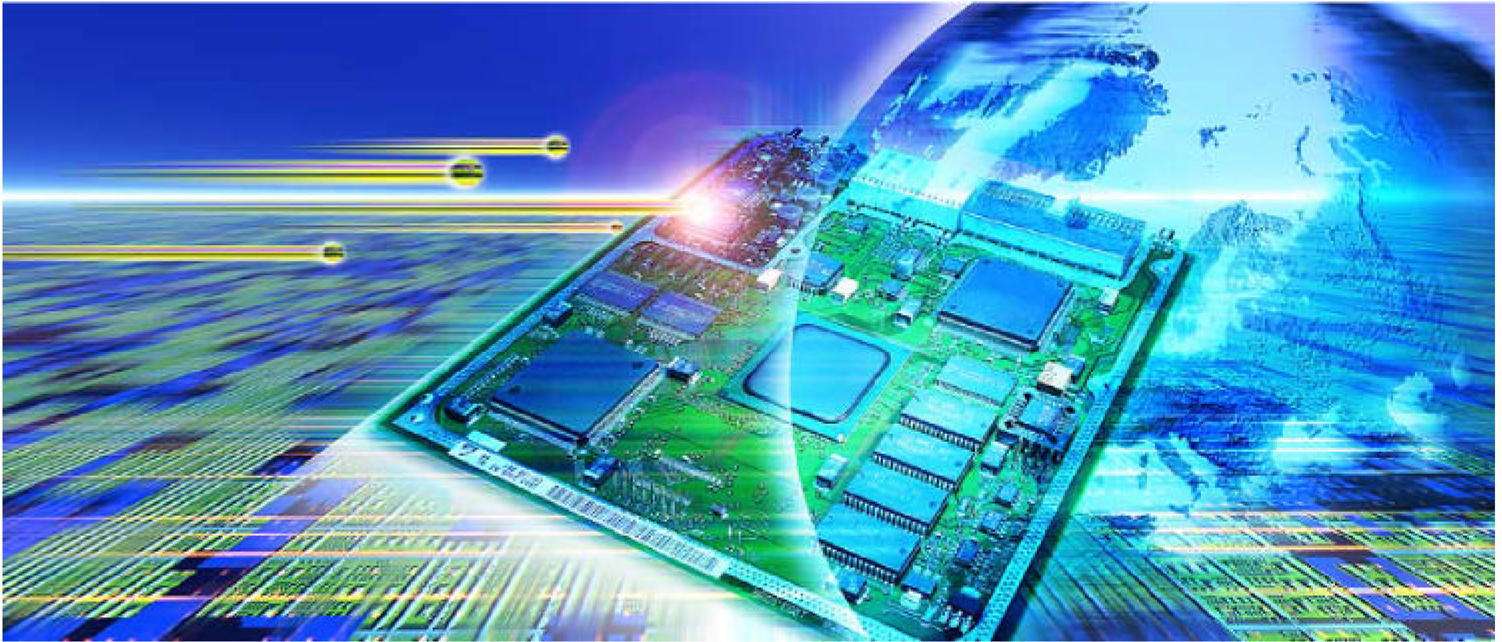


Product: GTSL

How to implement GTSL outside a TestStand ® environment

Application Note

GTSL (Generic Test Software Library) is part of the production test systems R&S TS7100/ TS7180, simplifies very much test programming with the CMU200 and can be used together with the “Test Executive” TestStand ® from National Instruments.

This application note shows how GTSL can be implemented outside a TestStand ® environment and describes the accompanying source code of a demo “Test Executive” developed in Microsoft ® Visual C++

Subject to change – Herman Gielen 06.2002 - 1SC01_0E

Contents

1 Overview 3
2 Software Features........ 4
3 Hardware and Software Requirements........ 5
Hardware Requirements 5
Software Requirements........ 5
4 Connecting the Computer and the Instrument 6
5 Required Skills........ 7
6 Installing the Software........ 7
Installation procedure........ 7
7 Starting the Software 10
8 Graphical User Interface 10
9 Compatibility Issues with GTSL libraries. 10
Header files. 10
GTSL Class Library........ 11
10 Launching a Test Sequence........ 13
11 Test Sequence Execution. 17
12 Utilities Library........ 20
Error Handling Methods. 20
Handling of User Interface. 21
Test Result Handling Methods. 26
13 Number of Sequences. 27
14 Literature 28
15 Additional Information 28
16 Ordering information 29

1 Overview

Although GTSL was specifically developed to be used with National Instruments TestStand®, a familiar name and well established platform in the world of Production Testing and Test Executives, GTSL can also be used on other platforms. In this application note we show that it can easily be integrated in almost any custom-made application program. Although it's clear that developing custom applications has its fare share of disadvantages (Long development time, high cost, etc…), it does not mean that there's only negative sides to it.

Commercially available test executives need to cater for a wide variety of potential customers working in different industries and using numerous platforms and systems.

It's obvious that overhead is never far away. The overhead in a custom made program is widely reduced since the required specifications are limited to the environment of one single end-user, in our example a manufacturer of a GSM mobile device. This advantage is reflected in simpler code and at the end faster execution times.

Another advantage is that, although the use of Test Executives is easier to master compared to lets say a programming language, it is also true that the pool of people mastering one or another programming language is definitely larger.

Windows based GUI applications can be developed on a wide variety of platforms, i.e. "Visual Basic", "Delphi", etc, or in our case "Visual C++". The given example makes use of the MFC (Microsoft Foundation Classes).

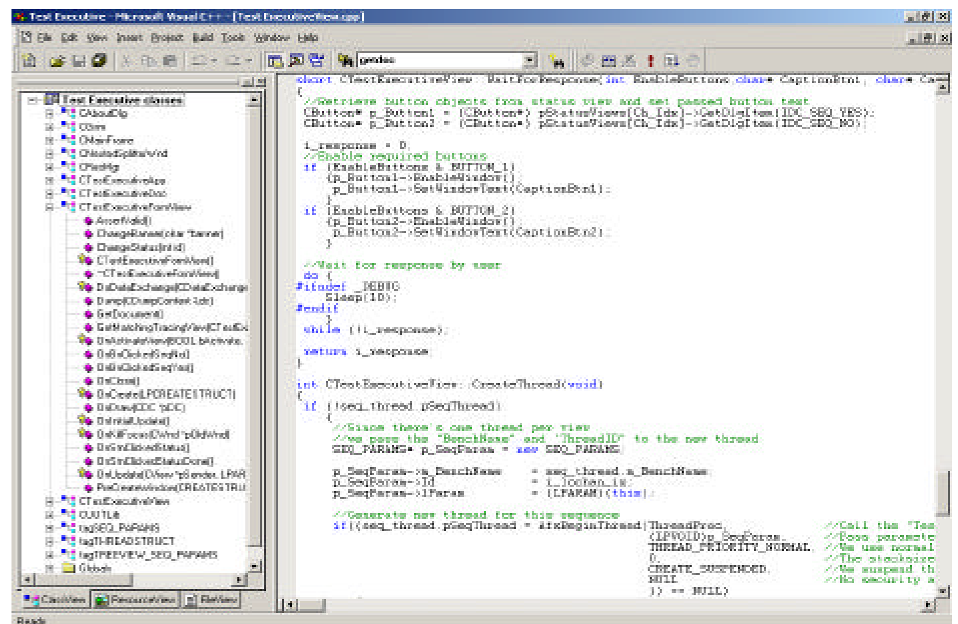


Figure 1. The demo was developed with MS Visual C++ ®.

