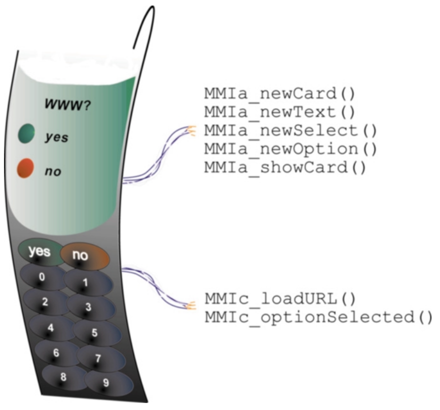
AUS WAP Browser User's Manual



Version 2.5

AU-SYSTEM

Document Reference: R109806019, version 2.5

AU-System AB

Scheelevägen 17, IDEON

SE – 223 70 Lund, SWEDEN

Telephone: +46 46 286 56 10

Fax: +46 46 286 56 20

http://www.ausys.se

Contents

1 Introduction

1.1 References

RFC2068	HTTP/1.1, http://ds.internic.net/rfc/rfc2068.txt
RFC2396	Uniform Resource Identifiers, http://ds.internic.net/rfc/rfc2396.txt
WAP-USSD	WAP over GSM USSD, version 29 April 1998, http://www.wapforum.org
WAP-WSP	WAP, WSP Specification, Version 30-April-1998 http://www.wapforum.org
WAP-WTLS	WAP WTLS, Version 5 Nov-1999 http://www.wapforum.org
X9.62	The Elliptic Curve Digital Signature Algorithm (ECDSA), ANSI X9.62 Working Draft, September 1998
P1363	Standard Specifications for Public Key Cryptography, IEEE P1363 / D1a (Draft Version 1a), February 1998. http://grouper.ieee.org/groups/1363/
X.509	The Directory – Authentication Framework, CCITT, Recommendation X.509, 1988.
X.968	
GSM0340	ETSI European Digital Cellular Telecommunication Systems (Phase 2+): Technical realisation of the Short Message Service (SMS) Point-to-Point (P) (GSM 03:40)

1.2 Abbreviations

API	Application Programming Interface
CSD	Circuit Switched Data
GUI	Graphical User Interface
MMI	Man Machine Interface
PDU	Protocol Data Unit
SMS	Short Message Service
SMSC	SMS Center, co-ordinator of SMS services
UDCP	USSD Dialog Control Protocol
UI	User Interface
URL	Uniform Resource Locator
USSD	Unstructured Supplementary Services Data
SMSC	USSD Center, co-ordinator of USSD services
WAE	Wireless Application Environment
WAP	Wireless Application Protocol
WDP	Wireless Datagram Protocol
WML	Wireless Mark-up Language
WMLS	Wireless Mark-up Language Script

2 Technical Specification

This chapter gives an overview of what is provided with the AUS WAP Browser, and a technical specification of the AUS WAP Browser.

2.1 The AUS WAP Browser package

The AUS WAP Browser is a software package that consists of:

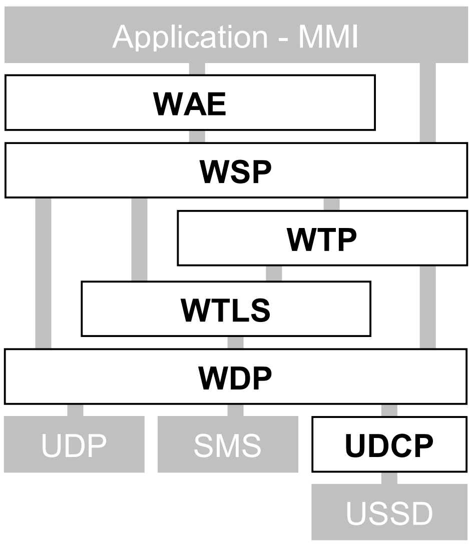
- Documentation – this manual.
- Adapter function interfaces – descriptions of all Adapter functions that must be implemented in the WAP application.
- Connector function interfaces – descriptions of all Connector functions that can be called by the WAP application.
- Source code – all sources needed to build the AUS WAP Browser.
- Stubs, i.e., empty function, for the Adapter functions.

2.2 Compatibility with WAP gateways

The AUS WAP Browser is full WAP-1.1 compliant. In order to ensure interoperability with WAP gateways, the AUS WAP Browser software is continously tested with the Ericsson WAP gateway, and with Ericsson competing WAP gateways.

2.3 Technical data

The WAP standard specifies a series of protocols. This section specifies what the AUS WAP Browser implements. Implementation specific details are given for each protocol layer in separate sections.



The picture above gives an overview of the layers, which can be included in a AUS WAP Browser configuration. The included layers are drawn in white. Read more about these layers below.

2.3.1 WAE

WAE is an application environment. It defines the functionality and dynamic behaviour for WAP applications. The WAE implementation that is used in the AUS WAP Browser supports full WML, WMLS and public WTA. It implements the cache model that HTTP/1.1 defines, and that WAP specialise.

The WML document character set can be UCS16 (Unicode), UTF-8 or ISO-8859-1. Internally is all sets converted to UCS16. Optional source code is delivered with the AUS WAP Browser, which extends the AUS WAP Browser to also read documents in the KSC 5601 character set.

Data can be sent from the AUS WAP Browser to the server with the HTTP post operation. See the following WML example:

```
<go href="www.jazz.com" method="post" accept-charset="utf8">
  <postfield name="theVariable" value="$theValue"/>
</go>
```

This source snippet can be put in a link, key or menu option. It takes the content of the variable "theValue" (represented in the UCS16 character set) and posts it to the server.

In which character set the data is posted is decided after these three rules:

- If the attribute "accept-charset" is not set, use the document charset.
- If the "accept-charset" value holds one of the character sets UCS16, UTF8 or ISO-8859-1, transcode to that particular charset.
- Otherwise, the data is transcoded to UCS16.

After the data has been transcoded, it is URL encoded and sent to the server with the content type application/x-www-form-urlencoded.

2.3.2 WSP

The WSP implementation of the AUS WAP Browser includes both the connection-less version and the connection-oriented version of WSP. WSP is specified to be a binary implementation of the Internet protocol HTTP/1.1, in a for wireless transmission optimised form. The WSP implementation may as well be used to retrieve content without going through the WAE layer. The WSP methods that are used by the AUS WAP Browser (i.e. WAE) are GET and POST.

2.3.3 WTP

The transaction layer. It adds functionality for retransmission and is used for connection-oriented transmission with WSP. The implementation supports synchronous transactions. It is also implemented to support transaction identifier verification, i.e. TID verification.

How many times retransmissions should be done and how long time it should be between each retransmission depends on the chosen bearer:

	UDP	**SMS**	**USSD**
Retransmissions	8	4	4
Interval in seconds	5	60	20

2.3.4 WDP

WDP is a transparent layer to UDP. It adds no functionality when UDP is used as bearer. The AUS WAP Browser supports also the two GSM bearers SMS and USSD. For these two bearers is additional functionality for assembling and segmentation added. The algorithm used is specified by GSM [GSM0340]. WDP has also the additional layer WCMP implemented. It serves the same purpose for the SMS and USSD bearers as ICMP does for UDP.

2.3.5 UDCP

The USSD bearer is a synchronous protocol where only one party at time is in control of the connection. UDCP implements an asynchronous connection, similar to UDP, over USSD.

2.4 Performance

This section tries to capture the overall performance of the AUR WAP Browser. RAM has been measured and execution has been timed for two different sized download operations. The measures are taken on a Pentium 233 MHz running Windows NT.

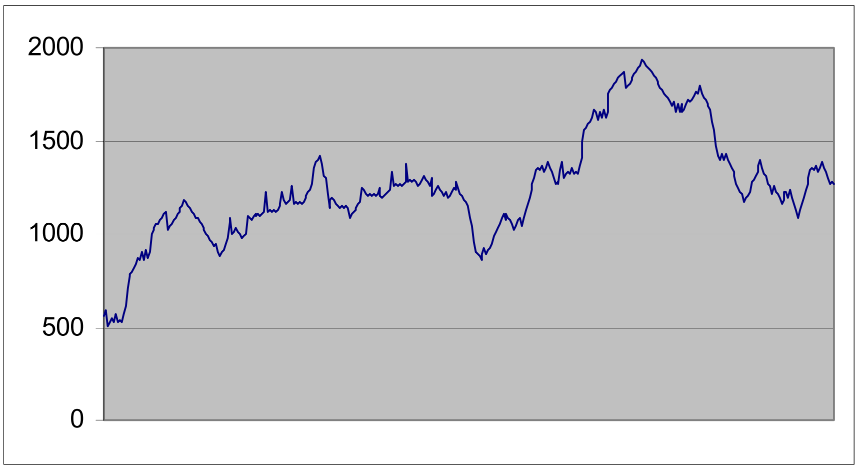
2.4.1 Downloading small data

The following data, which occupies 219 bytes (about 30 bytes when compiled), is downloaded in this example:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
                     "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
  <card>
    <p>
      Hello world!
    </p>
  </card>
</wml>
```

The AUS WAP Browser requires about 1000 microseconds to process the send operation and the final retrieve and display operations, network time excluded.

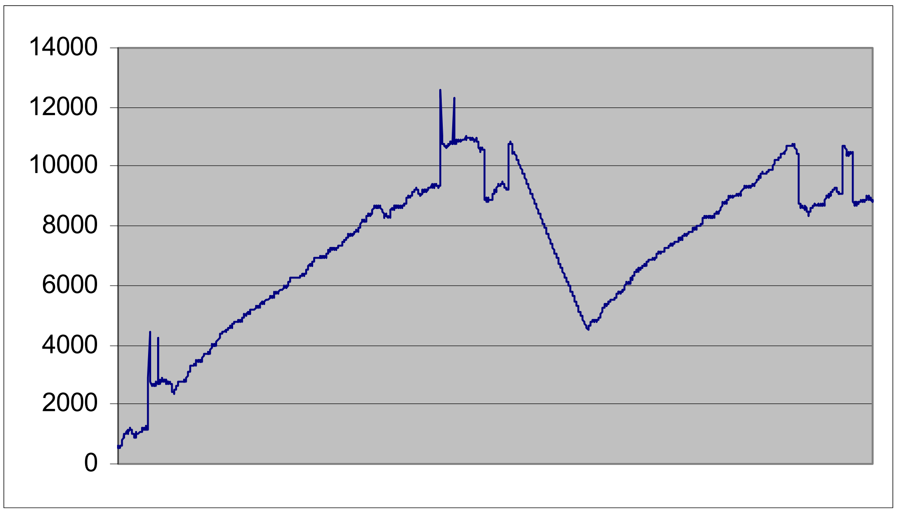
The first half of the next diagram gives the memory consumption when the AUS WAP Browser requests a small WML deck from the WAP Gateway, i.e., the time when the HTTP operation GET is propagated down the protocol stack. The diagram gives also the memory consumption when the AUS WAP Browser retrieves data from the WAP Gateway, i.e., when the data the URL (from the GET operation above) refers to is transported up through the protocol stack and finally displayed. In the second half an additional WML deck is downloaded.



The memory consumption is marginally higher than the memory consumption that the AUR WAP Browser consumes when idling and no WML deck has been downloaded previously (i.e., directly after start-up).

2.4.2 Downloading large data

This example illustrates the performance when large WML data is downloaded. The WML deck, which is 6333 bytes, is too large to be displayed in this manual. The deck, which contains one large WML card with all possible WML elements, takes about 1400 bytes when it is compiled. The deck has an image as well. This image is downloaded when the deck has been downloaded and parsed. The size of the image is 1400 bytes. The second half of the diagram shows how a equally sized WML deck is downloaded immediately after the first one. In this download the image is taken from the cache.



The AUS WAP Browser requires about 2900 microseconds to process the send operation and the final retrieve and display operations, network time excluded.

3 Host Device Requirements

3.1 Memory requirements

Memory requirements are broken into the following categories:

- ROM (Read Only Memory)
- Persistent Memory (hard disk, flash memory etc)
- RAM (Read Access Memory)

3.1.1 ROM

The memory that the executable program uses is static. It is therefore storable in ROM. If the Host Device does not support ROM, the persistent memory is used instead.

- The size of a compiled AUS WAP Browser configured for connection-less transmission is 250 Kbytes.
- The size of a compiled AUS WAP Browser configured for connection-oriented transmission is 300 Kbytes.

3.1.2 Persistent Memory

The persistent memory is used to store application data like downloaded WML decks. Persistent memory is used to optimise network usage by persistently caching content of the downloaded content. The memory requirement is left to decide by the WAP application that uses the AUS WAP Browser. Read more about this in the Memory API, in this manual.

3.1.3 RAM

The RAM is used for the WAP application heap. The memory usage is most intensive when WML content is opened and when WML scripts are executed. The AUS WAP Browser uses RAM to store a minimum of content data. Because the content that has been opened is directly related to how much RAM the AUS WAP Browser uses, only an average limit can be given. RAM usage is estimated to be 25 Kbytes. The RAM can be configured to be allocated from the system heap (with the ANSI C library function malloc), or to be allocated from an internal static storage of the AUS WAP Browser. When internal memory management is used, a maximum amount of RAM must be given at compile time (read more about this in the chapter about fine-tuning the AUS WAP Browser).

3.2 Operating system requirements

The AUS WAP Browser source code is written based on the assumption that the Host Device Operating System has support for 32-bit integers.

3.3 Task control

The WAP application that uses the AUS WAP Browser has the control of the tasks the AUS WAP Browser performs. The terminology is further explained in the next chapter. The following steps are performed in the task execution model the AUS WAP Browser implements:

- The WAP application reacts on events from the user, like button press events, etc. A AUS WAP Browser Connector function is called in such cases. It returns immediately after the function sent an event to the AUS WAP Browser.
- In order to let the AUS WAP Browser process the events, a task control function is to be called continuously. The AUS WAP Browser executes in that way the events in small steps until the last step of the event has been processed.
- Some Connector function calls cause the AUS WAP Browser to produce events to the WAP application. For instance to produce output for a new WML card. This is done by AUS WAP Browser defined Adapter function calls, which are implemented by the WAP application. The Adapter functions can be called at any time during a call of the task control function.

3.4 MMI Requirements on the Host Device

Below is a listing of a minimum set of GUI features that are needed to support WML.

- A character and graphics capable display when images are to be supported. A character capable display otherwise.
- Support for numeric and alphabetic data entry.
- Support of selecting hyper links.
- Support of selecting/deselecting one or several options in menus.
- Support of a backward navigation key.
- Support of the different WML keys (accept, prev, help, options, delete and reset). Each key can exist zero or more times in one WML card.

3.5 ANSI C requirements on the Host Device

The AUS WAP Browser makes use of several ANSI C library functions. They are not implemented in the AUS WAP Browser. They must be provided as a part of the porting work. The required functions do all, at least in the literature, sort under certain standard header files. They will do so also here. There is, however, not a requirement from the AUS WAP Browser, that the header file have these names. The proper files are during the adoption to the target OS included in the file ansilibs.h, situated in the include directory in the AUS WAP Browser source code tree.

3.5.1 stdlib.h

The ANSI C stdlib functions that are used are:

- int rand (void)
- void srand (unsigned int)

The AUS WAP Browser can be configured to handle memory management internally (read more about this in the chapter about fine-tuning the AUS WAP Browser). However, if this feature is not used the following library functions are used for memory management:

- void *malloc (size_t)
- void free(void *)

The AUS WAP Browser uses the following function:

- int abs(int)
- double strtod (const char *, char **);

3.5.2 math.h

Math functions are required if the HAS_FLOAT constant is defined. The constant is defined in the Common API. The ANSI C math functions that are used are:

- double pow (double, double)
- double sqrt (double)
- double ceil (double)
- double floor (double)

The functions should at overflow return HUGE_VAL, which is defined in math.h. Underflow and domain errors are detected by reading the errno variable (defined in errno.h).

Note that these functions use the type double but the AUS WAP Browser has only FLOAT32 (see the Common API). This means that the functions not need to be implemented with double precision.

3.5.3 errno.h

To control the result from the math functions, the AUS WAP Browser uses the constants EDOM and ERANGE, defined in the ANSI-C standard library file errno.h.

3.5.4 string.h

String functions are used for certain operations on memory blocks. WMLS uses also conversion routines from the standard library. The ANSI C string functions that are used are:

- void* memset (void* , int , size_t)
- void* memmove(void* , int , size_t)
- void* memcpy (void* , const void* , size_t)
- int memcmp (const void* , const void* , size_t)
- char* strcpy (char* , const char*)
- char* strncpy (char* , const char* , size_t)
- char* strcat (char* , const char*)
- int strcmp (const char* , const char*)
- size_t strlen (const char*)
- char* strstr (const char*, const char*)
- char* strchr (const char*, int)

3.5.5 stdio.h

I/O functions are used for certain conversion operations in WMLS library functions. The ANSI C I/O functions that is used is:

- int sprintf (char *s, const char *format, …)

3.5.6 setjmp.h

When the AUS WAP Browser is in a critical phase, due to a failed memory allocation, execution is rolled back to the state where execution were started in the AUS WAP Browser (CLNTc_run). The functions in use are:

- int setjmp (jmp_buf env)
- void longjmp (jmp_buf env, int value)

If these functions not are included in the standard libraries for a certain OS, they can be omitted by removing the C pre-compiler constant HAS_SETJMP, which is defined in target.h. The drawback is that the AUS WAP Browser allocates 1 Kbytes of RAM as a rescue buffer.

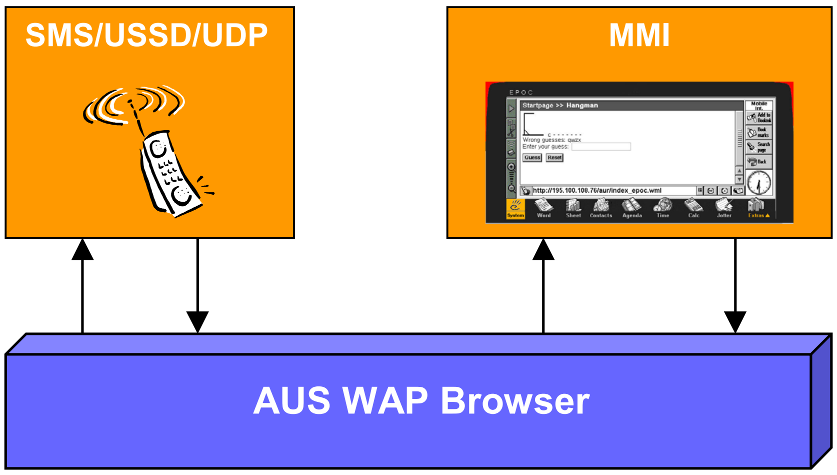
3.5.7 stdarg.h

The CLNTa_log function is defined to take an arbitrary number of arguments, and a format string that tells what arguments that comes. The ANSI C functionality required to implement that is defined in stdarg.h.

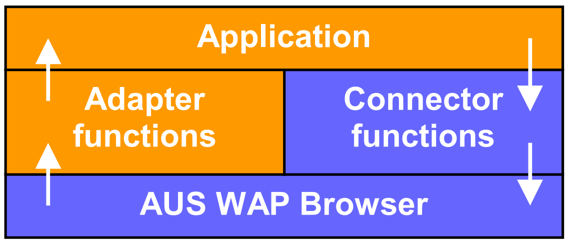
The CLNTa_log function calls are only included in the AUS WAP Browser if the compiler switch LOG_EXTERNAL is set in the makefiles that builds the AUS WAP Browser. The Adapter function should not be included in the WAP application code if the switch is not set. In this way stdarg.h will only be required when logging is required.

4 Overview

This section presents an overview of a WAP application that uses the AUS WAP Browser. This section is included to precisely define the technical terminology used throughout this manual, and the design artefacts that the AUS WAP Browser is build upon.



As the AUS WAP Browser is built to be reusable across all terminals, regardless of hardware and RTOS, a series of Connector and Adapter functions have been defined to provide a method of using the AUS WAP Browser in a WAP application for a specific terminal.

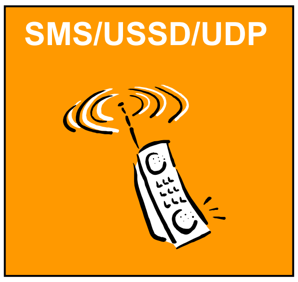


All communication from the WAP application to the AUS WAP Browser goes through Connector functions. On the reverse, all communication from the AUS WAP Browser to the WAP application passes through Adapter functions.

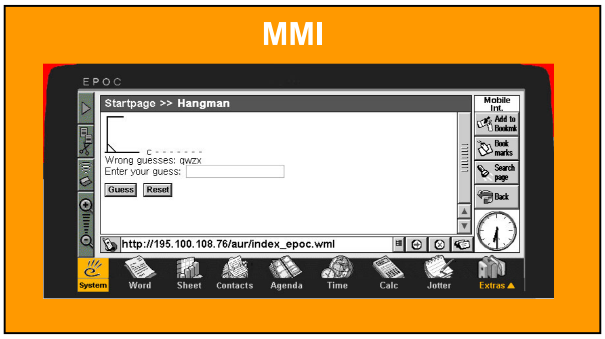
The colour scheme used in the illustrations goes through all illustrations in this manual. Orange means application implementation, i.e. device dependent implementation for a specific terminal. Blue means AUS WAP Browser implementation, i.e. implementation provided with the AUS WAP Browser product.

4.1 The WAP Application

The WAP application comprises an implementation of a WAP browser that makes use of the AUS WAP Browser.



The WAP application provides the AUS WAP Browser with access to the supported bearers (SMS, USSD or UDP). The bearers are accessed differently on different terminals. It may be in form of function calls to the operating system. It can also be signalling with a process or thread which controls the bearer. Either the way, glue has to be implemented between the AUS WAP Browser and the bearers.

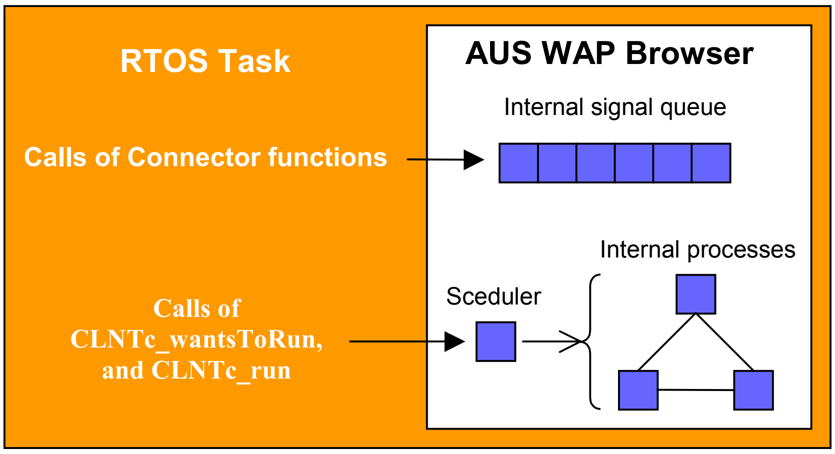


The WAP application implements the graphical user interface. Depending on the terminal, this application may appear in many shapes. The only requirement for it is that it should be able to display the WML graphical elements and take input from the user. Graphical elements are for instance text, images, menus and input fields. User input elements are menus and input fields. Most often take often text and images user input (when they are defined as Internet links).

The AUS WAP Browser does not make any assumptions on how the display is dimensioned and what capabilities the graphics software of the terminal have. This gives the WAP application developers freedom to implement the user interface for the browser in accordance with the user interface of other applications on the terminal. User interface guidelines and company policies can be followed.

4.2 The Connector functions

The Connector functions provide the WAP application with an interface to functionality of the AUS WAP Browser. The Connector functions are device independent implementations and come as part of the AUS WAP Browser. No Connector function returns data. If the result of a Connector function call shall be returned, an Adapter function provides it.



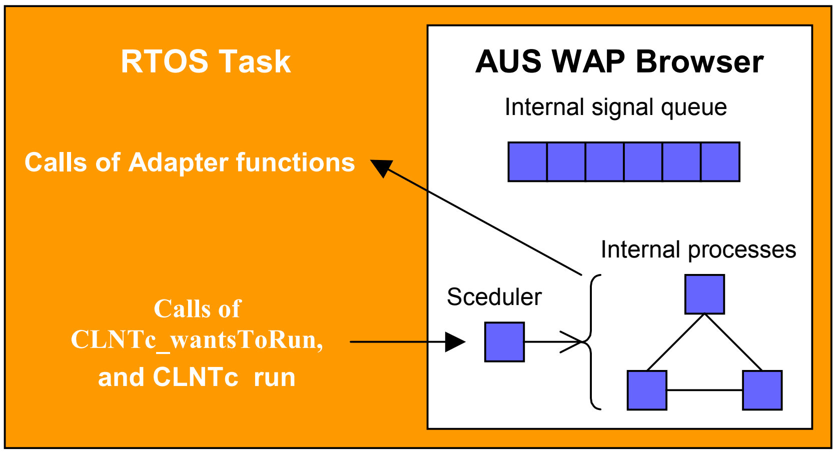
The task a certain Connector function has is not performed when the function is called. The call results only in an internal signal that is put in an internal signal queue. To process the signals, another function is used:

Two Connector functions, CLNTc_run and CLNTc_wantsToRun used in combination, provides the AUS WAP Browser with the CPU time necessary to process the internal signals in the internal signal queue. An internal scheduler keeps track of signals in the queue and executes internal processes that perform the actual task of the Connector functions when CLNTc_run is called. If there are no signals in the internal signal queue, there is nothing to do. CLNTc_wantsToRun checks that.

All Connector functions, including CLNTc_run and CLNTc_wantsToRun, are to be called from the same RTOS task in order to avoid that several RTOS tasks accesses the internal signal queue at the same time. Read more about this in the Client API.

4.3 The Adapter functions

To use AUS WAP Browser, Adapter functions must be implemented. Adapter functions are functions that are used by the AUS WAP Browser when communicating with the WAP application. Adapter functions shall be translated into application specific functionality.

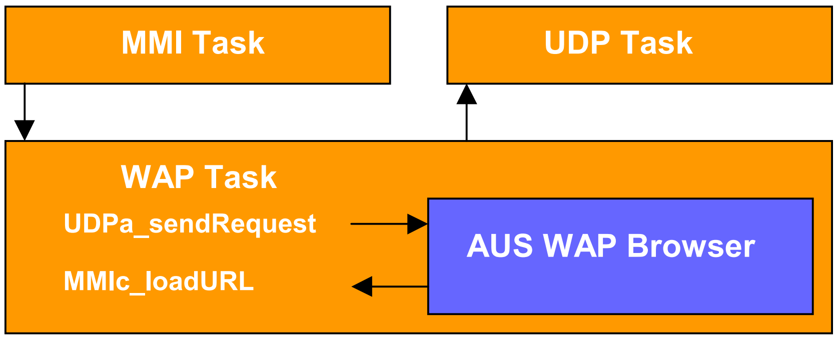


The AUS WAP Browser executes on control by a task in the WAP application (with CLNTc_run, as showed in the picture above). An Adapter function is only called when the function CLNTc_run is called. A few Adapter functions return data from the WAP application. The data retrieval time is, in these cases, assumed to be deterministic and not very long. Blocking the AUS WAP Browser (the call of CLNTc_run does not return) is regarded as acceptable in such cases. In all other cases are the data that shall be returned to the AUS WAP Browser provided through a Connector function call.

The adapter function implementations must be designed in a manner so that data belonging to other RTOS tasks (a MMI task, for instance) not is accessed directly from the adapter function. Implementing each Adapter function to send a signal to the receiving RTOS task does this. However, if the data belongs to the same RTOS task as from which CLNTc_run is called, the data can of course be accessed directly.

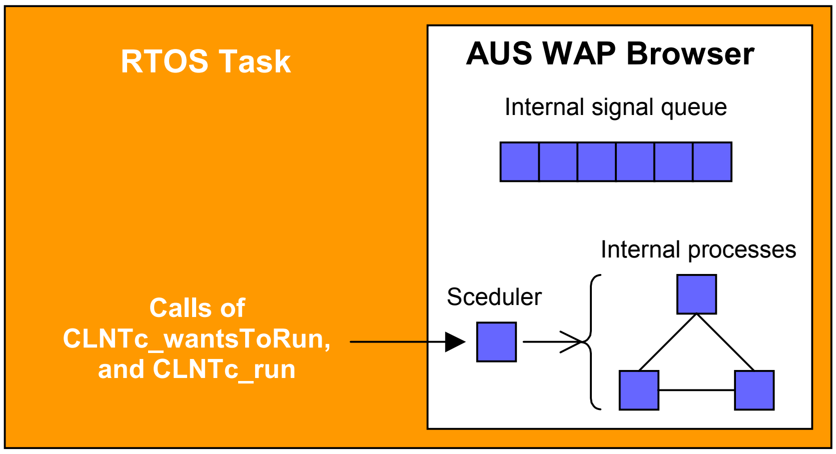
4.4 The AUS WAP Browser

The AUS WAP Browser is to be considered as a black box. It takes input from the WAP application. It responds and produces output to the WAP application in form of display instructions and instructions to send data over one, or several of the supported bearers.



The picture above gives one possible way to design RTOS tasks for the AUS WAP Browser, the display functionality and the bearer functionality.

The WAP application runs the AUS WAP Browser and in that way are events, initiated by Connector function calls, processed. Typical events from the WAP application are "Previous button pressed", "Link pressed" and "URL entered". Events from a service provider could be “received SMS”. The result of the event is for instance a set of draw instructions, sent back to the WAP application through Adapter function calls.



The AUS WAP Browser is implemented as a multi-process system. The processes can be regarded as a thread implementation in one RTOS task. It contains its own scheduler, signal queue and processes.

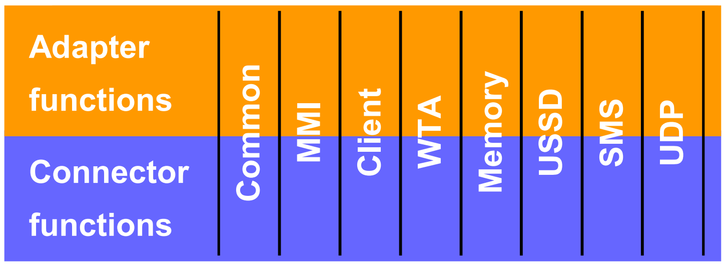
The system is signal driven. This means that if there are no signals in the signal queue, no process will be run by the scheduler. When there are signals, the scheduler processes them in FIFO order.

Each signal is aimed for a process, when the process is in a certain state. If the process is not ready, the signal is queued again. When the process is in the right state (it is ready) the scheduler runs a entire state of that process. Several signals might be sent during that state. The are put in the signal queue for later processing.

Nor does the signals or processes have different priorities. The only thing that counts is the order they are put in the signal queue.

4.5 AUS WAP Browser API

The Adapter and Connector functions of the AUS WAP Browser are logically divided into several API. Each one of the API, are described from a functional and a dynamical perspective in separate chapters, at the end of this manual.



All functions are preceded by a prefix identifying the API it belongs to. The prefix has either the letter 'a' or 'c' appended to indicate if the function is an Adapter or a Connector function. The headings for Connector and Adapter functions are in the manual highlighted without the prefix, for the sake of readability. The colour scheme is red and blue. See the examples below:

Connector function heading

Adapter function heading

The Common API contains only type definitions. No Connector or Adapter functions are defined here.

The MMI API contains all Connector and Adapter function in order to control the user interface of the WAP application.

The Client API contains the AUS WAP Browser kernel functionality. It contains also several utility Connector and Adapter functions like opening local files.

The WTA API contains an interface to telephony functionality, like setting up a call.

The Memory API is an interface to optional cache memory that the WAP application can provide the AUS WAP Browser.

The USSD, SMS and UDP API contain interfaces to the bearers on the terminal.

5 Building a WAP application

This chapter aims to guide the reader through the different API of the AUS WAP Browser. It explains, step by step, which Adapter functions to implement, and in which order.

The AUS WAP Browser is delivered with a simple test program. The program has all Adapter function implemented. Most Adapter functions contain a call to a log function (CLNTa_log). The Adapter functions for the log, timers and bearers have rudimentary implementation, in order to being able to run the program, which does a basic test of the AUS WAP Browser. The developers of the WAP application can make use of these functions and expand or re-implement them later.

5.1 Initialise, start and terminate

This section aims to guide the reader through what functionality in the WAP application that needs to be implemented in order to initialise, start and later on also terminate, the AUS WAP Browser.

WAP Browser
Menu
Select

A main view can be displayed when the WAP application is started. The example above displays a text and two keys, one to enter a menu with and one to make selections with. The keys do not have to be implemented in this phase, they are discussed in the next section.

At this step, implementations are required of the following adapter functions:

Adapter function	**Basic implementation**
MEMa_readCache	The basic implementation uses dynamic memory
MEMa_writeCache	The basic implementation uses dynamic memory
MEMa_cachePrepared	The basic implementation calls log function
CLNTa_terminated	The basic implementation calls log function
CLNTa_setTimer	The basic implementation uses native OS functionality
CLNTa_resetTimer	The basic implementation uses native OS functionality

CLNTa_currentTime	The basic implementation uses the ANSI C function time().
CLNTa_error	The basic implementation uses CLNTa_log
CLNTa_log	The basic implementation uses a native debug printing facility. Should be used if the compiler flag LOG_EXTERNAL is set.

The Adapter functions have all basic implementations that can be used in the beginning by the WAP application developers. However, real implementations must be considered early in the implementation in order to have them tailor made for the WAP application in matter.

5.1.1 Initialising the AUS WAP Browser

After the WAP application has been started, the AUS WAP Browser must be started as well. This is done in a well-defined way:

