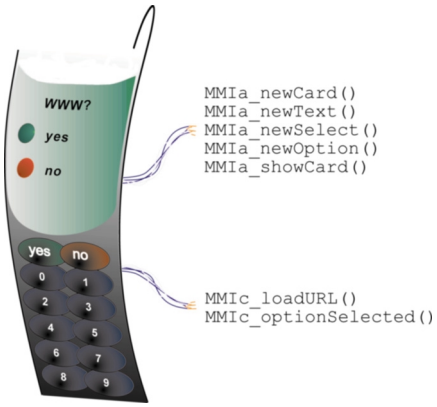
WAP

AUS WAP BROWSER USER'S MANUAL



VERSION 4.1

AU-SYSTEM

Document Reference: R109806019, version 4.1

AU-System AB

Scheelevägen 17, IDEON

SE – 223 70 Lund, SWEDEN

Telephone: +46 46 32 70 00

Fax: +46 46 32 70 01

http://www.ausys.com

Contents

1 Introduction

1.1 References

RFC2045	Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. http://ds.internic.net/rfc/rfc2045.txt
RFC2068	HTTP/1.1, http://ds.internic.net/rfc/rfc2068.txt
RFC2396	Uniform Resource Identifiers, http://ds.internic.net/rfc/rfc2396.txt
WMLS-CHRYPT	WMLScript Crypto Library Specification http://www.wapforum.org
WAP-USSD	WAP over GSM USSD http://www.wapforum.org
WAP-WSP	WAP, WSP Specification http://www.wapforum.org
WAP-WTP	WAP, WTP Specification http://www.wapforum.org
WAP-WTLS	WAP WTLS Specification http://www.wapforum.org
WAP-WDP	WAP WDP Specification http://www.wapforum.org
UAPROF	WAG UAPROF Specification 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	
DER	ISO/IEC 8825-2:1995 Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).
PKCS1	PKCS #1: RSA Encryption Standard", version 1.5, RSA Laboratories, November 1993.
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

PPG	Push Proxy Gateway
SI	Service Indication
SIA	Session Initiation Application
SL	Service Loading
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:

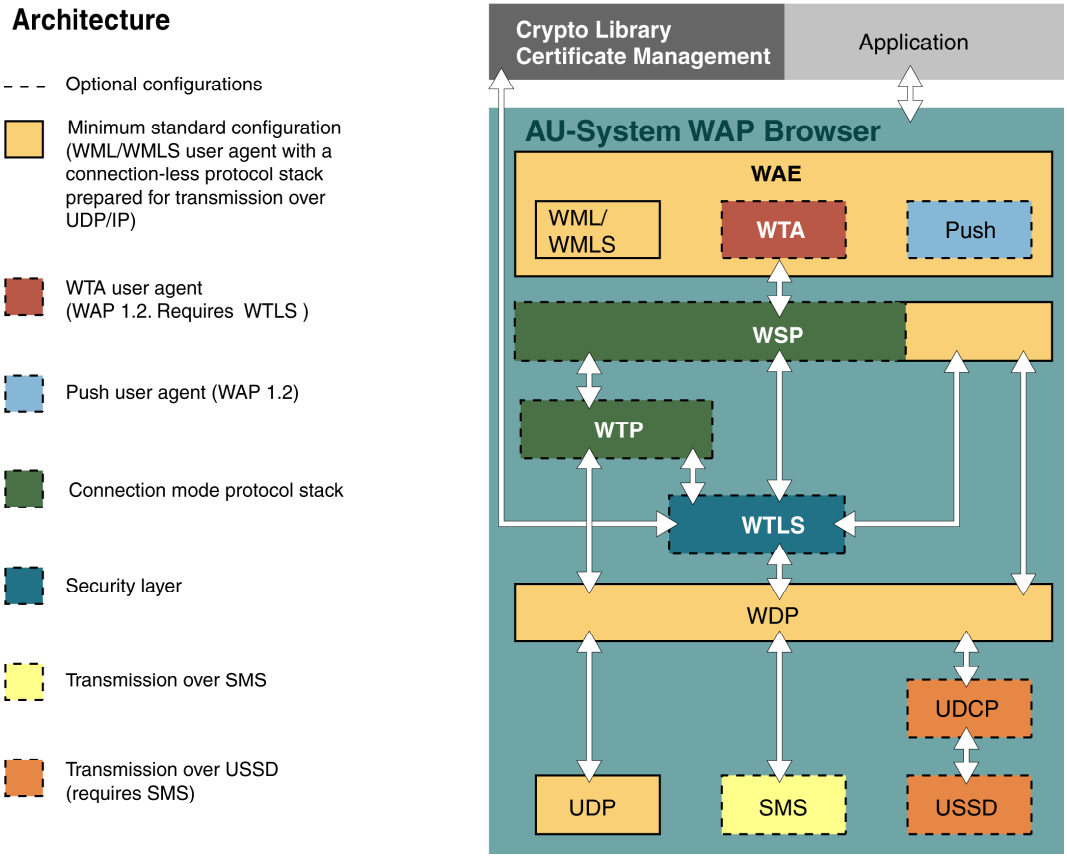
- Documentation – this manual.
- Adapter function interfaces – header files with definitions of all Adapter functions that must be implemented in the WAP application.
- Connector function interfaces – header files with definitions of all Connector functions that can be called by the WAP application.
- Source code – all sources needed to build the AUS WAP Browser.