For users who are unfamiliar with C++ and have a preference for other environments, this application note can also be used as a guideline to develop a similar Test Executive in a different environment. The only

prerequisite is that the chosen platform is able to import C or C++ style DLLs. Applications developed in "Visual Basic", "Delphi", or C based Windows applications developed with API libraries on a variety of compilers platforms can be just as successful.

The reason why we chose to develop the demo application with MFC is twofold. There are other object-oriented libraries available (OWL, VCL, etc.) however the MFC classes are widespread and can be considered as the de-facto standard for windows based application development in C++. As such it receives wide acceptance in the industry. In addition the MFC libraries contain the "Document/View" architecture that poses a considerable productivity advantage over other architectures. Although the understanding of the "Document/View" architecture requires serious initial efforts from the programmer, once understood it provides the ideal architecture for the multiple channel testing and device handling required for production testing, and already supported by GTSL.

As will become clear later on in this document, the link

- "Thread" (Required for multiple channel testing).
- "View" (Provided by the Document/View architecture).
- "Bench" (GTSL architecture for multiple channels).

can easily be made. By using the GTSL Libraries all the functionality of this platform (Path calibration, Multiple test channels, etc.) is integrated directly from the beginning. This relieves the programmer from reinventing the wheel.

2 Software Features

The simple custom-made Test Executive described in this application note was developed in a Visual C++ IDE and comes with all the required C++ source code. The demo code has the following functionality.

- Windows based Graphical User Interface.
- Multi-channel testing functionality.
- Advanced error, result, and status handling.
- Integrated Test sequence framework.
- Sample final QC test sequence for GSM900 mobile device.

Since most of the basic functionality is available, the demo program can be used as a foundation for any custom made test executive. If the user has the luxury of spending some extra development time, a more sophisticated version including Reporting, Statistical Process Control, etc can arise from it.

To make the initial development phase simpler, a "final QC test sequence" for a standard GSM900 Mobile device is already included.

For other type of devices, the available "Test sequence", "Error", and "Result" and "Status" handling framework can easily be adapted to fit the user's needs.

We mentioned earlier that the demo program comes in GUI form. Due to downward pressure on operator training time, in order for an application program to be successful in this era a GUI is not an option but merely a

must.

3 Hardware and Software Requirements

Hardware Requirements

- National Instruments ® GPIB adapter and cable
  (PCMCIA-GPIB (for notebook) or PCI-GPIB (For Desktop PC).
- Mobile device (GSM900 standard) with Test SIM.
- R&S CMU200 Universal Radio Communication Tester.
  (CMU-B21/K21 GSM option, firmware V3.07 or later).
- PC or Notebook.
  (With Windows NT or 2000 operating system
  Pentium 133 MHz / 64 MB Ram / Minimum 150MB Free disk space for GTSL and Test Executive).

Note: A CMU200 instrument supporting (a) different standard(s) and corresponding mobile device(s) can be used as an alternative as long as the test sequence in the demo program is modified accordingly.

Software Requirements

- Rohde & Schwarz GTSL V1.4, V1.5, or later.
  (And valid hardware key (iButton) and licence file).
- National Instruments ® NI-488.2 Drivers V1.6 or later.
  (Included with National Instruments GPIB adapter).
- Microsoft ©Visual C++ V6.0.
  (Or Visual Studio.Net)
- Rohde & Schwarz Test Executive Demo software.
  (Included with this application note).

4 Connecting the Computer with the Instrument

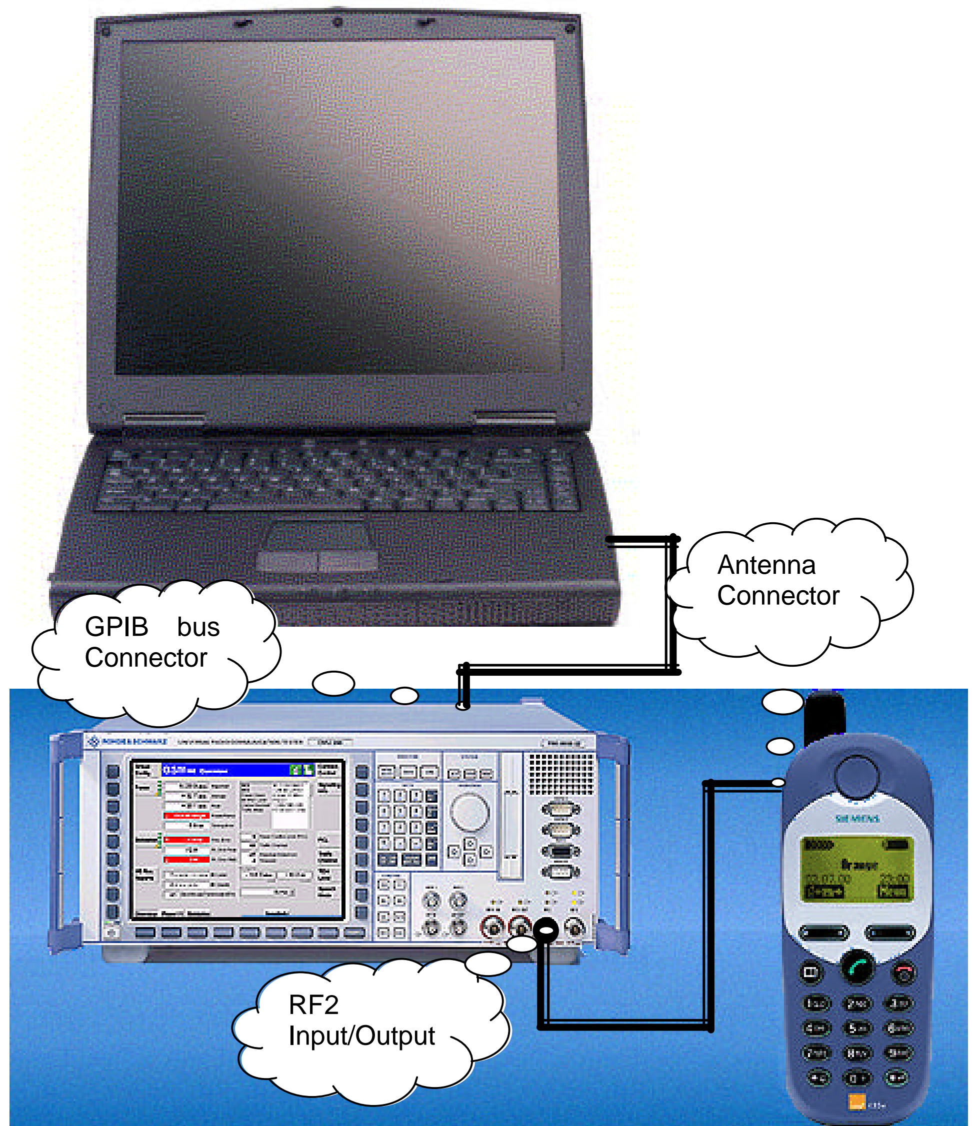


Figure 2: This is how you should connect the PC and Mobile.

