	Y	Y	Y	Y	Y	Y
count	Set’s the count value of a variable array.	N	N	Y	N	N	N	N	N
[] operator	Get element of array.	N	N	Y	N	N	N	N	N
-> operator	De-reference a pointer	N	N	N !	Y	N	N	Y	Y
_typename	Expands a struct and assign a local operator to the parent structure.	Y	Y	Y	Y	Y	Y	Y	Y

[17] If the statement do not contain a pass() or fail() the default behavior is as if it ended with a pass();
[18] This type is generated for special array[1] generated by the ASN1_TO_MDF compiler.
[19] Local declared variables in your test code are types from SAP, MSG and normal Condat types like U8. (Including function arguments and return values).

CONST_NAME()	Same as _set(PREFIX_*CONST_NAME*). There will be a one for each CONST_NAME .	N	N	N	N	N	Y	N	N

Table 10 Member functions

6.4 Trouble shouting

This section contains fixes / workarounds

6.4.1 No dot completion

(For a solution to this problem see section 4.1.1)

If you don't have dot completion (e.g. it doesn't work) in your TDC project you should "touch" all relevant header files. The dot completion compiler works in another way than the normal compiler. You can "tell" the dot completion compiler to include a header file by opening the file in visual studio, insert a space somewhere in the file – and undo it afterwards. The dot completion compiler then knows that this file is to be included as well. The files you should touch are:

All interfaces files

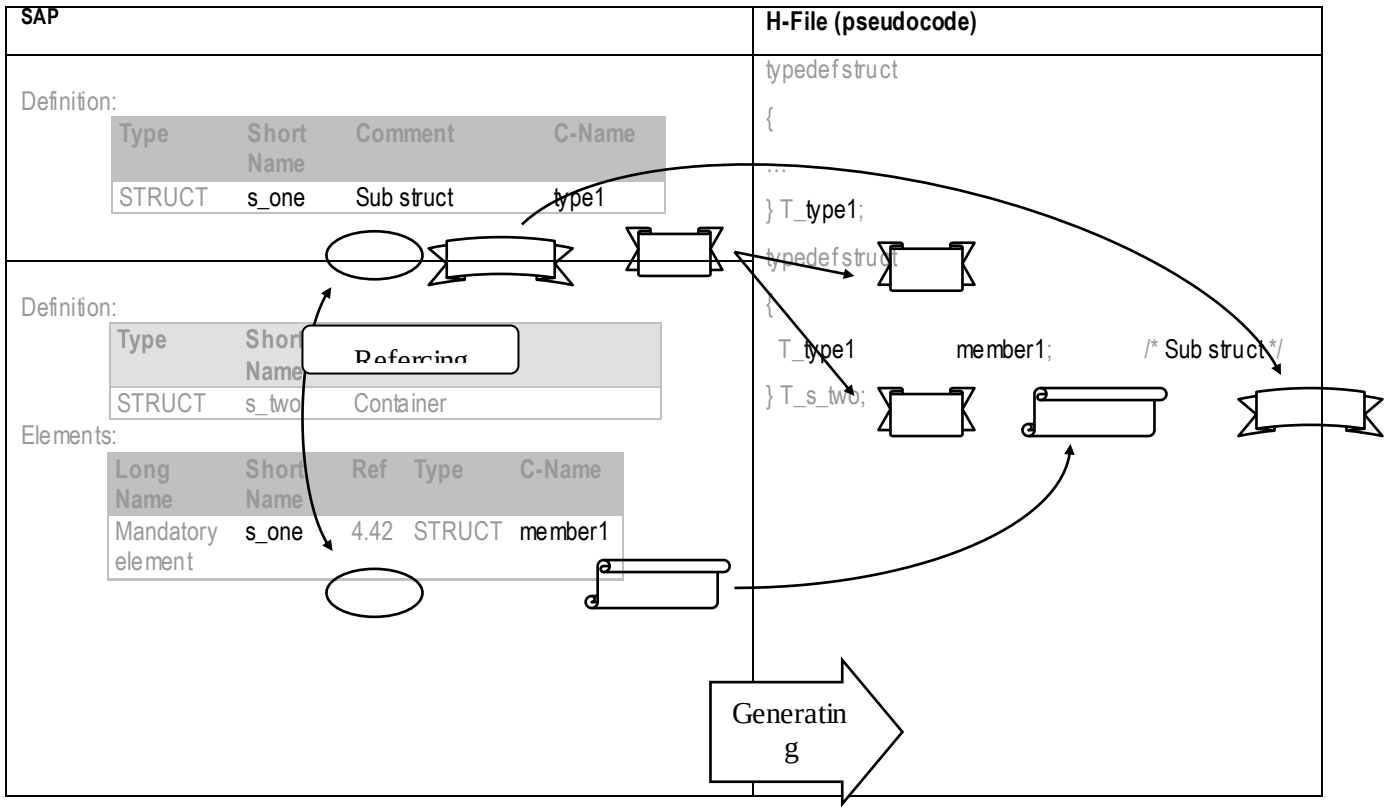
tdc.h

tdc_base.h

tdc_msg.h

tdc_prim.h

If 2 elements are basically the different stuff c-names should be used in definitions.



6.4.2 Linkage fails with error like "… unresolved external symbol "char BadLibVersionCheck_ …"

If you get something like the following error when trying to link your TDC test cases:

```
Linking...

Creating library rcm.lib and object rcm.exp

LINK : warning LNK4098: defaultlib "MSVCRT" conflicts with use of other libs; use /NODEFAULTLIB:library

interfaces.obj : error LNK2001: unresolved external symbol "char
BadLibVersionCheck____P_8010_115_RCM_SAP_VAL_H____Thu_Mar_06_11_31_22_2003"
(?BadLibVersionCheck____P_8010_115_RCM_SAP_VAL_H____Thu_Mar_06_11_31_22_2003@@3DA)

interfaces.obj : error LNK2001: unresolved external symbol "char
BadLibVersionCheck____P_8010_115_RCM_SAP_H____Thu_Mar_06_11_31_22_2003"
(?BadLibVersionCheck____P_8010_115_RCM_SAP_H____Thu_Mar_06_11_31_22_2003@@3DA)

\g23m\condat\ms\test\test_rcm\rcm.dll : fatal error LNK1120: 2 unresolved externals

Error executing link.exe.
```

This is coursed by using different versions of the SAP documents for compiling the TDC library and the TDC test cases.

Most likely you are looking on a different version of the SAP compared to when you compiled the TDC lib, or you are using a binary release version of TDC lib but looking on your private branch for the SAP

6.4.2.1 How lib version check works

A unik variable name are created for each SAP document in a common c-file, the names of these variables contain the file time stamp of the SAP word documents.

All SAP h-files references these variable, so if you use a different version of SAP file for compiling a the linker will be unable to solve this special variable since it have different names.

The variable is composed as follows:

BadLibVersionCheck____*filename____dayname_monthname_date_hour_minute_second_year*

6.4.3 All test-cases fails initial

Ensure you have a proper version of GPF (including any patches if you were told to include such in your config-spech)

Do **NOT** use the old tap (tap2_gprs.exe)

6.5 Know errors

For tdc version 1.2 following know errors / missing features exist:

6.5.1 Insufficient text in ~TDC traces

Some of the TDC traces are insufficient.

6.5.2 Value strings

The use of value strings, HEX, CHR and BIN is not implemented.

6.5.3 TDC lib handling

The TDC library generation is very time consuming. Therefore we recommend that the libraries are copied manually from the specified location (see section 2.4 TDC Libraries).

6.5.4 In deep copy

In deep copy is not implemented. Following example shows the problematic:

```
T_Y y;

y->a = 1;
y->b = 1;

x[0].y = y;
x[1].y = y;
x[1].y->a = 2; //Doing so will also change x[0].y->a to 2 as well.
```

6.5.5 Array handling

Please note that it is not possible to assign, _require, _skip, _forbid or _show to array members (bytes, shorts, longs or structs) in TDC version 1.