```
CLNTc_start();
```

This call initialises the AUS WAP Browser. The initialisation task is, however, not finished yet. There is still an optional cache and dynamic configuration variables to take care of. The corresponding terminate function (CLNTc_terminate) needs to be called when the WAP application later is terminated.

5.1.2 Initialising the cache

A cache of downloaded files may be used in order to increase the overall performance when running WML applications. The WAP application must initialise the cache memory first. The memory area can be a file, as well as a RAM or flash memory area. When the memory area have been initialised, the AUS WAP Browser needs to know the size of it. This is done with this function:

```
MEMc_initCache(sizeOfCache, restoredSize);
```

sizeOfCache is assigned the amount of available cache memory in bytes, and restoredSize is assigned the actual amount of bytes read into the cache memory.

The file AUS WAP Browser/develop/aapimem.c provides example code for a simple cache in RAM.

5.1.3 Initialising the view

Several WAP applications may use the AUS WAP Browser. In order to identify each application, the AUS WAP Browser identifies them through views. The WAP application opens a view by calling this function:

```
MMIc_openView(1, WML_UA);
```

When the WAP application has opened a view, an id that identifies it is given to the AUS WAP Browser. This view id will then be referred in all calls between the WAP application and the AUS WAP Browser. The constant WML_UA is used since this is a WML browser.

The corresponding close function needs not to be called when the WAP application terminates.

5.1.4 Initialising the application environment (WAE)

As a last step, in the initialising process, the WAP application provides the AUS WAP Browser with some essential parameters. This issue is described in the Client API. UDP/IP, which is used in this build example, requires these calls:

```
CLNTc_setIntConfig (1, configBEARER, 0);
CLNTc_setStrConfig (1, configUDP_IP_GW,
        "\x00\x00\x00\x00", 4);
CLNTc_setStrConfig (1, configAUTH_PASS_GW, "", 0);
CLNTc_setStrConfig (1, configAUTH_ID_GW, "", 0);
```

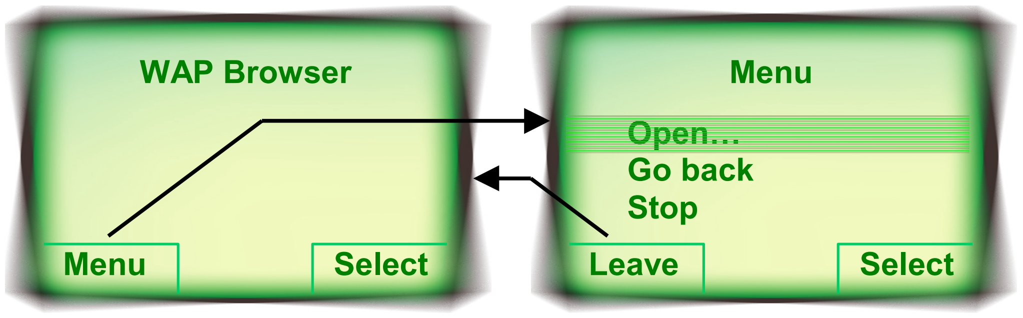
The application environment of the AUS WAP Browser needs also general information about how to handle the downloaded content, how to handle images, etc. The following variables have to be set to appropriate values:

```
CLNTc_setIntConfig (1, configHISTORY_SIZE, 10);
CLNTc_setIntConfig (1, configTIMEOUT, 30);
CLNTc_setIntConfig (1, configCACHE_AGE, 30);
CLNTc_setIntConfig (1, configCACHE_MODE, 2);
CLNTc_setIntConfig (1, configDISPLAY_IMAGES, FALSE);
CLNTc_setIntConfig (1, configUPDATE_IMAGES, FALSE);
CLNTc_setIntConfig (1, configQ_OTA, FALSE);
CLNTc_setStrConfig (1, configWSP_Language, "", 0);
```

All configuration variables are set for the view that was opened with MMIc_openView. When all configuration variables are set, the AUS WAP Browser is told so by the following call:

```
CLNTc_initialised();
```

5.2 Interactive elements



This build step aims to produce the three mandatory control-elements that must be present in a WAP application. There is a backward navigation key. There must also be a stop key. An input field to enter an URL is the third element.

As an example, the functionality discussed in this section can be controlled with a menu that is opened when the Menu key is pressed.

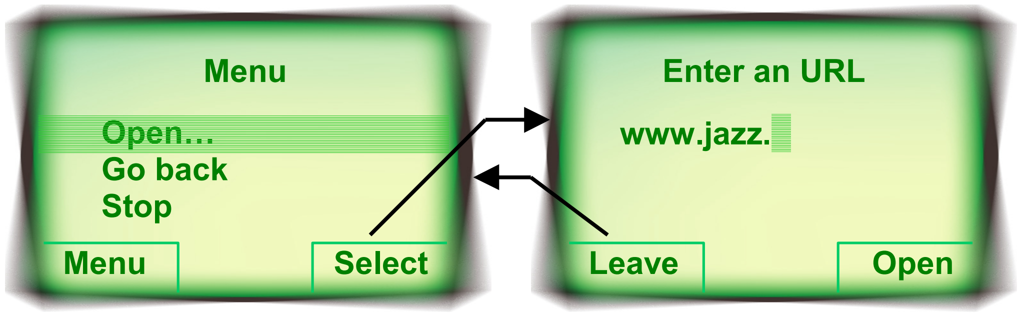
At this step implementation is required of the following adapter function:

Adapter function	**Basic implementation**
MMIa_wait	The basic implementation calls CLNTa_log

The MMIa_wait function is called from the AUS WAP Browser when it is i a critical phase. During that time shall no connector functions but those defined in this section be able to call. There is one connector function for each WAP application graphical user interface construct:

MMIc_loadURL
MMIc_stop
MMIc_goBack
MMIc_refresh

The input field to enter an URL in can be more or less complex. The example below displays the text input area in a separate view, which is entered when the Open option in the menu is chosen.



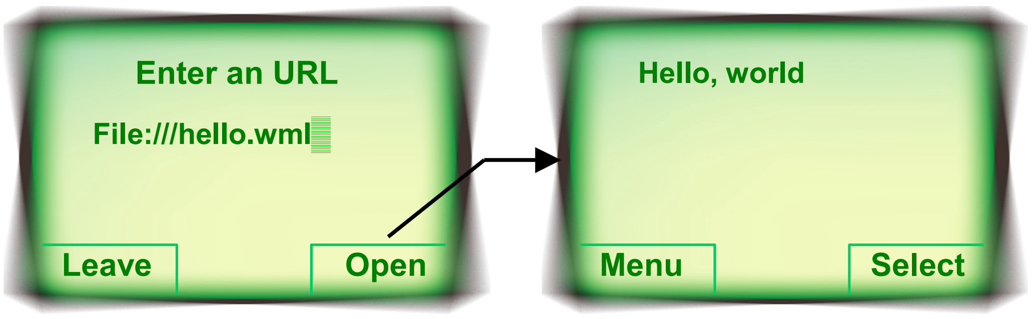
The stop and "go back" buttons in the example above are simple user interface constructs. The WAP application is designed to handle any of these events also when the AUS WAP Browser currently gives draw instructions (like MMIa_newText, which is discussed later). This means that the Menu key should be accessible, always.

5.3 Hello world

After the initialisation phase has been implemented and run and after the necessary user interface constructs are on place, implementation for a first test run can be made. A real UDP adaptation can start at any time during the WAP application development. The WML source for the “Hello, world” application is read from a local file. The “Hello, world” application has been compiled from this WML source:

```
<wml>
  <card>
    <p>
      Hello, world!
    </p>
  </card>
</wml>
```

It is the compiled byte code that is read. This is indeed a traditional start. A basic framework for the graphics is built during this phase. As an example, the following picture illustrates how the WML deck is opened and finally displayed.



At this step, implementations are required of the following adapter functions:

Adapter function	Basic implementation
CLNTa_getFile	The basic implementation calls CLNTa_log
MMIa_newCard	The basic implementation calls CLNTa_log
MMIa_newParagraph	The basic implementation calls CLNTa_log
MMIa_newText	The basic implementation calls CLNTa_log
MMIa_showCard	The basic implementation calls CLNTa_log

5.3.1 Open the WML source

When the test run is to be made, the function for entering URL strings has to be called. The function is to be called like this:

```
MMIc_loadURL (1, "file:///hello.wml", FALSE);
```

This function takes an URI string of a format as specified in [RFC2396]. However, in this case is the file scheme used. This means that this function is called:

```
CLNTa_getFile (1, "file:///hello.wml");
```

The "Hello, world" WML application is to be responded:

```
CLNTc_file (1, …, …, "application/vnd.wap.wmlc");
```

5.3.2 Display the WML source

At this point, the PDU is parsed into an internal WML deck structure in the AUS WAP Browser. The card is about to be displayed. The AUS WAP Browser will, based on the card content, call the functions:

```
MMIa_newCard (1, NULL, FALSE, FALSE, "file:///hello.wml",
           TRUE, titles);
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode("Hello, world!"), FALSE,
           TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

5.4 Interactive WML elements

Having the "Hello world" WML application successfully displayed in the WAP application eases the implementation of the remaining mandatory WML graphical elements.

At this step implementations are required of the following adapter functions:

Adapter function	**Basic implementation**
MMIa_newBreak	The basic implementation calls CLNTa_log
MMIa_newInput	The basic implementation calls CLNTa_log
MMIa_getInputString	The basic implementation calls CLNTa_log
MMIa_newSelect	The basic implementation calls CLNTa_log
MMIa_newOption	The basic implementation calls CLNTa_log
MMIa_newKey	The basic implementation calls CLNTa_log

5.4.1 Paragraphs and text

A WML application, which displays text samples, is opened with:

```
MMIc_loadURL (1, "file:///text.wml", FALSE);
```

This results in the following calls:

```
CLNTa_getFile (1, "file:///text.wml");
CLNTc_file (1, …, …, "application/vnd.wap.wmlc");
```

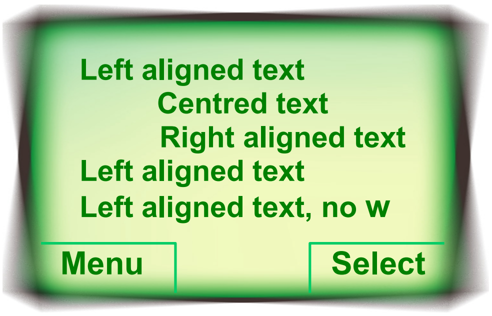
The WML source for this sample has this content:

```
<wml>
  <card>
    <p>Left aligned text</p>
    <p align="center">Centred text</p>
    <p align="right">Right aligned text</p>
    <p align="left">Left aligned text</p>
    <p mode="nowrap">Left aligned text, no wrapping is to be performed
    on this very long line.<br/>This is a new line.</p>
  </card>
</wml>
```

A series with MMIa_newParagraph calls, MMIa_newText calls and one MMIa_newBreak call will occur when this URL is loaded.

```
MMIa_newCard (1, NULL, FALSE, FALSE, "file:///text.wml",
             TRUE, titles);
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode("Left aligned text"), FALSE,
             TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_newParagraph (1, ALIGN_CENTER);
MMIa_newText (1, 0, Unicode("Centred text"), FALSE,
             TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_newParagraph (1, ALIGN_RIGHT);
MMIa_newText (1, 0, Unicode("Right aligned text"), FALSE,
             TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode("Left aligned text"), FALSE,
             TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode("Left aligned text, no
             wrapping is to be performed on this very long
             line."), FALSE, TXT_NORMAL);
MMIa_newBreak (1);
MMIa_newText (1, 0, Unicode("This is a new line."), FALSE,
             TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

This WML card could for instance be displayed in the view as the following picture illustrates:



Left aligned text
Centred text
Right aligned text
Left aligned text
Left aligned text, no w

How the non-wrapped text line is to be presented to the user depends on what capabilities the display software has. If the text can be scrolled sidewise, that is preferred. If not, the line must be wrapped to the next line.

5.4.2 Input field

A WML application, which displays an input field sample, is opened with:

```
MMIc_loadURL (1, "file:///input.wml", FALSE);
```

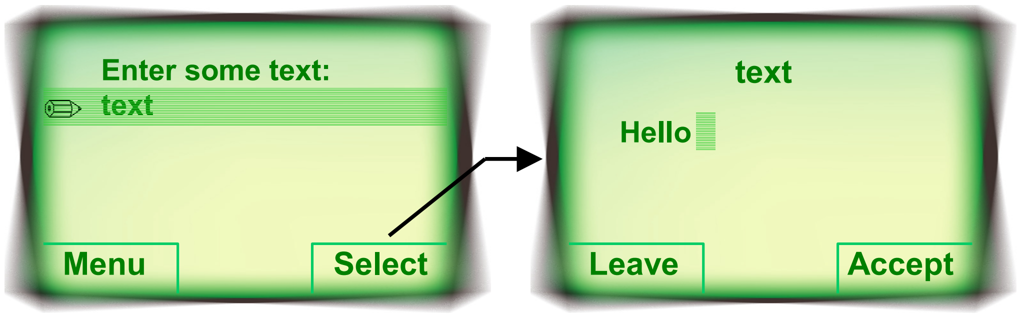
This results in the following calls:

```
CLNTa_getFile (1, "file:///input.wml");
CLNTc_file (1, …, …, "application/vnd.wap.wmlc");
```

The WML source for this sample has this content:

```
<wml>
  <card ontimer="#next">
    <p>
      Enter some text:
      <input title="text" name="var"/>
    </p>
    <timer value="600"/>
  </card>
  <card id="next">
    <p>
      The text entered: $(var)
    </p>
  </card>
</wml>
```

It may be displayed as follows:



In the example above the text is not entered directly in the WML card view but in a dedicated text input view.

A MMIa_newInput call will occur when this URL is loaded.

```
MMIa_newCard (1, NULL, FALSE, FALSE, "file:///input.wml",
          TRUE, titles);
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode("Enter some text:"), FALSE,
          TXT_NORMAL);
MMIa_newInput (1, 1, Unicode("text"), NULL, NULL, FALSE,
          TRUE, NULL, 0, 0, 0);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

After 60 seconds is card number two loaded. A string should, during card one is displayed, be entered in the input field. The entered string is then displayed in card two. Before card two is loaded, The AUS WAP Browser requests the input string from the WAP application input field:

```
MMIa_getInputString(1, 1);
```

This call should be replied with:

```
MMIc_inputString(1, 1, Unicode("string");
```

The sequence that now follows is:

```
MMIa_newCard (1, NULL, FALSE, FALSE,
          "file:///input.wml#next", TRUE, titles);
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode("The text entered: string"),
          FALSE, TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

The *string*, in the MMIa_newText call, is substituted with the entered string.

5.4.3 Selection menu

The first WML application, which displays a single choice menu sample, is opened with:

```
MMIc_loadURL (1, "file:///select1.wml", FALSE);
```

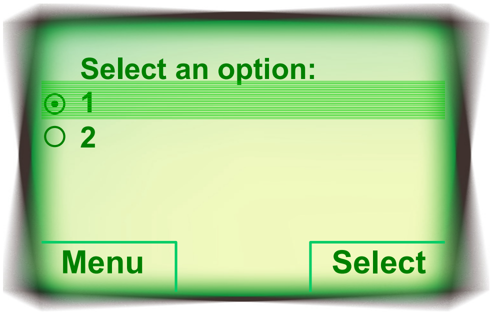
This results in the following calls:

```
CLNTa_getFile (1, "file:///select1.wml");
CLNTc_file (1, …, …, "application/vnd.wap.wmlc");
```

The WML source for this sample has this content:

```
<wml>
  <card ontimer="#next">
    <p>
      Select an option:
      <select name="select1">
        <option value="1">1</option>
        <option value="2">2</option>
      </select>
    </p>
    <timer value="600"/>
  </card>
  <card id="next">
    <p>
      Menu choice: $(select1)
    </p>
  </card>
</wml>
```

The first WML card could be displayed like this:


Select an option:

◉ 1

○ 2

In the example above, the menu option are displayed directly in the WML card view and not in a dedicated view for menus. The first option in this picture has been highlighted and then selected. This is illustrated with radio buttons.

MMIa_newSelect and MMIa_newOption calls will occur when the URL select1.wml is loaded.

```
MMIa_newCard (1, NULL, FALSE, FALSE,
             "file:///select1.wml", TRUE, titles);
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode("Select an option:"), FALSE,
             TXT_NORMAL);
```

```
MMIa_newSelect (1, NULL, FALSE, 0);
MMIa_newOption (1, 1, Unicode(“1”), NULL, TRUE);
MMIa_newOption (1, 2, Unicode(“2”), NULL, FALSE);
MMIa_closeSelect (1);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

After 60 seconds is card number two loaded. A selection should, during card one is displayed, be made in the menu. Example of resulting function call:

```
MMIc_optionSelected (1, 2);
```

The selection value is then displayed in card two. The sequence that now follows assumes that option 2 were chosen:

```
MMIa_newCard (1, NULL, FALSE, FALSE,
           “file:///select1.wml#next”, TRUE, titles);
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode(“Menu choice: 2”), FALSE,
           TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

The second WML application, which displays a multiple choice menu sample, is opened with:

```
MMIc_loadURL (1, “file:///select2.wml”, FALSE);
```

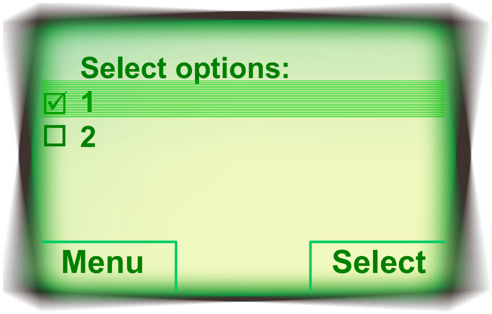
Which results in the following calls:

```
CLNTa_getFile (1, “file:///select2.wml”);
CLNTc_file (1, …, …, “application/vnd.wap.wmlc”);
```

The WML source for this sample has this content:

```
<wml>
  <card ontimer=“#next”>
    <p>
      Select options:
      <select multiple=“true” name=“select2”>
        <option value=“1”>1</option>
        <option value=“2”>2</option>
      </select>
    </p>
    <timer value=“600”/>
  </card>
  <card id=“next”>
    <p>
      Menu choice: $(select2)
    </p>
  </card>
</wml>
```

The first WML card in this deck looks like the first card in the previous deck. It could be displayed like this:



As in the previous example, the menu options are displayed directly in the WML card view. The first option has been highlighted and selected, as well. Since this is a multiple-choice menu, this is illustrated with check boxes.

There is no other difference from the single choice menus than the text and that the multiSelect attribute in set to true in the input field:

```
MMIa_newText (1, 0, Unicode("Select options:"), FALSE,
             TXT_NORMAL);
MMIa_newSelect (1, NULL, TRUE, 0);
```

Depending on for which options the MMIc_optionSelected function is called, the text in card two will be Menu choice: 1, Menu choice: 2 or Menu choice: 1;2.

The third WML application, which displays a sample of a single choice menu with sub-options, is opened with:

```
MMIc_loadURL (1, "file:///select3.wml", FALSE);
```

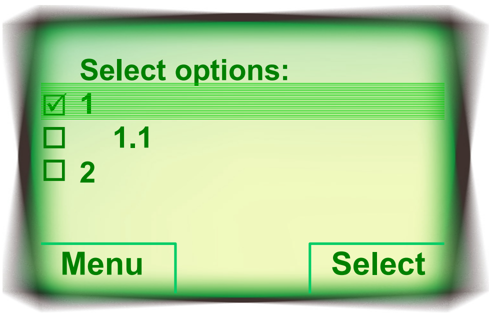
This results in the following calls:

```
CLNTa_getFile (1, "file:///select3.wml");
CLNTc_file (1, …, …, "application/vnd.wap.wmlc");
```

The WML source for this sample has this content:

```
<wml>
  <card ontimer="#next">
    <p>
      Select options:
      <select multiple="true" name="select3">
        <option value="1">1</option>
        <optiongroup>
          <option value="1.1">1.1</option>
        </optiongroup>
        <option value="2">2</option>
      </select>
    </p>
    <timer value="600"/>
  </card>
  <card id="next">
    <p>Menu choice: $(select3)</p>
  </card>
</wml>
```

It can, in accordance with the examples above, be displayed like this:



The option group is indented in this example.

The first WML card will produce the following calls:

```
MMIa_newCard (1, NULL, FALSE, FALSE,
           "file:///select3.wml", TRUE, titles);
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode("Select options:"), FALSE,
           TXT_NORMAL);
MMIa_newSelect (1, NULL, FALSE, 0);
MMIa_newOption (1, 1, Unicode("1"), NULL, TRUE);
MMIa_newOptionGroup (1, NULL);
MMIa_newOption (1, 2, Unicode("1.1"), NULL, TRUE);
MMIa_closeOptionGroup (1);
MMIa_newOption (1, 3, Unicode("2"), NULL, FALSE);
MMIa_closeSelect (1);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

Depending on for which options the MMIc_optionSelected function is called, the text in card two will be Menu choice: 1, Menu choice: 2 or Menu choice: 1;2.

5.4.4 Keys

There can be an arbitrary amount of keys in a WML application. The keys might be displayed inline, in the card content, or at another place on the display. They might as well be implemented as a combination of hardware keys and displayed menus of key options. A sample WML application, which displays a single key, is opened with:

```
MMIc_loadURL (1, "file:///key.wml", FALSE);
```

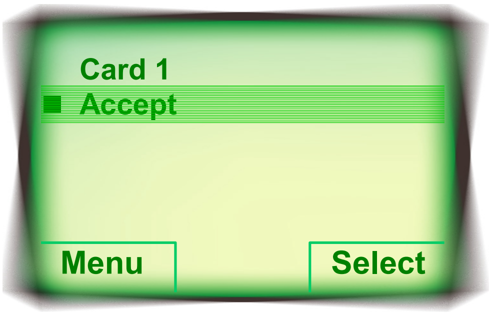
This results in the following calls:

```
CLNTa_getFile (1, "file:///key.wml");
CLNTc_file (1, …, …, "application/vnd.wap.wmlc");
```

The WML source for this sample has this content:

```
<wml>
  <card>
    <p>
      Card 1
      <do type="accept">
        <go href="#next"/>
      </do>
    </p>
  </card>
  <card id="next">
    <p>
      Card 2
      <do type="prev">
        <prev/>
      </do>
    </p>
  </card>
</wml>
```

If the WML source above should be displayed inline in the WML view, this example illustrates how it could look like.



The text that identifies the key in this example has the title Accept. This title is derived from the type of key that is used. The actual text that is displayed can be localised. The title can be substituted or combined with special symbols on the display. It can also be mapped to a keyboard.

A MMIa_newKey call will be executed when the URL is loaded.

```
MMIa_newCard (1, NULL, FALSE, FALSE, "file:///key.wml",
           TRUE, titles);
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode("Card 1"), FALSE, TXT_NORMAL);
MMIa_newKey (1, 1, Unicode("ACCEPT"), NULL, FALSE);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

When the key is selected, this call shall be made:

```
MMIc_keySelected(1, 1);
```

Now is card two opened:

```
MMIa_newCard (1, NULL, FALSE, FALSE,
           "file:///key.wml#next", TRUE, titles);
```

```
MMIa_newParagraph (1, ALIGN_LEFT);
MMIa_newText (1, 0, Unicode("Card 2"), FALSE, TXT_NORMAL);
MMIa_newKey (1, 1, Unicode("PREV"), NULL, FALSE);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

Selection of the key in this card causes the WAP application to navigate back to card one, again.

5.5 WML Script support

The WMLS interpreter is almost entirely hidden from the interface that the AUS WAP Browser has. However, some dialogs may be opened from a WML script that is currently running. The dialogs are to be implemented in this build step.

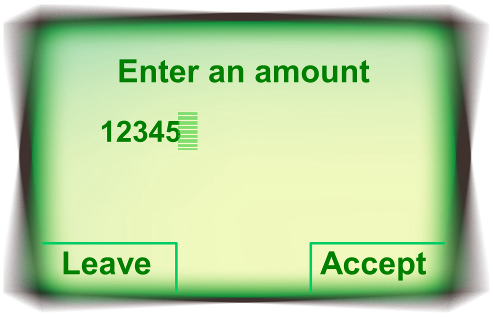
At this step implementations are required of the following adapter functions:

Adapter function	**Basic implementation**
MMIa_promptDialog	The basic implementation calls CLNTa_log and MMIc_promptDialogResponse
MMIa_confirmDialog	The basic implementation calls CLNTa_log and MMIc_confirmDialogResponse
MMIa_alertDialog	The basic implementation calls CLNTa_log and MMIc_alertDialogResponse

The basic implementations can be used in the beginning but should be exchanged with real implementation as soon as real dialog interaction is required.

5.5.1 Prompt dialog

A dialog that's prompts for input from the user might be opened. A prompt dialog may be displayed like this:



The dialog is left by either press the Leave key or Accept key. The Leave key is actually not necessary to have when it is not part of WML script. If it is chosen to be included, the action to take should be the same as when the Accept key is pressed. The exception should be that Leave returns an empty string to the AUS WAP Browser and that Accept returns the user entered string. Some dialogs are required to return a string. In that case, the Leave key must be disabled as long as the user has not entered a string.

A sample WML application, which uses such a WML script, is opened with:

```
MMIc_loadURL (1, “file:///script1.wml”, FALSE);
```

This results in the following calls:

```
CLNTa_getFile (1, “file:///script1.wml”);
CLNTc_file (1, …, …, “application/vnd.wap.wmlc”);
```

The WML source for this sample has this content:

```
<wml>
  <card>
    <p>
      Card 1
      <do type=“accept” name=“a”>
        <go href=“test.scr#openPromptDialog1()”/>
      </do>
      <do type=“accept” name=“b”>
        <go href=“ test.scr#openPromptDialog2()”/>
      </do>
      <do type=“accept” name=“c”>
        <go href=“ test.scr#openPromptDialog3()”/>
      </do>
      <do type=“accept” name=“d”>
        <go href=“#next”/>
      </do>
    </p>
  </card>
  <card id=“next”>
    <p>
      Card 2<br/>
      Answer 1: $(answer1)<br/>
      Answer 2: $(answer2)<br/>
      Answer 3: $(answer3)
    </p>
  </card>
</wml>
```

The three first keys will force three WMLS files to be read, and generate one call respectively to MMIa_promptDialog. The three calls will test the ways the input field in the dialog can be initially set. In each one of the three cases above shall the function open the dialog and directly return. The answer from each one of the dialogs is given to the AUS WAP Browser in a dedicated connector function.

Press key 1:

```
CLNTa_getFile (2, “file:///script11.wmls”);
CLNTc_file (2, …, …, “application/vnd.wap. wmlscriptc”);
MMIa_promptDialog (1, 1, NULL, NULL);
MMIc_promptDialogResponse (1, 1, Unicode(“answer”));
```

Press key 2:

```
CLNTa_getFile (3, "file:///script12.wmls");
CLNTc_file (3, …, …, "application/vnd.wap. wmlscriptc");
MMIa_promptDialog (1, 2, NULL, Unicode("default
         message"));
MMIc_promptDialogResponse (1, 2, Unicode("answer"));
```

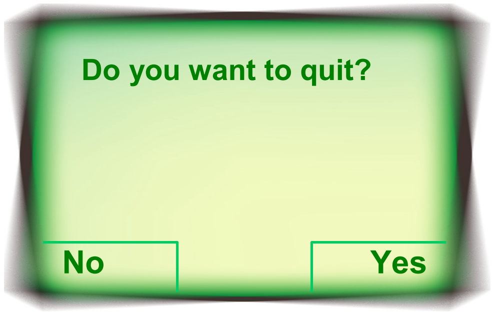
Press key 3:

```
CLNTa_getFile (4, "file:///script13.wmls");
CLNTc_file (4, …, …, "application/vnd.wap. wmlscriptc");
MMIa_promptDialog (1, 3, Unicode("message"),
         Unicode("default message"));
MMIc_promptDialogResponse (1, 3, Unicode("answer"));
```

By selecting the fourth key, card 2 will be displayed. The answers from each one of the dialogs shall be given there.

5.5.2 Confirm dialog

A dialog that's requesting confirmation from the user might as well be opened.



This example has keys with the labels No and Yes. The default names of the keys are not standardised and can be localised. They default labels should reflect the standard behaviour of the dialog, to decline or accept the given message.

A sample WML application, which uses such a WML script, is opened with:

```
MMIc_loadURL (1, "file:///script2.wml", FALSE);
```

This results in the following calls:

```
CLNTa_getFile (1, "file:///script2.wml");
CLNTc_file (1, …, …, "application/vnd.wap.wmlc");
```

The WML source for this sample has this content:

```
<wml>
  <card>
    <p>
      Card 1
      <do type="accept" name="a">
        <go href=" test.scr#openConfDialog1()"/>
      </do>
      <do type="accept" name="b">
        <go href=" test.scr#openConfDialog2()"/>
      </do>
      <do type="accept" name="c">
        <go href="#next"/>
      </do>
    </p>
  </card>
  <card id="next">
    <p>
      Card 2<br/>
      Answer 1: $(answer1)<br/>
      Answer 2: $(answer2)
    <p>
  </card>
</wml>
```

The two first keys will force two WMLS files to be read, and generate one call respectively to MMIa_confirmDialog. The two calls will test the ways the dialog can be closed. In each one of the two cases above shall the function open the dialog and directly return. The answer from each one of the dialogs is given to the AUS WAP Browser in a dedicated connector function.

Press button 1:

```
CLNTa_getFile (2, "file:///script21.wmls");
CLNTc_file (2, …, …, "application/vnd.wap.wmlscript");
MMIa_confirmDialog (1, 1, Unicode("message"),
             Unicode("ok"), Unicode("cancel"));
MMIc_confirmDialogResponse (1, 1, TRUE);
```

Press button 2:

```
CLNTa_getFile (3, "file:///script22.wmls");
CLNTc_file (3, …, …, "application/vnd.wap. wmlscriptc");
MMIa_confirmDialog (1, 2, Unicode("message"),
             Unicode("ok"), Unicode("cancel"));
MMIc_confirmDialogResponse (1, 2, FALSE);
```

By selecting the third key, card 2 will be displayed. The answers from the dialogs are to be given there.

5.5.3 Alert dialog

The last WMLS dialog is the simplest. It informs the user, which in turn only cancels the dialog. An alert dialog may be displayed like this:

Game over!
Leave
Accept

The dialog is left by either press the Leave key or Accept key. The Leave key is actually not necessary to have when it is not part of WML script. If it is chosen to be included, the action to take should be the same as when the Accept key is pressed. Alternatively, the entire script can be cancelled by calling CLNTc_stop when the Leave key is pressed.

A sample WML application, which uses a WML script with an alert dialog, is opened with:

```
MMIc_loadURL (1, "file:///script3.wml", FALSE);
```

This results in the following calls:

```
CLNTa_getFile (1, "file:///script3.wml");
CLNTc_file (1, …, …, "application/vnd.wap.wmlc");
```

The WML source for this sample has this content:

```
<wml>
  <card>
    <p>
      Card 1
      <do type="accept" name="a">
        <go href=" test.scr#openAlertDialog()"/>
      </do>
      <do type="accept" name="b">
        <go href="#next"/>
      </do>
    <p>
  </card>
  <card id="next">
    <p>
      Card 2
      The dialog is done: $(status)
    </p>
  </card>
</wml>
```

The first key will force one WMLS file to be read, and generate one call to MMIa_alertDialog. The call will test that the dialog is closed properly. In the case above shall the function open the dialog and directly return. The answer from the dialog is given to the AUS WAP Browser in a dedicated connector function.

Press key 1:

```
CLNTa_getFile (2, "file:///script31.wmls");
CLNTc_file (2, …, …, "application/vnd.wap. wmlscriptc");
MMIa_alertDialog (1, 1, Unicode("message"));
MMIc_alertDialogResponse (1, 1);
```

By selecting the second key, card 2 will be displayed. The text shall indicate whether the dialog was closed properly, or not.

5.6 Connecting the network

Now, when all mandatory functions are implemented, the adaptations to the available network bearers need to be implemented as well. The AUS WAP Browser has support for three bearers, UDP, SMS and USSD. If any of the bearers are omitted, further implementation of the particular API is not needed. However, the empty function stubs has still to be in the WAP application system, in order to compile and link properly.

At this step implementations are required of the following adapter functions:

Adapter function	**Basic implementation**
MMIa_passwordDialog	The basic implementation calls CLNTa_log and MMIc_passwordDialogResponse
UDPa_sendRequest	The basic implementation calls CLNTa_log and UDPc_recievedRequest, which responds with a hard wired WML deck like the "Hello, world" example in this chapter

5.6.1 Password dialog

One adapter function has to be implemented for all bearers:

MMIa_passwordDialog

The function is called when a content server or a WAP proxy server requires authorisation. The dialog implementation and behaviour shall be similar to the WML script dialog function MMIa_promptDialog. The response to that function is to be sent in i similar way as with the MMIc_promptDialogResponse function. The password counterpart is called:

MMIc_passwordDialogResponse

The main difference from the prompt dialog is that the password characters that the user enters shall not be displayed in the dialog. Instead shall any other character be used, asterisks, for instance.