Connect mobile's antenna connector to RF2 input/output. Connect GPIB connector at the back of CMU200 to PC's GPIB card.

5 Required Skills

The user should have the following skills if he/she wants to make modifications to the source code.

- Basic knowledge of C++ programming.
- Basic knowledge of Microsoft MFC.
- Basic knowledge of the Visual C++ IDE.
- Rudimentary knowledge of C.
- Familiarity with the Rohde & Schwarz GTSL platform.
- Rudimentary knowledge of GPIB bus technology.
- Basic knowledge of wireless technology.
- Basic use of CMU200 radio communication tester.

If you're not familiar with one of the above-mentioned technologies, equipment, or products refer to the literature paragraph.

6 Installing the Software

If one or more of the previously mentioned software components is missing on your PC you must install all the missing components.

Installation procedure.

1: Install NI-488.2 for Windows GPIB Driver software and GPIB card.

Test the correct operation with the “Getting started Wizard”.

You can find this tool under:

“Start->Programs->National Instruments NI-488.2”.

Figure 3:. NI’s Getting Started Wizard ®.

The "NI-488.2 User Manual" included with every GPIB adapter gives you additional information.

2: Install Rohde & Schwarz GTSL (V1.4, V1.5 or later).

Insert i-Button in COM port and verify presence of licence with the "License Viewer". This licence must at least include a Basic Licence "TS-LBAS" and one Option Licence. In the sample test sequence we use the GSM Libraries and as such the demo program requires the "TS-LGSM" option licence to be present. If you want to use the other available standards in GTSL the corresponding licence must be available instead.

You can find this tool under:

"Start->Programs->Rohde & Schwarz GTSL".

GTSL will not run without these licences!!!!!!.

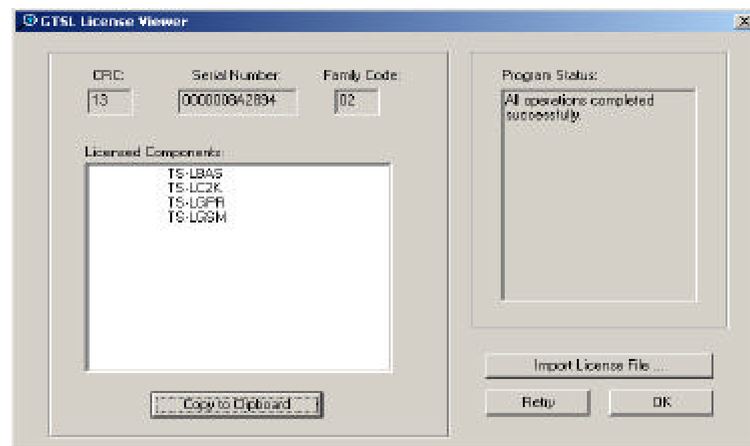


Figure 4: GTSL Licence Viewer.

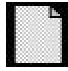
3: Install Visual Studio V6.0 or Visual Studio.Net.

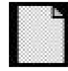
(If you plan to modify the demo program)

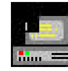
4: Install Rohde & Schwarz Demo Test Executive as follows:

Copy the 3 files included in this application note in any temporary folder and launch Setup.

Installation must take place in the **same folder as the installation folder of GTSL** (Default C:\Program Files\GTSL")
It's good practice to keep the default installation folders for both the ***Test Executive*** and ***GTSL.***







Notes: The Setup program installs the following files in the "Source Code" folder

All Project files, Make files, Source Code, etc....for Visual C++

- Test Executive executable.
- All the required libraries.
- Suitable "Application.ini" and "Physical.ini" files.
- Suitable Calibration files.

7 Starting the Software

You can find the Test Executive executable in the windows start menu under "Start->Programs->Rohde & Schwarz GTSL" after installation.
If you want to make modifications to the source code just double-click the Visual Studio workspace" Test Executive.dsw" in the "Source Code" sub-folder.

Remember that when you are using Visual Studio.Net a messagebox is prompted with the question if you want to convert the code to "dot.net projects" format.

8 Graphical User Interface

People familiar with MFC may notice that a close look reveals the "Document/View" architecture of the user interface. The framework of the source code was generated automatically with the "Visual Studio Wizard" and afterwards customised to suit our application. Since it is beyond the scope of this document to describe MFC, we limit ourselves to the code directly related to the test sequence. If you're interested to understand more about the GUI itself, you can refer to "Part II, Chapter 8 of Programming Windows 95 with MFC Jeff Prosise Microsoft Press. We recommend this since a good understanding of the architecture is essential to customise the application.

9 Compatibility Issues with GTSL libraries.

Header files.

The GTSL Libraries were developed with National Instruments Labwindows CVI ®. Most IDE compilers can handle C and C++ code, however Labwindows CVI ® restricts itself uniquely to C code. Although the Labwindow CVI ® compiler can generate Visual Studio C++ compatible code, not all header files are implemented this way. It is obvious that compatibility issues arise.

This means that for some of the GTSL header files the "extern "C" " directive must precede the inclusion.

```
extern "C"
{
#include "gsm.h"
}
```

Since the demo program comes with a ready made GSM900 test sequence, this has already been done for the "gsm" libaries, however if you plan make use of the other libaries and you refrain from including the "extern "C" " directive the linker might report an "unresolved external" error.

GTSL Class Library.

The header file incompatibility is not the only difference between the C and C++ environments.

Although C++ interprets C style functions perfectly it is object oriented in nature.

You have the option to use the C style functions directly, however this leaves a messy impression and contradicts with the object-oriented nature of C++. A better way is to convert the entire library directly from the beginning into a class library.

This leaves us with cleaner code afterwards. As one can observe from the workspace window, cGSM.cpp, cUUTlib.cpp and cResmgr.cpp with the corresponding header files cGSM.h, cUUTlib.h and cResmgr.h are present.

