BCC Group International

Programmer's Guide

Issue 1.3
Jan 2021

[ONE/HMS PROGRAMMER'S GUIDE]

An Introduction to the ONE/HMS API for .Net and C/C++

BCC Group

TABLE OF CONTENT

1. About this Document

1.1. Purpose

This document describes how the HMS/ONE-API can be used for programming market data applications. We show elementary example programs for publishing and subscribing within an ONE/HMS environment and explain them step by step. A basic knowledge of ONE/HMS as described in the HMS Essentials is assumed.

NOTE: The abbreviations ONE and HMS are used synonymously because the product has been renamed from HMS to ONE.

1.2. Audience

ONE/HMS application programmers.

1.3. Document history

Creation

Issue	Date	Name	Status	Comments
1.0	25.11.2011	M. González Evans / Dr. J. Hansper	Final	Next review: on demand

Modification

Issue	Date	Name	Modifications
1.1	29.05.2013	Dr. J. Hansper	Added chapter about HMS C API
1.1	29.05.2013	Klaus Raithel	Corrected tabs in the contents
1.1	14.01.2013	J. Hansper	Updated because of HMS C API changes.
1.1	06.03.2020	Rittik Wystup	Updated formatting
1.2	10.11.2020	MIGE	Updated formatting
1.3	07.01.2021	Rittik Wystup	Changed title to include *ONE*

Quality Assurance

No.	Date	Name
QA1 (1.1)	03.11.2014	J. Hansper
QA2 (1.1)	03.11.2014	V. di Leonardo

1.4. Related Documents

Refer to the following BCC Group documentations for additional information:

- HMS Essentials
- HMS-API Programmer's Reference
- MECS User Guide

1.5. Contact information

BCC Group International GmbH
Member of TechQuartier
Platz der Einheit 2
60327 Frankfurt am Main - Germany

E-Mail: info@bccgi.com
Webseite: http://bccgi.com

1.6. List of figures

2. Introduction

As described in the HMS Essentials, the basic functionality and conceptual simplicity of HMS can be explained using the visualization in Fig. 1. We see a simplified scenario of market data distribution and processing. As usual, the typical *participants* in this scenario are consumers, feeds, applications and authentication mechanisms.

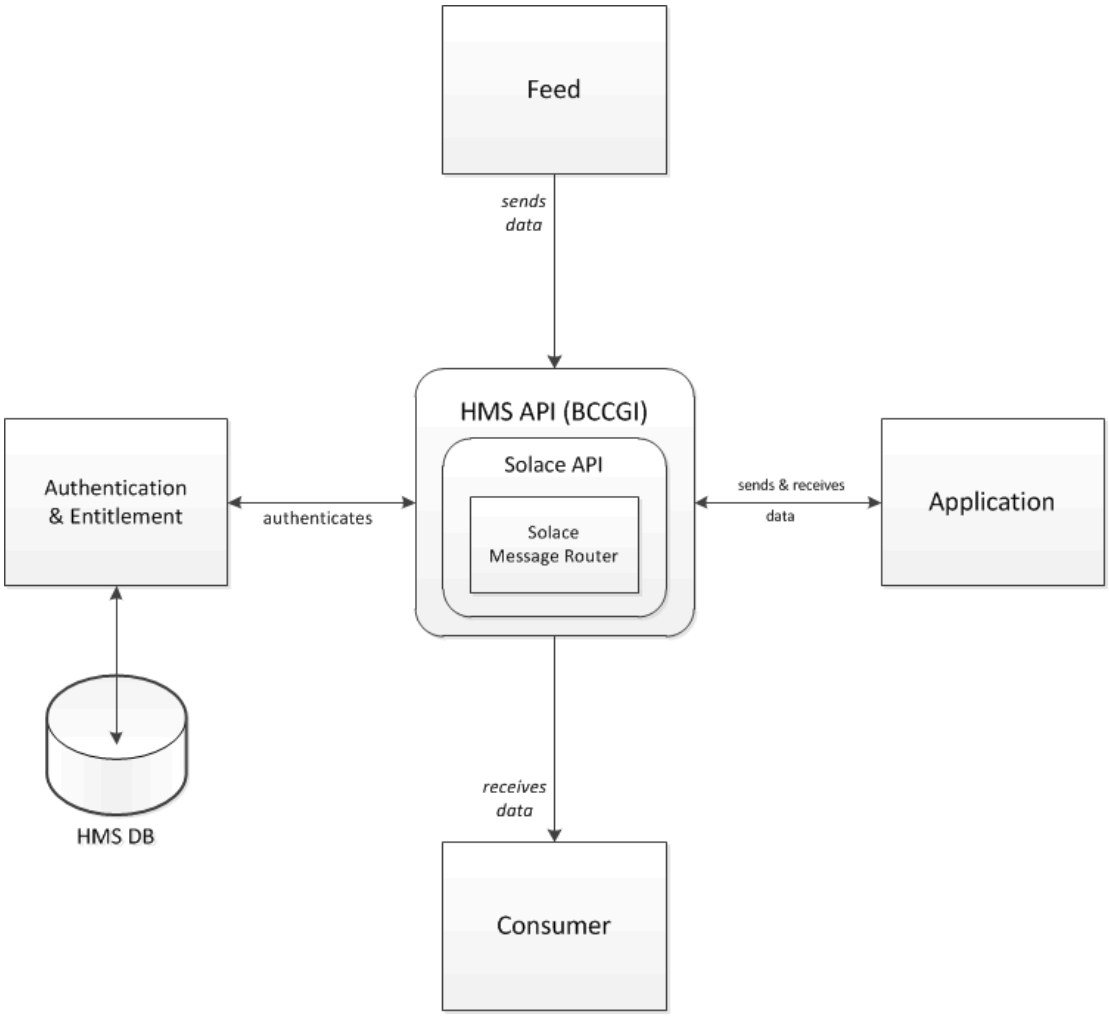

Figure 1: Idealized view of an HMS environment for market data distribution and processing

The central feature of HMS is that no participant (application, consumer, feed, etc.) is allowed to address the HMS hardware directly: *all communication occur indirectly by method calls or protocol provided by the HMS API.* This effectively isolates the complexity of the HMS hardware from the task of programming market data applications (see Fig. 2).