2.2 Compatibility with WAP gateways

The AUS WAP Browser is full WAP/1.1 as well as WAP/1.2 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, which of the AUS WAP Browser uses. This section specifies the protocols the AUS WAP Browser implements. Implementation specific details are given for each protocol layer in separate sub-sections.



The picture above gives an overview of the architecture, which WAP components that can be included in a AUS WAP Browser configuration.

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, WTA and Push. It implements the cache model that HTTP/1.1 defines, and that WAP specialises.

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 tells that one of the character sets UCS16, UTF8 or ISO-8859-1 shall be used, 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 adds functionality for retransmission. It 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 much 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

All values are default values taken from the WTP specification [WAP-WTP] and can be reconfigured (see chapter “Tuning the AUS WAP Browser”).

2.3.4 WTLS

WTLS is a class 3 implementation. It makes use of an external library with cryptographic functionality, to which an adoption must be made. The cryptographic functionality may be implemented by software as well as with a WIM.

2.3.5 WDP

WDP is a transparent layer to UDP. It adds no functionality when UDP is used as bearer. The AUS WAP Browser supports also Phase 2 of the two GSM bearers

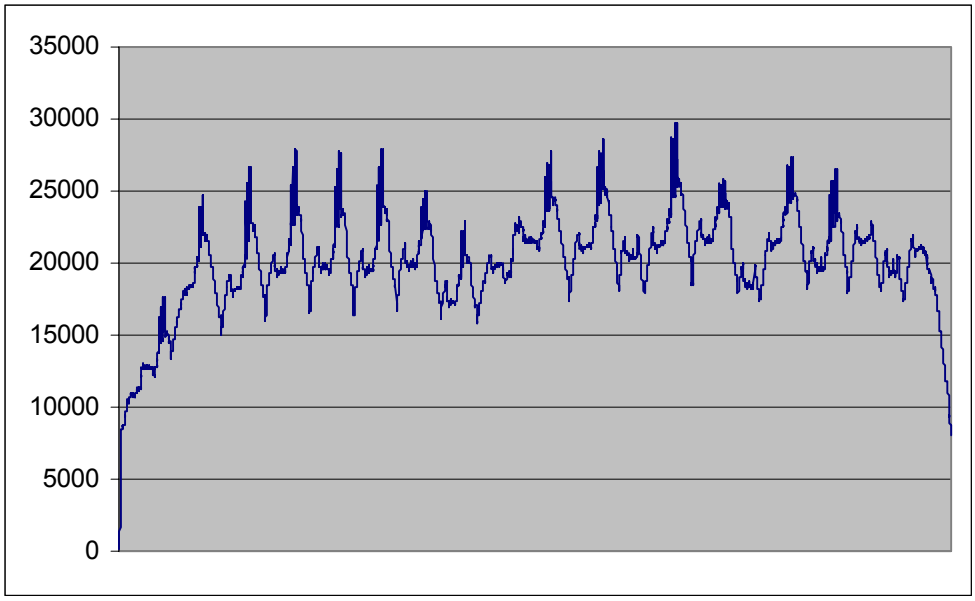
SMS and USSD (See UDCP below). For these two bearers is additional functionality for assembling and segmentation added. The algorithm used is specified by GSM [GSM0340]. The AUS WAP Browser has also the additional layer WCMP implemented. It serves the same purpose for the SMS and USSD bearers as ICMP does for UDP/IP.