To connect a bearer API read the chapter about it. Follow the guidelines given there. One thing to keep in mind is that if a bearer is implemented in a task, other than the WAP application task, bearer Connector function calls to the AUS WAP Browser have to be synchronised with other WAP application Connector function calls.

5.7 Optional functionality

By now, all base functionality is on place in order to be WAP conformant. The remaining oional fuctions are:

MMIc_refresh
MMIa_status
MMIa_unknownContent
MMIa_newImage
MMIa_completeImage
MMIc_imageSelected
MMIa_newFieldSet
MMIa_closeFieldSet
MMIa_linkInfo
MMIa_nonSupportedScheme
CLNTa_content
CLNTa_confirmDownload
CLNTc_confirmDownload
WTAa_publicMakeCall
WTAa_publicSendDTMF
WTAa_publicPBwrite

6 Tuning the AUS WAP Browser

Having the AUS WAP Browser compiled and linked for the Host Device guarantees only that it will work properly on it. Performance of the bearers can be quite different from one device, to another. In order to optimise the AUS WAP Browser for the Host Device, configurations variables can be adjusted. This section aims to clarify those variables, the purpose of them and the impact of the overall performance they have.

The file confvars.h contains constant configuration variables with default values that can be adjusted in order to fine-tune the AUS WAP Browser behaviour and performance for a specific Target Device.

All variables have default settings, which works for most target devices.

6.1 AUS WAP Browser

There are variables that configure the AUS WAP Browser kernel implementation. The kernel manages a set of processes that implements the AUS WAP Browser. The processes communicate by signals. On certain operating systems, where a dynamic memory management system not exists, the AUS WAP Browser implements that as well. Read more about this in the "Optional source code" chapter, at the end in this manual.

Variable name	**Default**	**Description**
USE_WIP_MALLOC	Not defined	Defined if the internal AUS WAP Browser memory management implementation is to be used in favour of malloc and free.
WIP_MALLOC_MEM_SIZE	25000	Size of memory that is to be used by the internal memory management routines.

The AUS WAP Browser has an optional character encoder that can be used. Read more about this in the "Optional source code" chapter, at the end in this manual.

Variable name	**Default**	**Description**
USE_CHARSET_ PLUGIN	Not defined	If KS C 5601 character set encoded data shall be read by the AUS WAP Browser, this variable should be defined.

The AUS WAP Browser has a supervising function called the Stack Manager. The Stack Manager controls start-up and termination of the AUS WAP Browser. It manages common functionality for the WAP stack in general.

Variable name	Default	Description
MaxStartUpTime	150	Time in 1/10 of seconds until start-up is considered failed

If error messages should be issued for certain levels of memory usage, the following constants should be defined:

Variable name	Default	Description
USE_MEMORY_GUARD	Not defined	If the constant is defined, memory count is turned on. This option costs MEM_ADDRESS_ALIGNMENT (device specific constant to be defined in tapicmmn.h) number of bytes per memory allocation.
MEMORY_WARNING	Not defined	The number of bytes where the AUS WAP Browser should issue a warning. The AUS WAP Browser resets the history of URLs and removes all WML variables.
MEMORY_LIMIT	Not defined	The number of bytes where the AUS WAP Browser should issue an error message. At this level, the AUS WAP Browser reset itself at this level. To proceed browsing, the AUS WAP Browser must be restarted (CLNTc_start) and re-initialised.

6.2 WAE

The WML user agent manages WML content. It parses downloaded WML files. The general behaviour of it can be adjusted with the following variables.

Variable name	Default	Description
cfg_wae_ua_ methodPostCharsetOverride	0	Defines if the post method should be WAP conformant or not. The non-WAP conformant way is de-facto standard on some older web servers. 0: WAP conformant. The "charset" parameter is set in

		http field "content-type". 1: Not WAP conformant. No "charset" parameter is set.
cfg_wae_ua_imageMaxNbr	30	Sets the maximum number of simultaneously requests for images that a view will queue for a WML card that is opened. All requests that are issued after this limit has been broken are ignored.
cfg_wae_ua_defaultPrio	2	Sets the priority of default requests over the air. A low value means a low priority. This constant is used if the request is not for an image or script.
cfg_wae_ua_imagePrio	1	Sets the priority when requesting images over the air. A low value indicates a lower priority, which implies a longer wait, if requests of other content types are issued. The default value is to have higher priority than the default priority.
cfg_wae_ua_scriptPrio	3	Sets the priority for script requests over the air. The default value is to have lower priority than the default priority.
cfg_wae_ua_fileCharEncoding	106	The default text encoding used in local files. 106 = UTF-8, i.e., Unicode
cfg_wae_ua_current_time_is_gmt	0	This constant tells if the time, the function CLNTa_currentTime returns, is GMT or local. If the time is GMT, the constant shall be set to 1, otherwise it shall be set to 0 (default).
cfg_wae_cc_cachePrivate	1	This variable is to be set to 0 if one application (or terminal) uses the AUS

		WAP Browser. If several applications use the AUS WAP Browser the cache is shared by all applications. This means that content with the cache-directive private not should be cached. In that case is this variable to be set to 1.
cfg_wae_cc_cacheCompact	0	This variable affects the way data posts is removed from the cache in order to make room for new ones. If the value of the variable is set to 0, the oldest data post is removed until enough space is available to store the new data post. If the value is set to 1, compaction is performed if enough space is not available for the new data post. If there still is not enough space, after the compaction, the oldest data post is removed until enough space is available.

The display differs among different devices. Therefore is it necessary to configure how the different user interface elements are to be rendered inline in the card layout, or not. This affects how white-space characters should be output to the display. For instance, the following example can be displayed in two ways, with the key displayed inline in the text or completely separated from the text:

```
<p>
    This is a WML key: <do type="accept" name="NAME"><prev/></do> with
    white-space characters on both sides.
</p>
```

If the key is to be displayed inline, the two white-space characters should be displayed as well. Otherwise, if the key should be displayed somewhere else, only one of the two white-space characters should be displayed.

Variable name	**Default**	**Description**
cfg_wml_disp_do_inline	0	1 if do elements should be rendered inline, 0 otherwise.

cfg_wml_disp_img_inline	1	1 if image elements should be rendered inline, 0 otherwise.
cfg_wml_disp_anchor_inline	1	1 if link elements should be rendered inline, 0 otherwise.
cfg_wml_disp_a_inline	1	1 if link elements should be rendered inline, 0 otherwise.
cfg_wml_disp_table_inline	1	1 if table elements should be rendered inline, 0 otherwise.
cfg_wml_disp_input_inline	1	1 if input field elements should be rendered inline, 0 otherwise.
cfg_wml_disp_select_inline	1	1 if selection menu elements should be rendered inline, 0 otherwise.

WAE has a WML Script interpreter. The interpreter operates under supervisory of the AUS WAP Browser. The behaviour of the interpreter can be configured in the way it shall execute. As with the other variables in confvars.h, the default setting is chosen to work on most target devices.

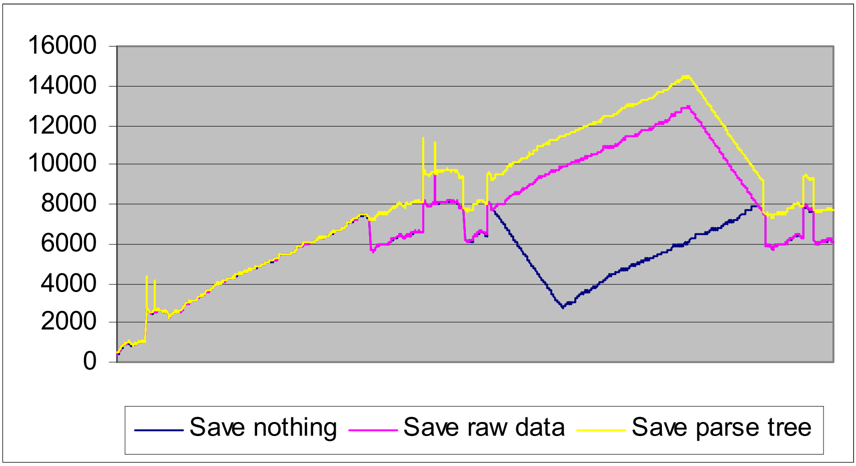
Variable name	**Default**	**Description**
cfg_wmls_timeSlice	10	How many time units the interpreter will execute before returning control to WAE. One time unit is equal to the time it takes to execute one WMLS byte code instruction or the time it takes to execute one WMLS library function. A rough estimate is that one line of WMLS code generates 3 byte code instructions.
cfg_wmls_roundRobin	0	0: Execute one scripts at time. If a view (for a WAP application) is started and a script is executed from that view as well, it will be queued if another script (belonging to another view) already executes. 1: Two or more scripts will execute in parallel. Round robin scheduling is used.
cfg_wmls_oneScriptPerUa	1	0: Allow execution of several scripts per view. This option is only present for future enhancements of the WAP standard. 1: Allow only execution of one script per view.

cfg_wmls_handleTopPriority	0	0: All script execution has equal priority. 1: Scripts execution from a WTA activity should be handled with higher priority than normal script execution. This option works also when the variable cfg_wmls_roundRobin is set to 0.

Memory optimisation variables that control how WML decks are saved between transitions, from one card to another. How the AUR WAP Browser shall do can be configured by the following variable:

Variable name	**Default**	**Description**
PARSE_MODE	1	Can be defined to 0, 1 or 2. 0: Do not save old card, i.e. throw it away before new card is opened. Saves a lot of RAM but makes the browser to not be WAP conformant. 1: Save raw data. If the new card fails to open, the old card is opened again from the raw data. Consumes as much extra memory as the WML deck takes in compiled format. 2: Save parse tree. When the raw data is parsed a parse tree is built. If the parse tree for a old card is saved when moving to a new card, and the new card fails to open, re-opening of the old card is much faster. The drawback is the memory consumption, which is the biggest of the three possible modes.

To be WAP conformant the old card shall be saved until the new card has been successfully opened.



The graph above illustrates the three different modes when parsing WML decks, regarding memory consumption. Each series describes a download of a (compiled) 1400 bytes large WML deck and a equal sized image, followed by a identical download but with the image taken from cache. The internal memory allocator is in use (see USE_WIP_MALLOC, above).

General variables:

Variable name	Default	Description
REFRESH_TASK_INFO	"REFRESH"	When using the "infolink" functionality this value indicates what will be displayed when no URL is present and a refresh task has been performed.
PREV_TASK_INFO	"PREVIOUS"	When using the "infolink" functionality this value indicates what will be displayed when no URL is present and a previous task (back) has been performed.

6.3 WSP

The WAE manages HTTP functionality in WSP. The following variable can be set to affect the standard behaviour.

Variable name	Default	Description
cfg_wae_wspif_ ReSendingTimeout	10	If the AUS WAP Browser does not receive the content of a requested image, the request is resent. This variable set the number of seconds before it is resent. A value less then 5 seconds is not recommended. This is repeated as long as the request

		does not time-out, or if content is not received. Each resent request gets a new identifier (tid) and is therefore different from the original request. In that way, confusions at the server side are avoided. This parameter is only used when the bearer is UDP. The functionality compensates the lack of WTP in a connection-less protocol stack, since the reliability of UDP is known to be poor.
cfg_wae_wspif_redirectPost	0	Which method shall be used when a redirect of a Post-request is performed? 0: GET, 1: POST. The HTTP specification (RFC2068) claims that POST should be used. It refers however also to the fact that some HTTP/1.0 browsers (*read practically all HTTP/1.0-1.1 browsers*) uses GET for the redirected operation. The default value for the AUS WAP Browser is therefore set to the de-facto standard that exists on internet.
cfg_wae_wspif_ FileTimeout	30	Number of seconds the AUS WAP Browser is waiting for a response after CLNTa_getFile was called. If the value is set to zero no timer is set.
cfg_wae_wspif_ FunctionTimeout	60	Number of seconds the AUS WAP Browser is waiting for a response after CLNTa_callFunction was called. If the value is set to zero no timer is set.
MaxPDUsize	5120	The maximum PDU size (in bytes) the WAP protocol stack should handle. This constant is used in the capability negotiation that takes place when a WSP session is to be established with a WAP gateway. It restricts however also the PDU size in connection-less WSP.

The identity and capabilities of the WAP Client that uses the AUS WAP Browser can be declared in the following C pre-processor definitions. The default values are initial settings and must be modified to describe the WAP Client capabilities.

Variable name	**Template**	**Description**
HTTP_HEADER_ USER_AGENT	"WAPPER"	Sets the identity of the WAP Client that is given to content servers
ACCEPT_IMAGE	"image/gif, image/jpg"	GIF and JPEG images accepted

6.4 WTP

The WTP layer of the protocol stack has some values that can be fine-tuned for a specific target device and specific bearer capabilities.

Variable name	**Default**	**Description**
no_of_retransmissions_ UDP_WTP	8	The number of retransmissions WTP does before the requested data is considered lost. This value is taken when the bearer is UDP.
retransmission_interval_ UDP_WTP	50	How long time, in 1/10 of seconds, shall it be between each retransmission? This value is taken when the bearer is UDP.
wait_timeout_interval_ UDP_WTP	400	How long time in 1/10 of seconds, after the requested data has arrived, shall WTP wait in order to catch duplicates of the requested data, etc? This value is taken when the bearer is UDP.
no_of_retransmissions_ SMS_WTP	4	The number of retransmissions WTP does before the requested data is considered lost. This value is taken when the bearer is SMS.
retransmission_interval_ SMS_WTP	600	How long time, in 1/10 of seconds, shall it be between each retransmission? This value is taken when the bearer is SMS.
wait_timeout_interval_ SMS_WTP	3000	How long time, in 1/10 of seconds, after the requested data has arrived, shall WTP wait in order to catch duplicates of the requested data, etc? This value is taken when the bearer is SMS.

no_of_retransmissions_USSD_WTP	4	The number of retransmissions WTP does before the requested data is considered lost. This value is taken when the bearer is USSD.
retransmission_interval_USSD_WTP	600	How long time, in 1/10 of seconds, shall it be between each retransmission? This value is taken when the bearer is USSD.
wait_timeout_interval_USSD_WTP	600	How long time, in 1/10 of seconds, after the requested data has arrived, shall WTP wait in order to catch duplicates of the requested data, etc? This value is taken when the bearer is USSD.

6.5 WDP

The WDP layer of the protocol stack has some values that can be fine-tuned for a specific target device and specific GSM capabilities.

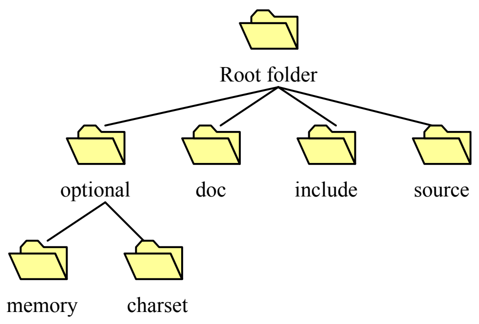
Variable name	**Default**	**Description**
MaxReassTime	3000	1/10 of seconds a SMS or USSD message segment is waited for, before it is considered lost.
LowestMaxUssdLength	70	The max length of an USSD text message may vary from country to country. It is not standardised. In order to determine when segmentation shall be performed and how large the segments shall be, this constant is to be set. It sets the upper limit of the length of each USSD text message that is sent from the AUS WAP Browser.

7 Makefile and source files

The AUS WAP Browser shall be compiled and linked with the WAP application that uses it. It can be compiled into a library, into a dynamic link library (DLL), or directly with the WAP application source code. Either the way, a makefile or a project file must be created or adopted for the AUS WAP Browser source code.

7.1 AUS WAP Browser source files

The directory structure of the AUS WAP Browser looks like this:



The folders and its content will be described shortly in below.

\include
In this folder we find all include files for the AUS WAP Browser:

File name	Description
aapiclnt.h	Header file for Adapter functions in the Client API.
aapimem.h	Header file for Adapter functions in the Memory API.
aapimmi.h	Header file for Adapter functions in the MMI API.
aapisms.h	Header file for Adapter functions in the SMS API.
aapiudp.h	Header file for Adapter functions in the UDP API.
aapiussd.h	Header file for Adapter functions in the USSD API.
aapiwta.h	Header file for Adapter functions in the WTA API.
aapicrpt.h	Header file for Adapter functions in the Crypto API.
capiclnt.h	Header file for Adapter functions in the Client API.
capimem.h	Header file for Adapter functions in the Memory API.
capimmi.h	Header file for Adapter functions in the MMI API.
capisms.h	Header file for Adapter functions in the SMS API.
capiudp.h	Header file for Adapter functions in the UDP API.
capiussd.h	Header file for Adapter functions in the USSD API.

capiwta.h	Header file for Connector functions in the WTA API.
tapicmmn.h	Header file for the Common API.
tapimem.h	Header file for types used in the Memory API.
tapimmi.h	Header file for types used in the MMI API.
errcodes.h	Header file with constants of all kinds of errors the Adapter function CLNTa_error may give.
logcodes.h	Header file with constants of all kinds of log information the Adapter function CLNTc_log may give.
confvars.h	Constants, to be configured for a specific application.
target.h	Header files for macros to help the AUS WAP Browser distinguish between different target environments like EPOC, OSE and REX.
ansilibs.h	Header file where all ANSI C library files are included.

\source
Source files that implement the AUS WAP Browser. All files should be included in the makefile or project.

\optional\memory
Source files for internal memory management. Does not to be included in the makefile or project if not the macro USE_WIP_MALLOC is defined (see \include\confvars.h).

\optional\charset
Source files for support of optional character sets. For the moment is KSC5601 (Korean characters) supported. The files must not be included in the makefile or project if not the macro USE_CHARSET_PLUGIN. (See \include\confvars.h.)

7.2 Makefile settings

The makefile or project must include all non-optional source code files, described in the section above. The source code is written in ANSI C and should be compiled as such.

The compiler must be given "include paths" so that the header files of the AUS WAP Browser can be found. Below is a listing of all needed paths:

\include
\source

If the macro USE_WIP_MALLOC is defined, this path must be added:

\optional\memory (see \include\confvars.h)

If the macro USE_WIP_MALLOC is defined, this path must be added:

\optional\charset (see \include\confvars.h)

If the LOG_EXTERNAL flag is given to the compiler, logging of the communication between the layers is enabled. This feature is used when the AUS WAP Browser integrated with a WAP application and when it is to be fine-tuned for a specific device. LOG_EXTERNAL should not be present in the release build, when the performance will be slightly affected. Read more about this in the Client API, where the CLNTa_log function is described.

8 Common API

The AUS WAP Browser uses different types depending of the magnitude and sign of the values they hold. The following types are to be defined to corresponding types on the Host Device.

8.1 Types

The AUS WAP Browser uses data types that can be defined for the Target Device development environment. The default definitions is found in tapicmmn.h:

INT8	8 bit signed integer
UINT8	8 bit unsigned integer
INT16	16 bit signed integer
UINT16	16 bit unsigned integer
INT32	32 bit signed integer
UINT32	32 bit unsigned integer
FLOAT32	32 bit floating point value. If no real type exist for the host device, map to INT32. See also the static configuration variable HAS_FLOAT, which is defined in target.h.
BOOL	1 bit integer
BYTE	8 bit unsigned integer
CHAR	8 bit character, signed or unsigned. The only criterion is that printable characters shall be positive.
UCHAR	8 bit unsigned character
WCHAR	16 bit unsigned character
VOID	A special type indicating the absence of any value.

8.2 Constants

These constants are already defined in tapicmmn.h:

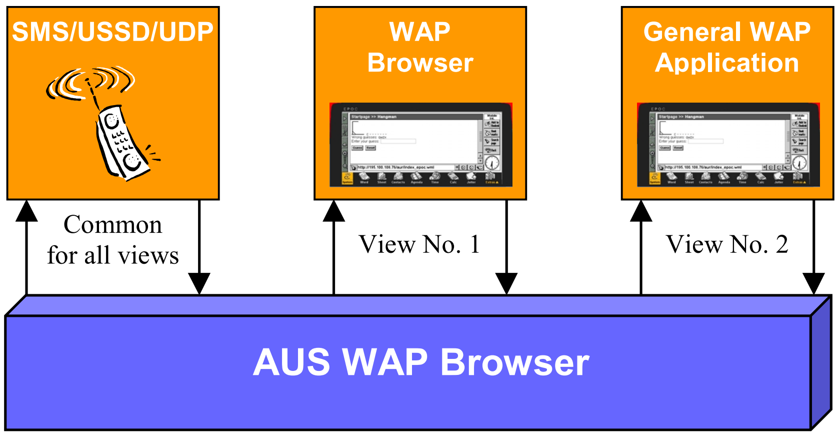
NULL	Integer value 0
TRUE	Integer value 1
FALSE	Integer value 0
MEM_ADDRESS_ALIGNMENT	Device dependent constant that describes the alignment for memory addresses that

	malloc return. Ex:
	`struct {`
	`        char *t; /* address 0x??????00 */`
	`        char *p; /* address 0x??????04 */`
	`};`
	As the example illustrates takes the two variables in the struct at least 8 bytes on a device with 32 bit addresses. Some devices need further padding. The least possible memory alignment is set to four bytes as default.

9 MMI API

For the implementation of the WAP application there will be a defined and implemented Connector interface and a defined but not implemented Adapter interface. For devices with sophisticated displays that already support some type of windowing/GUI interface, the Adapter functions will be very "thin". For less sophisticated displays the Adapter functions will be much "thicker" and will require extra effort to support those devices.

9.1 Views



Before a WAP application can start using the AUS WAP Browser, it must establish a view to the AUS WAP Browser. Several WAP applications can use the AUS WAP Browser simultanously. They will all share common resources like the cache and the bearers that are available for data transmission. The views are used by the AUS WAP Browser to distinguish one WAP application from another. They will have there own set of configuration variables.

openView

This function tells the AUS WAP Browser that a new view has been opened. A WAP application that only uses content retrieval (with CLNTc_getContent) can ignore this function since there is a predefined viewId constant /see table below) to be used in this case (e.g. to be used when a configuration variables shall be set).

```
VOID MMIc_openView (UINT8 viewId, INT8 uaMode)
```

`viewId`	An id, in the range from 1 to 127, of the view is passed in the argument. Other values for view ids are stated in the table below. These values can be used with the function CLNTc_setIntConfig and its string correspondant. These predefined values may also come as the viewId in the function CLNTa_error. The viewId argument may be used in subsequent calls to Connector functions as MMIc_loadURL. The viewId argument will then be used by the AUS WAP Browser in calls of Adapter functions for displaying WML cards.

`uaMode`	The view can be targeted for different applications. The different kinds are stated in the table below.

The argument uaMode can be set with the following constants (defined in tapimmi.h):

Value	Constant	Description
1	WML_UA	The constant WML_UA should be used when the application is a normal WML browser.
2	WTA_UA	The constant WTA_UA should be used when the application is a WTA browser.

Description of predefined view ids for which this function not needs to be used (defined in tapimmi.h):

Value	Constant	Description
0	GENERAL_VIEW	AUS WAP Browser regards the viewId value 0 as general. (0 is used as a viewId when error and log messages do not origin from a specific view).
128	CONTENT_VIEW	Reserved view id for WML browsing for content retrieval (with the functions CLNTc_getContent and CLNTa_content).
129	REPOSITORY_VIEW	Reserved view id for downloading activities to the WTAI repository.
130	PUSH_VIEW	Reserved view id for PUSH activities.

closeView

This function tells the AUS WAP Browser that the view identified by viewId has been closed. It cancels currently ongoing downloads and cleans up in the AUS WAP Browser. The function needs not to be called when the WAP application is closing. It is enough to call CLNTc_terminate.

```
VOID MMIc_closeView (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

9.2 Controls

Each view has a mandatory set of controls. They control the AUS WAP Browser by the following functions.

loadURL

Called from the view when the user enters an URL to navigate to.

```
VOID MMIc_loadURL (UINT8 viewId, const CHAR* url,
BOOL reload)
```

`viewId`	The view id received from a call to MMIc_openView.
`url`	The URL of the WML card that shall be opened. The caller may delete the string after the call.
`reload`	If the argument reload is set to FALSE, the source is first looked up in the cache. If the source is not found in the cache, or if the reload argument is set to TRUE, the source is downloaded. If a WML card inside a WML application shall be reloaded, the function MMIc_reload should be used instead, because a call to MMIc_loadURL resets the context of the view; i.e., all WML variables are removed and the internal history are reset.

reload

Called from the view when the user wants to reload the source of the WML card currently active. The call forces the AUS WAP Browser to download the source, without using the eventually cached source. The currently active card will then be redisplayed. Not only the WML deck is reloaded, images and WML scripts are reloaded as well. However, if it is an entire WML application that shall be reloaded, the function MMIc_loadURL with the reload argument set to true should be used instead. A call to this function does not reset the context of the view; i.e., all WML variables and the internal history are preserved.

```
VOID MMIc_reload (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

stop

This function shall be used when the current download of a new deck or the current downloads of a deck's images and scripts are to be cancelled, for a certain view. If it is called during the time a deck is loaded, the current card will remain active. However, if the downloading of the deck is done and the AUS WAP Browser waits for images and scripts, the target card of that deck is displayed with all available images, whether dynamic updates of images is supported or not. After all sources have been downloaded, the function can be used to stop execution of WML scripts. Not only the stop button can use this function, also when an error occurs in the WAP application during the display operation (lack of memory, for instance), it can be used.

```
VOID MMIc_stop (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

goBack

This function shall be used when the user has chosen to navigate backwards in a certain view. This kind of event is a default behaviour that must be supported by a WAP application. It has nothing in common with DO elements that might have been declared for the card currently being viewed. It simply takes the user back to the card previous to the current card in the history list.

```
VOID MMIc_goBack (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

9.3 Notifications

The AUS WAP Browser notifies the WAP application at different occasions. For that purpose these functions are defined.

wait

The AUS WAP Browser uses this function when the card control functions may not be operated. The only MMI control functions that may be operated during the pause are the MMIc_stop, MMIc_back and the MMIc_loadURL functions. All other functions, MMIc_textSelected, MMIc_imageSelected, MMIc_keySelected, and MMIc_optionSelected must be blocked (the AUS WAP Browser ignores all such calls that are done during the time of the pause). If the functions are not blocked, unpredictable results might occur. If the user for instance selects a menu option during that time, the AUS WAP Browser will not have the same options selected as the MMI.

```
VOID MMIa_wait (UINT8 viewId, BOOL isWait)
```

`viewId`	The view id received from a call to MMIc_openView.
`isWait`	TRUE if the WAP application shall pause. When the AUS WAP Browser is ready for input again, the function is called again, this time with this argument set to FALSE.

status

Called when status information is available from the AUS WAP Browser. See in the table of status codes below, what status information that can be sent and when.

```
VOID MMIa_status (UINT8 viewId, UINT8 status, const
CHAR *URL)
```

`viewId`	The view id received from a call to MMIc_openView.

`status`	Status code. All codes are described below in the section Constants.
`URL`	The URL the status is associated with is given in this argument. It is always given with the status. It is deleted after the function call.

Constants

Valid status values are given in the following table. The constants are declared in the file tapimmi.h.

WAE Status	**Value**	**Description**
ContentIsOpened	1	Used when a link or another navigation task is executed and content is to be downloaded or fetched from the cache. Content can be a WML deck, WMLS byte package or any unknown content.
ContentIsDone	2	Used when the content finally is downloaded.
ImageIsOpened	3	Used when an image in a WML deck is to be downloaded or fetched from the cache.
ImageIsDone	4	Used when the image finally is downloaded.
ScriptIsRunning	5	Used when a WML script has started its execution.
ScriptIsDone	6	Used when the WML script is done with the execution.
Redirect	7	Used when a Redirect status code from WSP is detected
ReadFromCache	8	Used when content is taken from the cache
ReadFromNetwork	9	Used when content is taken from a network server
CheckForNewerContent	10	Used when the server is queried for newer content, than the content in the cache that already exists
ReceivedFromNetwork	11	Used when the AUS WAP Browser receives data from the network
WSPSessionIsSetup	12	Indicates that WSP session currently is setup. The argument URL will always be NULL for this status code.
WSPSessionIsDone	13	Indicates that WSP session setup phase is done. It indicates not whether it was

		successfull, or not. The argument URL will always be NULL for this status code.
LoadingData	14	Indicates that data, i.e a deck, a script or unknown data is to be loaded. It does not indicate whether it is from the network or from the cache. Other status codes indicates that.
LoadingDataDone	15	Indicates that the data has been loaded. One indication matches all preceeded LoadingData indications.

The status codes above is given in the following order (BNF notation, with status codes in red):

Start ::= LoadingData Data LoadingDataDone
Data ::= ContentIsOpened How ContentIsDone ProcessData
How ::= (ReadFromNetwork ReadFromNetworkDone) | ReadFromCache
ProcessData ::= { Script | Images }
Script ::= ScriptIsRunning ScriptIsDone { Start }
Images ::= ImageIsOpened ProcessImages
ProcessImages ::= ImageIsOpened | ImageIsDone | Info { ProcessImages }
Info ::= ReadFromNetwork | ReadFromCache | ReadFromNetworkDone

Alternativy, the flow ca be described as follows:

Start: When MMIc_loadURL is called, when a link or key is selected or when a WML timer in a card expires, downloading of data is started.
Data: The data is first opened then processed.
How: The data can either be downloaded from the network or taken from the cache.
ProcessData: When the data has been downloaded it is to be processed. It might be a WML script, a WML deck or any other data (that not will be further processed).
Script: The script is executed. This may lead to that a new download activity is started.
Images: All images of the WML card that is to be, or has been opened, must be downloaded, as well.
ProcessImages: Images will be downloded asyncronously. Start opening as many images as possible, at once. Process incoming images in the order they arrive.
Info: Like data, images are downloaded from the network or taken from the cache. However, it is not possible to determine in what order the status codes will come.

unknownContent

Called when data of non-supported content type is downloaded or taken from the cache.

```
VOID MMIa_unknownContent (UINT8 viewId, const CHAR
*data, UINT16 length, const CHAR *contentType,
const CHAR *URL)
```

`viewId`	The view id received from a call to MMIc_openView.
`data`	The data, which is not NULL terminated. The data is deleted when the function returns.
`length`	The data length.
`contentType`	The contentType is taken from the WSP header [WAP-WSP] and gives the content type of the data. Content types are also defined in [RFC2068].
`URL`	The URL of the source is in order to identify the file that has been downloaded. The string is deleted when the function returns.

passwordDialog

Some content servers require a user id and a password in order to provide the requested data. This function is called in such cases. The function provides the WAP application the realm in which the requested data resist, and its purpose is to open a dialog that that tells that the data is locked and that the user must be authorised in order to get it. The dialog cannot be blocking. Therefore, when the user has finished, a call to MMIc_passwordDialogResponse must be made. However, if the user cancels the dialog operation, the MMIc_stop function should be called, instead of this function.

```
VOID MMIa_passwordDialog (UINT8 viewId, UINT8
dialogId, const CHAR *realm, INT8 type)
```

`viewId`	The view id received from a call to MMIc_openView.
`dialogId`	An id to be used in the corresponding Connector function call.
`realm`	The realm provides the user information in order to decide what id and password this particular server requires. The string is deleted when the function returns.
`type`	The type argument is set to AUTH_SERVER if it is a content server that requires authentication. WAP proxy server authentication is done with this function if the provided authentication data (configAUTH_PASS_GW and configAUTH_ID_GW) not is correct. In that case, the type argument is set to AUTH_PROXY. The WAP application should urge the user to change the configuration variables if this function is called with AUTH_PROXY, in order to avoid calls

of this type.

passwordDialogResponse

If a dialog for user name and password input has been opened (MMIa_passwordDialog) a call to this function must be made when the user closes the dialog. If the user cancels the dialog operation, the MMIc_stop function should be called instead of this function.

```
VOID MMIc_passwordDialogResponse (UINT8 viewId,
UINT8 dialogId, const CHAR *name, const CHAR
*password)
```

`viewId`	The view id received from a call to MMIc_openView.
`dialogId`	An id retrieved from the corresponding Adapter function call.
`name`	User name. The string can be deleted when the function returns.
`password`	User password. The string can be deleted when the function returns.

Constants

Constants that the Notification-functions use are (defined in apimmi.h):

AUTH_SERVER	1
AUTH_PROXY	2

9.4 WML Script dialogs

When WML scripts are running, dialogs can be opened in order to get a response from the user.

promptDialog

Opens a dialog and prompts for user input. The dialog cannot be blocking. Therefore, when the user has finished, a call to MMIc_promptDialogResponse must be made. There is no cancel option for this kind of dialog. The WML script can, however, be cancelled by calling CLNTc_stop.

```
VOID MMIa_promptDialog (UINT8 viewId, UINT8
dialogId, const WCHAR *message, const WCHAR
*defaultInput)
```

`viewId`	The view id received from a call to MMIc_openView.
`dialogId`	An id to be used in the corresponding Connector function call.