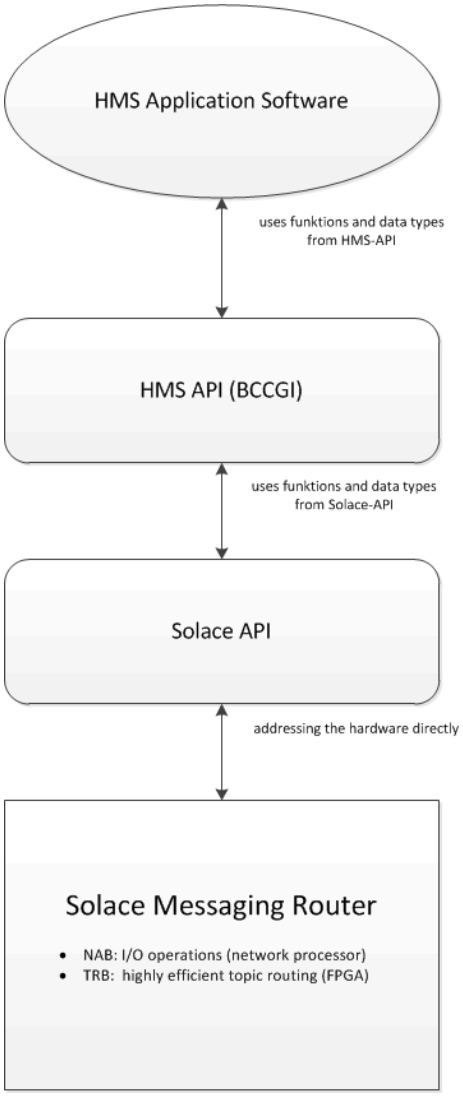

Figure 2: HMS application software using HMS hardware

3. First Steps in HMS Programming Using the .NET API

In order to give the reader an easily understandable introduction to the HMS API, we begin with two very simple cases:

1. a "consumer" subscribing to a single instrument and using a single data field
2. a "feed" publishing a single instrument into the HMS system and controlling the success by editing the consumer program for subscribing to, and displaying, the feed data.

In this section as well as throughout the whole document, we will use C# as our preferred programming language.

3.1. Simple Subscription

3.1.1. Application Settings

Before writing the program code, it is advisable to collect beforehand all the necessary information needed to run an application in an HMS environment:

- The IP address of the HMS router, e.g. "192.168.1.81", and the HMS domain of the application, e.g. "HMS-T" (If no HMS domain is given explicitly, the "default domain" is tacitly assumed). Domain and IP have to be combined (i.e. in our example: HMS-T\192.168.1.81).
- The HMS authentication data for this application (= the program to be written):
  - user name, e.g. "testuser",
  - password, e.g. "",
  - product key, e.g. "TEST_SUBS_1"

This information is required by the HMS MECS service.

- an application license token (which can be null)
- a timeout for the authentication, e.g. 1000 ms.

Additionally, the programmer has to make some choices concerning the market data:

- a provider including the corresponding ID in HMS, e.g. "Bloomberg" and "BB"
- an instrument from this provider, e.g. "DAX Index"
- a corresponding data field name for this instrument, e.g. "LAST_PRICE"

In part I of our program code all parameters above are coded in C# as editable constants:

```
////////////////////////////////////////////////////////////////////////////
//
//  PART I: application settings
//
////////////////////////////////////////////////////////////////////////////////

//
//  connection parameters
//

// HMS domain and IP address of the HMS Router
const string HMSRouterIP = "HMS-T\\192.168.1.81";

// user name and password associated with this application
const string UName = "testuser";
const string UPwd = "";

// application product key for this application
const string AppProdKey = "TEST_SUBS_1";

// license token for the application or null
const string AppLicenseToken = null;

// timeout in milliseconds for authentication of this application
const int replyTimeout = 5000;

//
//   subscription parameters
//

// provider (BB=Bloomberg)
const string DataProvider = "BB";

// instrument from the provider, e.g. DAX Index, EUR Curncy
const string Instrument = "DAX Index";

// field name of the instrument,
// e.g. BID, ASK or EUR Curncy or LAST_PRICE for DAX Index
const string FieldName = "LAST_PRICE";
```

NOTICE

It should be emphasized that the program will not work properly (i.e. will not be able to connect to the HMS system) if the authentication data have not been entered into the HMS MECS database by the HMS administrator! For further information see e.g. the section on the HMS MECS service in the HMS Essentials.

3.1.2. Connecting to the HMS System

As a next step, an initial administrative session to the HMS system has to be established. We proceed as described in the corresponding section of the HMS Essentials. These steps are programmed in part II of our example code. First, an HMS error event handler is introduced for all sorts of unhandled errors that might occur during program execution. The error handler is defined in the last part (V) of our program. Second, the authentication procedure is initiated using the *Init* call. It requires the IP address of the HMS router and opens the communication channel for authenticating the application within the HMS system. Third, the actual authentication is carried out by calling the *LogOn* procedure with the necessary authentication information, i.e. (i) user name, (ii) password, (iii) product key for the application (= this program), (iv) authentication timeout and (v) application license token. Both *Init* and *LogOn* have return values that indicate possible errors during the connection process. If no errors are reported, the application is properly authenticated within the HMS system and can start working.

```
// register the error event handler for errors that are unhandled otherwise
HmsApi.HmsErrorEvent += new HmsApi.HmsErrorEventEventHandler(OnErrorEvent);


////////////////////////////////////////////////////////////////////////////
//
//  PART II: initial authentication procedure for HMS
//
////////////////////////////////////////////////////////////////////////////

// initialize communication channel for authenticating this application
// within HMS
RetVal rv = HmsApi.Init(HMSRouterIP);

// possible error check:  if (rv != RetVal.ok) { ... }

// start authentication for accessing the HMS system
MsgType mgt = HmsApi.LogOn(UName, UPwd, AppProdKey, replyTimeout, AppLicenseToken);

// possible error check: if (mgt != MsgType.logOnOK) { ... }
```

3.1.3. Subscribing to an Instrument

Since we have now access permission to the HMS system (according to the entitlements given during the authentication process) we can start working with market data. We choose a simple subscription to a specific instrument in order to illustrate the general principle. This is shown in part III of our example program.

First of all, we take notice of the fact that market data are distributed in an asynchronous mode, i.e. it is impossible to predict when precisely the next data update will be delivered by the provider. Therefore, arriving data are indicated by data events for which a corresponding event handler has to be registered. Otherwise, the data could not be noticed and processed at all.

This is the reason why, as a first step, the data event handler "OnDataEvent" (defined in part V) is registered in the corresponding delegate before the actual subscription occurs.

In the next step we turn to the main purpose of this program, i.e. subscribing to an instrument (already chosen in part I: (i) provider "BB" (Bloomberg), (ii) instrument "Dax Index" and (iii) field "LAST_PRICE"). The subscription call is quite simple and needs the synchronization mode as an additional argument in addition to the other ones.