2.3.6 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 RAM usage of the AUR WAP Browser. RAM has been measured by browsing a set of WML pages (http://wap.tv4.se), which has an avererage size of 1400 bytes when compiled. Only dynamic RAM is messured.



When the AUS WAP Browser has started, the memory usage is about 13 Kbytes. During browsing, each download makes the AUS WAP Browser to peak at about 30 Kbytes.

3 Host Device Requirements

3.1 Memory requirements

The object code (reffered to as CODE in the table below) of the AUS WAP Browser uses is static. It is therefore storable in ROM. The constant data of the AUS WAP Browser is also storable in ROM (reffered to as CONST in the table below). If the Host Device does not support ROM, the persistent memory is used instead.

The RAM is used for the WAP application heap. It is divided in statically allocated RAM (reffered to as DATA in the table below), dynamically allocated RAM (reffered to as HEAP in the table below) and the call execution stack (reffered to as FUNC in the table below). 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. The dynamically allocated RAM can be configured to be allocated with the ANSI-C library function malloc, or to be allocated from a static storage of the AUS WAP Browser if malloc does not exist.

The size is dependent on the configuration of the AUS WAP Browser. The configuration parts are the following:

- **Base** is the AUS WAP Browser with WML, WMLS and a connection-less protocol stack.
- **WTA** is the integration to the telephony functionality like call control and the phonebook. This configuration part requires that WTLS is included.
- **Push** is server initiated content uploading to the browser.
- **CO** is WTP and session management in WSP.
- **WTLS** is the sequrity layer, WTLS.
- **SMS** is the bearer SMS.
- **USSD** is the bearer USSD. This configuration part requires that SMS is included.

The sizes can only be considerad as estimates since no two compilers produces the same object code. These sizes have been retrieved by using an IAR compiler for an AVR 8-bit microcontroller. The sizes (in Kbytes) of the configuration parts are:

Configuration	CODE	CONST	HEAP	DATA	FUNC
Base	227	10	30	5	2
WTA	52	3	ε	0.5	ε
Push	38	3	ε	ε	ε
CO	57	1	ε	8	ε

WTLS	28	0.5	ε	1	ε
SMS	10	ε	ε	1	ε
USSD	13	0.5	ε	1	ε

E.g., the size of the configuration Base and CO is 296 Kbyte, takes 30 Kbytes of the HEAP, 13 Kbytes of DATA and 2 Kbyte for the call stack (FUNC). The CODE, CONST and DATA sizes are obtained by compiling the source code with an IAR compiler targeting an AVR CPU. The flag –z9 (minimise code size) has been used.

The persistent memory is optionally 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.

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 event 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 of the Host Device

Below is a listing of a minimum set of GUI features that are needed in order 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 keys for stop operation and for backward navigation.
- 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 (in wiptrgt.h). 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 implementation of the functions must conform to the standard IEEE754. They should return HUGE_VAL at overflow. HUGE_VAL is defined in math.h. Underflow, overflow and domain errors are detected by reading the errno variable (defined in errno.h). When overflow or any other floating point exception is detected in the hardware due to a call to any floating point operation from the AUS WAP Browser, the harware must not generate an exception, or any other kind of interupt.