`message`	The message to present in the dialog. The string is deleted when the function returns.
`defaultInput`	The parameter contains the initial content for the user input. The string is deleted when the function returns.

promptDialogResponse

If a dialog for user input has been opened (MMIa_promptDialog) a call to this function must be made when the user closes the dialog. If CLNTc_stop is called and the WML script is cancelled, this function has not to be called.

```
VOID MMIc_promptDialogResponse (UINT8 viewId, UINT8
dialogId, const WCHAR *answer)
```

`viewId`	The view id received from a call to MMIc_openView.
`dialogId`	An id retrieved from the corresponding Adapter function call.
`answer`	The user's answer. The string can be deleted when the function returns.

confirmDialog

Displays the given message and two reply alternatives: ok and cancel. Note that cancel not cancels the WML script, it merely passes a cancel message to the script. The dialog cannot be blocking. Therefore, when the user has finished, a call to MMIc_confirmDialogResponse must be made.

```
VOID MMIa_confirmDialog (UINT8 viewId, UINT8
dialogId, const WCHAR *message, const WCHAR *ok,
const WCHAR *cancel)
```

`viewId`	The view id received from a call to MMIc_openView.
`dialogId`	An id to be used in the corresponding Connector function call.
`message`	The message to present in the dialog. The string is deleted when the function returns.
`ok`	The default implementation-dependent ok text may be replaced by alternative text. The string is deleted when the function returns.
`cancel`	The default implementation-dependent cancel text may be replaced by alternative text. The string is deleted when the function returns.

confirmDialogResponse

If a dialog for user confirmation has been opened in a view identified by viewId, (MMIa_confirmDialog) a call to this function must be made when the user selects one option in the dialog. If CLNTc_stop is called and the WML script is cancelled, this function has not to be called.

```
VOID MMIc_confirmDialogResponse (UINT8 viewId,
UINT8 dialogId, BOOL answer)
```

`viewId`	The view id received from a call to MMIc_openView.
`dialogId`	An id retrieved from the corresponding Adapter function call.
`answer`	The user's answer. If the user selected the OK button, TRUE is given and if the user selected the Cancel button, FALSE is given.

alertDialog

Displays the given message to the user. The function must not wait for the user confirmation, but must return immediately. The dialog cannot be blocking. Therefore, when the user has finished, a call to MMIc_alertDialogResponse must be made.

```
VOID MMIa_alertDialog (UINT8 viewId, UINT8
dialogId, const WCHAR *message)
```

`viewId`	The view id received from a call to MMIc_openView.
`dialogId`	An id to be used in the corresponding Connector function call.
`message`	The message to present in the dialog. The string is deleted when the function returns.

alertDialogResponse

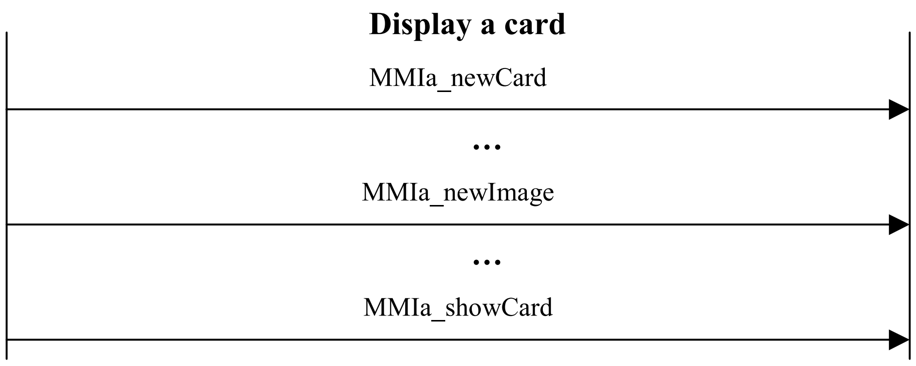
If an alert dialog has been opened (MMIa_alertDialog) a call to this function must be made when the user closes the dialog. If CLNTc_stop is called and the WML script is cancelled, this function has not to be called.

```
VOID MMIc_alertDialogResponse (UINT8 viewId, UINT8
dialogId)
```

`viewId`	The view id received from a call to MMIc_openView.
`dialogId`	An id retrieved from the corresponding Adapter function call.

9.5 WML Cards

A WML deck is composed of cards. One card at time is in focus of the AUS WAP Browser. This is maintained by the following functions.



newCard

Called when a new card is to be displayed. When this call is performed, the WAP application must prepare the view for a new card. The card that is displayed in the view when this function is called must be prepared for removal. It does not, however, need to be removed until the corresponding MMIa_showCard function is called.

```
VOID MMIa_newCard (UINT8 viewId, const WCHAR
*title, BOOL isList, BOOL isRefresh, const CHAR
*URL, BOOL isBookmarkable, const WCHAR * const
*history)
```

`viewId`	The view id received from a call to MMIc_openView.
`title`	The title (NULL if no title is available) may be used if the WAP application has a mean to display it. The string is deleted when the function returns.
`isList`	A hint regarding the structure of the card is also provided through the isList argument. A value of TRUE means that the workflow of the card is naturally organised as a linear sequence, i.e., a set of operations which are naturally processed in the order in which they appears in the deck. isList equal to FALSE means that the workflow of the card can be in any order.
`isRefresh`	If this argument is set to TRUE, the current card is about to be refreshed. This means that the state the current card is in, e.g., if an input field is active or not, can be saved and restored after the card has been fully refreshed (when MMIa_showCard has been called).
`URL`	The URL is given in order of having a way to store a bookmark of the current URL. The string is deleted when the function returns.

`isBookmarkable`	Whether it is possible to have the provided URL as a bookmark or not is given by this argument (TRUE if the card has the newcontext attribute set to true, otherwise FALSE). The URL is given also when it isn't possible to store the URL as a bookmark.
`history`	A list of all card titles, since this function was called with the isBookmarkable set to TRUE, is provided in the this argument. The list is a NULL terminated array of strings. Cards that not have titles are represented as empty strings in this array. The array, as well as the content of each entry of the array, is deleted after the function call.

showCard

Displays a created card. The function is called when no more card content is to be added.

```
VOID MMIa_showCard (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

cancelCard

When an error occurs during a card is displayed, the AUS WAP Browser calls this function instead of proceeding with the display routines and ending with a call to MMIa_showCard.

```
VOID MMIa_cancelCard (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

9.6 WML Keys

newKey

After the WAP application has received a MMIa_newCard call, calls to this function come for the every do element defined in the WML card, about to be opened. They come in the same order, as they are defined in the WML card and at the same position as they have in there. The keys can, in this way, be displayed inline in the card content at the position they have been defined. MMIa_newKey calls for template do elements are performed when the actual content of the card has been given to the WAP application, immediately before MMIa_showCard is called.

```
VOID MMIa_newKey (UINT8 viewId, UINT8 keyId, const
WCHAR *eventType, const WCHAR *label, BOOL
isOptional)
```

`viewId`	The view id received from a call to MMIc_openView.
`keyId`	This key can be referred to in the MMIc_keySelected function with the value of this argument.
`eventType`	The eventType argument identifies the type, e.g., “accept”. All possible values are given in the section Constants, in below. The string is deleted when the function returns.
`label`	If the key, a certain *WML do element* shall be associated with, shall have a label different than default, the label argument gives that string. The argument is set to NULL if the default name is to be used. The string is deleted when the function returns (if not NULL).
`isOptional`	This argument indicates whether the key must be present, or not.

Constants

Constants that this function use in the eventType argument are (defined in apimmi.h):

Event Type	Description
accept	Positive acknowledgement (acceptance).
prev	Backward history navigation.
help	Request for help. Context-sensitive.
reset	Clearing or resetting the WAP application (WML variables and WML application history).
options	Context-sensitive request for options or additional operations.
delete	Delete item or choice.
unknown	Generic event corresponding to the *do* type equal to the empty string (<do type=””>)
x-*	Experimental event. The name of the event is given in the eventName argument. The ‘*’ character is exchanged with the actual event name.
vnd.*	Vendor specific event. The name of the event is given in the eventName argument. The ‘*’ character is exchanged with the actual event name.

keySelected

Called from the WAP application when a key has been pressed.

```
VOID MMIc_keySelected (UINT8 viewId, UINT8 keyId)
```

`viewId`	The view id received from a call to MMIc_openView.
`keyId`	An id retrieved from the function MMIa_newKey.

9.7 WML Text, Images and Layout

9.7.1 Text

newText

This function instructs the WAP application to display a text. All consecutive white space (blanks, carriage-returns and tabs) have been reduced to one blank character. Word wrapping of the string must be performed according to the previous call of MMIa_newParagraph.

```
VOID MMIa_newText (UINT8 viewId, UINT8 textId,
const WCHAR *text, BOOL isLink, const WCHAR
*linkTitle, INT8 format)
```

`viewId`	The view id received from a call to MMIc_openView.
`textId`	An id to been used in the function MMIc_textSelected. This id is set to zero if the text is not a link.
`text`	The text is a zero terminated Unicode string. The string is deleted when the function returns.
`isLink`	If the isLink argument is set to TRUE, the text must be selectable; i.e. the text is a link. A selection of the text must then result in a call to MMIc_textSelected, with the id of this text as an argument.
`linkTitle`	A link may be associated with a title that may be displayed in various ways by the WAP application.
`format`	The format argument holds a value calculated by combining the constants TXT_NORMAL, TXT_SMALL, TXT_BIG, TXT_BOLD, TXT_ITALIC, TXT_UNDERLINE, TXT_EMPHASIS or TXT_STRONG with the logical or operator “|”. TXT_NORMAL, TXT_SMALL and TXT_BIG should never come together (WML actually allows that!). Example of how the format argument is used to determine if TXT_SMALL is set: `if (format & TXT_SMALL)` `setSmallFont();`

```
if (format & TXT_SMALL)
        setSmallFont();
```

textSelected

Shall be called when a text link has been selected.

```
VOID MMIc_textSelected (UINT8 viewId, UINT8 textId)
```

`viewId`	The view id received from a call to MMIc_openView.
`textId`	An id retrieved from the function MMIa_newText.

9.7.2 Images

newImage

Adds an image to a view.

```
VOID MMIa_newImage (UINT8 viewId, UINT8 imageId,
const CHAR *imageData, UINT16 imageSize, const CHAR
*imageType, const WCHAR *altText, const WCHAR
*localSrc, BOOL isLink, const WCHAR *linkTitle,
INT8 vSpace, INT8 hSpace, INT16 width, INT16
height, INT8 isPercent, INT8 align)
```

`viewId`	The view id received from a call to MMIc_openView.
`imageId`	An id to been used in the function MMIc_imageSelected.
`imageData`	The image data. If dynamic updates of images are supported, see the configuration variable configUPDATE_IMAGES, NULL is passed. In that case the image will be added using the MMIa_completeImage function. The imageData memory is deleted when the function returns. If the data is NULL and the configUPDATE_IMAGES variable is set to FALSE, an error has occurred during opening the image.
`imageSize`	Tells how many bytes the image is. This is needed since the data is not zero-terminated. If no image is passed, 0 is given.
`imageType`	Contains the suffix of the image filename (bmp, gif, etc). The string is deleted when the function returns. This variable is NULL if no image data is available at the moment.
`altText`	Can be used if the client not supports or wants to display images. This variable is NULL if the text is not provided in the WML source. If the text is provided, it is deleted when the function returns.
`localSrc`	Can be set to a name of an image, stored locally in the WAP application. If it is so, the WAP application must use that image. If not, the imageData argument shall be used as normally.
`isLink`	If the isLink argument is TRUE, the image must be selectable; i.e. the text is a link. A selection of the image must then result in a call to MMIc_imageSelected, with the id of this image as an argument.
`linkTitle`	The link may contain a title. The title may be displayed in various ways by the WAP application.
`vSpace`	This argument specifies the amount of white space to be inserted

	to the left and right of the image. The default value for this attribute is not specified, but is generally a small, non-zero length. If length is specified as a percentage value (see the argument isPercent), the resulting size is based on the available horizontal or vertical space, not on the natural size of the image. This attribute is only a hint to the WAP application and may be ignored.
`hSpace`	The same as vSpace, with the difference that it controls the horizontal spacing.
`width`	The argument provides the Application an idea of the size of an image so that they may reserve space for it and continue rendering the card while waiting for the image data. Applications may scale images to match these values if appropriate. If length is specified as a percentage value, the resulting size is based on the available horizontal space, not on the natural size of the image. This attribute is only a hint to the Application and may be ignored.
`height`	The same as width, with the difference that it controls the vertical spacing.
`isPercent`	Tells whether the height and width arguments are expressed in percent or not. The argument may be any combination of VSPACE_IS_PERCENT, HSPACE_IS_PERCENT, WIDTH_IS_PERCENT and HEIGHT_IS_PERCENT. NONE_IS_PERCENT is the default value. isPercent is evaluated with logical or "&": `if (format & VSPACE_IS_PERCENT) setVerticalSpaceInPercent(vSpace);`
`align`	Specifies image alignment within the text flow and with respect to the current insertion point. It has three possible values: ALIGN_BOTTOM: means that the bottom of the image should be vertically aligned with the current baseline. This is the default value. ALIGN_MIDDLE: means that the centre of the image should be vertically aligned with the centre of the current text line. ALIGN_TOP: means that the top of the image should be vertically aligned with the top of the current text line.

completeImage

Draws or updates an image identified by imageId if this functionality is supported (see the configuration variable configUPDATE_IMAGES). The image is displayed at its position in the view. If dynamic redrawing of cards cannot be implemented in the WAP application, this function can be implemented empty, since it never will be called.

```
VOID MMIa_completeImage (UINT8 viewId, UINT8
imageId, const CHAR *imageData, UINT16 imageSize,
const CHAR *imageType)
```

`viewId`	The view id received from a call to MMIc_openView.
`imageId`	An id, originating from a former call of the function MMIa_newImage.
`imageData`	The image data. If the data is NULL, an error has occurred during opening of the image. The imageData argument, if not NULL, is deleted when the function returns.
`imageSize`	Tells how many bytes the image is. This is needed since the data is not NULL terminated.
`imageType`	The imageType contains the suffix of the image filename (bmp, gif, etc). The string is deleted when the function returns.

imageSelected

Shall be called when an image link has been selected.

```
VOID MMIc_imageSelected (UINT8 viewId, UINT8
imageId)
```

`viewId`	The view id received from a call to MMIc_openView.
`imageId`	An id retrieved from the function MMIa_newImage.

9.7.3 Layout

newParagraph

This function indicates that a new paragraph shall be started. The MMI behaviour is to insert a line break at this particular place, if not the first one. If it is the first paragraph, only the alignment and wrap mode must be regarded.

```
VOID MMIa_newParagraph (UINT8 viewId, INT8 align,
BOOL wrap)
```

`viewId`	The view id received from a call to MMIc_openView.
`align`	The alignment of the flow in this paragraph may be determined with the align argument. Valid values are ALIGN_LEFT, ALIGN_CENTER and ALIGN_RIGHT. Default is ALIGN_LEFT.
`wrap`	The wrap argument tells whether word wrapping should be used or not. The possible values are TRUE for word wrapping mode and FALSE for non-”word wrapping” mode.

closeParagraph

Closes the current paragraph.

```
VOID MMIa_closeParagraph (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

newBreak

Adds a line break to a view. There is no defined distinction between the line breaks that this function shall produce and the line break that MMIa_newParagraph shall produce.

```
VOID MMIa_newBreak (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

newFieldSet

Tells the view that from now on, an optional frame can be drawn around the following card elements. The field set is closed by a call of the MMIa_closeFieldSet function.

```
VOID MMIa_newFieldSet (UINT8 viewId, const WCHAR
*title)
```

`viewId`	The view id received from a call to MMIc_openView.
`title`	The field set may have a title. It is deleted when the function returns.

closeFieldSet

Closes the current field set in the view.

```
VOID MMIa_closeFieldSet (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

9.7.4 Constants

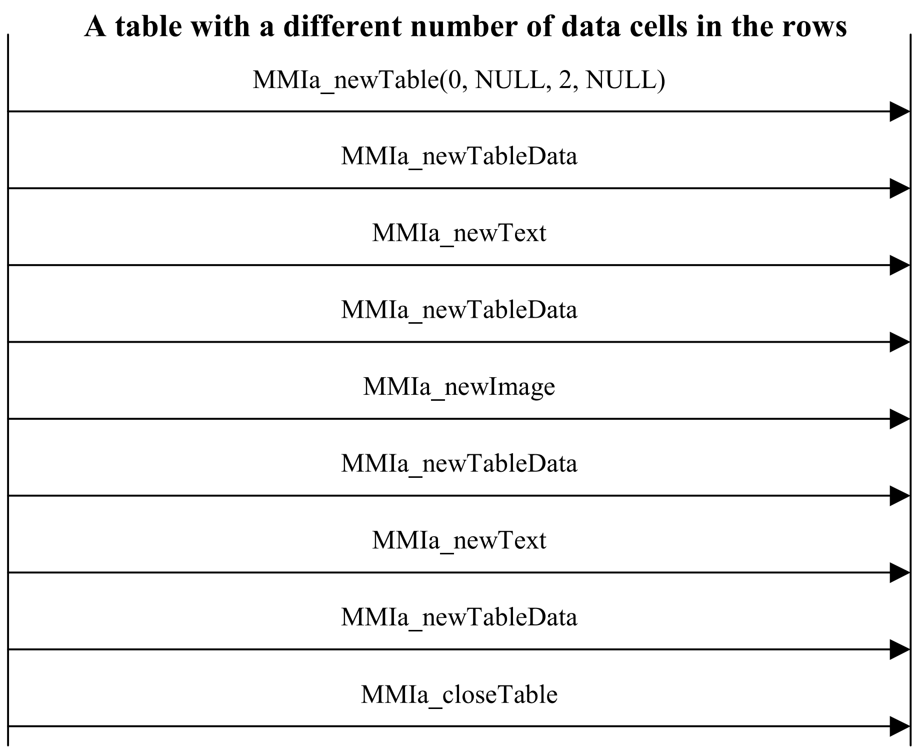
Constants that the text, image and layout functions use are (defined in tapimmi.h):

ALIGN_LEFT	0
ALIGN_CENTER	1
ALIGN_RIGHT	2

ALIGN_BOTTOM	0
ALIGN_MIDDLE	1
ALIGN_TOP	2
NONE_IS_PERCENT	0
WIDTH_IS_PERCENT	1
HEIGHT_IS_PERCENT	2
VSPACE_IS_PERCENT	4
HSPACE_IS_PERCENT	8
TXT_NORMAL	0
TXT_SMALL	1
TXT_BIG	2
TXT_BOLD	4
TXT_ITALIC	8
TXT_UNDERLINE	16
TXT_EMPHASIS	32
TXT_STRONG	64

9.8 WML Tables

A WML table is for the WAP application an optional feature to display. However, the content of the table have to be displayed either the WAP application supports tables, or not. If the WAP application not supports tables, the simplest way to display the content is simply to perform a line break, every time a call of the function MMIa_newTableData comes.



newTable

Adds a new table to a view.

```
VOID MMIa_newTable (UINT8 viewId, const WCHAR
*title, INT8 noOfColumns, const CHAR *align)
```

`viewId`	The view id received from a call to MMIc_openView.
`title`	The table may have a title. It may be used in the presentation of this table. It is deleted when the function returns.
`noOfColumns`	The argument noOfColumns is assigned with the numbers of data cells the rows of the table will contain. Each one of the columns might be left (default), centred or right aligned. The letters ‘L’, ‘C’ and ‘R’ are used as alignment descriptors. Columns are described from left to right. If the number of alignment descriptors is less than the number of columns, the default alignment (left) should be used for the last columns that were not described. If no descriptor exist, the argument is NULL. Ex: a three-column table with the leftmost column left aligned, the middle column centred aligned and the rightmost column right aligned is described with the string “LCR”. All strings are deleted when the function returns.

newTableData

Indicates that a new data cell is to be started in the table, currently being diplayed. The data cells shall be displayed from left to right, in the order they come. The number of cells in a row is determined in the noOfColumns argument in the MMIa_newTable function call. Any number of calls of MMIa_newText, MMIa_newImage and MMIa_newBreak may come, after MMIa_newTableData has been called. The cell can be empty. Either a MMIa_newTableData call or a MMIa_closeTable call terminates the data cell.

```
VOID MMIa_newTableData (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

closeTable

Indicates that no more data cells will come and that the table is finished.

```
VOID MMIa_closeTable (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

9.9 WML Menus

newSelect

Adds a single or multiple choice menus for a variable number of option elements, to a view. The options are added with the function MMIa_newOption.

```
VOID MMIa_newSelect (UINT8 viewId, const WCHAR
*title, BOOL multiSelect, INT8 tabIndex)
```

`viewId`	The view id received from a call to MMIc_openView.
`title`	The menu may have an optional title, which can be used by the WAP application in the presentation of the menu. The title is deleted when the function returns.
`multiselect`	FALSE means a single choice menu and TRUE means a multiple-choice menu.
`tabIndex`	The tabIndex tells which order this particular menu has in the card. If the tabIndex is zero, the menu is either first in order or whiteout order. The WAP application may ignore the tab index.

closeSelect

Closes the selection menu.

```
VOID MMIa_closeSelect (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

newOption

Adds an option with a label to a selection menu. The option is then selected or deselected by calls to the function MMIc_optionSelected. If an option is set, and the user selects that particular option, the function shall be called anyway, even if a selection of that option not has a visual effect.

```
VOID MMIa_newOption (UINT8 viewId, UINT8 optionId,
const WCHAR *label, const WCHAR *title, BOOL
isSelected)
```

`viewId`	The view id received from a call to MMIc_openView.
`optionId`	An id of the option is given in the optionId argument. It is to be used in the function MMIc_optionSelected when this option has been selected. The label is deleted when the function returns.
`label`	The label argument holds the text that shall be displayed in the option. The title is deleted when the function returns.
`title`	The WML application author sometimes gives a title to an option. The WAP application might use the title for additional visual feedback.
`isSelected`	The option is initially set if isSelected is TRUE.

newOptionGroup

Tells that the following options are a submenu of the current menu. The submenu may have a title.

```
VOID MMIa_newOptionGroup (UINT8 viewId, const WCHAR
*label)
```

`viewId`	The view id received from a call to MMIc_openView.
`label`	The option group may have a label. The label is deleted when the function returns.

closeOptionGroup

Closes the option group initiated by the former call to MMIa_newOptionGroup.

```
VOID MMIa_closeOptionGroup (UINT8 viewId)
```

`viewId`	The view id received from a call to MMIc_openView.

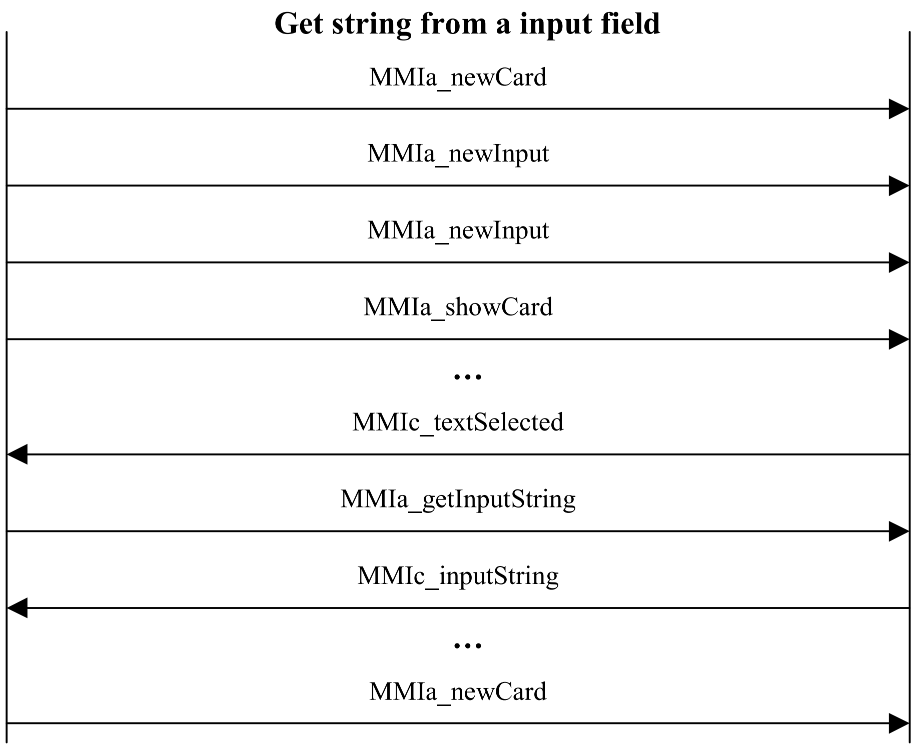
optionSelected

Changes the state of an option in a selection menu. The function is used when an option is selected or deselected in multiple-choice and single-choice menus and when an option is selected in single-choice menus. It shall be called also when the menu is single-choice, and the option is already on.

```
VOID MMIc_optionSelected (UINT8 viewId, UINT8
optionId)
```

`viewId`	The view id received from a call to MMIc_openView.
`optionId`	The id of the option that has been selected.

9.10 WML Input fields



newInput

The function adds an input field to a view.

```
VOID MMIa_newInput (UINT8 viewId, UINT8 inputId,
const WCHAR *title, const WCHAR *text, BOOL
isPassword, BOOL emptyOk, const WCHAR *format, INT8
size, INT8 nChars, INT8 tabIndex)
```

`viewId`	The view id received from a call to MMIc_openView.
`inputId`	The id of the input field. It will be used in calls of the functions MMIa_getInputString and MMIc_inputString.

`title`	A title may be given to the input field. If no title exist, NULL is passed in the argument. It may be used by the WAP application in the presentation of the input field. It is deleted when the function returns.
`text`	The text argument is to be displayed in the input field if it conforms to the description in the format argument. Read more about this argument in the format argument. It is deleted when the function returns.
`isPassword`	If TRUE, the argument indicates that the entered characters should be hidden.
`emptyOk`	The emptyOk argument tells if an empty string is accepted as input. A MMIa_getInputString call is done when a new card is about to be downloaded. If the input field text is empty and this argument is set to FALSE, the WAP application must prompt the user for input.
`format`	How the text shall be formatted is to be read in the format argument. The following rules must be followed when this function is called: 1. If the text argument conforms to the rules in the format string, display the text. 2. If the text argument not conforms to rule 1 and the default text argument conforms to rule 1, display the default text. 3. If neither the text argument nor the default text argument conforms to rule 1, display the empty string. This string is deleted when the function returns.
`size`	Tells how many characters that should be visible.
`nChars`	How many characters the input field should be able to handle is given in the nChars argument. If it is equal to –1, any number of characters is accepted.
`tabIndex`	The tabIndex tells which order this particular widget has in the card. If the tabIndex is zero, the menu is either first in order or whiteout order. The WAP application may ignore the tab index.

The format string, given in the argument format, deserves more explanation. It is composed as a set formatting control characters specifying the data format expected to be entered by the user. The default format is "*M", i.e., any number of characters. The format codes that can be used in such a string are:

- **A** entry of any upper-case alphabetic or punctuation character (i.e., upper-case non-numeric character)
- **a** entry of any lower-case alphabetic or punctuation character (i.e., lower-case non-numeric character)
- **N** entry of any numeric character
- **X** entry of any upper case character

- **x** entry of any lower-case character
- **M** entry of any character; the user agent may choose to assume that the character is upper-case for the purposes of simple data entry, but must allow entry of any character
- **m** entry of any character; the user agent may choose to assume that the character is lower-case for the purposes of simple data entry, but must allow entry of any character
- ***f** entry of any number of characters; f is one of the above format codes and specifies what kind of characters can be entered. *Note: This format may only be specified once and must appear at the end of the format string*
- ***nf*** entry of n characters where n is from 1 to 9; f is one of the above format codes and specifies what kind of characters can be entered. *Note: This format may only be specified once and must appear at the end of the format string*
- **\c** display the next character, c, in the entry field; allows quoting of the format codes so they can be displayed in the entry area.

Examples of format strings are:

NNNNNN Six digits (could also been expressed as 6N)

NN\-NN\-NN A date string. The minus characters are static characters and cannot be omitted.

getInputString

When the user performs an operation, which causes the AUS WAP Browser to navigate to another card, for instance when a link is selected, this function is called in order to get the user entered string in the input field. The WAP application must call the Connector function MMIc_inputString in order to supply the AUS WAP Browser with the string. The string must be formatted according to rules originating from the MMIa_newInput. The string shall not be returned to the Generic WAP Clien if not the text conforms to the format specified. If the input field text is empty and the isEmpty argument in the MMIa_newInput was set to TRUE, the WAP application must prompt the user for new input.

```
VOID MMIa_getInputString (UINT8 viewId, UINT8
inputId)
```

`viewId`	The view id received from a call to MMIc_openView.
`inputId`	The id of the input field of which the string is requested.

inputString

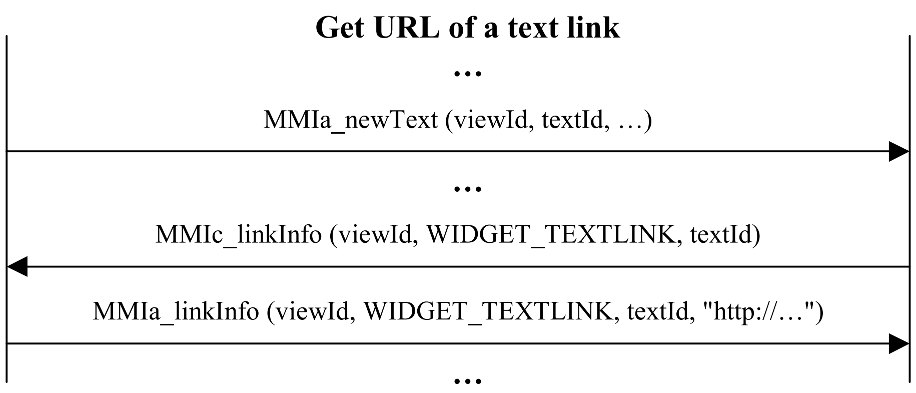
This function is called in order to return the string requested in a MMIa_getInputString call. It is an error to not respond on this function. This will block the downloading activity, when it is dependent upon the input string. However, if the answer of any reason not has been received by the AUS WAP Browser, the MMIc_stop, MMIc_back and MMIc_loadURL functions will cancel the download operation. They will leave the AUS WAP Browser in a stable state again.

```
VOID MMIc_inputString (UINT8 viewId, UINT8 inputId,
const WCHAR *text)
```

`viewId`	The view id received from a call to MMIc_openView.
`inputId`	The id of the input field of which the string is requested.
`text`	The string from the input field. If the string is empty, NULL is returned in the text argument. If the string is formatted according to rules originating from the MMIa_newInput, the formatting characters should be including as well. The string can be deleted after the call.

9.11 The URL of a link

Since an URL that contains WML variables can differ from time to time (depending on the current value of the variable), the URL cannot be given for the functions MMIa_newText, MMIa_newImage, MMIa_newOption and MMIa_newKey. The functions in this section provide the WAP application with a mean to get the URL that a link is associated with, at any moment after the link has been displayed.


Get URL of a text link
…
MMIa_newText (viewId, textId, …)
…
MMIc_linkInfo (viewId, WIDGET_TEXTLINK, textId)
MMIa_linkInfo (viewId, WIDGET_TEXTLINK, textId, "http://…")
…

linkInfo

Used by the WAP application when the URL, a certain link is associated with, shall be displayed.

```
VOID MMIc_linkInfo (UINT8 viewID, UINT8 widgetType,
UINT8 widgetID)
```

`viewId`	The view id received from a call to MMIc_openView.
`widgetType`	There exist four types of WML elements that can be links:

- text links
- image links
- options

- keys

	Each of them has a predefined type id that shall be assigned this argument (see the constants in below).
`widgetId`	The last argument, widgetId, shall be assigned the id of the widget (i.e. textId, imageId, optionId or keyId) of which the URL shall be retrieved. The URL will be retrieved by the Adapter function call MMIa_linkInfo.

linkInfo

The AUS WAP Browser calls this function in order to return an URL that the WAP application has requested in a MMIc_linkInfo call.

```
VOID MMIa_linkInfo (UINT8 viewID, UINT8 widgetType,
UINT8 widgetID, const CHAR* URL)
```

`viewId`	The view id received from a call to MMIc_openView.
`widgetType`	The same values as in the corresponding MMIc_linkInfo call.
`WidgetId`	The same values as in the corresponding MMIc_linkInfo call.
`URL`	The argument URL is set to the current URL. The string is deleted when the function returns.