Finally, part III is terminated by some output statements.

```
////////////////////////////////////////////////////////////////////////////////
//
//  PART III: doing a simple subscription in HMS
//
////////////////////////////////////////////////////////////////////////////////

// register the HMS data event handler for receiving subscribed data
HmsApi.HmsDataEvent  += new HmsApi.HmsDataEventEventHandler(OnDataEvent);

// execute a simple subscription call:
rv = HmsApi.Subscribe(DataProvider, Instrument, SyncMode.async, FieldName);

// possible error check: if (rv != RetVal.ok) { ... }

// subscription infos
Console.WriteLine();
Console.WriteLine("Subscription sucessful!");
Console.WriteLine("Displaying subscribed data for\n");
Console.WriteLine("  Provider:   " + DataProvider);
Console.WriteLine("  Instrument: " + Instrument);
Console.WriteLine("  Field name: " + FieldName);
Console.WriteLine();
Console.WriteLine("Hit [Esc] to terminate...");
Console.WriteLine();
```

3.1.4. Processing the Subscribed Data

After a successful subscription we face the problem of how the subscribed data can be handled continually upon arrival until the program is told to stop.

In part IV of the example program this is solved by an infinite loop over a sleep call (for limiting the processor load) and a check if the ESC-key was pressed in order to exit the loop and terminate the program. Before the program ends, some clean-up operations are performed: (i) deregistering the data and error event handlers and (ii) a *LogOff* call for cancelling all subscriptions and exiting the HMS system.

```
////////////////////////////////////////////////////////////////////////////////
//
//  PART IV:
//  run an infinite loop in order to collect the subscribed data using event
//  handlers
//  until the ESCAPE button is pressed
//
////////////////////////////////////////////////////////////////////////////////

// run the loop
while (true) {
      Thread.Sleep(10);
      if (Console.KeyAvailable) {
            if (Console.ReadKey(true).Key == ConsoleKey.Escape)
                  break;
      }
}

// deregister obsolete event handlers after leaving the loop
HmsApi.HmsDataEvent  -= OnDataEvent;
HmsApi.HmsErrorEvent -= OnErrorEvent;

// log off from HMS
HmsApi.LogOff(true);

Console.WriteLine("\nEnd of Demo. Press any key to exit ...");
Console.ReadKey();
```

3.1.5. Defining Event Handlers

In the final part V of our program, the missing event handlers are defined. Both are quite trivial. The data event handler just checks for the right instrument and field name and prints them together with the associated value. The error handler merely prints the error message without any further action.

```
////////////////////////////////////////////////////////////////////////////////
//
//  PART V: define the event handlers registered in Main for receiving subscribed
//  data
//
////////////////////////////////////////////////////////////////////////////////

// event handler for data
static void OnDataEvent(DataObject Data) {

      if (Data.Instrument.Equals(Instrument)) {
            double? lastPrice = Data.getValueOf(FieldName) as double?;
            // lastPrice contains a double value or null
            if (lastPrice != null) {
               Console.WriteLine("{0}, {1} = {2:00000.00}", Instrument, FieldName,
               lastPrice);
            }
      }

}

// event handler for errors
static void OnErrorEvent(HmsErrorObject Error) {
      Console.WriteLine();
      Console.WriteLine("***** Error occurred: {0} *****", Error.Message);
      Console.WriteLine();
}
```

3.1.6. Complete C# Code Listing

After having explained all five parts of our program, we present here the listing of the whole source code at once. It should be compilable as it stands together with the necessary libraries and it should run in any HMS environment that is prepared accordingly.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using Hms;


namespace SampleSubscription1 {

class Program {

////////////////////////////////////////////////////////////////////////////////
//
//  PART I: application settings
//
////////////////////////////////////////////////////////////////////////////////

//
//  connection parameters
//

// IP address of the HMS Router
const string HMSRouterIP = "HMS-T\\192.168.1.81";

// user name and password associated with this application
const string UName = "testuser";
const string UPwd = "";

// application product key for this application
const string AppProdKey = "TEST_SUBS_1";

// license token for the application or null
const string AppLicenseToken = null;

// timeout in milliseconds for authentication of this application
const int replyTimeout = 5000;

//
//   subscription parameters
//

// provider (BB=Bloomberg)
const string DataProvider = "BB";

// instrument from the provider, e.g. DAX Index, EUR Curncy
const string Instrument = "DAX Index";

// field name of the instrument,
// e.g. BID, ASK or EUR Curncy or LAST_PRICE for DAX Index
const string FieldName = "LAST_PRICE";
```

```
static void Main(string[] args) {

      // register the error event handler for errors that are unhandled otherwise
      HmsApi.HmsErrorEvent += new HmsApi.HmsErrorEventEventHandler(OnErrorEvent);


      ////////////////////////////////////////////////////////////////////////////
      //
      //  PART II: initial authentication procedure for HMS
      //
      ////////////////////////////////////////////////////////////////////////////

      // initialize communication channel for authenticating this application
      //  within HMS
      RetVal rv = HmsApi.Init(HMSRouterIP);

      // possible error check:  if (rv != RetVal.ok) { ... }

      // start authentication for accessing the HMS system
      MsgType mgt = HmsApi.LogOn(UName, UPwd, AppProdKey, replyTimeout,
      AppLicenseToken);

      // possible error check: if (mgt != MsgType.logOnOK) { ... }


      ////////////////////////////////////////////////////////////////////////////
      //
      //  PART III: doing a simple subscription in HMS
      //
      ////////////////////////////////////////////////////////////////////////////

      // register the HMS data event handler for receiving subscribed data
      HmsApi.HmsDataEvent  += new HmsApi.HmsDataEventEventHandler(OnDataEvent);

      // execute a simple subscription call:
      rv = HmsApi.Subscribe(DataProvider, Instrument, SyncMode.async, FieldName);

      // possible error check: if (rv != RetVal.ok) { ... }

      // subscription infos
      Console.WriteLine();
      Console.WriteLine("Subscription sucessful!");
      Console.WriteLine("Displaying subscribed data for\n");
      Console.WriteLine("  Provider:   " + DataProvider);
      Console.WriteLine("  Instrument: " + Instrument);
      Console.WriteLine("  Field name: " + FieldName);
      Console.WriteLine();
      Console.WriteLine("Hit [Esc] to terminate...");
      Console.WriteLine();
```

```
        ////////////////////////////////////////////////////////////////////////////
        //
        //  PART IV:
        //  run an infinite loop in order to collect the subscribed data using event
        //  handlers
        //  until the ESCAPE button is pressed
        //
        ////////////////////////////////////////////////////////////////////////////

        // run the loop
        while (true) {
                Thread.Sleep(10);
                if (Console.KeyAvailable) {
                        if (Console.ReadKey(true).Key == ConsoleKey.Escape)
                                break;
                }
        }

        // deregister obsolete event handlers after leaving the loop
        HmsApi.HmsDataEvent -= OnDataEvent;
        HmsApi.HmsErrorEvent -= OnErrorEvent;

        // log off from HMS
        HmsApi.LogOff(true);

        Console.WriteLine("\nEnd of Demo. Press any key to exit ...");
                Console.ReadKey();

                } // Main


        ////////////////////////////////////////////////////////////////////////////
        //
        //  PART V: define the event handlers registered in Main for receiving
        //  subscribed data
        //
        ////////////////////////////////////////////////////////////////////////////