Note that these functions, as they are defined by ANSI, use the type double. The AUS WAP Browser uses only FLOAT32 (see the Common API), which means that they only need to have support for the type float.

3.5.3 errno.h

To control the result (underflow, overflow and domain errors) from the math functions, the AUS WAP Browser uses the constants EDOM and ERANGE, defined in the ANSI-C 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* , const void*, 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 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 wiptrgt.h. The drawback is that the AUS WAP Browser allocates one Kbyte of RAM as a safety buffer.

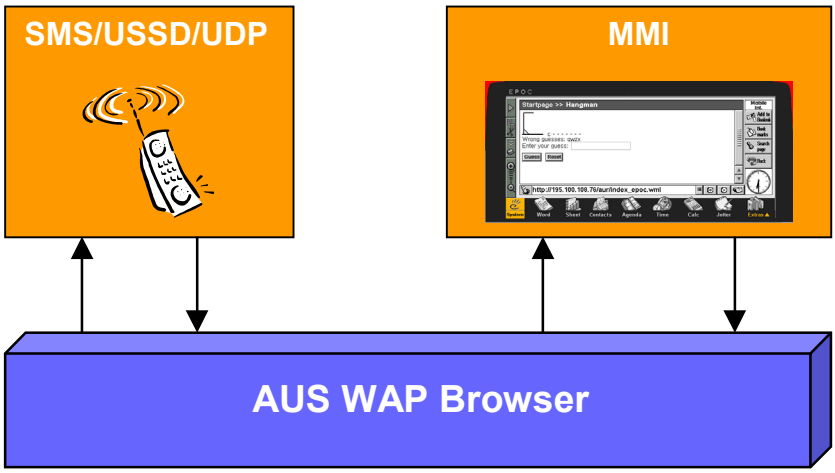
3.5.7 stdarg.h

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.

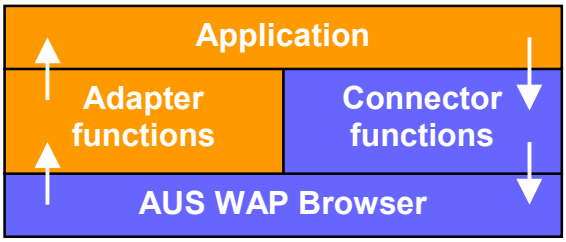
The CLNTa_log function calls are only included in the AUS WAP Browser if the compiler switch LOG_EXTERNAL is set in the makefiles that builds the AUS WAP Browser. The Adapter function should not be included in the WAP application code if the switch is not set. In this way, stdarg.h will only be required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

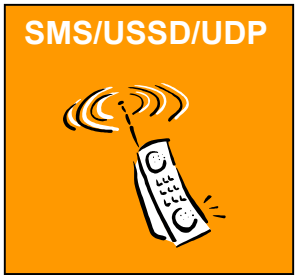


All communication from the WAP application to the AUS WAP Browser goes through Connector functions. On the reverse, all communication from the AUS WAP Browser to the WAP application passes through Adapter functions.

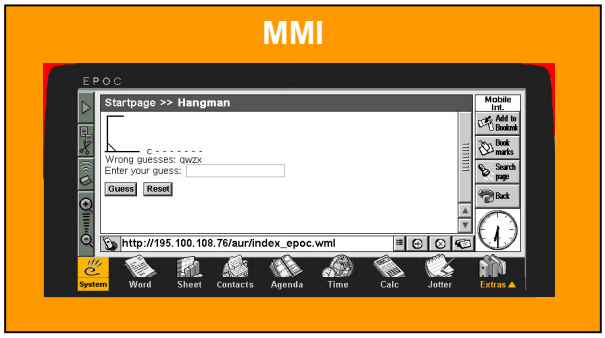
The colour scheme used in the illustrations goes through all illustrations in this manual. Orange means application implementation, i.e. device dependent implementation for a specific terminal. Blue means AUS WAP Browser implementation, i.e. implementation provided with the AUS WAP Browser product.

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.