Constants

Constants that these functions use for the widgetType are (defined in tapimmi.h):

WIDGET_IMAGELINK	1
WIDGET_TEXTLINK	2
WIDGET_DOLINK	3
WIDGET_OPTIONLINK	4

10 Client API

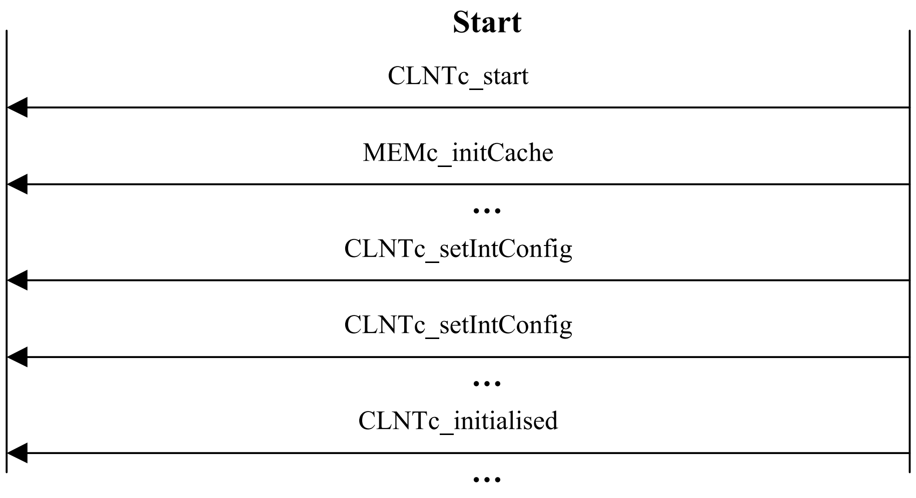
This API defines the general interface between the WAP application and the AUS WAP Browser. The functionality cover areas like:

- Start, initialise and closing down
- Control of execution
- Time
- Dynamic configuration variables
- Downloading non supported content types (vcard, vcalendar, etc)
- File interface (file://)
- Other interfaces (e.g. mailto:)

10.1 Control of the AUS WAP Browser

10.1.1 Start and initialise

The AUS WAP Browser must be notified to start. The AUS WAP Browser must also be notified when its time to close down.



start

This function is used to start the AUS WAP Browser. If the AUS WAP Browser fails to start, the CLNTa_error function is called. It is important that this function is called before any other Connector function is called. The two functions CLNTc_run and CLNTc_wantsToRun must not be run before this function has been called. It is, however, free to call the other initialising Connector functions, i.e., MEMc_initCache, CLNTc_setStrConfig and CLNTc_setIntConfig. And when they have been called, the CLNTc_initialised function shall be called.

```
VOID CLNTc_start (VOID)
```

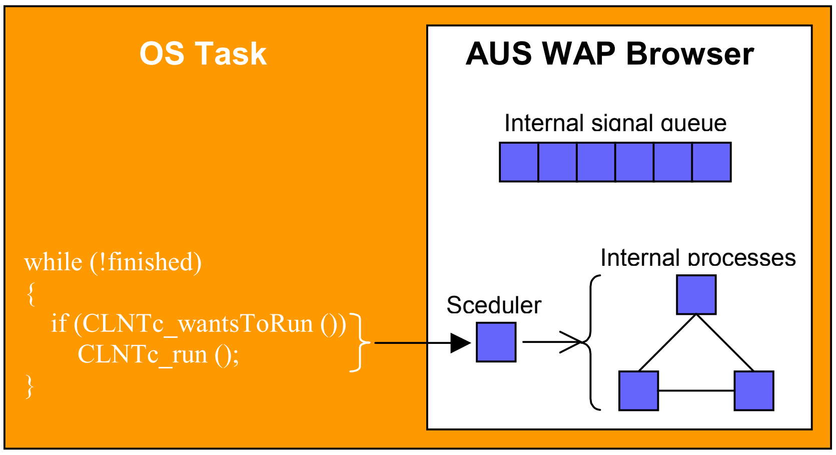
initialised

This function shall be used when the AUS WAP Browser has been started and initialised and when it is reconfigured. Initialised means that the Connector functions MEMc_initCache, CLNTc_setStrConfig and CLNTc_setIntConfig have been called. Reconfigured means that one or several configuration values have been changed (with CLNTc_setStrConfig and CLNTc_setIntConfig). The function shall after that be called in order to tell the AUS WAP Browser that the reconfiguration is finished and that the new value or values shall be used instead.

```
VOID CLNTc_initialised (VOID)
```

10.1.2 Control of execution

The AUS WAP Browser is event driven. An event is sent to the AUS WAP Browser by a Connector function call. A special Connector function shall be called on a regular basis, in order to process the events.

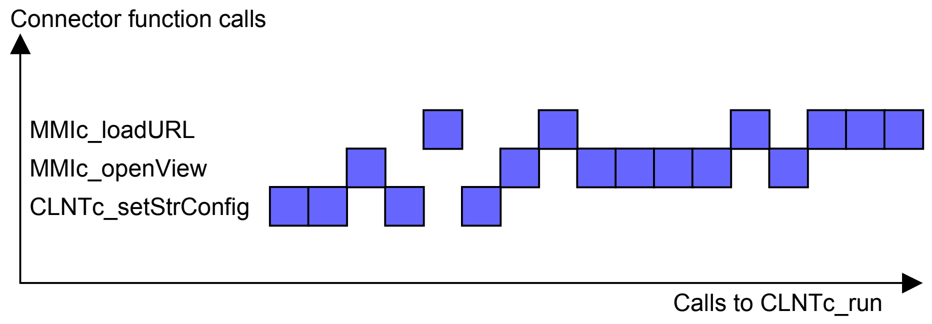


The picture above gives a very simple example of how the Connector functions might be called. The Connector function CLNTc_run processes the events (in form of internal signals) in the internal message queue. Connector functions or internal processes put the signals there.

If many Connector functions are called in a sequence, the signal queue will contains many signals. Since they are processed in the order they are put in the signal queue, and they requires different amount of execution to fulfil the task, they are executed simultaneously.

Read more about this in the Overview chapter, at the beginning of this manual.

The image below illustrates how three different Connector function tasks are processed by the function CLNTc_run. Each blue box represents execution of one state in one internal process. CLNTc_setStrConfig requires execution of four states, according to the picture.



In order to find out whether there is anything to process the Connector function CLNTc_wantsToRun shall be used. It checks if there are any signals in the internal signal queue.

run

This function runs the AUS WAP Browser. The call to it should be done at a place where processing time can be guaranteed continuously. The priority of the RTOS task that runs the AUS WAP Browser is not required to be high. A normal priority is in most cases more than enough. The AUS WAP Browser gains in high priority when it executes WML script and parsing WML decks (when opening WML decks). Otherwise, the priority can be very low. Since there is no way to distinguish between different kind of execution in the AUS WAP Browser, a trade-off must be made. If WMLS execution and opening of WML decks is very important tasks a high priority should be chosen. Otherwise a normal, or even low priority (only if the tasks is of minor importance) can be chosen.

Note 1: It is important that CLNTc_start is called before this function is called.

Note 2: When the AUS WAP Browser not protects the data that is accessed by the Connector functions, it must be ensured that not more than one Connector function is called at once.

```
VOID CLNTc_run (VOID)
```

Example:

Example of usage in a tight loop that is iterated forever:

```
While (notQuit)
{
    RTOS_signal *s;

    /* Get RTOS signal from RTOS signal queue for this
       RTOS process */
    While (s = RTOS_getSignal())
    {
        switch (s->kind)
        {
            …
            case loadURL:
                MMIc_loadURL(s->…);
            …
        }
    }

    if (CLNTc_wantsToRun())
        CLNTc_run();
}
```

This loop loads the CPU only when the RTOS signal queue are checked and when the AUS WAP Browser has something to do:

```
    While (notQuit)
    {
        RTOS_signal *s;

        /* Get RTOS signal from RTOS signal queue for this
           RTOS process */
        While (s = RTOS_getSignal())
        {
            switch (s->kind)
            {
                …
                case loadURL:
                    MMIc_loadURL(s->…);
                …
            }
        }

        if (CLNTc_wantsToRun())
            CLNTc_run();
        else
            /* The time the AUS WAP Browser can
               afford to lose without loss of accuracy */
            Sleep(100); /* milliseconds */
    }
```

When the AUS WAP Browser is event driven, the RTOS process that hosts this loop may be put to sleep when the AUS WAP Browser is inactive. The example above uses 100 milliseconds, which is the time the AUS WAP Browser can lose without significant loss of accuracy. This is because of WML timers that have a resolution of one 10:th of a second. There is actually not a requirement that this accuracy must be held. Furthermore, the only thing that is affected when a long sleeping time is used is the wake-up time. When the RTOS process has started to execute again, it will not stop until there is nothing to do in the AUS WAP Browser.

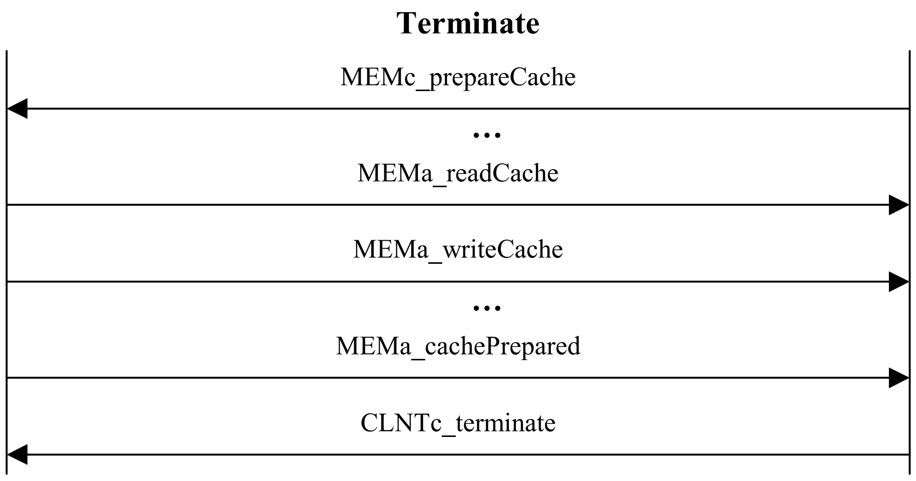
wantsToRun

This function shall be used in order to avoid unnecessary calls to CLNTc_run. If this function returns TRUE, the AUS WAP Browser has not processed everything yet, and the CLNTc_run shall be run again. If it returns FALSE, there is no need for calling CLNTc_run. It will only return TRUE again if a new call to a Connector function is performed.

Note: It is important that CLNTc_start has been called in the initialisation phase, before this function is called.

```
BOOL CLNTc_wantsToRun (VOID)
```

10.1.3 Closing down



terminate

This function is used when terminating the AUS WAP Browser. Before a call to this function is done, be sure the function MEMc_prepareCache has been called first and that the acknowledgement has been received (MEMa_cachePrepared). The termination is regarded as finished when the function CLNTa_terminated is called.

```
VOID CLNTc_terminate (VOID)
```

terminated

This function is used when the AUS WAP Browser termination is done. It is the acknowledgement on the CLNTc_terminate function call.

```
VOID CLNTa_terminated (VOID)
```

10.1.4 Suspend and resume

There are occasions when the AUS WAP Browser could be suspended. For instance when a voice call is to be set-up when the AUS WAP Browser is used. If the WAP browser shall be put in the background during the voice call, the AUS WAP Browser should be suspended during that time. When the voice call terminates the AUS WAP Browser should be resumed again.

There are no Connector functions for suspending and resuming the AUS WAP Browser. The WAP application must stop calling CLNTc_run when suspending and start calling it again when resuming. All other Connector functions can still be called. They will be processed when the AUS WAP Browser is resumed.

If the user has activated the AUS WAP Browser by selecting a hyper link or such like, and the request for a new WML deck has been sent over the network, a timer has been set to expire after, say 60 seconds. This is done with CLNTc_setTimer, CLNTa_timerExpired and the time interval is set with the configuration variable configTIMEOUT. If the AUS WAP Browser is suspended during this timer is active, the timer ought to be suspended as well. If it is not, the timer will most likely expire when the AUS WAP Browser is suspended. In that case will the user activated request time out immediately when the AUS WAP Browser is resumed. On the other hand, if the timer is suspended and then resumed when the AUS WAP Browser is resumed, it is not likely that the requested WML deck will arrive. So the time out will occur anyway. This holds, of course only when the bearer is UDP, because the UDP bearer is normally set-up over a data call. This means that the IP network will be down during the voice call.

The recommendation is therefore to suspend and resume timers only when SMS and USSD are the active bearers. If UDP is the active bearer, time out of requests is practically impossible to avoid.

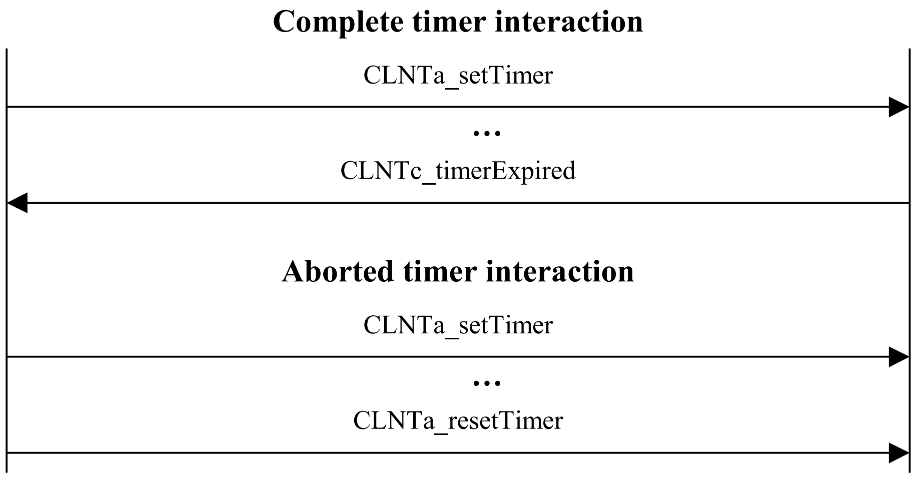
10.2 Time

currentTime

The function returns seconds ellapsed since 1970-01-01 00.00.00. The time can be of any format, GMT as well as local time (which is assumed as default). A configuration variable in confvars.h (cfg_wae_ua_current_time_is_gmt) is by default set to 0 (i.e. the time is GMT). If the implementation of this function returns the time in GMT, the configuration variable shall be set to 1.

```
UINT32 CLNTa_currentTime (VOID)
```

10.3 Timers



setTimer

The function is called when a timer shall be started. There will never be more than one active timer, at once. If the timer not is reset, i.e. aborted by a

CLNTa_resetTimer call, it will expire. When it expires, the function CLNTc_timerExpired shall be called.

```
VOID CLNTa_setTimer (UINT32 timeInterval)
```

`timeInterval`	The time is given as an interval of 100 millisecond units, e.g., if a timer shall expire in one second an interval of 10 is given.

timerExpired

The function is called when the timer, initiated by a previous call to CLNTa_setTimer, has expired.

```
VOID CLNTc_timerExpired (VOID)
```

resetTimer

The function is called when a timer, previously started with a call to CLNTa_setTimer, shall be aborted.

```
VOID CLNTa_resetTimer (VOID)
```

10.4 Configuration

When the Generic WAP Stack has been started, certain configuration variables have to be set. These variables can then be changed during run time. The function CLNTc_initialised must be called after the last variable in this case, as well.

setIntConfig

Called by the WAP application at start up and when a configuration value is changed. When the last call, in a sequence of many, call of this function has been done, it is important that CLNTc_initialised is called again.

```
VOID CLNTc_setIntConfig (UINT8 viewId, ConfigInt
kind, UINT32 value)
```

`viewId`	Each view has its own set of configuration values; therefore must a view id be provided with the function. If the view id is zero, the configuration value will be used by all views. The view id has been received from a call to MMIc_openView.
`kind`	The valid kinds of configuration variables are found in the table below. The integer kinds are all of the ConfigInt type.
`value`	The value the variable shall take.

setStrConfig

Called by the WAP application at start up and when a configuration value is changed. When the last call, in a sequence of many, call of this function has been done, it is important that CLNTc_initialised is called again.

```
VOID CLNTc_setStrConfig (UINT8 viewId, ConfigStr
kind, const CHAR *value, UINT8 length)
```

`viewId`	Each view has its own set of configuration values; therefore must a view id be provided with the function. If the view id is zero, the configuration value will be used by all views. The view id has been received from a call to MMIc_openView.
`kind`	The valid kinds of configuration variables are found in the table below. The string kinds are all of the ConfigStr type.
`value`	The value the variable shall take. The memory may be deleted when the function returns.
`length`	The length of the string (a terminating zero byte shall not be counted) is to be given in the length attribute.

Type definitions

The Client API uses types that are defined in the header file capiclnt.h:

ConfigInt	The type of the enumerator constants, taken by the function CLNTc_setIntConfig.
ConfigStr	The type of the enumerator constants, taken by the function CLNTc_setStrConfig.

Constants

Configuration variables that the Client API defines (in capiclnt.h) are:

Name	**Type**	**Description**
configBEARER	ConfigInt	Currently supported bearers are: UDP = 0, USSD = 2 and SMS = 3.
configUDP _IP_SRC	ConfigStr	Internet addresses in IP's network order (bytes ordered from left to right)
configUDP _IP_GW	ConfigStr	Internet addresses in IP's network order (bytes ordered from left to right)
configSMS_C	ConfigStr	BCD encoded msisdn number [GSM0340] of SMS center

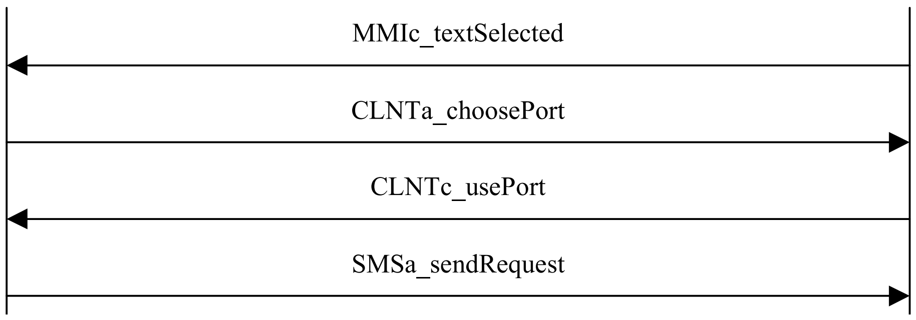
configSMS_GW	ConfigStr	BCD encoded msisdn number [GSM0340] for WAP gateway server (SME)
configUSSD_C	ConfigStr	BCD encoded msisdn number [GSM0340] for USSD center
configUSSD_SC_TYPE	ConfigInt	Currently supported service code types are: none= 0, msisdn= 1 and IP= 2.
configUSSD_SC	ConfigStr	Three types of service codes, depending on the value of the CONFIG_INT_USSD_SC_TYPE variable. Examples: no SC. “” msisdn: BCD encoded msisdn number [GSM0340] IP: Internet addresses in IP’s network order (bytes ordered from left to right)
configHISTORY_SIZE	ConfigInt	How many URLs should be held in the history?
configTIMEOUT	ConfigInt	The time in seconds the client can wait, when downloading has stalled, before the transaction is cancelled. If a IP network is to be set-up when a UDP request is first done, this constant should hold a time interval enough to both set-up the network and wait for the requested WML deck. After the first request it can be reset to its normal time interval again.
configAUTH_PASS_GW	ConfigStr	The password for the WAP proxy server.
configAUTH_ID_GW	ConfigStr	The user id for the WAP proxy server.
ConfigWSP_Language	ConfigStr	A string that contains WSP codes that describe what languages the WAP application is able to handle. The codes shall be given as encoded octets, as defined in [WAP-WSP].
configCACHE_AGE	ConfigInt	The time in seconds a cached item shall be in the cache, if no “expires date” is given with the downloaded

		item. This variable is only possible to set for all views.
configCACHE_MODE	ConfigInt	Supported cache modes are: 0: If the item in cache has expired, always check if a newer version of the item is available on the server. 1: If the item in cache has expired, check the first time after the AUS WAP Browser has been started, if a newer version of the item is available on the server. 2: Never check if a newer version is available on the server. Always use cached version. This variable is only possible to set for all views.
configDISPLAY_IMAGES	ConfigInt	1 if images can be displayed, 0 otherwise.
configUPDATE_IMAGES	ConfigInt	1 if images can be displayed after the card has been displayed, 0 otherwise.
configQ_OTA	ConfigInt	1 if "over the air" questions shall be issued, 0 otherwise.
configSTACKMODE	ConfigInt	Supported stack modes are: Connection-less MODE_CL_WSP = 9200 Connection-less with security layer MODE_CL_WTLS = 9202 Connection-less with WTA MODE_CL_WTA = 30 Connection mode MODE_CO_WSP = 9201 Connection mode with security layer MODE_CO_WTLS = 9203 Connection mode with WTA MODE_CO_WTA = 60 All constants are defined in capiclnt.h.

10.5 Client Port Management

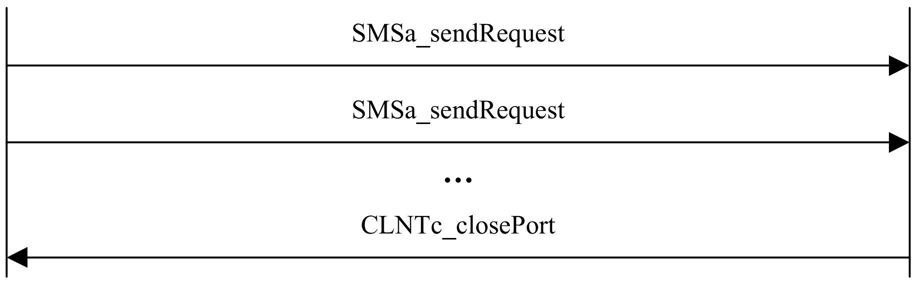
In order to provide the AUS WAP Browser with the client port number that has been opened for a certain bearer, and to control the usage of it, the following set of functions are to be used.

First download from a view



In the diagram above, the AUS WAP Browser requests a client port in order to be able to perform the request operation over SMS. In case that the current bearer is UDP, a setup of the network might be nessecary to do, before the AUS WAP Browser is notified which port to use.

Closing a port



The client port is after that used in all calls to the sending operation with the current bearer. The port shall be closed when:

- When the network is about to be closed
- When the network has already been closed
- When the view is about to be closed

During the time when a client port is considered open, the protocol stack may be re-configured at any time by the WAP application. The client port will be re-used in the new configuration, even if the bearer is changed or if the protocol stack mode is changed.

choosePort

After a view has been opened, the first time a request is about to be sent over the bearer for that view, this function will be called. The WAP application shall choose a client port number to do the request over. The bearer has to be setup, if that not has been done previously. When the bearer is setup, and a client port has been opened, the function CLNTc_usePort is to be called.

```
VOID CLNTa_choosePort (UINT8 viewId, UINT8 id)
```

`viewId`	The view id retrieved from a previous call of MMIc_openView or the default view id of the content handler (referred to by the constant CONTENT_VIEW).
`id`	An id of this call. It is to be used in the corresponding call to CLNTc_usePort.

usePort

This function is called by the WAP application when all preparations, as a response to a call to the function CLNTa_choosePort, have been finnished. The client port number will be used in subsequent calls to send operations of the bearer APIs. It is valid for usage by the AUS WAP Browser until CLNTc_closePort is called. After that, CLNTa_choosePort must be called again, if data is about to be requested over the bearer.

```
VOID CLNTc_usePort (UINT8 viewId, UINT8 id, UINT16
port, BOOL success)
```

`viewId`	The view id retrieved from a previous call of MMIc_openView or the default view id of the content handler (128).
`id`	The id from the corresponding call of CLNTa_choosePort. The client port numbers must be unique. There cannot exist two views with equal port numbers.
`port`	The client port number that has been choosen.
`success`	If CLNTa_choosePort succeds, this pargument is set to TRUE. Otherwise, for instance if no IP-network was able to setup, this argument is set to FALSE.

closePort

When the port number for a view becomes invalid, the AUS WAP Browser needs to be notified about that in order to be able to terminate ongoing activities. This situation arises for instance when the IP-network is about to be, or has already been disconnected. Function calls of the sending operations of the bearer APIs, when the bearer is disconnected shall be ignored. Note that this function must be called before MMIc_closeView is called, if a client port still is in use by the view.

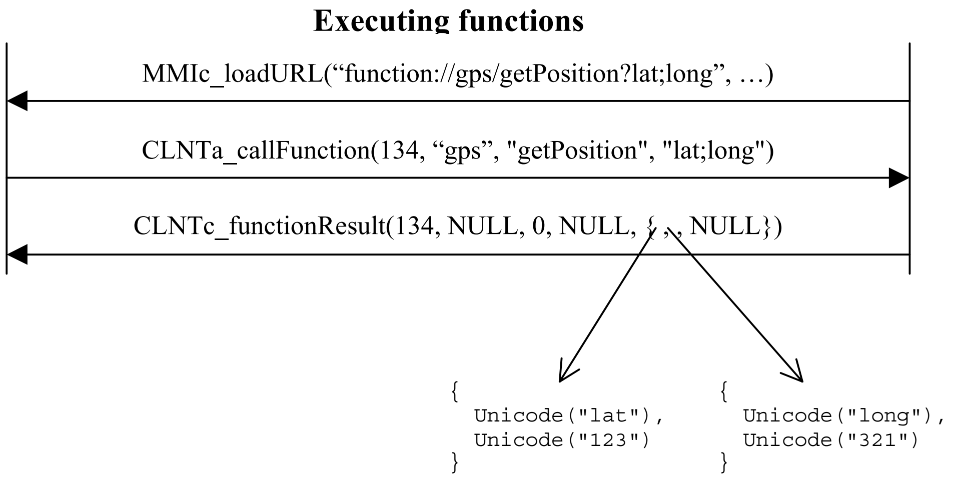
```
VOID CLNTc_closePort (UINT16 port)
```

`port`	The port number that is about to be closed, or in certain cases, has already become closed.

10.6 Support of local functions

It is possible to execute a function on the Host Device by using a URL identifying a function. The URLs should be stated on the following form:

```
<go href="function://device/function?variables">
```


Executing functions
MMIc_loadURL(“function://gps/getPosition?lat;long”, …)
CLNTa_callFunction(134, “gps”, "getPosition", "lat;long")
CLNTc_functionResult(134, NULL, 0, NULL, { , , NULL})
{ Unicode("lat"), Unicode("123") }
{ Unicode("long"), Unicode("321") }

callFunction

Called by the AUS WAP Browser when a function on a device should be executed. The original URL is splitted and stored in the parts *device*, *function* and *attributes*. The parts are not URL decoded. See the figure above for a reference.

```
VOID CLNTa_callFunction (UINT8 functionId, const
CHAR *device, const CHAR *function, const CHAR
*attributes)
```

`functionId`	The application should respond to the AUS WAP Browser by calling CLTNc_functionResult with the functionId having the same value as this argument.
`device`	The device, which is addressed by the URL (see the figure above for a reference). The string is deleted after the function call.
`function`	The function on the device, which is addressed by the URL (see the figure above for a reference). The string is deleted after the function call.
`attributes`	The device, which is addressed by the URL (see the figure above for a reference). The list with its content is deleted after the function call.

functionResult

Called by the WAP client as a response to a CLNTa_callFunction call.

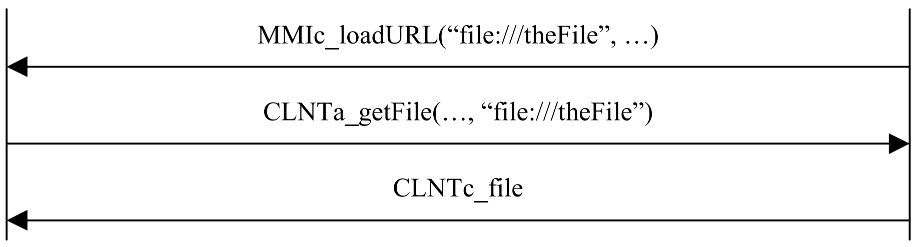
VOID CLNTc_functionResult (UINT8 functionId, const CHAR *data, UINT16 length, const CHAR *contentType, const variableType * const *variables)

`functionId`	The functionId corresponds to the id sent in the adapter function call.
`data`	The data argument holds the content of a file, if not set to NULL. The data can be deleted after the function call.
`length`	The length of the data is given by the length argument.
`contentType`	The content type should be set to "application/vnd.wap.wmlc" for WML files and "application/vnd.wap.wmlscriptc" for WML script files. For images the content should be set to "image/xxx", where xxx gives the image file type ("xbmp", "gif" or "jpg", for instance). The AUS WAP Browser in the MMIa_newImage uses the image type and MMIa_completeImage calls. The string can be deleted after the function call.
`variableType`	If there are variables to set from the function (for instance, "lat" and "long" from the figure above), they should be returned to the AUS WAP Browser in a NULL-terminated list of pointers to C structures of the type variableType, which is declared in capiclnt.h. The variables are set before the data is processed (navigation to the returned WML deck, for instance). The list and its content can be deleted after the function call.

10.7 Support of local files

It is possible to open a file on the device by using the MMIc_loadURL function with a URL to a local file (file://[host | "localhost"]/path). If a card from a local file contains relative links to scripts and images, these content types will be fetched from local files as well.

Getting files


MMIc_loadURL("file:///theFile", …)
CLNTa_getFile(…, "file:///theFile")
CLNTc_file

getFile

Called by the AUS WAP Browser with an URL referring to a local file (file:/// …). The result is to be with the corresponding function CLNTc_file.

```
VOID CLNTa_getFile (UINT8 fileId, const CHAR *URL)
```

`fileId`	The file should be returned to the AUS WAP Browser by calling CLTNc_file with the fileId given with this argument.
`URL`	The URL of the file, which is requested. The URL is deleted after the function call.

file

Called by the WAP application as a response to a CLNTa_getFile call.

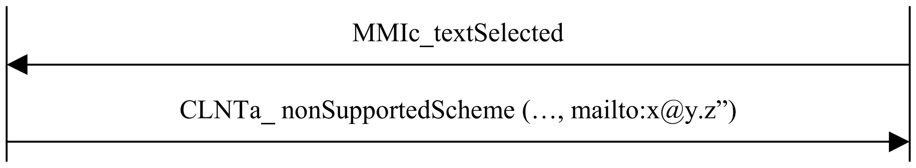
VOID CLNTc_file (UINT8 fileId, CHAR *data, UINT16 length, const CHAR *contentType);

`fileId`	Should be set to the file id retrieved from a CLTNa_getFile call.
`data`	The argument data is to be set to the content of a local file on the device. The data is deleted after the function call.
`length`	The length of the data is to be assigned to the length argument.
`contentType`	The contentType argument is to be assigned the kind of content the data argument is assigned. contentType should be set to "application/vnd.wap.wmlc" for WML files and "application/vnd.wap.wmlscriptc" for WML script files. For images the content should be set to "image/xxx", where xxx gives the image file type ("xbmp", "gif" or "jpg", for instance). The AUS WAP Browser, in the MMIa_newImage and MMIa_completeImage calls, uses the image type. If the file is not found, the contentType should be set to NULL.

10.8 Support of other URL schemes

The WAP application may have support of other URL scheme types than the AUS WAP Browser handles (namely http, file, wtai, function and wapdevice). In that case, the following routines may be used to hook on this extended functionality.

Connecting e-mail applications to WML links



nonSupportedScheme

Called by the AUS WAP Browser when an URL is of a non-supported type (e.g. mailto:). The AUS WAP Browser supports only the http, file, wtai, function and wapdevice schemes.

```
VOID CLNTa_nonSupportedScheme (UINT8 viewId, const
CHAR *URL)
```

`viewId`	The view id received from a call to MMIc_openView.
`URL`	The non-supported URL. The URL is deleted after the function call.

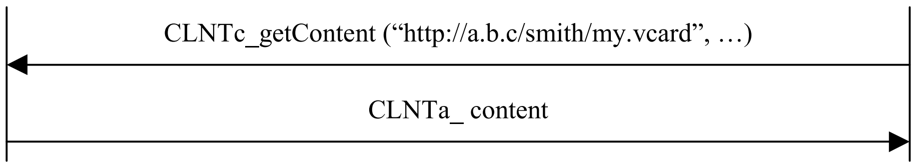
10.9 Retrieval of arbitrary content types

If other content types than WML and WMLS should be downloaded, without going through WAE, these functions should be used. The function MMIa_unknownContent will not be called when these functions are used.

In order to use these functions, a view does not have to be opened. However, the configuration variables for the addresses of the client and the WAP gateway need to be set. The bearer must be chosen, as well. This is either done for all open views at once (by help of the special view id GENERAL_VIEW) or for content retrieval handler directly (by help of the special view id CONTENT_VIEW).

getContent

Open arbitrary content



This function instructs the AUS WAP Browser to retrieve content. In response to this function the AUS WAP Browser will call the Adapter function CLNTa_content. Note that this function bypasses the WAE layer of the AUS WAP Browser. Actually, there is no requirement that a view is opened, to use this function. This means that a download operation started with this function not can be stopped with the MMIc_stop function. If a WAP server or a content server requires authentication, the MMIa_passwordDialog will not be called. The content cannot be retrieved in that case.

```
VOID CLNTc_getContent (const CHAR *url, UINT8
urlID, BOOL isOTAallowed)
```

`url`	The URL of the content to be retrieved. The URL may be deleted after the call.
`urlID`	The urlID identifies this request in the WAP application. It is used in the corresponding Adapter function CLNTa_content.

`isOTOallowed`	The isOTAallowed flag informs whether it is allowed to retrieve the content “over the air”. If this is FALSE, only content from the cache can be opened.

Content

This function is used by the AUS WAP Browser to provide a WAP application with data requested in a previous call to the function CLNTc_getContent.

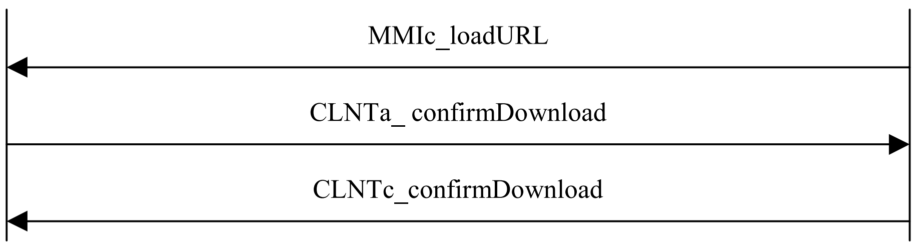
```
VOID CLNTa_content (UINT8 urlID, const CHAR *data,
UINT16 length, const CHAR *contentType, INT16
errorNo)
```

`urlID`	The urlID retrieved in the Connector function CLNTc_getContent.
`data`	The data parameter is a pointer to the content. If the data parameter is NULL then an error has occurred. The data is deleted when the function returns.
`length`	The length parameter gives the length in bytes of the data.
`contentType`	The contentType is taken from the WSP header [WAP-WSP] and gives the content type of the data. Content types are defined in [RFC2068]. The string is deleted when the function returns.
`errorNo`	If the data parameter is NULL then an error has occurred. The errorNo parameter indicates the type of error. Constants for them are the same as for the function CLNTa_error (defined in errcodes.h).