        // event handler for data
        static void OnDataEvent(DataObject Data) {

                if (Data.Instrument.Equals(Instrument)) {
                        double? lastPrice = Data.getValueOf(FieldName) as double?;
                        // lastPrice contains a double value or null
                        if (lastPrice != null) {
                Conole.WriteLine("{0}, {1} = {2:00000.00}", Instrument, FieldName,
                lastPrice);
                        }
                }

        }

        // event handler for errors
        static void OnErrorEvent(HmsErrorObject Error) {
                Console.WriteLine();
                Console.WriteLine("***** Error occurred: {0} *****", Error.Message);
                Console.WriteLine();
        }

        } // Program

} // namespace
```

3.2. Simple Publication

Any HMS API based software that only sends data into the system is called a (data) *feed* ("publications only"). All rules for feeds apply to contributions in the same way, i.e. contributions must belong to a feed registered in an HMS system.

As a simple example, we choose a test feed publishing random data, e.g. consisting of a random double precision number and a string of random length containing random characters. Both values are published together in a "random instrument".

3.2.1. Application Settings

As already pointed out in the preceding case, we have to collect beforehand the necessary information before we can run any HMS program:

- The HMS domain and the IP address of the HMS router, e.g. "HMS-T" and "192.168.1.81", in combined form, e.g. "HMS-T\192.168.1.81" (no domain means default domain)
- The authentication data for the application (= the feed program):
  - user name, e.g. "testfeed",
  - password, e.g. "",
  - (application) product key, e.g. "TST_RND_FEED"
  - (data) product key, e.g. "TST_RND_FEED_DTA"

This information is required by the HMS MECS service.

- an application license token (which can be null)
- a timeout for the authentication, e.g. 1000 ms.

Additionally, the programmer has to make some choices concerning the feed data. In our specific example, we need

- the feed ID used to identify the feed within the HMS system, e.g. "TRFD"
- the instrument ID for the feed data, e.g. "RND_INSTR"
- the corresponding data field name for the double value, e.g. "RND_DBL"

In part I of our program code all parameters above are coded in C# again as editable constants:

```
////////////////////////////////////////////////////////////////////////////////
//
//  PART I: application settings
//
////////////////////////////////////////////////////////////////////////////////

//
//  connection parameters
//

// HMS domain and IP address of the HMS Solace Messaging Router
const string HMSRouterIP = "HMS-T\\192.168.1.81";

// user name and password associated with this application
const string UName = "testfeed";
const string UPwd  = "";


// application product key for this application
```

```
const string AppProdKey = "TST_RND_FEED";

// data product key for data to be published and subscribed
const string DtaProdKey = "TST_RND_FEED_DTA";

// license token for the application or null
const string AppLicenseToken = null;

// timeout in ms for authentication of this application
const int replyTimeout = 5000;

//
//   publication parameters
//

// provider (this application as publisher/feed)
const string DataProvider = "TRFD";

// instrument to subscribe to (from this application as a provider/feed)
const string Instrument = "RND_INSTR";

// field names of published data
static String[] FieldNames = new String[]{ "RND_DBL", "RND_STR" };

// delay in ms for published data
const int pubDelay = 50;
```

3.2.2. Auxiliary Variables and Methods

Part II of our program is used to declare or define some auxiliary variables and methods for later use. First, two static auxiliary variables for random generators are declared. Second, since there is no standard method in C# for generating random strings, we have to write our own one. Finally, a simple error event handler is defined.

```
////////////////////////////////////////////////////////////////////////////
//
//  PART II: auxiliary variables and methods
//
////////////////////////////////////////////////////////////////////////////

// random generators for double and string
static Random rdm1, rdm2;

// generate random string
static string RndString() {
  // minimal and maximal length of random string
  const int minlen = 3;
  const int maxlen = 24;

  // create Char-array with randomly chosen length
  int strlen = rdm2.Next(minlen, maxlen + 1);
  Char[] rndcharr = new Char[strlen];

  // fill Char-array with random characters
  for (int i = 0; i < strlen; i++)
    rndcharr[i] = (Char) rdm2.Next(32,127);
```

```
 return new string(rndcharr);

}

// event handler for errors
static void OnErrorEvent(HmsErrorObject Error) {
  Console.WriteLine();
  Console.WriteLine("***** Error occurred: {0} *****", Error.Message);
  Console.WriteLine();
}
```

3.2.3. Connecting to the HMS System

The steps are the same as already described in section 3.1.2 and need no further comments.

```
////////////////////////////////////////////////////////////////////////////
//
//  PART III: initial authentication procedure for HMS
//
////////////////////////////////////////////////////////////////////////////

// initialize communication channel for authenticating this application within HMS
RetVal rv = HmsApi.Init(HMSRouterIP);

// possible error check:  if (rv != RetVal.ok) { ... }

// start authentication for accessing the HMS system
MsgType mgt = HmsApi.LogOn(UName, UPwd, AppProdKey, replyTimeout, AppLicenseToken);

// possible error check:  if (mgt != MsgType.LogOnOK) { ... }
```

3.2.4. Publishing Data into the HMS System

In the last part of our program data are published into the HMS system. Since we want the program to behave like a (broadcast) feed, we do this within an infinite loop (that can be exited by pressing the escape button):

First, the random data are generated using the random number generators defined earlier. Then, the data (in our case a double value and a string of variable length) are prepared for publication by storing them in the object array *Fields*. Using the parameters defined in the first part of the program, we have already all information to invoke the HMS publish call *HmsApi.Publish*. This call packs the string array *FieldNames* and the object array *Fields* into a data object which is published as instrument using the ID *Instrument*. Furthermore, the call needs the product key *DtaProdKey* and the feed ID *DataProvider* for HMS MECS and potential subscribers.

After leaving the loop, some clean-up operations are performed: (i) deregistering the error event handlers and (ii) a *LogOff* call for exiting the HMS system.