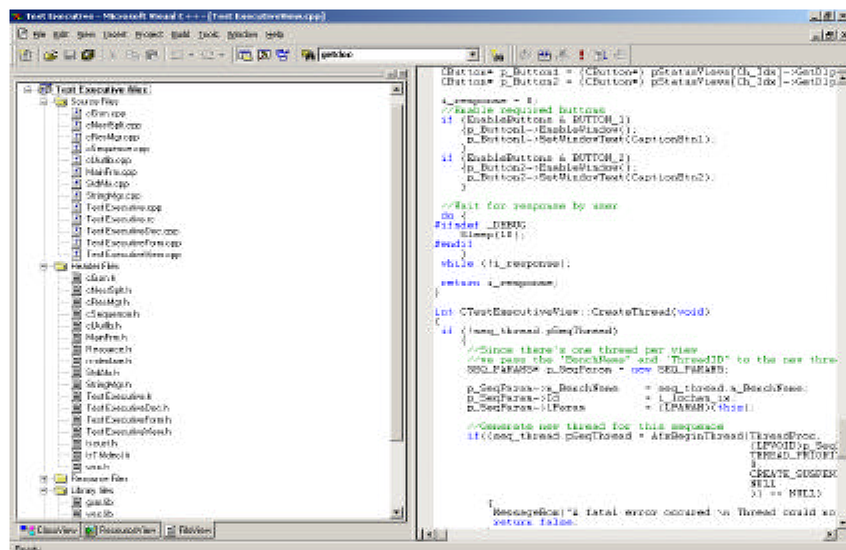


Figure 5: The FileView shows the GTSL Class Library Files.

These files simply contain the code conversion of the identically named original GTSL libraries

	GSM.dll	GSM.h
	UUTlib.dll	GSM.h
and	RESMGR.dll	GSM.h

As is obvious from the sample code below, the source code and header files with "c" prefix cGSM, cRESMGR, cUUTLIB contain the newly defined GTSL classes

The methods in the above-defined objects simply call upon the original GTSL library function. In addition a constructor and destructor is added to form a complete class.

The names are kept the same in order to keep the link with the original GTSL function clear.

Example: The equivalent method of GSM_Setup is CGsm::Setup

To keep the code simple, only the functions directly required by the demo program are converted and included in the classes. Again if additional functions are required in your test sequence, just add extra methods to the

class and call upon the corresponding GTSL function.

If you intend to use other GTSL libraries you need to build your own GTSL class libraries. Again this task should not take long. Just open a blank source code file, set-up a class and copy and past the code from the original GTSL code.

Further down in the demo program, objects of these classes are instantiated and used by the test sequence code. See later.

Ex. cGSM class library.

```
//Constructor
CGsm::CGsm(void)
{
}
//Destructor
CGsm::~CGsm(void)
{
}
//SetUp
void CGsm::Setup (CAObjHandle seqContextCVI,
                  char    *benchName,
                  long    *resourceID,
                  short   *errorOccurred,
                  long    *errorCode,
                  char    errorMessage[]
                 )
{
 GSM_Setup (seqContextCVI, benchName,resourceID,errorOccurred, errorCode, errorMessage);
}
```

10 Launching a Test Sequence.

The actual test sequence is implemented in the "cSequence.cpp" file. The methods declared in this file are however part of the "CTestExecutiveView" class and not a separate class. Why? Very simple, we mentioned already that the "Document/View" architecture was used in this application program. Later on it will become clear that every "View" (The window representing a particular channel) is linked to test sequence of that particular channel.

It's obvious that in this case it makes sense to declare the methods related to the execution of a sequence as members of the corresponding view.

Whenever a test sequence is started (By clicking the "Start Sequence" or corresponding menu item) a new thread is generated and launched only if no previously generated thread linked to that particular channel exists at the moment.

This is done in the "::OnSequenceStartsequence()" method of the "CtestExecutiveView" class.

```
void CTestExecutiveView::OnSequenceStartsequence()
{
 if (!seq_thread.pSeqThread)
        {ClearTree();
         //Create new thread;
         CreateThread();
         if (seq_thread.pSeqThread)seq_thread.pSeqThread->ResumeThread();
        }
 return;
}
```

This function calls the function that actually generates the thread

```
int CTestExecutiveView::CreateThread(void)
{
 if (!seq_thread.pSeqThread)
        {
         //Since there's one thread per view
         //we pass the "BenchName" and 'ThreadID" to the new thread
         SEQ_PARAMS* p_SeqParam = new SEQ_PARAMS;
         p_SeqParam->m_BenchName        = seq_thread.m_BenchName;
         p_SeqParam->Id                 = i_lochan_ix;
         p_SeqParam->lParam             = (LPARAM)(this);
```

```
        //Generate new thread for this sequence
        if((seq_thread.pSeqThread = AfxBeginThread
        (ThreadProc,        //Call the "Test Exeutive's Sequencer" function
        (LPVOID)p_SeqParam,             //Pass parameters to sequencer (thread)
        THREAD_PRIORITY_NORMAL,         //We use normal priority for all threads
        0,                              //The stacksize is equal to thread stack
        CREATE_SUSPENDED,               //We suspend the thread at the beginning
        NULL                            //No security attributes
        )) == NULL)
                {
                MessageBox("A fatal error occured:\n Thread could not be generated","Fatal
                Error",  MB_ICONSTOP | MB_OK);
                 return false;
                }
 seq_thread.pSeqThread->m_bAutoDelete = true;
}
return true;
}
```

In order for the thread to have easy access to all the necessary data required by the GTSL functions, the "p_SeqParam" structure is initialised with the "Bench name" and the "Channel number" and then passed further down to the function "AfxBeginThread". This function creates the actual thread.

```
SEQ_PARAMS* p_SeqParam = new SEQ_PARAMS;
p_SeqParam->m_BenchName        = seq_thread.m_BenchName;
p_SeqParam->Id                 = i_lochan_ix;
p_SeqParam->lParam             = (LPARAM)(this);
```

The "Start All" button and associated menu item operates in a slightly different way. Since all test sequences have to be launched if the user clicks this button, a windows message is sent to the message map of the "Document" and not the "View" .The reason behind this is simple. It's rather complicated to access a particular view from within another view. It's obvious that to guarantee the integrity of the views, the variables inside views are kept separated as much as possible. As such it would not make much sense to use the message map of the currently active view since the active and non-active view(s) don't have easy access to each other. The "Document" however does indeed have easy access to all its views.

Below the message map for the document window.

```
/////////////////////////////////////////////////////////////////////////////
// CTestExecutiveDoc
IMPLEMENT_DYNCREATE(CTestExecutiveDoc, CDocument)
BEGIN_MESSAGE_MAP(CTestExecutiveDoc, CDocument)
ON_COMMAND(ID_SEQUENCE_STARTALLSEQUENCES,OnSequenceStartallsequences)
ON_COMMAND(ID_SEQUENCE_STOPALLSEQUENCES,  OnSequenceStopallsequences)
```

```
 END_MESSAGE_MAP()
void CTestExecutiveDoc::OnSequenceStartallsequences()
{
 ToggleThread(true);   //Start all sequences
}
```