10.10 Control of the “over the air” activities

On bearers with rates, the following functions provide the WAP application a mean to let the user confirm if a file shall be downloaded, or not. The functions are used whenever a download task is about to be executed. There is a configuration variable, configQ_OTA, which controls whether the feature shall be used or not.

configQ_OTA == true



confirmDownload

The function will be invoked from the AUS WAP Browser just before an "over the air" activity will start. The function expects the answer TRUE if it is allowed to go "over the air", otherwise FALSE. In case of false the current request will not be performed. The answer is delivered to the AUS WAP Browser by calling the function CLNTc_confirmDownload. If this function not shall be called, the configuration variable configQ_OTA must be set to zero. The AUS WAP Browser checks this flag in order to decide if it is necessary to confirm the download.

```
VOID CLNTa_confirmDownload (INT16 confirmId)
```

`confirmId`	The answer is delivered to the AUS WAP Browser by calling the function CLNTc_confirmDownload with the confirmId given by this function.

confirmDownload

The function will be called from the WAP application in order to deliver the answer if an "over the air" activity is allowed to be performed. The question in matter comes from the function CLNTa_confirmDownload.

```
VOID CLNTc_confirmDownload (INT16 confirmId, BOOL
isOTAallowed)
```

`confirmId`	The confirmId retrieved by the Adapter function CLNTa_confirmDownload.
`isOTAallowed`	The function sets the isOTAallowed argument to TRUE if it is allowed to go "over the air", otherwise it sets it to FALSE.

10.11 Messages

error

The AUS WAP Browser calls the function when an error occurred during an operation from the view.

```
VOID CLNTa_error (UINT8 viewId, INT16 errorNo,
UINT8 errorType)
```

`viewId`	The view id received from a call to MMIc_openView. If the view id is equal to 0, the error is general for all views.
`errorNo`	The error number gives what kind of error it is. All errors are described in Appendix 2.
`errorType`	The error type indicates the severity of the error. All types are described in Appendix 2.

log

The AUS WAP Browser calls the function when log or debug information about the system is to be given. All communication from and to the lower end of each protocol layer is logged. The function equals in functionality with ANSI C printf().The ANSI C function vprintf() is thought to be used if it is available, or to be taken as model for the implementation. The CLNTa_log function calls are only included in the AUS WAP Browser if the compiler switch LOG_EXTERNAL is set when the AUS WAP Browser is compiled. This Adapter function needs not to be implemented if the switch is not set.

```
VOID CLNTa_log (UINT8 viewId, INT16 logNo, const
CHAR *format, …)
```

`viewId`	The view id received from a call to MMIc_openView. If the view id is equal to 0, the error is general for all views.
`logNo`	The log number gives what kind of operation it is. The log numbers are defined in logcodes.h.
`format`	The format tells how the following arguments shall be formatted, e.g. "%d\n". The format argument is always given.
…	There might be zero or more arguments following the format string. All passed strings are deleted when the function returns.

10.12 Support of character sets

If any further character sets than UTF-8, ISO-8859-1 or UCS16 (Unicode) shall be supported by the WAP application, a set of transcoding functions must be implemented for the character sets in matter.

setTranscoders

Provides the AUS WAP Browser with function pointers to external transcoding functions. These external functions will be used if the internal functions can not perform the transcoding. The functions are described in the four following sub-chapters.

```
VOID CLNTc_setTranscoders (
fPtr_Iana2Unicode_canConvert canConvert,
fPtr_Iana2Unicode_calcLen calcLen,
fPtr_Iana2Unicode_convert convert,
fPtr_Iana2Unicode_getNullTermByteLen nullLen)
```

`canConvert`	Pointer to function described below.
`calcLen`	Pointer to function described below.
`convert`	Pointer to function described below.
`nullLen`	Pointer to function described below.

10.12.1 canConvert

In order to being able to determine if the WAP application has support for a certain character set, a function must be implemented for that purpose. A pointer to it, defined as:

```
 typedef BOOL (*fPtr_Iana2Unicode_canConvert)( INT16 );
```

is assigned to an argument, canConvert, of the function CLNTc_ setTranscoders. The function is to be implemented according to the following:

```
BOOL Iana2Unicode_canConvert  (INT16 charset)
```

`charset`	The argument holds the MIBenum IANA code that corresponds to a specific character encoding.

The function shall return TRUE if transcoding of the character set is available, otherwise FALSE.

10.12.2 calcLen

In order to being able to determine the number of characters in a string of a certain character set, a function must be implemented for that purpose. A pointer to it, defined as:

```
 typedef INT32 (*fPtr_Iana2Unicode_calcLen)( BYTE*, INT16,
          BOOL, UINT32, UINT32* );
```

is assigned to an argument, calcLen, of the function CLNTc_ setTranscoders. The function is to be implemented according to the following:

```
INT32 Iana2Unicode_calcLen (BYTE *str, INT16
charset, BOOL isNullTerminated, UINT32 readBytes,
UINT32 *strByteLen)
```

`str`	The argument str holds the string to be transcoded from the character set that is given in the argument charset.
`charset`	The argument holds the MIBenum IANA code that corresponds to a specific character encoding.
`isNullTerminated`	The argument isNullTerminated is set to FALSE if the size of the argument str not is known.
`readBytes`	In order to avoid searching infinitely after a termination of the string, readBytes should be assigned to a nonzero value. This will then be the upper limit of bytes that will be read in search of the termination.
`strByteLen`	The argument strByteLen will after call to the function contain the size of the argument str (in number of bytes). This argument will thus have the same value as

the argument readBytes if isNullTerminated is set to FALSE. This parameter should contain a correct result even if the Iana2Unicode_calcLen function failed due to incorrect characters in the argument str. If the byte length could not be decided, 0 should be returned.

Returns the number of characters str contains or -1 if an error occurs during the calculation (strByteLen must, however, be valid).

10.12.3 convert

In order to convert a string of a certain character set to a Unicode string, a function must be implemented for that purpose. A pointer to it, defined as:

```
typedef BOOL (*fPtr_Iana2Unicode_convert)( BYTE*, INT16,
             UINT32, WCHAR*, UINT32 );
```

is assigned to an argument, convert, of the function CLNTc_ setTranscoders. The function is to be implemented according to the following:

```
BOOL Iana2Unicode_convert(BYTE *str, INT16 charset,
UINT32 strByteLen, WCHAR *resultBuffer, UINT32
resultBufferLen)
```

`str`	The argument str holds the string to be transcoded from the character set that is given in the argument charset.
`charset`	The argument holds the MIBenum IANA code that corresponds to a specific character encoding.
`strByteLen`	The number of bytes the string takes is given in the argument strByteLen.
`resultBuffer`	The resulting Unicode string should be retrieved in the argument resultBuffer.
`resultBufferLen`	The length (in number of Unicode characters) of the string is given in the argument resultBufferLen.

The function returns TRUE if the conversion went ok, FALSE if something went wrong.

10.12.4 nullLen

In order to determine the length of the terminating character that a certain character set has, a function must be implemented for that purpose. A pointer to it, defined as:

```
typedef UINT8 (*fPtr_Iana2Unicode_getNullTermByteLen)(
             INT16 );
```

is assigned to an argument, nullLen, of the function CLNTc_ setTranscoders. The function is to be implemented according to the following:

```
UINT8 Iana2Unicode_nullLen(INT16 charset)
```

`charset`	The argument holds the MIBenum IANA code that corresponds to a specific character encoding.

Returns the length (in bytes) that a string terminating character occupies in a string encoded with iIANAcharset. If the character set is unknown zero should be returned.

11 WTA API

This API hosts the telephony functionality. The API is optional and may be implemented, or not. If not needed, empty Adapter functions are to be implemented.

WTA functionality is divided into two parts: *Public WTAI* that is included in all configurations of the AUS WAP Browser, and all other WTAI functionality that is included in AUS WAP Browser configurations that includes full WTAI (*will come in future versions*).

11.1 Public WTAI

The function in this section describes public WTAI telephony functionality. It is an optional API that may be implemented, or not. If not needed, empty Adapter functions are to be implemented. However, if the configuration of the AUS WAP Browser configuration contains full WTA, the public function should be implemented as well.

publicMakeCall

WTAI Lib/Func ID: 512.0

This function is called when a voice call should be set-up. The user must be prompted if this operation shall be performed, or not. If a data call currently is established in order to supply the IP network, it must be closed down before this operation is performed. If the WAP application shall live in the background during the voice call, the AUS WAP Browser can be suspended. When the voice call is closed down, the AUS WAP Browser can be resumed and put in front again. Suspend and resume is further described in the Client API.

```
VOID WTAIa_publicMakeCall (const CHAR *number)
```

`number`	The parameter argument holds the phone number. The string is deleted after the function has returned.

publicSendDTMF

WTAI Lib/Func ID: 512.1

This function is called when a tone sequence should be sent. The user must be prompted if this operation shall be performed, or not. If a data call currently is established in order to supply the IP network, it must be closed down before this operation is performed.

```
VOID WTAIa_publicSendDTMF (const CHAR *number)
```

`number`	The parameter argument holds the numbers to be sent as a tone sequence. The string is deleted after the function has returned.

publicPBwrite

WTAI Lib/Func ID: 512.2

This function is called when a new entry should be added to the phone book. The user must be prompted if this operation shall be performed, or not.

```
INT8 WTAIa_publicPBwrite (const CHAR *number, const
WCHAR *name)
```

`number`	The parameter argument holds the phone number. The string is deleted after the function has returned.
`name`	The parameter argument holds the name to be associated with the phone number. The string is deleted after the function has returned.

The return value is zero if successful or a value below zero indicating the WTAI error code. Predefined errors are found in the next section.

11.2 Error handling of Adapter function calls

In order to control the operation of the mobile phone corresponding Adapter functions have been defined. They are to be implemented in a way that control is given when possible. When not possible, an error code is to be returned by the Adapter function that failed. The following table states all possible error codes.

Value	**Constant**	**Description**
-1	WTA_RES_ID_NOT_ FOUND	Call id not found. Function could not be completed.
-2	WTA_RES_ILLEGAL_ PARAMETERS	Illegal number of parameters, function could not be resolved due to missing parameters.
-3	WTA_RES_SERVICE_ NOT_AVAILABLE	Service not available or non-existent function.
-4	WTA_RES_SERVICE_ TEMP_UNAVAILABLE	Service temporarily unavailable.
-5	WTA_RES_BUSY	Called party is busy.
-6	WTA_RES_ NETWORK_BUSY	Network is busy.
-7	WTA_RES_NO_ ANSWER	No answer, i.e. call setup timed out.
-8	WTA_RES_ UNKNOWN	Unknown.

-9	WTA_RES_OUT_OF_ MEMORY	Out of memory.
-10 to -63		Reserved for future use by WTA standard library functions.
-64	WTA_RES_USSD_ PROGRESS	USSD dialogue in progress.
-65	WTA_RES_ILLEGAL_ CHARACTERS	Illegal characters.
-66 to -128		Network specific error codes.

12 Memory API

This API defines functionality in order to access three kinds of storage.

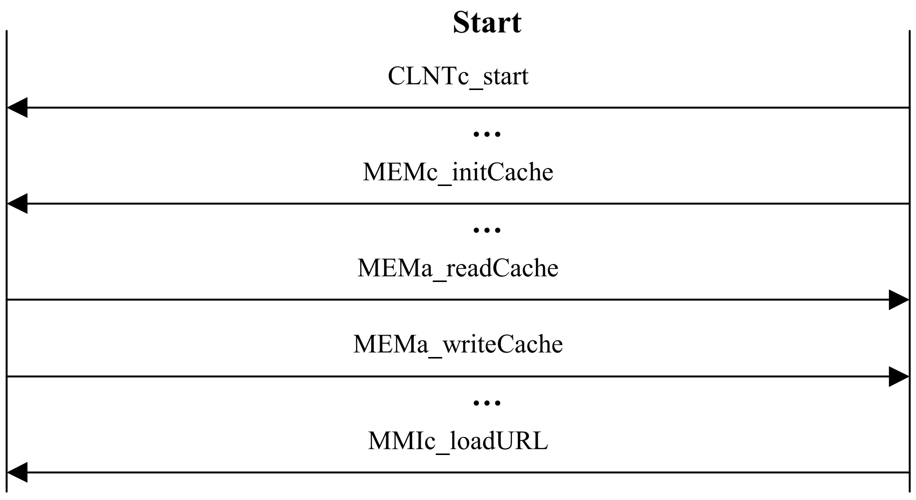
The first kind is the cache, where the downloaded WML, WMLS content and images are stored. The cache is optional in the sense that if the size is zero, the Generic WAP consider the cache to be non-existent.

The next kind of memory is the storage where the AUS WAP Browser is supposed to store WTAI services. This storage type is only necessary in configurations of the AUS WAP Browser that includes full WTA.

Last, there is the storage for Pushed content, i.e. content that has been uploaded to the AUS WAP Browser from a WAP content server, and is stored temporary to be viewed later. This storage type is only necessary in configurations of the AUS WAP Browser that includes Push functionality.

12.1 Initiate the cache

After the AUS WAP Browser has been started (CLNTc_start), after the cache memory has been restored, but before any content is downloaded (MMIc_loadURL), the AUS WAP Browser must be notified about how much cache memory that has been restored and how much it now can use. The AUS WAP Browser needs also this notification during runtime when the cache has been resized. Read more about this in the next section.



initCache

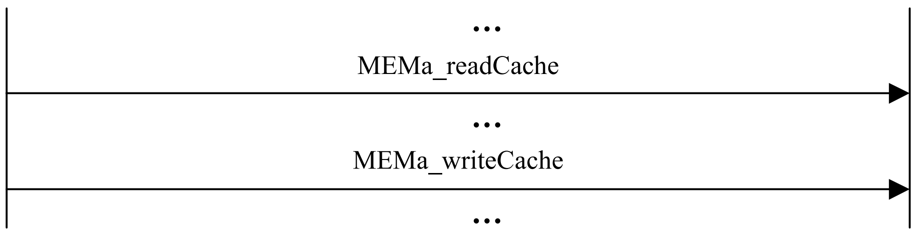
The WAP client calls this function when the cache memory has been restored. If the cache memory is new, i.e., the WAP Client has created it before this call, the eight leading bytes of the memory must be set to zero. If the eight leading bytes are not set to zero, the AUS WAP Browser assumes the memory to contain data. If this function is never called, the AUS WAP Browser assumes that no cache is available and runs without it.

```
VOID MEMc_initCache (memSizeType cacheSize,
memSizeType restoredSize)
```

`cacheSize`	The available size of cache memory is given in the cacheSize argument.
`restoredSize`	The memory size that has been restored from persistent memory is given in the restoredSize argument.

12.2 Accessing the cached content repository

There are functions with wich to access the cached content repository. There are one for writing and one for reading. They are never called simultaneously from the AUS WAP Browser, mutual exclusion of the common data is not necessary from the point of view of the AUS WAP Browser.



readCache

Read a number of bytes from the cache content repository.

```
UINT32 MEMa_readCache (UINT32 pos, UINT32 size,
CHAR *buffer)
```

`pos`	The position in the cache content repository from where to read.
`size`	The number of bytes to read.
`buffer`	A, by the AUS WAP Browser, allocated buffer to store the read bytes in.

The function returns the actual number bytes read.

writeCache

Write a number of bytes to the cache content repository.

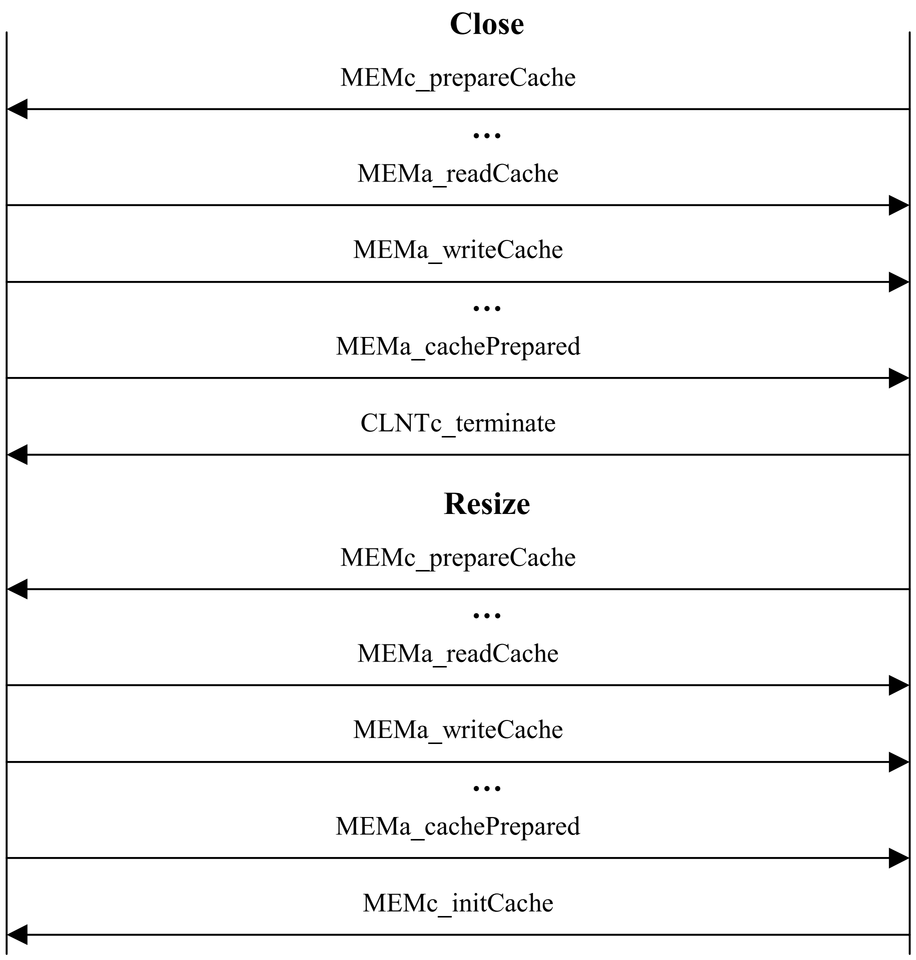
```
UINT32 MEMa_writeCache (UINT32 pos, UINT32 size,
const CHAR *buffer)
```

`pos`	The position in the cache content repository to write from.
`size`	The number of bytes to write.
`buffer`	The bytes to write.

The function returns the actual number bytes written.

12.3 Close the cache

The following two functions are used immediately before the cache is to be closed (CLNTc_terminate), and the AUS WAP Browser must make preparations on the cache content. They are also used when the cache is to be resized. In that case, a call to MEMc_initCache must be done again, when the WAP Client has received the MEMc_prepareCache acknowledgement (MEMa_cachePrepared) and the new memory area is initialised.



prepareCache

The function is called when the WAP client is closing down, before the cache is to be stored in persistent memory. This function may as well be called during runtime when the cache is to be resized. When the AUS WAP Browser is done with the preparations, the function MEMa_cachePrepared is called as an acknowledgement.

```
VOID MEMc_prepareCache (memSizeType
availablePersistentMemory)
```

cachePrepared

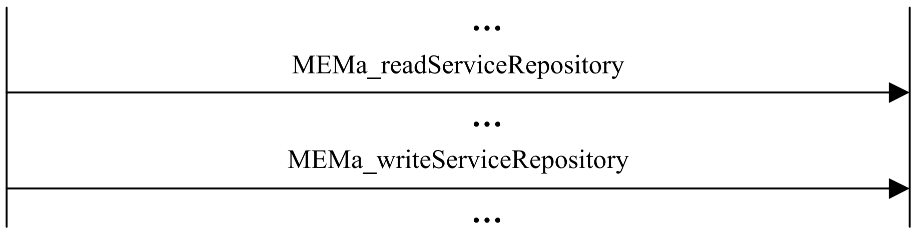
The function is called when the Generic WAP has prepared the cache and the WAP client is allowed to finally store the cache. Note that a call to this function must be preceded by a call of the function MEMc_prepareCache.

```
VOID MEMa_cachePrepared (VOID)
```

12.4 Accessing the WTA services repository

The functions in this section are only necessary to implement in configurations of the AUS WAP Browser that includes full WTA.

There are functions with wich to access the WTAI services repository. There are one for writing and one for reading. They are never called simultaneously from the AUS WAP Browser, mutual exclusion of the common data is not necessary from the point of view of the AUS WAP Browser.



readServiceRepository

Read a number of bytes from the WTAI services repository.

```
UINT32 MEMa_readServiceRepository (UINT32 pos,
UINT32 size, CHAR *buffer)
```

`pos`	The position in the WTAI services repository from where to read.
`size`	The number of bytes to read.
`buffer`	A, by the AUS WAP Browser, allocated buffer to store the read bytes in.

The function returns the actual number bytes read.

writeServiceRepository

Write a number of bytes to the WTAI services repository.

```
UINT32 MEMa_writeServiceRepository (UINT32 pos,
UINT32 size, const CHAR *buffer)
```

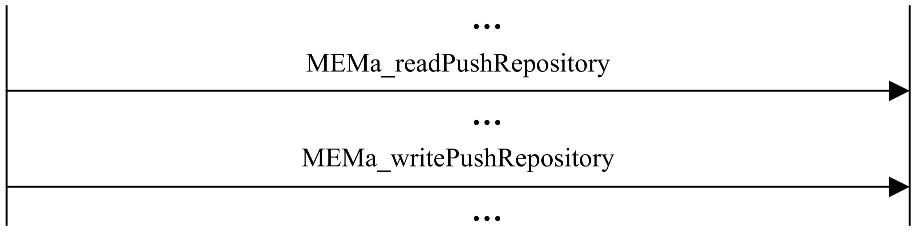
`pos`	The position in the WTAI services repository where to write.
`size`	The number of bytes to write.
`buffer`	The bytes to write.

The function returns the actual number bytes written.

12.5 Accessing the pushed content repository

The functions in this section are only necessary to implement in configurations of the AUS WAP Browser that includes Push functionality.

There are functions with which to access the pushed content repository. There are one for writing and one for reading. They are never called simultaneously from the AUS WAP Browser, mutual exclusion of the common data is not necessary from the point of view of the AUS WAP Browser.



readPushRepository

Read a number of bytes from the pushed content repository.

```
UINT32 MEMa_readPushRepository (UINT32 pos, UINT32
size, CHAR *buffer)
```

`pos`	The position in the pushed content repository from where to read.
`size`	The number of bytes to read.
`buffer`	A, by the AUS WAP Browser, allocated buffer to store the read bytes in.

The function returns the actual number bytes read.

writePushRepository

Write a number of bytes to the pushed content repository.

```
UINT32 MEMa_writePushRepository (UINT32 pos, UINT32
size, const CHAR *buffer)
```

`pos`	The position in the pushed content repository where to write.

`size`	The number of bytes to write.
`buffer`	The bytes to write.

The function returns the actual number bytes written.

12.6 Type definition

The Memory API uses a type that remains to be defined (in tapimem.h):

memSizeType	If the memory sizes the device supports is less than 32kBytes, “typedef INT16 memSizeType;” could be used as type definition.

13 Crypto API

The security layer in the AUS WAP Browser, WTLS, needs access to a library of cryptographic functions. For the AUS WAP Browser, this is realised as a set of Adapter functions. This chapter describes the cryptographic functions that the AUS WAP Browser makes use of.

13.1 Overview

13.1.1 Design principles

The API to the cryptographic routines described herein is tailored to meet the needs of the WTLS client. It is expected that it might be realised with the aid of WIM or other smart card based implementations. One of the chief advantages of a smart card device, is its ability to protect secrets. For example, the private half of a private/public key pair might be stored on the card, and would then not be retrievable. Instead, the user would have to pass data to be encrypted/decrypted to the device, have the computation be performed and the result be passed back. A device that offers this kind of services and protection is called *tamper-resistant*.

In the design of WIM, one has utilised this feature and decided that the so-called *master secret* should be kept inside the WIM at all times. To be compatible with a WIM implementation, the API to the cryptographic routines makes the same assumption. That is, it is assumed throughout that the *master secret* is held internally in the library of the cryptographic routines. This implies that all functions that use or compute the master secret have to use a parameter set consistent with this principle. See for example CRYPTa_PRF.

Some encryption operations will probably take a long time. This is especially true for the public-key algorithms, like RSA and Diffie-Hellman. The AUS WAP Browser cannot wait for such lengthy operations to complete because it would interfere with the scheduling in the system. Instead, the routines that are most likely to require long computation times have been divided into adaptor-connector function pairs. The adaptor function is expected to return immediately, and then the computation should be carried out asynchronously and the answer delivered via a call to the connector function

The bulk encryption methods and the key exchange methods used in WTLS all have variants with reduced key lengths. The decision to use a method with reduced key size is usually based on legal, political and/or commercial considerations. From a technical perspective, there would be no compelling reason to reduce the key size, and consequently also reduce security. Hence, it was considered appropriate that this type of policy decision should be located outside of WTLS itself. On the other hand, due to the way in which reduction in key size is handled in WTLS, the responsibility for executing a decision to reduce the key size rests with the WTLS implementation.

The current implementation of WTLS is of implementation class 2, i.e, it handles server authentication through the use of certificates but not client authentication.

To verify a server certificate, the client needs one or more root certificates. The management of such certificates (and in the future client certificates) will likely involve storage on WIM or other smart card devices. Hence, the verification of certificates has been placed wholly inside the library of the cryptographic routines.

13.1.2 How the functions are used

The AUS WAP Browser calls the functions in this Crypto API in the following order:

Initialisation:

```
CRYPTa_initialise
CRYPTa_getCipherMethods
CRYPTa_getClientKeyExchangeIds
CRYPTa_getTrustedKeyIds
```

Handshake, that is establishing a connection:

```
CRYPTa_generateRandom
CRYPTa_verifyCertificateChain
CRYPTa_keyAgreement
CRYPTa_hashInit
CRYPTa_hashUpdate
CRYPTa_hashFinal
```

Computing encryption keys:

```
CRYPTa_PRF
CRYPTa_hash
```

Encrypting and decrypting data:

```
CRYPTa_encryptInit
CRYPTa_encryptUpdate
CRYPTa_encryptFinal
CRYPTa_decrypt
CRYPTa_hashInit
CRYPTa_hashUpdate
CRYPTa_hashFinal
```

13.1.3 Conventions for output-producing functions

A number of functions in this Crypto API produce output, as the result of some cryptographic algorithm. The output produced by such a function is placed in a supplied buffer. Two of the arguments to such a function are a pointer to the output buffer (say *buf*) and a pointer to the length of the output produced (say *bufLen*). There are two ways such a function can be called:

1. If *buf* is NULL, then all that the function does is return (in **bufLen*) a number of bytes that would suffice to hold the output produced by the cryptographic

algorithm. This number may exceed the precise number of bytes needed, but not by a large amount. The function returns CRV_OK.

2. If *buf* is not NULL, then **bufLen* must contain the size in bytes of the buffer pointed to by *buf*. If that size is large enough the output produced is placed there, and the function returns CRV_OK. If it is not large enough, then CRV_BUFFER_TOO_SMALL is returned. In either case, **bufLen* is set to the exact number of bytes needed to hold the cryptographic output produced by the function.

13.1.4 Function return values

Most functions in this crypto library return an integer value. Successful return is indicated by the value CRV_OK. Other values indicate some sort of failure or problem. The AUS WAP Browser treats all failures as equal, i.e., a certain library function that fails will be handled in a uniform way regardless of the actual reason for failure. However, the return code will be logged with CLNTc_log when LOG_EXTERNAL is defined. The function CLNTa_error will, as well, be called. The error code will be ERR_WTLS_CRYPTOLIB.

It is recommended that the following constants (defined in aapicrpt.h) be used as return values for the Adapter functions of this API.

Constant	**Value**	**Description**
CRV_OK	0	No error
CRV_GENERAL_ERROR	1	Error not covered by any other return value
CRV_BUFFER_TOO_SMALL	2	The output produced does not fit in the supplied buffer
CRV_UNSUPPORTED_ METHOD	3	A cryptographic method that is not supported has been requested
CRV_ALREADY_INITIALISED	4	Trying to initialise the system a second time
CRV_INSUFFICIENT_ MEMORY	5	Memory allocation failed
CRV_CRYPTOLIB_NOT_ INITIALISED	6	Trying to use a method in the library without prior initialisation
CRV_KEY_TOO_LONG	7	The key supplied to a cryptographic method is too long
CRV_NOT_ IMPLEMENTED	8	The called function has not been implemented
CRV_INVALID_PARAMETER	9	One or more of the supplied parameters has an illegal value
CRV_DATA_LENGTH	10	Encryption or decryption using a block method where the length of

		the data is not a multiple of the block length
CRV_INVALID_KEY	11	A supplied key has an illegal format or value
CRV_INVALID_HANDLE	12	A handle used in, e.g., CRYPTa_EncryptUpdate, has an illegal value
CRV_KEY_LENGTH	13	The length of the key does not match what the method in use requires
CRV_MISSING_KEY	14	The requested secret key is not present in the library
CRV_UNKNOWN_ CERTIFICATE_TYPE	15	A certificate of unknown type has been specified
CRV_NO_MATCHING_ROOT_ CERTIFICATE	16	Authentication cannot be carried out due to a missing root certificate
CRV_BAD_CERTIFICATE	17	Some other unspecified error prevents authentication of the certificate
CRV_CERTIFICATE_EXPIRED	18	The certificate to be authenticated has expired

13.2 General functions

initialise

Perform necessary initialisation tasks; for example, seed the random number generator. This function must be called (exactly once) before any other function in the Crypto API is used.

```
INT16 CRYPTa_initialise (VOID)
```

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

13.3 Bulk encryption algorithms

getCipherMethods

Retrieve a list of all Cipher Methods that the Crypto API supports.

```
INT16 CRYPTa_getCipherMethods (BYTE *methodList,
UINT16 *bufSize)
```

`methodList`	The Cipher Methods are stored in this byte buffer. The type of the output is a byte-encoded sequence of elements of type CipherMethod, defined below. This function uses the convention for output-producing functions described above. The AUS WAP Browser is responsible for the de-allocation of the data.
`bufSize`	See the convention for output-producing functions described above.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

encrypt

Encrypt single-part data.

```
INT16 CRYPTa_encrypt (BulkCipherAlgorithm method,
KeyObject key, BYTE *data, UINT16 dataLen, BYTE
*encryptedData, UINT16 *encryptedDataLen)
```

`method`	The encryption algorithm.
`key`	The encryption key.
`data`	The data to encrypt. The AUS WAP Browser deletes the data.
`dataLen`	The size in bytes of the data. For some encryption methods, the input data has certain length constraints (e.g., the length should be a multiple of 8). If these constraints are not satisfied, then CRYPTa_encrypt shall fail with return code CRV_DATA_LENGTH.
`encryptedData`	The encrypted data. This argument can, depending on the encryption method used, hold a reference to the same memory as the data argument. This function uses the convention for output-producing functions described above. The AUS WAP Browser is responsible for the de-allocation of the data.
`encryptedDataLen`	See the convention for output-producing functions described above.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

encryptInit

Initialise an encryption operation. After the AUS WAP Browser has called CRYPTa_encryptInit, it calls CRYPTa_encryptUpdate zero or more times, followed by CRYPTa_encryptFinal, to encrypt data in multiple parts. The encryption operation is active until the AUS WAP Browser makes a call to

CRYPTa_encryptFinal. To process additional data (in single or multiple parts), the AUS WAP Browser will call CRYPTa_encryptInit again.

```
INT16 CRYPTa_encryptInit (BulkCipherAlgorithm
method, KeyObject key, EncryptHandle *handlePtr)
```

`method`	The encryption algorithm.
`key`	The encryption key.
`handlePtr`	Will point to a new handle to be used in subsequent encryption operations.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

encryptUpdate

Continue a multiple-part encryption operation, processing another data part. A call to CRYPTa_encryptUpdate that results in an error terminates the current encryption operation.

```
INT16 CRYPTa_encryptUpdate (EncryptHandle handle,
BYTE *data, UINT16 dataLen, BYTE *encryptedData,
UINT16 *encryptedDataLen)
```

`handle`	The handle of the encryption operation received by a previous call to CRYPTa_encryptInit.
`data`	The data to encrypt. The AUS WAP Browser is responsible for the de-allocation of the data.
`dataLen`	The size in bytes of the data.
`encryptedData`	The encrypted data. This function uses the convention for output-producing functions described above. The AUS WAP Browser is responsible for the de-allocation of the data.
`encryptedDataLen`	See the convention for output-producing functions described above.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

encryptFinal

Finish a multiple-part encryption operation. A call to CRYPTa_encryptFinal always terminates the active encryption operation unless it returns CRV_BUFFER_TOO_SMALL or is a successful call to determine the length of the buffer needed to hold the encrypted data.

```
INT16 CRYPTa_encryptFinal (EncryptHandle handle,
BYTE *lastEncryptedPart, UINT16
*lastEncryptedPartLen)
```