```
////////////////////////////////////////////////////////////////////////////////
//
//  PART IV: run in an infinite loop for generating and publishing data
//
////////////////////////////////////////////////////////////////////////////////

while (true) {

  // generate random data ...
  double dbl = 100.0 * rdm1.NextDouble();
  string str = RndString();

  // ... and put them in an object array for publication as instrument
  object[] Fields = new object[] { dbl, str };

  // publish the random data as data provider "DataProvider" with product key
  // "DtaProdKey" under the instrument name "Instrument"

 rv = HmsApi.Publish( DataProvider, DtaProdKey, Instrument, FieldNames, Fields,
 false );

  // control output
  Console.WriteLine("Random double: {0:00.0000}", dbl);
  Console.WriteLine("Random string: " + str);
  Console.WriteLine();

  // leave loop if [Esc] is pressed
  if (Console.KeyAvailable) {
    if (Console.ReadKey(true).Key == ConsoleKey.Escape)
      break;
  }
  // time delay for publication
  Thread.Sleep(pubDelay);
}

// deregister obsolete event handlers after leaving the loop
HmsApi.HmsErrorEvent -= OnErrorEvent;
HmsApi.HmsSystemErrorEvent -= OnErrorEvent;

// log off from HMS
HmsApi.LogOff(true);

Console.WriteLine("\nEnd of Demo. Press any key to exit ... ");
Console.ReadKey();
```

3.2.5. Complete C# Code Listing

After having explained all five parts of our program, we present here the listing of the whole source code at once. It should be compilable as it stands together with the necessary libraries and it should run in any HMS environment that is prepared accordingly.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using Hms;
namespace SampleFeed1 {
```

```
class Program {

////////////////////////////////////////////////////////////////////////////
//
//  PART I: application settings
//
////////////////////////////////////////////////////////////////////////////

//
//  connection parameters
//

// HMS domain and IP address of the HMS Solace Messaging Router
const string HMSRouterIP = "HMS-T\\192.168.1.81";

// user name and password associated with this application
const string UName = "testfeed";
const string UPwd  = "";

// application product key for this application
const string AppProdKey = "TST_RND_FEED";

// data product key for data to be published
const string DtaProdKey = "TST_RND_FEED_DTA";

// license token for the application or null
const string AppLicenseToken = null;

// timeout in ms for authentication of this application
const int replyTimeout = 5000;

//
//   publication parameters
//

// provider (this application as publisher/feed)
const string DataProvider = "TRFD";


// instrument to subscribe to (from this application as a provider/feed)
const string Instrument = "RND_INSTR";

// field names of published data
static String[] FieldNames = new String[]{ "RND_DBL", "RND_STR" };

// delay in ms for published data
const int pubDelay = 50;



////////////////////////////////////////////////////////////////////////////
//
//  PART II: auxiliary variables and methods
//
////////////////////////////////////////////////////////////////////////////

// random generators for double and string
static Random rdm1, rdm2;
```

```
// generate random string
static string RndString(){
	// minimal and maximal length of random string
	const int minlen = 3;
	const int maxlen = 24;

	// create Char-array with randomly chosen length
	int strlen = rdm2.Next(minlen, maxlen + 1);
	Char[] rndcharr = new Char[strlen];

	// fill Char-array with random characters
	for (int i = 0; i < strlen; i++)
		rndcharr[i] = (Char) rdm2.Next(32,127);

	return new string(rndcharr);
}

// event handler for errors
static void OnErrorEvent(HmsErrorObject Error) {
	Console.WriteLine();
	Console.WriteLine("***** Error occurred: {0} *****", Error.Message);
	Console.WriteLine();
}


// main routine
static void Main(string[] args) {

	// initialize random number generators
	rdm1 = new Random(); rdm2 = new Random();

	// register error event handlers
	HmsApi.HmsErrorEvent +=
		new HmsApi.HmsErrorEventEventHandler(OnErrorEvent);
	HmsApi.HmsSystemErrorEvent +=
		new HmsApi.HmsSystemErrorEventEventHandler(OnErrorEvent);



	////////////////////////////////////////////////////////////////////
	//
	//  PART III: initial authentication procedure for HMS
	//
	////////////////////////////////////////////////////////////////////

	// initialize communication channel for authenticating this
	// application within HMS
	RetVal rv = HmsApi.Init(HMSRouterIP);

	// possible error check:  if (rv != RetVal.ok) { ... }

	// start authentication for accessing the HMS system
	MsgType mgt = HmsApi.LogOn(UName, UPwd, AppProdKey, replyTimeout,
	AppLicenseToken);


	// possible error check:  if (mgt != MsgType.LogOnOK) { ... }

	Console.WriteLine("Sample Random Feed started.");
```

```
            ////////////////////////////////////////////////////////////////////
            //
            //  PART IV: run in an infinite loop for generating and publishing
            //  data
            //
            ////////////////////////////////////////////////////////////////////

            while (true) {

                  // generate random data ...
                  double dbl = 100.0 * rdm1.NextDouble();

                  string str = RndString();
                  // ... and put them in an object array for publication as
                  // instrument
                  object[] Fields = new object[] { dbl, str };


                  // publish the random data as data provider "DataProvider" with
                  // product key
                  // "DtaProdKey" under the instrument name "Instrument"
                  rv = HmsApi.Publish( DataProvider, DtaProdKey,
                  Instrument, FieldNames, Fields, false );

                  // control output
                  Console.WriteLine("Random double: {0:00.0000}", dbl);
                  Console.WriteLine("Random string: " + str);
                  Console.WriteLine();

                  // leave loop if [Esc] is pressed
                  if (Console.KeyAvailable) {
                        if (Console.ReadKey(true).Key == ConsoleKey.Escape)
                              break;
                  }
                  // time delay for publication
                  Thread.Sleep(pubDelay);
            }

            // deregister obsolete event handlers after leaving the loop
            HmsApi.HmsErrorEvent -= OnErrorEvent;
            HmsApi.HmsSystemErrorEvent -= OnErrorEvent;

            // log off from HMS
            HmsApi.LogOff(true);

            Console.WriteLine("\nEnd of Demo. Press any key to exit ... ");
            Console.ReadKey();

      } // Main

      }
}
```

3.2.6. Client Program for Feed Data

We can easily convince ourselves that the feed program works as expected: We merely have to modify the program parameters in the application settings part of our subscription program (see section 3.1). The result looks as follows:

```
////////////////////////////////////////////////////////////////////////////////
//
//  PART I: application settings
//
////////////////////////////////////////////////////////////////////////////////

//
//  connection parameters
//

// HMS domain and IP address of the HMS Router
const string HMSRouterIP = "HMS-T\\192.168.1.81";

// user name and password associated with this application
const string UName = "testfeed";
const string UPwd = "";

// application product key for this application
const string AppProdKey = "TST_RND_FEED";

// license token for the application or null
const string AppLicenseToken = null;

// timeout in milliseconds for authentication of this application
const int replyTimeout = 5000;

//
//   subscription parameters
//

// provider (BB=Bloomberg)
const string DataProvider = "TRFD";

// instrument from the provider, e.g. DAX Index, EUR Curncy
const string Instrument = "RND_INSTR";

// field name of the instrument,
// e.g. BID, ASK or EUR Curncy or LAST_PRICE for DAX Index

const string FieldName = "RND_DBL";
```

NOTICE

We emphasize that this is the only part of the program that has to be modified. The rest remains completely unchanged which is the reason why we do not show it here.

3.3. Simple Application (Publication and Subscription)

After having considered the simplest possible cases of pure subscription and publication, we now turn to a more general case of an application that both publishes and subscribes. Its main part consists of a for-loop publishing data into the HMS system that are retrieved from HMS via a corresponding subscription.