This means that the "CtestExecutiveDoc" object has a similar method to launch the test sequences namely "::ToggleThread".

```
void CTestExecutiveDoc::ToggleThread(bool Resume)
{
 //Loop through all views and toggle the attached thread
 POSITION pos = GetFirstViewPosition();
 while (pos != NULL)
  {
   CTestExecutiveView* pView = (CTestExecutiveView*) GetNextView(pos);
   CTestExecutiveFormView* pFormView =  (CTestExecutiveFormView*)
                                   GetNextView(pos);
   //User wants to start sequence
   if (Resume & (!pView->seq_thread.pSeqThread))
                 {
                  //Generate and launch new thread
                  pView->ClearTree();
                  pView->CreateThread();
                  if(pView->seq_thread.pSeqThread)pView->
                         seq_thread.pSeqThread->ResumeThread();
                 }
                 //User wants to stop sequence
   else if (!Resume & (pView->executing))
                 {
                  //Inform user that the sequence is about to be cancelled
                  pView->terminate = true;
                  pFormView->ChangeBanner("Sequence aborted. Please wait...");
                  pFormView->ChangeStatus(IDB_ABORT);
                 }
  }
  return;
}
```

The function retrieves the first view with the "GetFirstViewPosition" function, and then loops through all the remaining "Views" (and thus all channels) attached to the "Document". A pointer to the attached thread is then retrieved and subsequently launched.

```
//Generate and launch new thread
pView->ClearTree();
pView->CreateThread();
if(pView->seq_thread.pSeqThread)pView
                         ->seq_thread.pSeqThread->ResumeThread();
}
```

Note that the "::ToggleThread" function actually handles the launching and termination of a test sequence. This is achieved through the "Resume" flag. When the user clicks the "Stop All" button the function is called with this flag turned off and as is obvious from the code in the 2nd part of the function, a "terminate" flag is set to true. This flag in continuously polled inside the test sequence. Setting this flag causes the sequence and subsequently the thread to terminate.

11 Test Sequence Execution.

As is obvious from the previous code samples that the creation of the new thread gets the "ThreadProc" function name as an argument. The "ThreadProc" function in our demo program is declared as a “global” function. We do not have many alternatives here, since a thread function can only be declared as a method of a class if it’s declared as "static". This however, introduces multithreading problems since “static” declared methods are not created on the stack however are part of the class rather then the object and as such are inherently common for all threads.

```
UINT CTestExecutiveView::ThreadProc( LPVOID pParam )
```

Below is the body of the actual sequence execution.

```
UINT ThreadProc( LPVOID pParam )
{
 SEQ_PARAMS* seq_params = (SEQ_PARAMS*)pParam;
 CTestExecutiveView*  m_View  = (CTestExecutiveView*)seq_params->lParam;

 //Do "Pre-DUT-Test-Loop" sub-section
 if (!m_View->Pre_DUT_Test_Loop(NULL, seq_params->m_BenchName.GetBuffer(
            seq_params->m_BenchName.GetLength()),seq_params->Id)) goto RETURN;

 //Do main test loop
 do{
   m_View->ClearTree();
            //Do "Pre-DUT-Test" sub-section
            if (!m_View->Pre_DUT_Test(NULL, seq_params->m_BenchName.GetBuffer(
            seq_params->m_BenchName.GetLength()),seq_params->Id)) goto RETURN;
            //Do "Main-Test" sub-section
            else if (!m_View->Main(NULL, seq_params->m_BenchName.GetBuffer(
            seq_params->m_BenchName.GetLength()),seq_params->Id)) goto RETURN;
            //Do "Post-DUT-Test" sub-section
            else if (!m_View->Post_DUT_Test(NULL, seq_params->m_BenchName.GetBuffer(
            seq_params->m_BenchName.GetLength()),seq_params->Id))goto RETURN;
   }
//Prompt user "Continue" or "Stop"
#if defined(USEMESSAGEBOXES)
  while (IDYES == m_View->MessageBox("Do you want to test the next UUT ?",
                        seq_params->m_BenchName..GetBuffer(seq_params
                        ->m_BenchName.GetLength()), MB_ICONINFORMATION |
                        MB_YESNO));
#else
```

```
while (m_View->WaitForResponse(BUTTON_1_2,"Continue", "Stop", seq_params->Id)
            == BUTTON_1);
#endif
RETURN:
 //Do "Post-DUT-Test Loop"
 if (!m_View->Post_DUT_Test_Loop(NULL, seq_params->m_BenchName.GetBuffer(
            seq_params->m_BenchName.GetLength()),seq_params->Id)) return 0;
 m_View->seq_thread.pSeqThread = NULL;
 return 1;
}
```

The body of the "ThreadProc" function might look a bit confusing at the beginning though a simple architecture commonly used in production testing has been introduced here. This architecture is basically a simple loop.

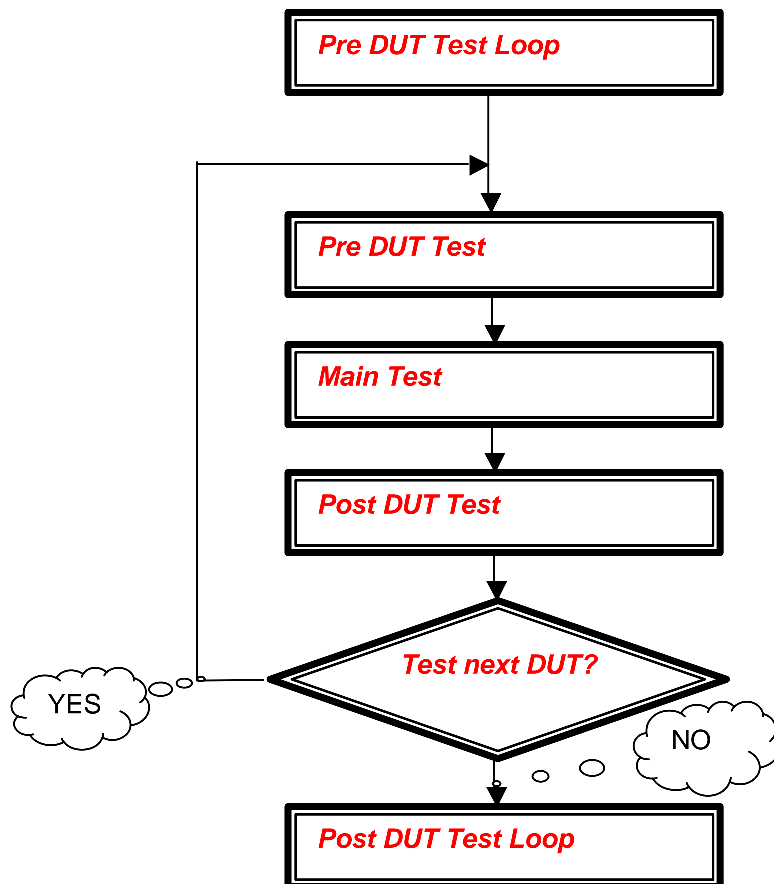


Figure 6: Basic architecture of test sequence..

The "Pre DUT Test Loop" typically contains tasks which must be done only once before the actual testing starts. Ex. Initialising instruments etc. The "Setup" function included in every GTSL Class library (CGsm::Setup, CResmgr::Setup,….) must be called in this section.

The "Pre DUT Test" typically contains tasks that must be done once for every DUT before the actual testing starts. Ex. Initialising DUT etc.

The “Main” contains the actual test sequence.

The “Post DUT Test” typically contains tasks that must be done after every test cycle before testing of the next DUT starts. Ex. Print DUT Test reports etc.