`handle`	The handle of the encryption operation received by a previous call to CRYPTa_encryptInit.
`lastEncryptedPart`	The location in memory that receives the last bit of encrypted data. This function uses the convention for output-producing functions described above. The AUS WAP Browser is responsible for the de-allocation of the data.
`lastEncryptedPartLen`	See the convention for output-producing functions described above. For some encryption methods, the input data has certain length constraints (e.g., the length should be a multiple of 8). If these constraints are not satisfied, then CRYPTa_encrypt shall fail with return code CRV_DATA_LENGTH.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

decrypt

Decrypt single-part data.

```
INT16 CRYPTa_decrypt (BulkCipherAlgorithm method,
KeyObject key, BYTE *data, UINT16 dataLen, BYTE
*decryptedData, UINT16 *decryptedDataLen)
```

`method`	The decryption algorithm.
`key`	The decryption key.
`data`	The data to decrypt. The AUS WAP Browser is responsible for the de-allocation of the data.
`dataLen`	The size in bytes of the data. For some decryption methods, the input data has certain length constraints (e.g., the length should be a multiple of 8). If these constraints are not satisfied, then CRYPTa_encrypt will fail with return code CRV_DATA_LENGTH.
`decryptedData`	The encrypted data. This argument can, depending on the encryption method used, hold a reference to the same memory as the data argument. This function uses the convention for output-producing functions described above. The AUS WAP Browser is responsible for the de-allocation of the data.
`decryptedDataLen`	See the convention for output-producing functions described above.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

13.4 Secure hash functions

hash

Compute a hash digest of given single-part data. CRYPTa_hash is equivalent to a call to CRYPTa_hashInit, followed by a sequence of CRYPTa_hashUpdate operations, and terminated by a call to CRYPTa_hashFinal.

```
INT16 CRYPTa_hash (HashAlgorithm method, BYTE
*data, UINT16 dataLen, BYTE *digest, UINT16
*digestLen)
```

`method`	The hash algorithm to use.
`data`	The input data. The AUS WAP Browser is responsible for the de-allocation of the data.
`dataLen`	The length in bytes of the input data.
`digest`	The digest output. This argument can, depending on the encryption method used, hold a reference to the same memory as the data argument. This function uses the convention for output-producing functions described above. The AUS WAP Browser is responsible for the de-allocation of the data.
`digestLen`	See the convention for output-producing functions described above.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

hashInit

Initialise a hash operation. After calling CRYPTa_hashInit, the AUS WAP Browser calls CRYPTa_hashUpdate zero or more times, followed by CRYPTa_hashFinal, to digest data in multiple parts. The hash operation is active until the AUS WAP Browser makes a call to CRYPTa_hashFinal to actually obtain the final piece of ciphertext. To process additional data (in single or multiple parts), one must call CRYPTa_hashInit again.

```
INT16 CRYPTa_hashInit (HashAlgorithm method,
HashHandle *handlePtr);
```

`method`	The hash algorithm to use.
`handlePtr`	Will point to a new handle to be used in subsequent operations.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

hashUpdate

Continue a multiple-part hash operation, processing another data part. A call to CRYPTa_hashUpdate that results in an error terminates the current hash operation.

```
INT16 CRYPTa_hashUpdate (HashHandle handle, BYTE
*part, UINT16 partLen)
```

`handle`	The handle of the hash operation previously received by a call of CRYPTa_hashInit.
`part`	The data part. The AUS WAP Browser is responsible for the de-allocation of the data.
`partLen`	The length in bytes of the data part.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

hashFinal

Finish a multiple-part hash operation. The hash operation must have been initialised with CRYPTa_hashInit. A call to CRYPTa_hashFinal always terminates the active hash operation unless it returns CRV_BUFFER_TOO_SMALL or is a successful call to determine the length of the buffer needed to hold the message digest.

```
INT16 CRYPTa_hashFinal (HashHandle handle, BYTE
*digest, UINT16 *digestLen)
```

`handle`	The handle of the hash operation previously received by a call of CRYPTa_hashInit.
`digest`	The digest output. This argument can, depending on the encryption method used, hold a reference to the same memory as the data argument. This function uses the convention for output-producing functions described above. The AUS WAP Browser is responsible for the de-allocation of the data.
`digestLen`	See the convention for output-producing functions described above.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

13.5 Key Exchange and Key Generation

getClientKeyExchangeIds

Retrieve a list of all key exchange ids that the library of the cryptographic routines supports, possibly together with parameters and certificate identifiers.

```
INT16 CRYPTa_getClientKeyExchangeIds (BYTE
*keyExchangeIds, UINT16 *bufSize)
```

`keyExchangeIds`	The output is to be stored as a byte stream in this buffer. The type of the output is a byte-encoded sequence of elements of type KeyExchangeId, defined below. This function uses the convention for output-producing functions described above. The AUS WAP Browser is responsible for the de-allocation of the data.
`bufSize`	See the convention for output-producing functions described above.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

keyExchange

Using the appropriate key exchange algorithm, perform a key exchange operation and calculate the master secret. The result is returned by calling the connector function CRYPTc_keyExchangeResponse. The public key to use either is given explicitly in the parameters, or must be retrieved from a certificate passed to this routine.

```
VOID CRYPTa_keyExchange (UINT16 id,
KeyExchangeParameters parameters, HashAlgorithm
alg, const BYTE *data, UINT16 len)
```

`id`	An identifier that is passed back in the call to CRYPTc_keyExchangeResponse.
`parameters`	Parameter that holds the public key and indicates which key exchange method to use.
`alg`	The secure hash algorithm to use.
`data`	Additional data that the key exchange method may require (see parameters), as follows:

Null	empty
Secret key	empty
Diffie-Hellman	public value from server side
RSA	1 byte additional data
ECDH	empty

`len`	The length of the data supplied.

keyExchangeResponse

The result of the key exchange operation is returned by a call to this connector function.

```
VOID CRYPTc_keyExchangeResponse (UINT16 id, INT16
result, UINT16 masterSecretID, const BYTE
*publicValue, UINT16 publicValueLen)
```

`id`	The identifier that was supplied in the call to CRYPTa_keyExchange.
`result`	One of the return codes, as specified above.
`masterSecretID`	The master secret is kept internally in the library of the cryptographic routines. An integer is used to identify the master secret in subsequent operations requiring its use, e.g., CRYPTa_PRF.
`publicValue`	A public value computed by the key exchange method to be sent to the server side. Only relevant for Diffie-Hellman, RSA and ECDH methods.
`publicValueLen`	The length of public value.

PRF

Calculate a hash of a block of data using the PRF function defined in the WTLS spec, and with the master secret as a first parameter.

```
INT16 CRYPTa_PRF (HashAlgorithm method, UINT16
masterSecretID, BYTE *label, BYTE *seed, UINT16
seedLen, BYTE *output, UINT16 outputLen)
```

`method`	The secure hash algorithm to use.
`masterSecretId`	The pre-master secret is kept internally in the library of the cryptographic routines. An identifier retrieved from CRYPTc_keyExchangeResponse is used to identify the master secret.
`label`	The AUS WAP Browser is responsible for the de-allocation of the data.
`seed`	The AUS WAP Browser is responsible for the de-allocation of the data.
`seedLen`	The length of the seed data.
`output`	The AUS WAP Browser is responsible for the de-allocation of the data.

`outputLen`	The length of the output data.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

13.6 Certificates

getTrustedKeyExchangeIds

Retrieve a list of the key exchange ids of all trusted certificates.

```
INT16 CRYPTa_getTrustedKeyExchangeIds (BYTE
*trustedKeyExchangeIds, UINT16 *bufSize)
```

`trustedKeyExchangeIds`	This argument holds the identifier of a certificate, plus information about a matching key exchange method. The type of the output should be a byte-encoded sequence of elements of type KeyExchangeId, defined in the WTLS specification [WAP-WTLS]. This function uses the convention for output-producing functions described above. The AUS WAP Browser is responsible for the de-allocation of the data.
`bufSize`	See the convention for output-producing functions described above.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

certificateListUpdated

If the set of root certificates is updated, e.g., by downloading new certificates, then this function should be called to invalidate the AUS WAP Browsers cache of certificates. The AUS WAP Browser will then respond by calling CRYPTa_getTrustedKeyExchangeIds the next time a connection is to be established.

```
VOID CRYPTc_certificateListUpdated (VOID)
```

verifyCertificateChain

Verify a chain of certificates. The result is delivered via a call to CRYPTc_verifyCertificateChainResponse.

```
VOID CRYPTa_verifyCertificateChain (UINT16 id,
const BYTE *buf, UINT16 bufSize)
```

`id`	An identifier that is to be passed back in the call to

	CRYPTc_verifyCertificateChainResponse.
`buf`	The input data. The type of the input is a byte-encoded sequence of elements of type Certificate. The AUS WAP Browser is responsible for the de-allocation of the data. Note, the definition of how the Certificate data is to be parsed is found in the standard specifications WTLS [WAP-WTLS], where the header and the WTLS certificate is described. The two other possible kinds of certificates are described in [X.509] and [X.968].
`bufSize`	The size in bytes of the input data.

verifyCertificateChainResponse

The result of the certificate verification operation is returned by a call to this connector function.

```
VOID CRYPTc_verifyCertificateChainResponse (UINT16
id, INT16 result)
```

`id`	The identifier that was supplied in the call to CRYPTc_verifyCertificateChain.
`result`	One of the return codes, as specified above.

13.7 Random number generation

generateRandom

Generate random (or pseudo-random) data.

```
INT16 CRYPTa_generateRandom (BYTE *randomData,
UINT16 randomLen)
```

`randomData`	Points to the location that receives the data. The AUS WAP Browser is responsible for the de-allocation of the data.
`randomLen`	The size in bytes of the random data to be generated.

The function should return CRV_OK on success or any of the constants defined in the table above on failure.

13.8 Types and constants

KeyExchangeParameters structure

This structure is used by the functions CRYPTa_keyTransportRSA, CRYPTa_keyAgreementDH and CRYPTa_keyAgreementECDH.

Type	Name	Description
UINT8	paramType	A constant from the table ParamType
KeyExchangeSuite	keyExchangeSuite	A constant from the table KeyExchangeSuite
union	_u	C union of the structures KeyParam, Certificates and SecretKey, defined below

paramType

The paramType variable of the KeyExchangeParameters structure can be set to the following constants:

Constant	Value	Description
PARAMTYPE_KEY	1	Used if the parameter is a key
PARAMTYPE_CERTIFICATE	2	Used if the parameter is a certificate

KeyExchangeSuite

The keyExchangeSuite variable of the KeyExchangeParameters structure can be set to the following constants (described in the WTLS specification [WAP-WTLS], Appendix A):

Constant	Value
KEY_EXCH_NULL	0
KEY_EXCH_SHARED_SECRET	1
KEY_EXCH_DH_ANON	2
KEY_EXCH_DH_ANON_512	3
KEY_EXCH_DH_ANON_768	4
KEY_EXCH_RSA_ANON	5
KEY_EXCH_RSA_ANON_512	6
KEY_EXCH_RSA_ANON_768	7

KEY_EXCH_RSA	8
KEY_EXCH_RSA_512	9
KEY_EXCH_RSA_768	10
KEY_EXCH_ECDH_ANON	11
KEY_EXCH_ECDH_ANON_113	12
KEY_EXCH_ECDH_ANON_131	13
KEY_EXCH_ECDH_ECDSA	14

Certificates structure

The structure is used in the structure KeyExchangeParameters.

Type	**Name**	**Description**
UINT16	bufLen	Size of buffer.
BYTE *	buf	The certificates as an octet string. Note, the definition of how the Certificate data is to be parsed is found in the standard specifications WTLS [WAP-WTLS], where the header and the WTLS certificate is described. The two other possible kinds of certificates are described in [X.509] and [X.968]. (The same kind of byte string is passed from the function CRYPTa_verifyCertificateChain.)

KeyParam structure

The structure is used in the structure KeyExchangeParameters.

Type	**Name**	**Description**
PublicKey	pubKey	Public key that is being certified. See definition below.
ParameterSpecifier	parameterSpecifier	Specifies parameter relevant for the public key. See definition below.

PublicKey structure

The structure is used in the structure KeyParam.

Type	Name	Description
union	_u	C union of the types PublicKey_RSA, PublicKey_DH and PublicKey_EC.

PublicKey_RSA structure

The structure is used in the structure PublicKey.

Type	Name	Description
UINT16	expLen	Length of the exponent string
BYTE *	exponent	The exponent of the RSA key, using the big-endian (network-byte order) representation of the integer as octet string
UINT16	modLen	Length of the modulus string
BYTE *	modulus	The exponent of the RSA key, using the big-endian (network-byte order) representation of the integer as octet string

PublicKey_DH structure

The structure is used in the structure PublicKey.

Type	Name	Description
UINT16	len	The length of the string "y"
BYTE *	y	The Diffie-Hellman public value (Y) as octet string

PublicKey_EC structure

The structure is used in the structure PublicKey.

Type	Name	Description
UINT16	len	
BYTE *	point	The EC public key W = sG [P1363] as octet string. The representation format is defined in [X9.62], section 4.3.6.

ParameterSpecifier structure

The structure is used in the structure KeyParam.

Type	Name	Description
BYTE	parameterIndex	Indicates parameters relevant for this key exchange suite: 0 = not applicable, or specified elsewhere. 1-254 = assigned number of a parameter set, defined in [WAP-WTLS], Appendix A. 255 = explicit parameters are present in the variable params.
UINT16	paramLen	The length of the data that the variable params holds
BYTE *	params	Explicit parameters, e.g., Diffie-Hellman or ECDH parameters

SecretKey structure

The structure is used in the structure KeyExchangeParameters.

Type	Name	Description
BYTE *	identifier	An id of a secret key that shall be used.
INT16	idLen	The number of bytes the (not zero-terminated) id takes.

CipherMethod structure

The type CipherMethod, used in the function CRYPTa_getCipherMethods, is defined as follows:

Type	Name	Description
BulkCipherAlgorithm	bulkCipherAlg	Described below
HashAlgorithm	hashAlg	Described below

BulkCipherAlgorithm

Structure to be used for bulk encryption. Bulk cipher algorithms are listed in the WTLS specification, Appendix A. The type CipherMethod and the functions CRYPTa_decrypt, CRYPTa_encryptInit and CRYPTa_encrypt have arguments of the type BulkCipherAlgorithm that can be called with the following set of constants (described in the WTLS specification, Appendix A):

Constant	Value
CIPHER_NULL	0
CIPHER_RC5_CBC_40	1
CIPHER_RC5_CBC_56	2
CIPHER_RC5_CBC	3
CIPHER_DES_CBC_40	4
CIPHER_DES_CBC	5
CIPHER_3DES_CBC_EDE	6
CIPHER_IDEA_CBC_40	7
CIPHER_IDEA_CBC_56	8
CIPHER_IDEA_CBC	9

HashAlgorithm

The type CipherMethod and the functions CRYPTa_hash, CRYPTa_hashInit and CRYPTa_PRF use the following constants with the arguments of type HashAlgorithm:

Constant	Value	Description
HASH_SHA	1	The type of hash algorithm is SHA-1
HASH_MD5	2	The type of hash algorithm is MD5

KeyObject structure

The bulk encryption routines CRYPTa_encrypt, CRYPTa_encryptInit and CRYPTa_decrypt uses the type KeyObject to transfer the parameters:

Type	Name	Description
BYTE *	key	The key as octet string
UINT16	keyLen	The length of the key
BYTE *	iv	Initialisation vector
UINT16	ivLen	The length of the initialisation vector

EncryptHandle

A handle of this type is given to the AUS WAP Browser with the function CRYPTa_encryptInit. The AUS WAP Browser then uses the handle in subsequent calls to CRYPTa_encryptUpdate and CRYPTa_encryptFinal. The handle is defined as a pointer to the type VOID.

HashHandle

A handle of this type is given to the AUS WAP Browser with the function CRYPTa_hashInit. The AUS WAP Browser then uses the handle in subsequent calls to CRYPTa_hashUpdate and CRYPTa_hashFinal. The handle is defined as a pointer to the type VOID.

14 USSD API

This API defines the interface between the AUS WAP Browser and the USSD service in the Host Device environment. The AUS WAP Browser is implemented to use GSM phase 2 USSD.

USSD is a dialog based GSM supplementary service. The dialogs may be both mobile and network initiated, i.e. may be use for both content retrieval and content push operations. An USSD operation consists of two parts, the DCS (Data Coding Scheme) and the data to send. The data is assembled by the AUS WAP Browser. Depending on which party that the operation originates from, the DCS is different. If it is mobile originated, the DCS is required by GSM 03.38 to be 0x0F, which indicates “Language unspecified” and “Default alphabet”. The DCS value (the four most significant bits) for network originated operations is operator specific. However, 1110 is used for WAP. The four least significant bits may be set as follows:

xxxx 00xx	Reserved
xxxx 01xx	8.bit data
xxxx 10xx	Reserved
xxxx 11xx	Reserved
xxxx xx00	No message class
xxxx xx01	Class 1 Default meaning: ME-specific
xxxx xx10	Class 2 SIM specific message
xxxx xx11	Class 3 Default meaning: TE-specific

A recommendation is that the DSC is set to 0xE5. The data coding scheme and the transmitted data of the receiving and sending operations are described and defined in the specification [WAP-USSD].

Addressing of the gateway through USSD network node is done with a service code (see the configuration variable configUSSD_C) that identifies the USSD network node. This is a network operator dependent address string. The address is then completed with an IP number or a MSISDN number (which one is determined through configUSSD_GW_TYPE) identifying the WAP gateway (configUSSD_GW).

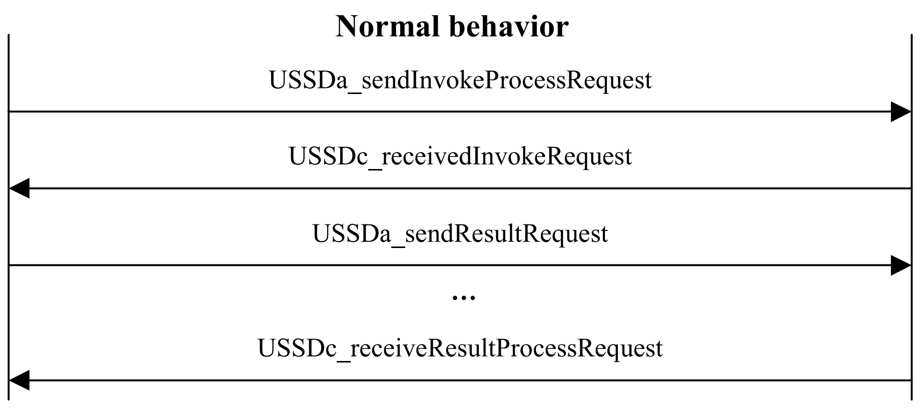
In WAP 1.2 is both address parts mandatory. In WAP 1.1 is the second part, the IP number or the MSISDN number that identifies the WAP gateway, optional. The AUS WAP Browser supports both modes. If a gateway, which is accessed with both addresses, fails, the AUS WAP Browser tryes again, this time using the service code only.

14.1 USSD dialogue scenarios

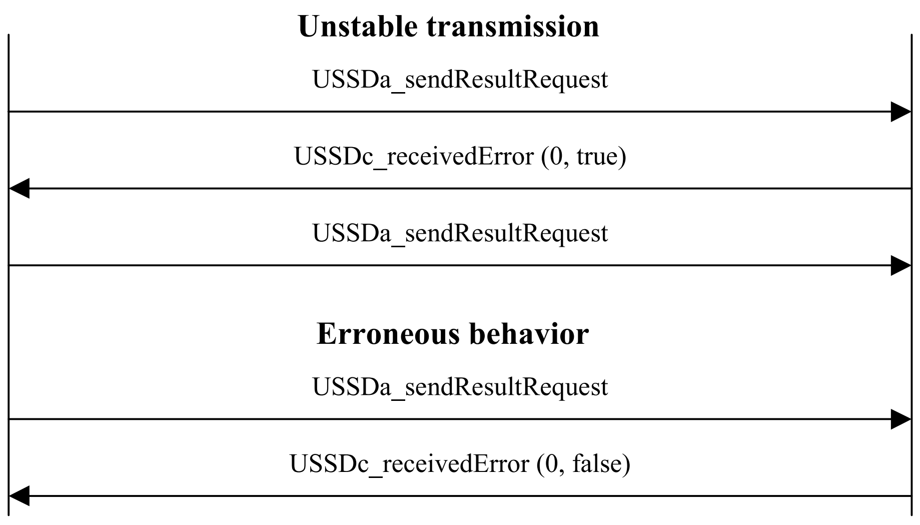
The functions in this API will be used differently dependent on whether the dialogue is mobile or network initiated.

14.1.1 Mobile initiated dialogues

The following diagram states the normal behavior of a mobile initiated dialogue. The dialogue is started with USSDa_sendInvokeProcessRequest. The dialogue continues by exchanging of a number of USSDc_receiveInvokeRequest and USSDa_sendInvokeProcessRequest between the mobile and the network. The dialogue is ended with the network sending USSDc_receiveResultProcessRequest.

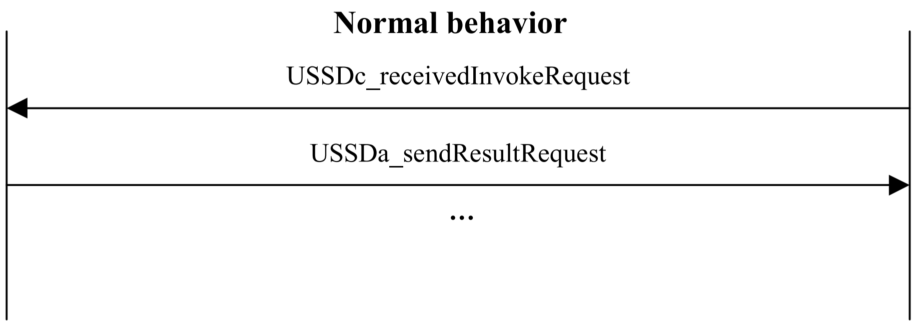

Normal behavior
USSDa_sendInvokeProcessRequest
USSDc_receivedInvokeRequest
USSDa_sendResultRequest
...
USSDc_receiveResultProcessRequest

If an error occurs in the Host Device Environment (the mobile) or if the transmission is unstable, an error message must be propagated to the AUS WAP Browser. The dialogue is assumed terminated when the AUS WAP Browser receives the error message and isKept is false.



14.1.2 Network initiated dialogues

Network initiated dialogs do not differ in any other mean that the special invocation procedure from the mobile is omitted:



While the dialogue proceeds there is no difference from a mobile initiated dialogue. Error handling is done in the same way.

14.2 Mobile initiated dialogues

The following functions are only used in mobile initiated dialogues. They are used in order to setup the dialog, when it is mobile initiated, as well as disconnecting it when the dialogue is done. The first datagram will be sent during this setup phase. The following datagrams are managed by the functions described in the next section.

sendInvokeProcessRequest

The AUS WAP Browser uses the following function when an USSD dialogue is to be initiated. It is done when the first datagram segment is to be sent.

```
VOID USSDa_sendInvokeProcessRequest (const CHAR
*data, UINT8 stringLength)
```

`data`	The data to send. The data is deleted when the function returns.
`stringLength`	The data is not zero-terminated. The length is therefore given by this argument.

receivedResultProcessRequest

This function is used when the USSD WAP gateway has terminated a mobile initiated USSD dialogue.

```
VOID USSDc_receivedResultProcessRequest (const CHAR
*data, UINT8 stringLength)
```

`data`	The data to send. The data may be deleted when the function returns.

`stringLength`	The data is not zero-terminated. The length is therefore given by this argument.

14.3 Mobile and and Network initiated dialogues

The AUS WAP Browser uses the following functions to proceed a mobile initiated USSD dialogue. The functions are also used directly from the beginning in a network-initiated dialogue.

receivedInvokeRequest

This function is used when an USSD string has been received.

```
VOID USSDc_receivedInvokeRequest (const CHAR *data,
UINT8 stringLength)
```

`data`	The data to send. The data may be deleted when the function returns.
`stringLength`	The data is not zero-terminated. The length is therefore given by this argument.

sendResultRequest

This function is used to send an USSD string within an established USSD dialogue.

```
VOID USSDa_sendResultRequest (const CHAR *data,
UINT8 stringLength)
```

`data`	The data to send. The data is deleted when the function returns.
`stringLength`	The data is not zero-terminated. The length is therefore given by this argument.

sendAbort

The AUS WAP Browser uses this function in order to abort the USSD dialogue when an internal error occurs in the AUS WAP Browser, or when the AUS WAP Browser is terminated and a dialogue is running.

```
VOID USSDa_sendAbort (VOID)
```

receivedError

This function is used when an error or reject reply has been received.

```
VOID USSDc_receivedError (UINT8 message, BOOL
isKept)
```

`message`	The error message can be any numeric code that is appropriate for a particular device. When the AUS WAP Browser cannot make any qualitative judgements over the codes, they are passed to the WAP application via the CLNTa_error function.
`isKept`	The USSD dialogue is kept or released depending on the parameter isKept.

receivedRelease

This function is called when a USSD dialogue has been terminated in the network.

```
VOID USSDc_receivedRelease (VOID)
```

15 SMS API

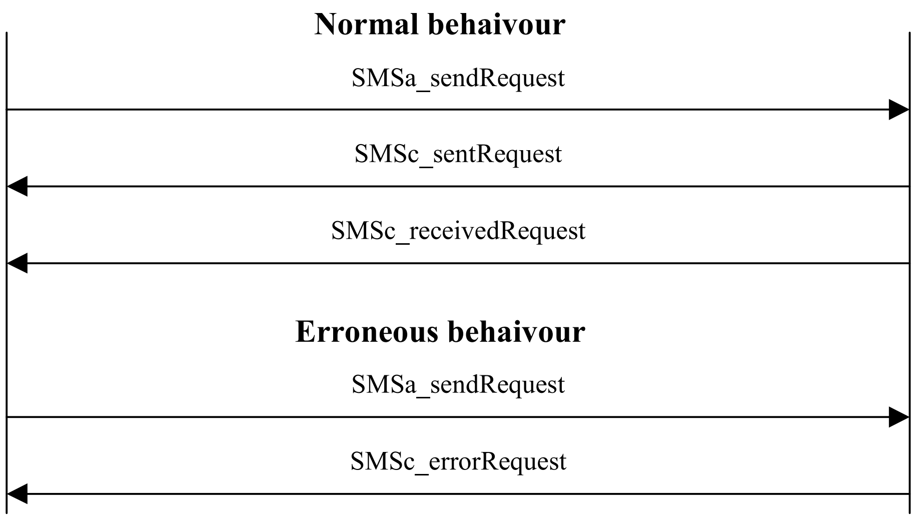
This API defines the interface between the AUS WAP Browser and the SMS service in the Host Device environment. The AUS WAP Browser is implemented to send and receive GSM phase 2 SMS with binary User Data Headers (UDH). When a SMS is sent from the AUS WAP Browser, the WAP application must add the remaining header information, before it is sent to the SMSC. When a SMS arrives from the SMSC to the WAP application, all header information but the UDH and the user data must be removed, before the SMS is sent further to the AUS WAP Browser.

Segmentation and reassembly of large data is handled within the AUS WAP Browser, as defined in GSM 03.40, i.e. SMS that is sent from or received to the AUS WAP Browser do not exceed 140 bytes.

In order to distinguish a SMS that is aimed for another application than a WAP application (like the inbox of the mobile phone), a router of SMS must be used. To recognise a WAP SMS, the following statements can be tested, one by one, until a statement is verified:

1. The UDHI field of the SMS header is given the value 1, if the SMS is a WAP SMS.
2. The originator port number (byte 6 – 7 in the UDH) is one of 9200 – 9203 if the SMS originates from a WAP SMS-C.
3. The destinator port number (byte 4 – 5 in the UDH) is one of 2948 or 2949 if the SMS is a pushed SMS, originating from a WAP SMS-C.
4. Compare addresses of originator in the received SMS with a list of addresses of destinators for SMS that have been sent from the WAP application. If the address exist in the list, it originates from a WAP SMS-C. However, this criterion cannot be used if the SMS-C is used for both ordinary SMS and WAP SMS.

The functions are called as in the figure below. The arguments are given as fictive values in order to illustrate how the values in the Adapter function are used in the resulting Connector function.



sendRequest

This function is used to send a SMS string. There cannot be two calls in a sequence to this function without waiting for a call of the function SMSc_sentRequest in between.

```
VOID SMSa_sendRequest (const CHAR *smsc, UINT8
smscLength, const CHAR *destination, UINT8
destLength, const CHAR *data, UINT8 dataLength)
```

`smsc`	The address (msisdn number) for the smsc. The string is deleted when the function returns.
`smscLength`	The length of the smsc string.
`destination`	The msisdn or IP number of the WAP gateway. The string is deleted when the function returns.
`destLength`	The length of the destination address string.
`data`	The data to send. The string is deleted when the function returns.
`dataLength`	The length of the data string.

sentRequest

This function is used when a SMS string has been sent and received by the SMSC. When the function has been called, calls to SMSa_sendRequest may come again. If the SMS service, of any reason, not can be accessed, the SMSc_receivedError must be used. If the neither the SMSc_sentRequest or the SMSc_receivedError functions are called, the AUS WAP Browser is blocked for communication over SMS until the application environment performs timeout of the transaction.

```
VOID SMSc_sentRequest (VOID)
```

receivedRequest

This function is used when a SMS string has been received. In order to distinguish between SMS that targets the WAP application and SMS that targets the ordinary inbox of the mobile phone, it is necessary to read the SMS header. If the SMS is targeted to the inbox of the mobile phone and the SMS contains a user data header, the SMS can be a WAP message. However, it can also be another application specific message, if other applications are supported that use SMS as data transmitter. Because of that, it might be necessary to check the servers port number in the user data header, as well. This header is specified in the WDP specification (appendix A and B).

```
VOID SMSc_receivedRequest (const CHAR *source,
UINT8 sourceLength, const CHAR *data, UINT8
dataLength)
```

`source`	The msisdn or IP number of the WAP gateway. The string may be deleted when the function returns. The source is equal to the destination of the SMSa_sendRequest call this call responds.
`sourceLength`	The length of the source string.
`data`	The data received. The string may be deleted when the function returns.
`dataLength`	The length of the data string.

receivedError

This function is used when a SMS datagram cannot be delivered to the SMSC. The reason can be any internal or external. When the AUS WAP Browser cannot make any qualitative judgements over the codes, they are passed to the WAP application via the CLNTa_error function with the type argument set to BEARER.

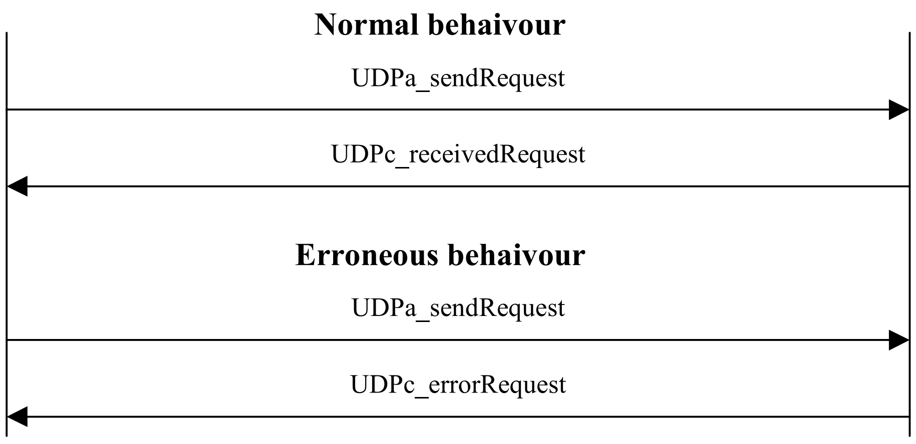
```
VOID SMSc_receivedError (UINT8 message)
```

`message`	The error message can be any numeric code that is appropriate for a particular device.

16 UDP API

This API defines the interface between the AUS WAP Browser and the UDP service in the Host Device environment. The UDP Adapter functions must be implemented for a specific Host Device. The Connector functions are Host Device independent and may be used immediately.

The functions are called as in the figure below. The arguments are given as fictive values in order to illustrate how the values in the Adapter function are used in the resulting Connector function.



sendRequest

This function is used to send an UDP datagram.
Note that the IP network already must be established when this call is done. If it is not, the request can time out before the data call and the network has been set up. This will happen if the set-up time exceeds the value of the configuration value configTIMEOUT. If the data call will be set-up during this call, the easiest way to avoid time out is to set the configuration variable to a reasonable higher value while the network not has been set up. 2 minutes is a normal interval if the set-up time is 60 seconds and the normal time out interval 60 seconds. After it is established, it can be decreased to its normal level (60 seconds).

```
VOID UDPa_sendRequest (const CHAR *data, UINT16
dataLength, const CHAR *destination, UINT8
destLength, const CHAR *source, UINT8 sourceLength,
UINT16 destinationPort, UINT16 sourcePort)
```

`data`	The data to send. The string is deleted when the function returns.
`dataLength`	The length of the data string.
`destination`	The IP number of the WAP gateway. The address is taken as is from the value of the configuration variable configUDP _IP_GW. The string is deleted when the function returns.