The program does not introduce any new idea that is not contained in the preceding examples. Therefore, it is not necessary to explain all its parts separately. Instead, we present the complete listing that should be self-explanatory using the knowledge from sections 3.1 and 3.2:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using Hms;


namespace SamplePubSub1 {


   class Program {

      ////////////////////////////////////////////////////////////////////////
      //
      //  PART I: application settings
      //
      ////////////////////////////////////////////////////////////////////////

      //
      //  connection settings for this application
      //

      // HMS domain and IP address of the HMS Solace Messaging Router
      const string HMSRouterIP = "HMS-T\\192.168.1.81";

      // user name and password associated with this application
      const string UName = "testuser";

      const string UPwd  = "";

      // application product key for this application
      const string AppProdKey = "TEST_PUBSUB_1";

      // data product key for data to be published and subscribed
      const string DtaProdKey = "TEST_PUBSUB_1_DATA";

      // license token for the application or null
      const string AppLicenseToken = null;

      // timeout in ms for authentication of this application
      const int replyTimeout = 5000;

      //
      //   define the subscription/publication  parameters
      //

      // provider (this application as publisher/feed)
      const string DataProvider = "TPS";

      // instrument to subscribe from the provider (this application!)
      const string Instrument = "HEARTBEAT";

      // chosen field name of the instrument (for later subscription)
      const string FieldName = "TIMESTAMP";

      // complete field names of the instrument to be published
      static String[] FieldNames = new String[]{ FieldName, "DATE" };
```

```
// number of published instruments
const int numPub = 15;

// delay in ms for published data
const int pubDelay = 30;

static void Main(string[] args) {

        //
        // register the data and error event handlers in their corresponding
        // delegates
        //

        HmsApi.HmsDataEvent  +=
               new HmsApi.HmsDataEventEventHandler(OnDataEvent);
        HmsApi.HmsErrorEvent +=
               new HmsApi.HmsErrorEventEventHandler(OnErrorEvent);
        HmsApi.HmsSystemErrorEvent +=
               new HmsApi.HmsSystemErrorEventEventHandler(OnErrorEvent);

        ////////////////////////////////////////////////////////////////////
        //
        //  PART II: initial authentication procedure for HMS
        //
        ////////////////////////////////////////////////////////////////////

        // initialize communication channel for authenticating this
        // application within HMS
        RetVal rv = HmsApi.Init(HMSRouterIP);

        // possible error check:  if (rv != RetVal.ok) { ... }

        // start authentication for accessing the HMS system

        MsgType mgt = HmsApi.LogOn(UName, UPwd, AppProdKey, replyTimeout,
        AppLicenseToken);


        // possible error check: if (mgt != MsgType.logOnOK) { ... }

        // subscription infos
        Console.WriteLine();
        Console.WriteLine("Displaying subscribed data for\n");
        Console.WriteLine("  Provider:   " + DataProvider);
        Console.WriteLine("  Instrument: " + Instrument);
        Console.WriteLine("  Field name: " + FieldName);
        Console.WriteLine();
        Thread.Sleep(500);


        ////////////////////////////////////////////////////////////////////
        //
        //  PART III: doing a simple subscription in HMS
        //
        ////////////////////////////////////////////////////////////////////

        // SUBSCRIBE CALL HERE:
        rv = HmsApi.Subscribe(DataProvider, Instrument, SyncMode.async,
        FieldName);

        // possible error check: if (rv != RetVal.ok) { ... }

        Thread.Sleep(pubDelay);

        // the application can now receive data
```

```
        ////////////////////////////////////////////////////////////////////
        //
        //  PART IV:
        //  run a loop in order to publish the data to the HMS system and
        //  receive them again as subscribed data using event handlers
        //
        ////////////////////////////////////////////////////////////////////

        // publish the data into the HMS system

        for (int i = 0; i < numPub; i++) {

                // prepare data object for publication as instrument
                double dbl_value = (double) i;
                object[] Fields  = new object[]
                        { dbl_value, DateTime.Today.ToLongDateString() };

                // PUBLISH CALL HERE:
                HmsApi.Publish( DataProvider, DtaProdKey, Instrument,
                FieldNames, Fields, false );

                Console.WriteLine(
                        "<< {0}, {1} = {2:00.00} published.",Instrument,
                        FieldName, dbl_value);


                // wait some time for next publication and arriving data
                Thread.Sleep(pubDelay);
        }

        // deregister obsolete event handlers after leaving the loop
        HmsApi.HmsDataEvent  -= OnDataEvent;
        HmsApi.HmsErrorEvent -= OnErrorEvent;
        HmsApi.HmsSystemErrorEvent -= OnErrorEvent;

        // log off from HMS
        HmsApi.LogOff(true);

        Console.WriteLine("\nEnd of Demo. Press any key to exit ... ");
        Console.ReadKey();

} // Main

////////////////////////////////////////////////////////////////////////////
//
//  PART V: define the event handlers registered in Main for receiving
//  subscribed data
//
////////////////////////////////////////////////////////////////////////////

// event handler for data
static void OnDataEvent(DataObject Data) {

        if (Data.Instrument.Equals(Instrument)) {
                double? lastValue = Data.getValueOf(FieldName) as double?;
                // lastValue contains a double value or null
                if (lastValue != null) {
                        Console.WriteLine(
                        ">> {0}, {1} = {2:00.00} received.\n", Instrument,
                        FieldName, lastValue);
                }
        }

}

// event handler for errors
```

```
        static void OnErrorEvent(HmsErrorObject Error) {
                Console.WriteLine();
                Console.WriteLine("***** Error occurred: {0} *****", Error.Message);
                Console.WriteLine();
        }

    } // Program

} // namespace
```

4. HMS Programming Using the C API

The HMS C API was designed with the goal in mind to be as similar to the HMS C#/.NET API as possible. However, in some cases major differences are inevitable due to the different structure of the programming languages C and C#.

The first major difference is the absence of event handling classes in C, so that HMS event processing is realized by means of user supplied callback functions for each event type. Analogously, the HMS C API uses the callback mechanism for logging.

A second important difference is the fact that there are no dynamic elementary data structures in C. As a consequence, buffers must be used in callback functions that have to be chosen large enough for processing all received data. The size of these buffers has to be adapted to the needs of each specific application.

Finally, there is no function overloading in C so that, for example, similar but different *Publish()* commands need unique names each.

No new aspects are introduced by the C API, so that all basic concepts explained for the .NET API retain their validity. *We remind the reader, however, that a careful planning and implementation of the HMS entitlement configuration is indispensable for writing a fully functional HMS application program.* Since all necessary explanations have been provided in detail in the preceding chapter about .NET programming, we will not repeat this discussion here.

4.1. Simple Application (Publication and Subscription) in C

We will use an application that is quite analogous to the last example of the preceding chapter: data is published to a topic that was subscribed to before. We explain the program step by step, focusing on the new aspects required by the C language. NOTICE: Details about the C implementation can be found in the *HMS C API online documentation.*

4.1.1. Application Configuration

The configuration part determines log file name and log level, user data buffer sizes for the user defined HMS event callbacks as well as further parameters for convenience. Among other C standard header files, HmsCApi.h *must* be included:

```
/*********************************
  PART I
  HMS C application configuration
 *********************************/
#include <stdio.h>
#include <string.h>
#include <time.h>

#include "HmsCApi.h"

/* use either log file or callback for logging */
#define USELOGFILE

/* HMS log file name */
const char* HmsLogFile = "C:\\Temp\\HmsCApi.log";
/* HMS log level */
HmsLogLevel_t HmsLogLevel = HMS_LOG_WARNING;

/* constants for user defined buffer sizes */
#define DataBufLength     64  /* key/value pairs for field names and values */
#define SessEvtBufLength 512  /* bytes */
#define LogBufLength     512  /* bytes */
```

```
/* number of publications */
const int npubs = 3;

/* constants for controlling screen output */
const int waitms  = 1000;
const char* nl    = "\n";
const char* hline = "--------------------------------------------------------------------------------";
```

4.1.2. Writing User Defined Event and Logging Callbacks

In the HMS C API events of type Data, Message and Error are implemented. Furthermore, a logging mechanism exists that uses standard output, a log file or a callback. Each callback has an associated buffer type and requires that a buffer of this type is created and passed to the API before initialization (for further details please consult the *HMS C API online documentation*). The following code gives examples for these callbacks:

```
/***********************************************
  PART II
  User defined callbacks required by the HMS C API
 ***********************************************/

/* HMS logging callback */
void MyLogCB(void* pVoid) {
  HmsLogInfo_t* pHmsLogInfo;
  if (pVoid) {
            /* cast to correct type */
            pHmsLogInfo = (HmsLogInfo_t*)pVoid;
            if (pHmsLogInfo->msgBuf)
                      printf( "MyLogCB: %s\n", pHmsLogInfo->msgBuf );
            else
                      puts( "MyLogCB: HmsLogInfo: Message buffer NULL!\n" );
  } else
            printf( "MyLogCB: Log info buffer pointer NULL!\n" );
}

/*
  HMS event handling callbacks,
  to be stored later in struct of type HmsCEvent_t
*/
void MyDtaCB(void* pVoid){ /* HMS data event callback */
  if (pVoid) {
            int i;
            HmsDataInfo_t* pMsgUserData;
            HmsFieldType_t HmsFieldType;

            /* cast pointer to correct type */
            pMsgUserData = (HmsDataInfo_t*) pVoid;

            printf("Number of received key/value pairs: %d\n", pMsgUserData->length);

            /* loop through received key/value pairs */
            for(i=0; i<pMsgUserData->length; i++) {

                      printf("Key: %10s , ", pMsgUserData->Keys[i]);

                      HmsFieldType = pMsgUserData->Values[i].type;
                      if (HmsFieldType==HMS_STRING)
                        printf("Value: %10s\n", pMsgUserData-<Values[i].value.string);
                      else if (HmsFieldType==HMS_INT)
                        printf("Value: %10d\n", pMsgUserData->Values[i].value.int32);
                      else if (HmsFieldType==HMS_BOOL)
                        printf("Value: %10s\n",
                               (pMsgUserData->Values[i].value.boolean ? "TRUE" : "FALSE") );
                      else if (HmsFieldType==HMS_FLOAT)
                        printf("Value: %10f\n", pMsgUserData-Values[i].value.float32);
                      else if (HmsFieldType==HMS_DOUBLE)
                        printf("Value: %10f\n", pMsgUserData->Values[i].value.float64);
                      else
                        printf("Value: output not (yet) implemented!");
            }
            puts("");
```

```
  }
}

void MyMsgCB(void* pVoid) { /* HMS message event callback */
  if (pVoid) printf( "HMS Msg: %s\n", ((HmsSessionEventInfo_t*)pVoid)->msgBuf );
}

void MyErrCB(void* pVoid) { /* HMS error event callback */
  if (pVoid) printf( "HMS Err: %s\n", ((HmsSessionEventInfo_t*)pVoid)->msgBuf );
}
```

4.1.3. HMS Connection Parameters

The definition of the connection parameters is completely analogous to the C# case and needs no further explanation:

```
/*******************************
  PART III
  HMS connection parameters in C
 *******************************/
char* initRouter = "HMS-T\\192.168.1.81";

char* HmsUser    = "hmsusr";
char* HmsPwd     = "hmspw";
char* ProductKey = "hmsappkey";

int replyTO = 1500;
char* LicenseToken = "";

char* Provider   = "CTST";
char* PubProdKey = "CTST_DATA";
char* Instrument = "TestData";

HmsBool_t useRawDta    = False;
HmsSyncMode_t SyncMode = Sync;
```

4.1.4. Preparing HMS Data Payload

In contrast to C#, C has no data type "object". In order to send field values of different types in one array, the type **HmsField_t** is used instead. The next code section shows how such a key/value pair array is prepared for use in a *Publish()* command:

```
/*********************************************
  PART IV
  Prepare data payload for HmsPublish() command
 *********************************************/

/* array for keys (data field names) */
char* names[] = { "Language", "Message", "int1", "int2", "float1", "float2", "double1",
                  "double2", "bool1", "bool2" };

/* various field values of different types */
char* svals[]           = { "Ansi C", "Hello, Hms C Api User!" };  /* string values */
int ivals[]             = { 5, 4 };                               /* integer values */
float fvals[]           = { 1.01f, 2.02f };                       /* float values */
double dvals[]          = { 1.001, 299000.0 };                    /* double values */
HmsBool_t bvals[]       = { False, True };                        /* boolean values */

/* length of array pair */
#define pairlen sizeof(names)/sizeof(char*)

/* corresponding array for data fields (to be filled later in main()) */
HmsField_t fields[pairlen];
```

4.1.5. Defining and Filling Required C Data Structures

In the fifth code section, all data structures required for successfully initializing and using the HMS C API are filled with data:

```
int main(int argc, const char* argv[]) {


  /*************************************
            PART V
            Define and fill required data structs
   *************************************/

  int i;
  HmsRetVal_t rv;
  HmsMsg_t mt;

  /* struct variable for storing all necessary HMS C API event callbacks */
  HmsCEvents_t CEvents;


  /* define buffer variables for user data */
  HmsDataInfo_t         MsgUserDataBuf;
  HmsSessionEventInfo_t SessEvtUserDataBuf;
# ifndef USELOGFILE
  HmsLogInfo_t          LogUserDataBuf;
# endif