The “Post DUT Test Loop” typically contains tasks that must be done before termination of the test program. Ex. closing instruments etc. The “Cleanup” function included in every GTSL Class library (CGsm::Cleanup, Cresmgr::Cleanup,….) must be called in this section.

You’ll find the corresponding 4 core methods of the “CtestExecutiveView” class in the “cSequence.cpp” defined as:

"Pre_DUT_Test_Loop"

"Pre_DUT_Test"

“Main”
"Post_DUT_Test"

"Post_DUT_Test_Loop"

Inside the "ThreadProc" procedure, the above-mentioned methods are called in the correct order to resemble the test flow. The “decision” present in the flowchart was added to the code in order to let the user decide to continue or stop testing after every DUT. The user interface can be customized slightly. By choosing the USEMESSAGEBOXES macro definition message boxes pop-up to prompt the user. Otherwise the buttons in the bottom windows function as interaction between user and PC.

Note that if the user decides to continue with the next DUT the thread function does not return and continues to exist. Only when the decision is made to end testing the thread ceases to exist.

```
//Prompt user "Continue" or "Stop"
#if defined(USEMESSAGEBOXES)
  while (IDYES == m_View->MessageBox("Do you want to test the next UUT ?"
,seq_params->m_BenchName..GetBuffer(seq_params->m_BenchName.GetLength()),
MB_ICONINFORMATION | MB_YESNO));
#else
  while (m_View->WaitForResponse(BUTTON_1_2,"Continue", "Stop", seq_params->Id) ==
BUTTON_1);
#endif
```

We mentioned earlier that the actual function bodies of

"Pre_DUT_Test_Loop"

"Pre_DUT_Test"

“Main”
"Post_DUT_Test"

"Post_DUT_Test_Loop"

are implemented in the “cSequence.cpp” The user is free to modify the contents of these function according to his/her needs. The demo program has a sample body to test any standard GSM900 mobile device.

12 Utilities Library.

To interact with the user (interface) all the required functions are already included. This however requires that the programmer complies with some basic coding rules pertaining to "error handling", "result collection", and the "user interface ".

De methods made available for this purpose are split into a few groups. I.e.

Error Handling Methods.

The functions available in the GTSL library have a standard error reporting format. In the event of an error the error is reported by means of 3 variables.

```
&errorOccurred, &errorCode, errorMessage
```

To report this error to the user properly, it is necessary to call the "Error" method after every GTSL function call. This "Error" function takes the "errorMessage" parameters and reports this message to the user if the "errorOccurred" flag was set. If you stick with this concept, there's no additional error handling required.

```
if (errorOccurred) {Error(hWnd, hStep, BenchName, errorMessage, Ch_Idx); return (false);}
```

If you add additional test steps and thus additional error handling you just need to make a copy of this code. There's no need to change any of the variables passed to the parameters of the function. The "hStep, BenchName, Ch_Idx" are included in order for the user to trace the channel that actually generated the error.

Handling of User Interface.

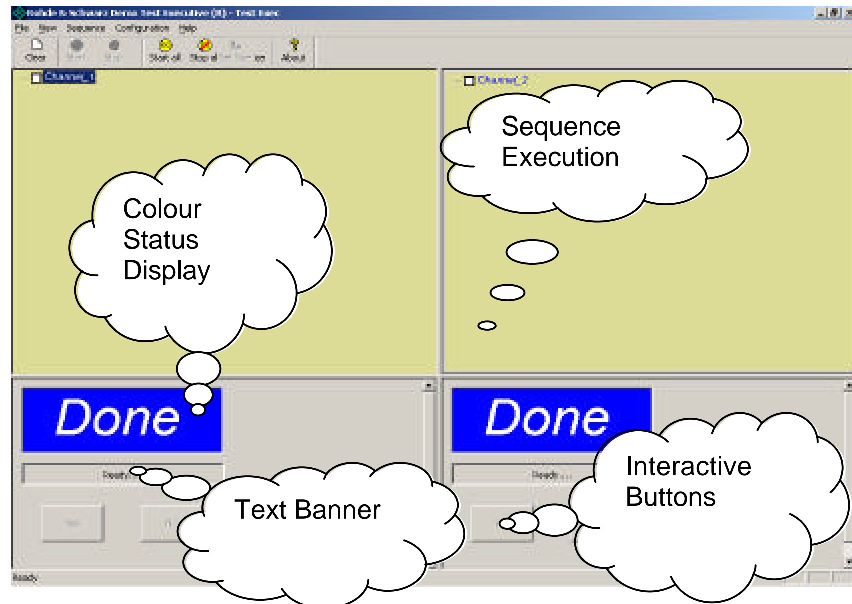


Figure 7: The MMI has multiple features.

To communicate with the user there are several options available.

The lower "Status" view has the following man machine dialogs available.

1 Status window, 1 Banner Text message, 2 Buttons.

The upper "Execution" view has the following man machine dialogs available.

1 Treeview

1: Status window

The banner panels in the bottom windows can be changed during test sequence execution with the "ChangeStatus" function.

"ChangeStatus" is a member of "CtestExecutiveFormView". This is the object that invokes the lower status views. Since the test sequence has no direct access to the member methods of this object we need another solution to call this method from within the "cSequence". Pointers to the "CtestExecutiveFormView" objects are stored in a global variable (array) "pStatusViews[NUM_SEQS]" when the view is instantiated. This global variable is used to access the

"CtestExecutiveFormView" methods like the "ChangeStatus" and others.

```
pStatusViews[Ch_Idx]->ChangeStatus(IDB_TERMINATE);
```

The "ChangeStatus" function has the following predefined banner selections.

IDB_TERMINATE	Displays blue TERMINATE banner. (Used to show that test sequence is done)
IDB_FAIL	Displays red FAIL banner. (Used to show that DUT did not pass test)
IDB_ERROR	Displays yellow ERROR banner (Used to show that an ERROR occurred during the test cycle)
IDB_PASS	Displays green PASS banner. (Used to show that the DUT passed the test successfully)
IDB_TESTING	Displays cyan TESTING banner. (Used to show that DUT test cycle is in progress)
IDB_INSERT_DUT	Displays cyan INSERT DUT banner. (Used to inform the operator that he/she must connect the DUT)
IDB_CONFIGURATION	Displays cyan CONFIGURATION banner. (Displayed during the initialisation cycle: Pre DUT Test loop, and/or Pre DUT Test)
IDB_ABORT	Displays blue ABORT banner. (Used to announce that user terminated the sequence.

The different banners can be used randomly throughout the test sequence however it is obvious that there's some "logic" involved here. The location of the banners in the demo sample GSM900 test sequence are logically placed and should not require too much modification. Some banners like the IDB_ERROR, IDB_FAIL, etc do not need to be called at all inside the sequence since they are already called by the error and result handler.

2: Banner Text.