`destLength`	The length of the destination address string.
`source`	The IP number for the WAP application. The address is taken as is from the value of the configuration variable configUDP _IP_SRC. The string is deleted when the function returns.
`sourceLength`	The length of the source address string.
`destinationPort`	The destination port is the physical port on the server, e.g., 9200 for connection-less, non-secure transmission.
`sourcePort`	The source port is the port number of the device. The number has been retrieved with the function CLNTc_usePort.

receivedRequest

This function is used when an UDP datagram has been received.

```
VOID UDPc_receivedRequest (const CHAR *data, UINT16
dataLength, const CHAR *destination, UINT8
destLength, const CHAR *source UINT8 sourceLength,
UINT16 destinationPort, UINT16 sourcePort)
```

`data`	The data received. The string may be deleted when the function returns.
`dataLength`	The length of the data string.
`destination`	The destination is the address of the WAP application. The string may be deleted when the function returns.
`destLength`	The length of the destination address string.
`source`	The source is the address of the WAP gateway. The string may be deleted when the function returns.
`sourceLength`	The length of the source address string.
`destinationPort`	The destination port is the same port number as were given as source port in the UDPa_sendRequest call.
`sourcePort`	The source port is the physical port where the datagram origins, e.g., 9200 for connection-less, non-secure transmission.

errorRequest

This function is used when UDP datagram cannot be delivered from a certain port.

```
VOID UDPc_errorRequest (UINT8 message, UINT16
destinationPort)
```

`message`	The reason, which can be any internal or external, is passed in the message argument. The error message can be any numeric code that is appropriate for a particular device. When the AUS WAP Browser cannot make any qualitative judgements over the codes, they are passed to the WAP application via the CLNTa_error function with the type argument set to BEARER.
`destinationPort`	The destinationPort is the port of the WAP application that the datagram should have been delivered to, i.e., the same port number as were given as source port in the UDPa_sendRequest call.

17 WAP Device API

This API defines the interface between the AUS WAP Browser and a generic bearer in the Host Device environment. It is supposed to be used when accessing local WAP devices.

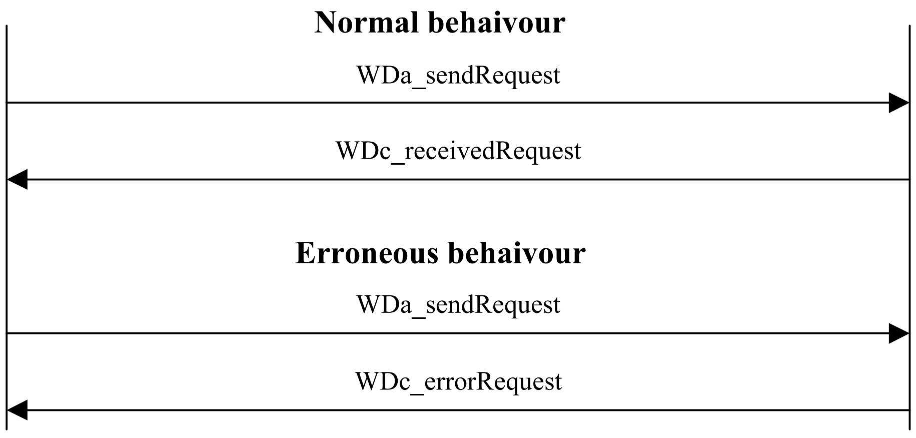
The protocol over this API is connection-less, also when the dynamic configuration variable configSTACKMODE has been set with the constants MODE_CO_WSP, MODE_CO_WTLS and MODE_CO_WTA.

The API can be accessed at any time regardless of the chosen bearer (the dynamic configuration variable configBEARER). To access it, a URL with the specific scheme wapdevice is to be used. Ex:

```
<go href="wapdevice://host/path/file?variables#fragment">
```

The WAP Device adapter functions must be implemented for a specific Host Device and a specific bearer. The connector functions are Host Device independent and may be used immediately.

The functions are called as in the figure below. The arguments are given as fictive values in order to illustrate how the values in the adapter function are used in the resulting connector function.



sendRequest

This function is used to send a datagram over a generic network.
Note that the generic network already must be established when this call is done. If it is not, the request can time out before the data call and the network has been set up. This will happen if the set-up time exceeds the value of the configuration value configTIMEOUT. If the data call will be set-up during this call, the easiest way to avoid time out is to set the configuration variable to a reasonable higher value while the network not has been set up. 2 minutes is a normal interval if the set-up time is 60 seconds and the normal time out interval 60 seconds. After it is established it can be decreased to its normal level (60 seconds).

```
VOID WDa_sendRequest (const CHAR *data, UINT16
dataLength, UINT16 destinationPort, UINT16
sourcePort)
```

`data`	The data to send. The string is deleted when the function returns.
`dataLength`	The length of the data string.
`destinationPort`	The destination port is the physical port on the server, e.g., 9200 for connection-less, non-secure transmission.
`sourcePort`	The source port is a logical port number that doesn’t have to correspond to a physical port with the same number on the device. Two different calls to WDa_sendRequest, with different source port numbers, must also use two different physical port numbers.

receivedRequest

This function is used when a datagram has been received.

```
VOID WDc_receivedRequest (const CHAR *data, UINT16
dataLength, UINT16 destinationPort, UINT16
sourcePort)
```

`data`	The data received. The string may be deleted when the function returns.
`dataLength`	The length of the data string.
`destinationPort`	The destination port is the same logical port number as were given as source port in the WDa_sendRequest call.
`sourcePort`	The source port is the physical port where the datagram origins, e.g., 9200 for connection-less, non-secure transmission.

errorRequest

This function is used when a datagram cannot be delivered.

```
VOID WDc_errorRequest (UINT8 message, UINT16
destinationPort)
```

`message`	The reason, which can be any internal or external, is passed in the message argument. The error message can be any numeric code that is appropriate for a particular device. When the AUS WAP Browser cannot make any qualitative judgements over the codes, they are passed to the WAP application via the CLNTa_error function with the type argument set to BEARER.
`destinationPort`	The destinationPort is the port of the WAP application that the datagram should have been delivered to, i.e., the same port number as were given as source port in the WDa_sendRequest call.

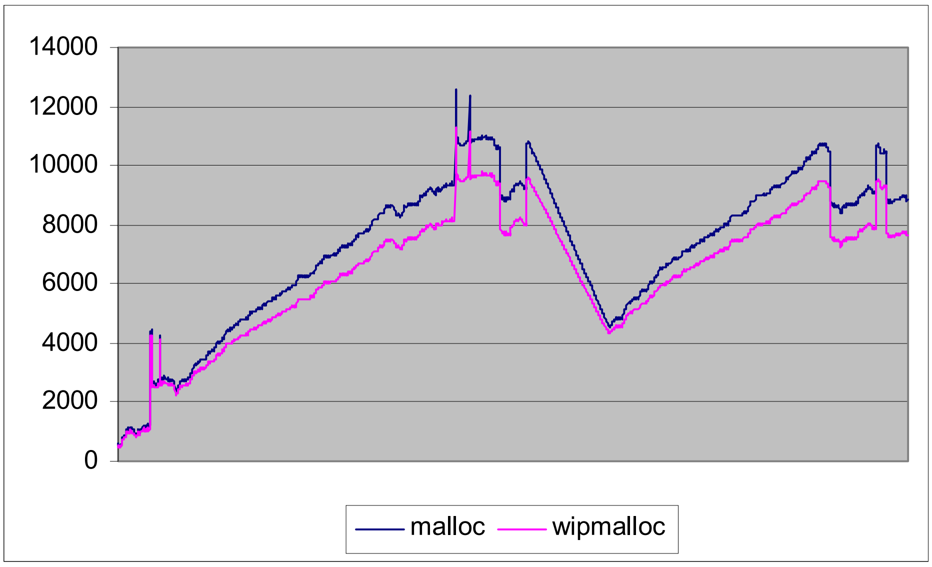
18 Optional source code

The AUS WAP Browser is delivered with optional source code that can be compiled and linked with the AUS WAP Browser. The AUS WAP Browser generally uses the optional source code when certain C macros have been set.

18.1 Memory

If the environment, the WAP application using the AUS WAP Browser is developed for, not has a memory allocator. Or if the memory allocator returns fixed sized memory blocks, which generally are much larger than the requested size. Or if memory allocator exists and there is not much RAM memory in general.

The optional memory allocator can be used in these cases. It allocates the requested memory with an overhead of 4 extra bytes plus alignment to nearest larger multiple of 4. It never allocates a chunk less than 8 bytes. Comparing the ANSI C memory allocator *malloc* with the AUS WAP Browser memory allocator, *malloc* consumes 8 bytes more per allocation due to extra overhead. This means that the AUS WAP Browser memory allocator consumes about 2 Kbytes less than malloc when large WML decks are opened. See diagram below.



The diagram illustrates the total memory consumption when two, in a sequence equally sized WML decks (compiled size is 1400 bytes) is downloaded. The AUS WAP Browser uses a garbage collector in situations when it must be reset, in order to release memory allocated elsewhere in the AUS WAP Browser, at any occasion. The garbage collector is integrated in the AUS WAP Browser allocator but not *malloc*, hence the greater overhead for *malloc*.

The file include\confvars.h contains the C macros necessary to set-up the AUS WAP Browser for using its own memory allocator. If USE_WIP_MALLOC is defined the internal memory allocator will be used. The size of the total memory is set by another macro in that file, WIP_MALLOC_MEM_SIZE. Its default size is 25 Kbytes.

The memory allocator can be used by the WAP application as well. In this case the memory allocator must normally be initialised by the WAP application, before the AUS WAP Browser is started (CLNTc_start). The API for the functions that can be used by the WAP application is stated in below. The header file \source\optional\memory\wip_mem.h should be included where these functions are called.

initmalloc

This function is used to initialise the memory allocator with. This should only be used if the memory allocator shall be used outside the scope of the AUS WAP Browser. If the WAP application shall call this function, the AUS WAP Browser shall be told to not do. This is achieved by remove the definition of the C macro INITIALISE_WIP_MALLOC from \include\confvars.h. If it is not defined, the AUS WAP Browser does not call this function and does not use the macro WIP_MALLOC_MEM_SIZE.

```
VOID *wip_initmalloc (VOID *memory, UINT32 size)
```

`memory`	A pointer to the variable holding the memory is past to the argument
`Size`	The size of the memory

malloc

This function is used to allocate memory. It corresponds to the ANSI C library function malloc.

```
VOID *wip_malloc (UINT32 size)
```

`size`	The size of the memory

free

This function is used to de-allocate memory. It corresponds to the ANSI C library function free.

```
VOID wip_free (VOID *memory)
```

`memory`	A pointer to the memory (allocated by wip_malloc) is given in the memory argument. If NULL is passed, nothing is done.

18.2 Charset

In the Client API, there is a Connector function for adding additional character transcoding algorithms. The transcoder should convert from any given character set to the Unicode character set (UCS16). The AUS WAP Browser has transcoding algorithms for US-ASCII, UTF-8 and ISO-8859-1.

This optional source code package contains algorithms and tables to convert from KS C 5601 (Korean character set) to Unicode. The implementation is performed after the descriptions in the Client API.

If the C macro USE_CHARSET_PLUGIN is defined in confvars.h (or makefile or project file), the software will be used by the AUS WAP Browser when a WML deck encoded with the KS C 5601 character set is received.

The software can be used for transcoding text strings in the MMI API, as well. For instance, MMIa_newText supplies the text string in Unicode format. In order to convert it to KS C 5601 before putting it on the display, functions in this software can be used. On the reverse, MMIc_getInputString requires the input string being in Unicode format. In order to convert from KS C 5601 to Unicode before this function is called, functions in this software can be used. Before these functions are used, \source\optional\charset\HCodeCnvt.h should be included.

Uni2KSCString

This function converts an Unicode string to KS C 5601 string. This function returns the length (number of bytes) of kscString.

```
int Uni2KSCString (WCHAR *uniString, UCHAR
*kscString)
```

`uniString`	Should be set to a pointer a double zero terminated array string that holds the Unicode encoded string that shall be converted to a KS C 5601 encoded String
`kscString`	Points to the string buffer. Before this argument is used, the memory space must be allocated to this pointer to contain the converted string.

KSC2UniString

This function converts a KS C 5601 string to an Unicode string. This function returns the length of uniString (number of WCHAR characters).

```
int KSC2UniString (UCHAR *kscString, WCHAR
*uniString)
```

`kscString`	Should be set to a pointer to the zero terminated KS C 5601 encoded string that will be converted to Unicode encoded string
`uniString`	Points to the string buffer. Before this argument is used, the memory space must be allocated to this pointer to contain the converted string (double zero terminated).

KSCStrLenOfUni

This function calculates the number of bytes that requires converting an Unicode encoded string to a corresponding KS C 5601 encoded String. It returns the number of bytes that must be allocated to store the corresponding KS C 5601 string as a Unicode string. The value this function returns are not exactly the same as the memory space that is needed to convert Unicode String to KS C 5601 string, but is equivalent to or greater than the minimum memory space needed.

```
int KSCStrLenOfUni (WCHAR *uStr)
```

`uStr`	Shall be set to a Unicode encoded string

UniLenOfKSCStr

This function calculates the number of WCHAR characters that requires converting a KS C 5601 encoded string to a corresponding Unicode encoded string. It returns the number of bytes that must be allocated to store the Unicode string as a KS C 5601 string.

```
int UniLenOfKSCStr (CHAR *kStr)
```

`kStr`	Shall be set to a KS C 5601 encoded string

Appendix, Error codes

This appendix states all error codes for the Client API Adapter function CLNTa_error, which is defined like this:

```
CLNTa_error (UINT8 viewId, INT16 errorNo, UINT8
             errorType);
```

The error codes are described as follows:

C enum: Constant representing the error number (declared in errcodes.h)
Type: The type of error. They are described in the following section.
Source: The possible source or sources of the problem. They are described in the following section.
Description: A description of the error.

Possible kinds of error

The attribute errorType can take the following values (defined in errcodes.h):

Type	**Description**
ERRTYPE_ INFORMATION	*Type I in the section below, with error codes.* Indicates this is an error the user might want to know about, but it is in no way critical to the WAP application, i.e. no actions besides showing the information to the user is needed. Example: WML file not found, error in WML parse.
ERRTYPE_ CRITICAL	*Type C in the section below, with error codes.* This is a type of error, which indicates when something has gone wrong in the AUS WAP Browser. It is not directly fatal – but may be if left unattended. This could be due to a misuse of connector functions by the WAP application (typically when debugging). Example. To many views started.
ERRTYPE_ FATAL	*Type F in the section below, with error codes.* A serious error has occurred and the AUS WAP Browser should be shut down immediately. Ex. out of memory.
ERRTYPE_ BEARER	*Type B will not be found in the section below since a bearer related error has occurred.* The error code is in this case related to the actual bearer rather than the error codes defined by the AUS WAP Browser.

Possible sources of error

Each error code (errorNo) can be derived to one or several of the following sources (defined in errcodes.h):

Source	Description
WML/WMLS content	*Category C in the section below, with error codes.* Indicates that an error were encountered when the browser tried to read the WML/WMLS content.
Content server	*Category S in the section below, with error codes.* Indicates that an error was encountered when the content server tried to retrieve the requested file.
WAP proxy server	*Category P in the section below, with error codes.* Indicates that an error was encountered when the proxy (WAP) server tried to transfer the requested file to the AUS WAP Browser by help of the WAP protocol stack.
Application	*Category A in the section below, with error codes.* Indicates that an internal error in the browser has occurred.
User	*Category U in the section below, with error codes.* Indicates that an error was encountered when the user has performed any illegal actions, like entering a malformed URL.
Development	*Category D in the section below, with error codes.* This is a type of error that occurs during development of a browser. It indicates when something has gone wrong in the AUS WAP Browser. Example. To many views started.
Transmission	*Category T in the section below, with error codes*. The error occurred in the underlying transmission protocols that the protocols in WAP use, i.e., SMS, UDP or USSD.

Error codes

This section states the error numbers that the attribute errorNo can have (defined in errcodes.h). The types are described in a section above, as well as the possible sources of errors.

0 **C enum:** ERR_WTP_UNKNOWN
Type: I
Source: P, A
Description: WTP Abort reason according to WTP Specification: A generic error code indicating an unexpected error.

1 **C enum:** ERR_WTP_PROTOERR
Type: I
Source: P, A

Description: WTP Abort reason according to WTP Specification: The received PDU could not be interpreted. The structure MAY be wrong.

2 **C enum:** ERR_WTP_INVALIDTID
Type: I
Source: P, A
Description: WTP Abort reason according to WTP Specification: Only used by the Initiator as a negative result to the TID verification.

3 **C enum:** ERR_WTP_NOTIMPLEMENTEDCL2
Type: I
Source: P, A
Description: WTP Abort reason according to WTP Specification: The transaction could not be completed since the Responder does not support Class 2 transactions.

4 **C enum:** ERR_WTP_NOTIMPLEMENTEDSAR
Type: I
Source: P, A
Description: WTP Abort reason according to WTP Specification: The transaction could not be completed since the Responder does not support SAR.

5 **C enum:** ERR_WTP_NOTIMPLEMENTEDUACK
Type: I
Source: P, A
Description: WTP Abort reason according to WTP Specification: The transaction could not be completed since the Responder does not support User acknowledgements.

6 **C enum:** ERR_WTP_WTPVERSIONONE
Type: I
Source: P, A
Description: WTP Abort reason according to WTP Specification: Current version is 0. The initiator requested a different version that is not supported.

7 **C enum:** ERR_WTP_CAPTEMPEXCEEDED
Type: I
Source: P, A
Description: WTP Abort reason according to WTP Specification: Due to an overload situation the transaction can not be completed.

8 **C enum:** ERR_WTP_NORESPONSE
Type: I
Source: P, A
Description: WTP Abort reason according to WTP Specification: A User acknowledgement was requested but the WTP user did not respond.

9 **C enum:** ERR_WTP_MESSAGETOOLARGE
Type: I
Source: P, A

Description: WTP Abort reason according to WTP Specification: Due to a message size bigger than the capabilities of the receiver the transaction cannot be completed.

64 **C enum:** HTTPBadRequest
Type: I
Source: S
Description: See HTTP 1.1 RFC (2068) – NOTE: these errors are NOT sent if the enclosed entity contains valid WML content

65 **C enum:** HTTPUnauthorized
Type: I
Source: S, P
Description: See BadRequest

66 **C enum:** HTTPPaymentRequired
Type: I
Source: S, P
Description: See BadRequest

67 **C enum:** HTTPForbidden
Type: I
Source: S, P
Description: See BadRequest

68 **C enum:** HTTPFileNotFound
Type: I
Source: S
Description: See BadRequest

69 **C enum:** HTTPMethodNotAllowed
Type: I
Source: S
Description: See BadRequest

70 **C enum:** HTTPNotAcceptable
Type: I
Source: S
Description: See BadRequest

71 **C enum:** HTTPProxyAuthenticationRequired
Type: I
Source: S, P
Description: See BadRequest

72 **C enum:** HTTPRequestTimeout
Type: I
Source: S
Description: See BadRequest

73	**C enum:** HTTPConflict **Type:** I **Source:** S **Description:** See BadRequest
74	**C enum:** HTTPGone **Type:** I **Source:** S **Description:** See BadRequest
75	**C enum:** HTTPLengthRequired **Type:** I **Source:** S **Description:** See BadRequest
76	**C enum:** HTTPPreconditionFailed **Type:** I **Source:** S **Description:** See BadRequest
77	**C enum:** HTTPRequestedEntityTooLarge **Type:** I **Source:** S **Description:** See BadRequest
78	**C enum:** HTTPRequestURITooLarge **Type:** I **Source:** S **Description:** See BadRequest
79	**C enum:** HTTPUnsupportedMediaType **Type:** I **Source:** S **Description:** See BadRequest
96	**C enum:** HTTPInternalServerError **Type:** I **Source:** S, P **Description:** See BadRequest
97	**C enum:** HTTPNotImplemented **Type:** I **Source:** S **Description:** See BadRequest
98	**C enum:** HTTPBadGateway **Type:** I **Source:** S **Description:** See BadRequest

99 **C enum:** HTTPServiceUnavailable
Type: I
Source: S
Description: See BadRequest

100 **C enum:** HTTPGatewayTimeout
Type: I
Source: S
Description: See BadRequest

101 **C enum:** HTTPVerNotSupported
Type: I
Source: S
Description: See BadRequest

224 **C enum:** ERR_WSP_PROTOERR
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: The rules of the protocol prevented the peer from performing the operation in its current state. For example, the used PDU was not allowed.

225 **C enum:** ERR_WSP_DISCONNECT
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: The session was disconnected while the operation was still in progress.

226 **C enum:** ERR_WSP_SUSPEND
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: The session was suspended while the operation was still in progress.

227 **C enum:** ERR_WSP_RESUME
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: The session was resumed while the operation was still in progress.

228 **C enum:** ERR_WSP_CONGESTION
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: The peer implementation could not process the request due to lack of resources.

229 **C enum:** ERR_WSP_CONNECTERR
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: An error prevented session creation.

230 **C enum:** ERR_WSP_MRUEXCEEDED
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: The SDU size in a request was larger than the Maximum Receive Unit negotiated with the peer.

231 **C enum:** ERR_WSP_MOREXCEEDED
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: The negotiated upper limit on the number of simultaneously outstanding method or push requests was exceeded.

232 **C enum:** ERR_WSP_PEERREQ
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: The service peer requested the operation to be aborted.

233 **C enum:** ERR_WSP_NETERR
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: An underlying network error prevented completion of a request.

234 **C enum:** ERR_WSP_USERREQ
Type: I
Source: P, A
Description: WSP Abort reason according to WSP Specification: An action of the local service user was the cause of the indication.

1003 **C enum:** ERR_WAE_UA_VIEWID_INVALID
Type: C
Source: D
Description: Invalid view id was used

1004 **C enum:** ERR_WAE_UA_MAX_EXCEEDED
Type: C
Source: D
Description: Too many active user agents

1006 **C enum:** ERR_WAE_UA_DISPLAY_ERROR
Type:

	Source: **Description:** MMI engine is not able to process content due to syntactically erroneous WML.
1007	**C enum:** ERR_WAE_UA_RESPONSE_BODY_INVALID **Type:** I **Source:** P **Description:** No proper WML response body, possibly empty
1008	**C enum:** ERR_WAE_UA_URL_INVALID **Type:** I **Source:** U, C **Description:** Malformed URL was encountered by the MMIc_loadURL function or in the active WML card.
1009	**C enum:** ERR_WAE_UA_URL_TIMEOUT **Type:** I **Source:** S, P **Description:** URL request timed out
1010	**C enum:** ERR_WAE_UA_WSP_RESPONSE_INVALID **Type:** I **Source:** S, P **Description:** Error in WSP header, e.g. no valid content type, non recognisable HTTP-WSP/B response was encountered
1011	**C enum:** ERR_WAE_UA_WMLDECK_ACCESS_DENIED **Type:** I **Source:** C **Description:** WML deck contained access restrictions not fulfilled by the user agent
1012	**C enum:** ERR_WAE_UA_URL_NONSUPPORTED_SCHEME **Type:** I **Source:** C **Description:** A non-supported scheme was used in the wrong context. This code is used in a CLNTa_content function call.
1013	**C enum:** ERR_WAE_UA_REDIRECT_ERROR **Type:** I **Source:** S **Description:** A request has been redirected more than 5 times
1014	**C enum:** ERR_WAE_UA_SESSION_NOT_CONNECTED **Type:** **Source:** **Description:**
1015	**C enum:** ERR_WAE_UA_MAX_NR_OF_SESSIONS_REACHED **Type:**

Source:
Description:

1001 **C enum:** ERR_WAE_WML_INSTREAM_FAULT
Type:
Source:
Description:

1002 **C enum:** ERR_WAE_WML_CONTENT_CHARSET_ERROR
Type: I
Source: S, P
Description: Error in the character set coding. This can be originated to the content server or to the proxy server if it performs transcoding.

1003 **C enum:** ERR_WAE_WML_CONTENT_CHARSET_NOT_SUPPORTED
Type: I
Source: A
Description: Character set coding not supported

1004 **C enum:** ERR_WAE_WML_UNKNOWN_TOKEN
Type: I
Source: P
Description: Illegal WML token

1005 **C enum:** ERR_WAE_WML_WML_ERROR
Type: I
Source: C, S, P
Description: Illegal WML, e.g. unexpected end of file

1006 **C enum:** ERR_WAE_WML_CONTENT_VERSION_WARNING
Type: I
Source: S, P
Description: If the version number of the WBXML specification is different than 01, a warning is issued. Parsing is not cancelled.

1007 **C enum:** ERR_WAE_WML_CONTENT_PUBLIC_ID_WARNING
Type: I
Source: S, P
Description: The public id is used to identify a well-known document type contained within the WBXML entity. The following numbers are accepted:

- Any number if the content-type is set to application/vnd.wap.wmlc and the WSP parameter "level", if present, is specified to 1.1
- The number 04 (WML/1.1)
- The number 00 if used with any of the two public identifier strings: "-//WAPFORUM//DTD wml 1.1//EN" and "http://www.wapforum.org/DTD/wml_1.1.xml".

A warning is issued and parsing is proceeded if the above stated criterions not can be verified and the public id is:

- The number 00 if used with a public identifier string different from the accepted ones.
- The number 01 (i.e. unknown, e.g. the WAP proxy could not validate the WML source)
- The number 02 (WML 1.0)

All other numbers are rejected (see the next error code).

1008 **C enum:** ERR_WAE_WML_CONTENT_PUBLIC_ID_ERROR
Type: I
Source: S, P
Description: The public id is used to identify a well-known document type contained within the WBXML entity. An error message is issued if the criterions for an acceptable public identifier fail. See the previous error code. Parsing is not proceeded if this message is issued.

1102 **C enum:** ERR_WAE_WMLS_VERIFICATION
Type: I
Source: C, P
Description: Verification failed, not proper byte-code

1103 **C enum:** ERR_WAE_WMLS_LIB
Type: I
Source: A
Description: Fatal library function error

1104 **C enum:** ERR_WAE_WMLS_FUNC_ARGS
Type: I
Source: C
Description: Invalid function arguments

1105 **C enum:** ERR_WAE_WMLS_EXTERNAL
Type: I
Source: C
Description: External function not found

1106 **C enum:** ERR_WAE_WMLS_LOAD
Type: I
Source: C
Description: Unable to load compilation unit

1107 **C enum:** ERR_WAE_WMLS_ACCESS
Type: I
Source: C
Description: Access violation

1108 **C enum:** ERR_WAE_WMLS_STACK_UNDERFLOW
Type: I
Source: C, P
Description: Stack underflow

1109	**C enum:** ERR_WAE_WMLS_ABORT **Type:** I **Source:** C **Description:** Programmed abort
1110	**C enum:** ERR_WAE_WMLS_STACK_OVRFLW **Type:** I **Source:** A **Description:** Stack overflow
1111	**C enum:** ERR_WAE_WMLS_USER_ABORT **Type:** I **Source:** U **Description:** User initiated abort
1112	**C enum:** ERR_WAE_WMLS_SYSTEM_ABORT **Type:** I **Source:** A **Description:** System initiated
1113	**C enum:** ERR_WAE_WMLS_NULL **Type:** I **Source:** C **Description:** Some component was inaccessible
2001	**C enum:** ERR_WSPCL_ErrorInAddressFromWAE **Type:** I **Source:** D **Description:** Address received from WAE incorrect
2002	**C enum:** ERR_WSPCL_ErrorExtractReplyPDU **Type:** I **Source:** P, A **Description:** Received reply PDU incorrect or no memory available for extraction
2004	**C enum:** ERR_WSPCL_ErrorNoBuffersAvailable **Type:** I **Source:** A **Description:** No free memory available
2005	**C enum:** ERR_WSPCL_ErrorMethodNotSupported **Type:** I **Source:** D **Description:** Requested method not supported.
3001	**C enum:** ERR_WSPCM_ErrorNoMemoryAvailableForPDUPacking **Type:** I **Source:** A **Description:** No memory available for PDU construction

3002

C enum: ERR_WSPCM_ErrorNoMemoryAvailableForPDUUnPacking
Type: I
Source: A
Description: No memory available for PDU extraction.

3003

C enum: ERR_WSPCM_ErrorInDataFromWAE
Type: I
Source: D
Description: Data from WAE erroneous.

3004

C enum: ERR_WSPCM_ErrorInReplyFromServer
Type: I
Source: P
Description: Data from server erroneous.

3005

C enum: ERR_WSPCM_ErrorMaxSessionsAlreadyReached
Type: I
Source: D
Description: Max number of active sessions already reached.

3006

C enum: ERR_WSPCM_ErrorMOMAlreadyReached
Type: I
Source: D
Description: Number of outstanding methods already reached.

3007

C enum: ERR_WSPCM_WAEErrorNoPIdFoundMatchingSession
Type: I
Source: D
Description: The request does not conform to an existing session..

3008

C enum: ERR_WSPCM_ErrorNoPIdFoundMatchingMethod
Type: I
Source: P
Description: Non requested data received, or a second answer of an old request.

3009

C enum: ERR_WSPCM_ErrorNoPIdFoundMatchingPush
Type: I
Source: P
Description: Non requested Push acknowledgement received.

3010

C enum: ERR_WSPCM_ErrorStoreOMInfoFailed
Type: I
Source: D
Description: Session data could not be stored.

3011

C enum: ERR_WSPCM_ErrorStoreHandleFailed
Type: I
Source: D
Description: Handle could not be stored.

3012	**C enum:** ERR_WSPCM_ErrorMethodNotSupported **Type:** I **Source:** D **Description:** Requested method not supported.
3013	**C enum:** ERR_WSPCM_ErrorSameAQUsed **Type:** I **Source:** D **Description:** Session disconnected because a new one is started using the same WAP gateway.
3014	**C enum:** ERR_WSPCM_WAEErrorNoPIdFoundMatchingMethod **Type:** I **Source:** D **Description:** WAE has given acknowledgement of a non-received request.
4001	**C enum:** ERR_WTP_ErrorNORESPONSE **Type:** I **Source:** P, S **Description:** No response PDU has been received, and no more re-transmissions to do.
4002	**C enum:** ERR_WTP_ErrorNOFREEBUFF **Type:** I **Source:** A **Description:** Out of memory, cannot proceed processing of transaction.
4003	**C enum:** ERR_WTP_ErrorINVALID_BEARER **Type:** I **Source:** D **Description:** Invalid bearer.
4004	**C enum:** ERR_WTP_ErrorINVALID_CLASS **Type:** I **Source:** D **Description:** Invalid transaction class.
4005	**C enum:** ERR_WTP_ErrorINVALID_ACKTYPE **Type:** I **Source:** D **Description:** Invalid user acknowledgement type.
5001-5099	**C enum:** None **Type:** I **Source:** A **Description:** General WTLS error. The WTLS connection is in most cases dropped.
5100-5199	**C enum:** None **Type:** I

Source: A
Description: Handshake failure. A WTLS connection could not be established.

5200-5299 **C enum:** None
Type: I
Source: A
Description: Error in crypty library. In most cases is the WTLS connection dropped.

5300-5399 **C enum:** None
Type: I
Source: P
Description: WTLS error message from WAP gateway. WTLS connection is dropped.

5400-5499 **C enum:** None
Type: I
Source: A
Description: Internal WTLS error. WTLS connection is dropped.

6001 **C enum:** ERR_WDP_ErrorInDatafromWSP
Type: I
Source: D
Description: The data received from WSP was faulty

6002 **C enum:** ERR_WDP_ErrorInDatafromWTP
Type: I
Source: D
Description: The data received from WTP was faulty

6003 **C enum:** ERR_WDP_ErrorBearerNotSupported
Type: I
Source: D
Description: The bearer is not supported

6004 **C enum:** ERR_WDP_WDErrorInd
Type: I
Source: T
Description: Bearer WD error

6005 **C enum:** ERR_WDP_UDPErrorInd
Type: I
Source: T
Description: UDP error

6006 **C enum:** ERR_WDP_UDPBigBuffer
Type: I
Source: A
Description: Received UDP datagram too big

6007	**C enum:** ERR_WDP_SMSErrorInd **Type:** I **Source:** T **Description:** SMS error
6008	**C enum:** ERR_WDP_SMSBigBuffer **Type:** I **Source:** A **Description:** Received SMS datagram too big
6009	**C enum:** ERR_WCMP_PortUnreachable **Type:** I **Source:** P, A **Description:** Erroneous port number used by AUS WAP Browser.
6010	**C enum:** ERR_WDP_WCMP_MessageTooBig **Type:** I **Source:** P **Description:** Datagram sent by AUS WAP Browser too big.
8001	**C enum:** ERR_MEMORY_WARNING **Type:** C **Source:** A **Description:** The memory has reached the level specified by the configuration constant MEMORY_WARNING in the file confvars.h. The AUS WAP Browser resets all of its context, i.e., the internal history and all WML variables. Can only be received if USE_MEMORY_GUARD is defined (in confvars.h).
8002	**C enum:** ERR_OUT_OF_MEMORY **Type:** F **Source:** A **Description:** Out of memory, i.e., the AUS WAP Browser cannot allocate more memory or, if memory guard is enabled, the memory usage has reached the upper limit. The AUS WAP Browser resets itself. The WAP application must reset its data and user interface. At this state, the AUS WAP Browser can be restarted again, without restarting the entire system.

Index