  /* assign values of different types to the array of field values (defined previously) */
  for(i=0; i<2; i++) {
            fields[i]=createField(HMS_STRING, svals[i]);
            fields[i+2]=createField(HMS_INT, &ivals[i]);
            fields[i+4]=createField(HMS_FLOAT, &fvals[i]);
            fields[i+6]=createField(HMS_DOUBLE, &dvals[i]);
            fields[i+8]=createField(HMS_BOOL, &bvals[i]);
  }


  puts(nl);


  puts(nl); puts(hline); puts(nl);
  sleepMs(waitms);



#ifdef USELOGFILE
  /* set HMS C API log file */
  rv = HmsSetLogFile(HmsLogFile);
  /* error check */
  if ( rv != HMS_RETVAL_OK ) {
    printf("HmsSetLogFile(): ERROR: returned code : %s\n", HmsRetValToCstr(rv) );
    return rv;
  } else
    printf("HMS log file successfully set to: %s\n", HmsLogFile);
#else
  /* set HMS C API logging callback and buffer */
  LogUserDataBuf = createHmsLogInfoBuffer(LogBufLength);
  rv = HmsSetLoggingCallback( &MyLogCB, (void*) &LogUserDataBuf );
  /* error check */
  if ( rv != HMS_RETVAL_OK ) {
    printf("HmsSetLoggingCallback(): ERROR: returned code : %s\n", HmsRetValToCstr(rv));
    return rv;
  } else
    printf("HMS loggging callback successfully set.\n");
#endif



  /* set HMS C API log level */
  rv = HmsSetLogLevel(HmsLogLevel);
  /* error check */
  if ( rv != RetValOK ) {
```

```
  printf("HmsSetLogLevel(): ERROR: returned code : %s\n", HmsRetValToCstr(rv));
  return rv;
} else
  printf("HMS log level successfully set to: %2d\n", (int) HmsLogLevel);

puts(nl); puts(hline); puts(nl);
sleepMs(waitms);


/*
          set HMS event buffers
*/

/* assign user data struct for received messages */
MsgUserDataBuf = createHmsDataInfoBuffer(DataBufLength);
/* pass user defined buffer pointer to the API */
HmsSetMsgUserDataPtr((void*) &MsgUserDataBuf);

/* assign user data struct for received session events */
SessEvtUserDataBuf = createHmsSessionEventInfoBuffer(SessEvtBufLength);
/* pass user defined buffer pointer to the API */
HmsSetSessUserDataPtr((void*) &SessEvtUserDataBuf);


/*
          assign HMS C API event callbacks
*/
CEvents.OnData    = MyDtaCB;
CEvents.OnError   = MyErrCB;
CEvents.OnMessage = MyMsgCB;
```

4.1.6. Initializing the HMS C API and Authentication

Initializing the C API and user authentication follow the lines of the .NET API with one significant exception: the C command *HmsInit()* requires the pointer to an **HmsCEvents_t** struct storing all HMS event callback pointers:

```
/************************************************************
          PART VI
          Initialization and authentication procedure for HMS
 ************************************************************/

puts("Calling HmsInit:");
rv = HmsInit( initRouter, &CEvents, useRawDta );
/* error check */
if ( rv != HMS_RETVAL_OK ) {
          printf("HmsInit: ERROR: returned code : %s\n", HmsRetValToCstr(rv));
          return rv;
} else
          puts("HmsInit: OK.");
puts(nl); puts(hline); puts(nl);
sleepMs(waitms);


puts("Calling HmsLogOn:");
mt = HmsLogOn( HmsUser, HmsPwd, ProductKey, replyTO, LicenseToken, False );
/* error check */
if (mt != HMS_MSG_LOGON_OK) {
          printf("HmsLogOn: ERROR: returned code : %s\n", HmsMsgToCstr(mt));
          return rv;
} else
          puts("HmsLogOn: OK.");
puts(nl);puts(hline); puts(nl);
sleepMs(waitms);


puts("Just logged in to HMS! Will now execute HMS commands from within ANSI C!");
puts(nl);puts(hline); puts(nl);
sleepMs(waitms);
```

4.1.7. Subscribing and Publishing using the HMS C API

Subscribing and Publishing in the C API follows along the lines of the .NET API and does not need further explanation:

```
/**********************************
  PART VII
  Doing a simple subscription in HMS
 **********************************/
puts("Calling HmsSubscribe:");
rv = HmsSubscribe( Provider, Instrument, SyncMode );
/* error check */
if (rv != HMS_RETVAL_OK) {
          printf("HmsSubscribe: ERROR: returned code : %s\n", HmsRetValToCstr(rv));
          return rv;
} else
          puts("HmsSubscribe: OK.");
puts(nl);puts(hline); puts(nl);
sleepMs(waitms);


/*****************************************************************************
  PART VIII
  Run a loop in order to publish the data to the HMS system and receive them again
  as subscribed data via the data event callback
 *****************************************************************************/
for(i=0; i<npubs; i++) {
          puts("Calling HmsPublish:");
          rv = HmsPublish( Provider, PubProdKey, Instrument, pairlen, names, fields );
          /* error check */
          if (rv != HMS_RETVAL_OK) {
                    printf("HmsPublish: ERROR: returned code : %s\n", HmsRetValToCstr(rv));
                    return rv;
          }
          sleepMs(waitms/2);
          puts("HmsPublish: OK.");
          puts(nl);puts(hline); puts(nl);
}
sleepMs(waitms);
```

4.1.8. Terminating the Program Regularly

The same applies to the last code section with the only exception that buffer space allocated using the *createHmsXyBuffer()* commands should be released with the corresponding *clearHmsXyBuffer()* commands:

```
/**********************************
  PART IX
  Terminating the program regularly
 **********************************/

puts("Calling HmsUnSubscribe:");
rv = HmsUnSubscribe( Provider, Instrument, SyncMode );
/* error check */
if (rv != HMS_RETVAL_OK) {
          printf("HmsUnSubscribe: ERROR: returned code : %s\n", HmsRetValToCstr(rv));
          return rv;
} else
          puts("HmsUnSubscribe: OK.");
puts(nl);puts(hline); puts(nl);
sleepMs(waitms);


puts("Calling HmsLogOff:");
mt = HmsLogOff(True);
/* error check */
if (mt != HMS_MSG_LOGOFF_OK) {
      printf("HmsLogOff: ERROR: returned code : %s\n", HmsMsgToCstr(mt) );
       return rv;
} else
      puts("HmsLogOff: OK.");
```

```
  puts(nl);puts(hline); puts(nl);
  sleepMs(waitms);


  /* release allocated buffer memory */
  clearHmsDataInfoBuffer(&MsgUserDataBuf);
  clearHmsSessionEventInfoBuffer(&SessEvtUserDataBuf);
# ifndef USELOGFILE
  clearHmsLogInfoBuffer(&LogUserDataBuf);
# endif


  puts("Program terminated.");
  puts(nl);

  return 0;
}
```