Identical to the status window, the banner text has a corresponding function to give the user the ability to inform the user namely "ChangeBanner"

The "ChangeBanner" has only one parameter, which is a pointer to a character string with the text that must be displayed. The way the function is called is identical to the previous one.

```
pStatusViews[Ch_Idx]->ChangeBanner("Configuration. Please wait...");
```

Use this function whenever you want to inform the user.

3: Buttons.

To give the user the ability to respond to the messages prompted by the test sequence in the banner text, there are 2 options available. Either user interaction takes place through "message boxes" or through the "2 available button controls". Selection can be made with the macro definition: "USEMESSAGEBOXES".

Inside the test sequence the following structure must be respected to handle the 2 different ways of interaction.

```
#if defined(USEMESSAGEBOXES)
   MessageBox  ("Switch on Mobile Device", Channel, MB_ICONINFORMATION | MB_OK);
#else
   WaitForResponse(BUTTON_1,"Ok", "", Ch_Idx);
```

```
#endif
```

In case of message boxes program flow is handled inherently by the messagebox itself. In case of the buttons the "WaitForResponse" method has all the required code to handle the flow.

If you need user interaction in your sequence you can copy, paste, and amend the code wherever necessary in your sequence.

Parameter description:

EnableButtons:	Selects which button must be active Options: BUTTON_1_2, BUTTON_1, BUTTON_2
CaptionBtn1:	Button 1's caption text.
CaptionBtn2:	Button 2's caption text.
Ch_Idx:	Pass the id of the corresponding view.

The function returns BUTTON_1, or BUTTON_2 (The ID of the button the user clicked). The function automatically dims the buttons after operation.

```
Short WaitForResponse (int EnableButtons,char* CaptionBtn1, char* CaptionBtn2, int Ch_Idx);
```

Below the function body.

```
short CTestExecutiveView::WaitForResponse(int EnableButtons,char* CaptionBtn1, char*
CaptionBtn2, int Ch_Idx)
{
 //Retrieve button objects from status view and set passed button text
 CButton* p_Button1 = (CButton*) pStatusViews[Ch_Idx]->GetDlgItem(IDC_SEQ_YES);
 CButton* p_Button2 = (CButton*) pStatusViews[Ch_Idx]->GetDlgItem(IDC_SEQ_NO);

 i_response = 0;
 //Enable required buttons
 if (EnableButtons & BUTTON_1)
       {
        p_Button1->EnableWindow();
        p_Button1->SetWindowText(CaptionBtn1);
       }
 if (EnableButtons & BUTTON_2)
       {
        p_Button2->EnableWindow();
        p_Button2->SetWindowText(CaptionBtn2);
      }
 //Wait for response by user
 do {
#ifndef _DEBUG
   Sleep(10);
#endif
     }
 while (!i_response);

 return i_response;
}
```

4: TreeView.

The Treeview has a 2 level nesting structure. The 1st level represents the name of the "Subsequence". The 2nd level the "Step".

A subsequence can be added in the treeview with the "AddSubSequence" function. Since there are a total of 5 possible subsequences the "SubSequence" parameter can only have the following values to display

the corresponding treeview node.

PRE_DUT_TEST_LOOP

PRE_DUT_TEST

MAIN_DUT_TEST

POST_DUT_TEST

POST_DUT_TEST_LOOP

The predefined sample sequence includes the code to insert of all the treeview nodes at the correct location in the individual subsequences and as such only little or no modification at all is required here. It's a different story for the steps. Since the amount of steps depends on the required test sequence, ultimately the programmer must take care of this. The "AddStep" function handles this.

Parameter description:

SubSequence:	Name of the subsequence where you want to add the step.
StepName:	Name of step (Any name)

```
hStep = AddStep(PRE_DUT_TEST_LOOP,"Resource Manager Set-up");
```

The function returns a handle to the inserted step. The reason for this is to have access to its properties especially name and the attached checkbox.

Since a tree node represents the execution of one particular test step, the node must be modified after the test to inform the user about the outcome. This is possible with

"AddResult". Just pass the just obtained handle of the step to the function together with one of the following results.

FAIL

PASS

ERR

DONE

As result the following programming sample shows the pattern that is followed throughout the entire test sequence to obtain a fully-fledged test sequence with error handling and user interaction.

```
//Set-up GSM
hStep = AddStep(PRE_DUT_TEST_LOOP,"GSM Set-up");
ThisGsm.Setup (0, BenchName, &resource_ID_gsm, &errorOccurred,
                    &errorCode,errorMessage);
if (errorOccurred){Error(hWnd, hStep, BenchName, errorMessage, Ch_Idx); return (false);}
AddResult(hStep, DONE);
```

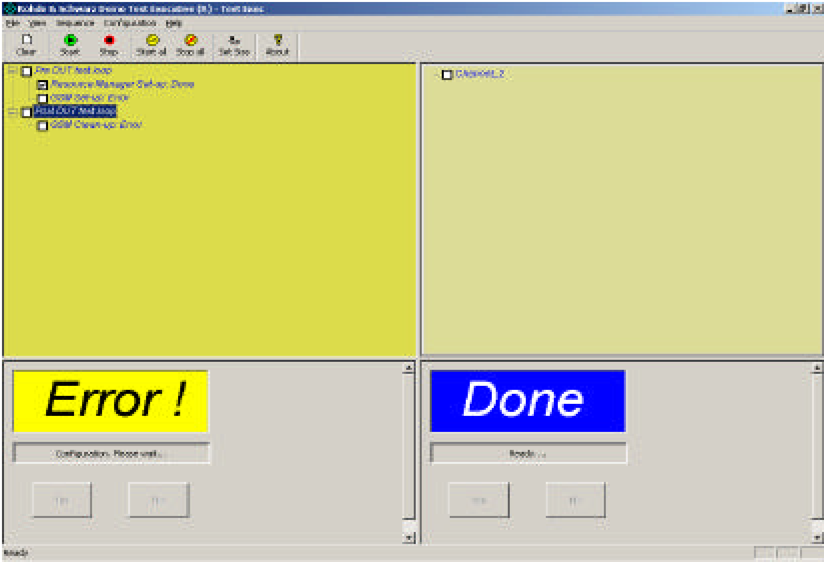


Figure 8: A test sequence terminates with an error.

Test Result Handling Methods.

If a particular measurement step returns a result (Actually all tests do), the obtained test value(s) must be recorded and processed. A standard function can be used to handle this.

```
void Record_Result (double UpperLimit, double LowerLimit,char* Stepname, double result,bool is_bool);
```

You must pass the Upper Limit, Lower Limit, (Normally constant values) and test Result to the function. The Upper and Lower Limits are used to make the actual "Pass/Fail "decision, If the result you want to include is already a "Pass/Fail" result instead of a numeric value you must set the "is_bool" flag. Doing so ignores "Within limits" checking. Passing a "Pass/Fail" requires type casting since all results are stored in "double" format.

The function automatically inserts a new step into the treeview and attaches a "Pass/Fail result" to the step. In addition the Checkbox of the treeview is checked or left blank according to the result of the test. If you like the test limits to be included in every step, uncomment the "INCLUDE_LIMITS" macro.
A Typical code segment of a test step looks as follows

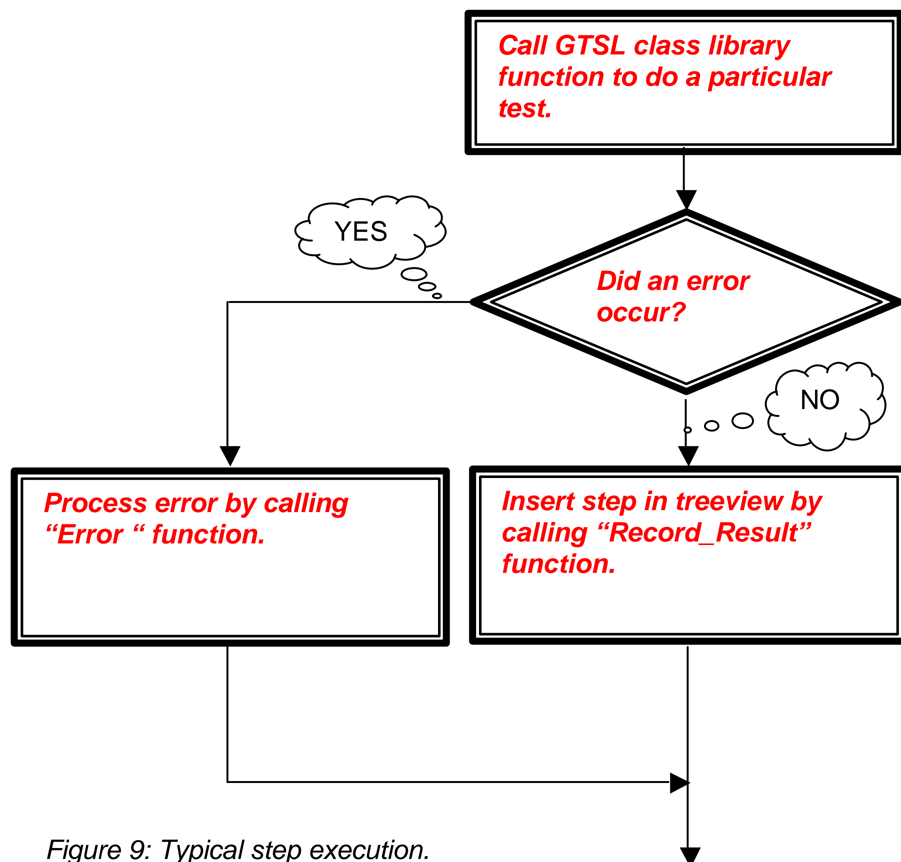


Figure 9: Typical step execution.

The sample below is a small part of the GSM900 test sequence resembling one particular test step.

```
//Set TCH and PCL
ThisGsm.Sig_Conf_PCL_TCH_TS (        0,
                                      resource_ID_gsm,
                                     PCL[i],
                                     TCH[i],
                                     TS[i],
                                     &errorOccurred,
                                     &errorCode,
                                     errorMessage);
//Get Frequency Error
ThisGsm.Sig_Fetch_Freq_Error (       0,
                                     resource_ID_gsm,
                                     0,
                                     (result_value + j),
                                     &errorOccurred,
                                     &errorCode,
                                     errorMessage);

if (errorOccurred) {Error(hWnd, hStep, Channel, errorMessage, Ch_Idx); return (false);}

Record_Result(     FREQ_ERROR_UPPER_LIM_RX,
                   FREQ_ERROR_LOWER_LIM_RX,
                   "Frequency Error",
                   *(result_value + (j++)),false);

Sleep(500);
```

The following test is done in the sample code.
The ThisGsm.Sig_Conf_PCL_TCH_TS, instructs the mobile device to switch to a particular power level, timeslot, and frequency channel. ThisGsm.Sig_Fetch_Freq_Error retrieves the frequency error of the mobile's signal, and finally the frequency error together with upper and lower limit values FREQ_ERROR_UPPER_LIM_RX, and FREQ_ERROR_LOWER_LIM_RX are passed to the Record_Result function. Note that the actual result is stored in an array "*(result_value + (j++))". The array index must be incremented after every test to ensure that all results are stored in a unique location. This is required to keep the test results until the last moment, so that they can eventually be used in a report or transferred to a database.

Note that some programming steps include a "sleep" function. This is often required to give the DUT time to respond to the modified channel, level, etc.… settings. If the measurement takes place to early, the result might be erroneous.

13 Number of Sequences.

Due to the previously described concept, the demo Test Executive is able to run an unlimited (At least in theory) number of sequences in parallel. The number of sequences is defined in the "NUM_SEQS" macro definition. You are free to modify this number however from a practical point of view it should be limited to 4.
"NUM_SEQS" is defined in "TestExecutive.h".

14 Literature

	Title	Author	Publisher
•	Programming Windows 95 with MFC	Jeff Prosise	Microsoft. Press
	Very good book on MFC. Although Windows 95 appears in the title, it's applicable to NT and other Windows versions as well.		
•	ANSI C++ in 21 days	Jesse Liberty and J.Mark Hord	Sams Publishing
	Although it's very unlikely that you'll master C++ in 21 days, the book itself is very well written.		
•	GTSL Software Description	Michael Kammerer	Rohde & Schwarz
	Describes the use of GTSL Libraries. Available in PDF file format after installation of GTSL in "documentation" sub folder.		
•	C Primer Plus 3rd Edition	Stephen Prata	Sams
	Very well written book on C.		
•	CMU200 Universal Radio Communication Tester Operating Manual		Rohde & Schwarz
•	CMU200 Software Option GSM900/1800/1900 for CMU-B21		Rohde & Schwarz
•	Wireless Technician's Handbook	Andrew Miceli	Artech House
	Just enough to make you understand a bit of GSM Technology.		
•	MSDN		Microsoft.
•	Getting started with your plug and play GPIB Hardware		National Instruments.
	NI-488.2 User Manual for Windows		National Instruments
	The last 2 manuals are included with the GPIB adapter		

15 Additional Information

Please contact

ROHDE & SCHWARZ Support Centre Asia Pte Ltd
1 Kaki Bukit View #04-01/07 TECHVIEW"
Singapore 415941

Tel: (65) 6 8463 710

Fax: (65) 6 8460 029

Website: www.rohde-schwarz.com.sg

for comments and further suggestions.

16 Ordering information

Type of instrument		
CMU200	Radiocomtester	1100.0008.02
CMU-B21	Signalling Unit	1100.5200.02
CMU-K21	GSM900, R-GSM, E-GSM firmware	1115.6007.02
Software		
GTSL	Software Libraries	

ROHDE & SCHWARZ GmbH & Co. KG · Mühldorfstraße 15 · D-81671 München · P.O.B 80 14 69 · D-81614 München ·
Telephone +49 89 4129 -0 · Fax +49 89 4129 - 13777 · Internet: http://www.rohde-schwarz.com

This application note and the supplied programs may only be used subject to the conditions of use set forth in the download area of the Rohde & Schwarz website.