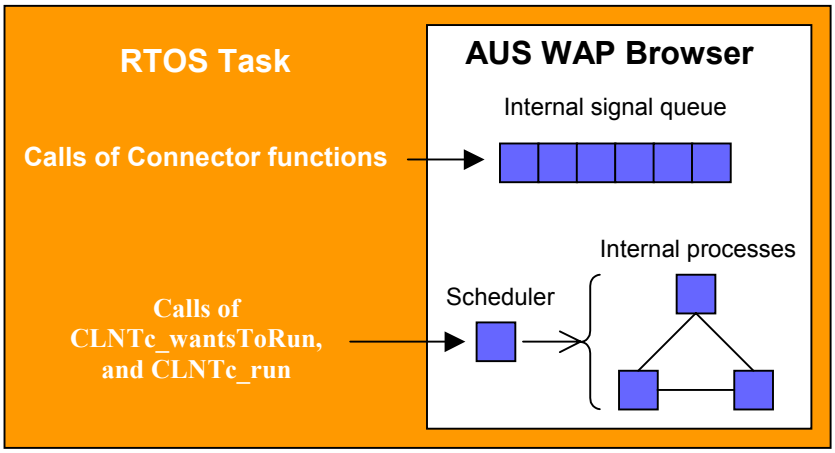
The WAP application implements the graphical user interface. Depending on the terminal, this application may appear in many shapes. The only requirement for it is that it should be able to display the WML graphical elements and take input from the user. Graphical elements are for instance text, images,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

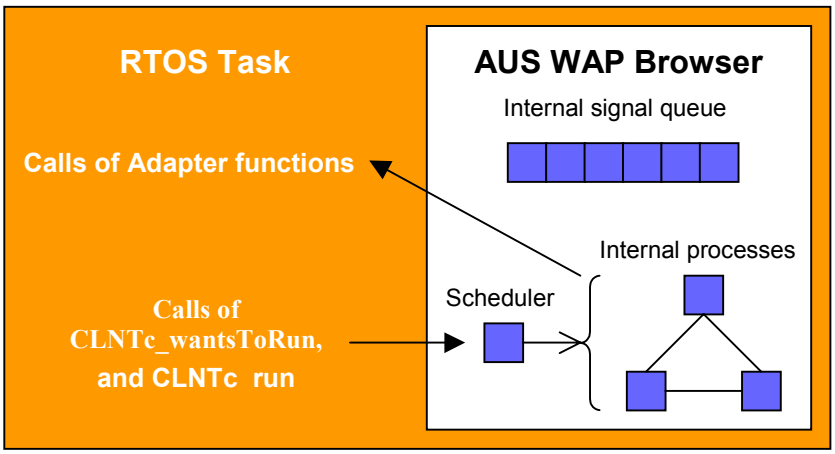


Two Connector functions, CLNTc_run and CLNTc_wantsToRun used in combination, provides the AUS WAP Browser with the CPU time necessary to process the internal signals in the internal signal queue. An internal scheduler keeps track of signals in the queue and executes internal processes that perform the 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.

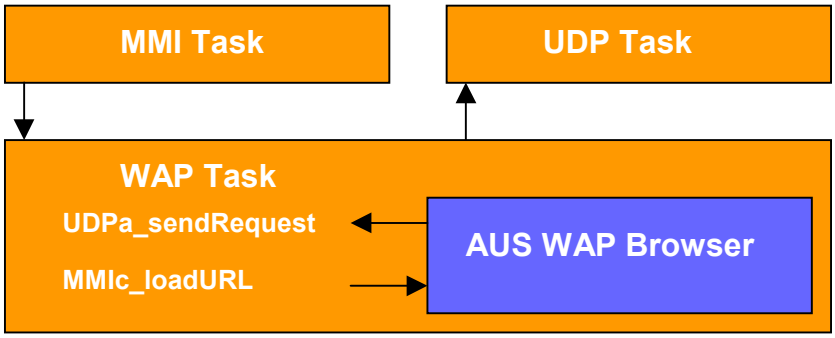


The AUS WAP Browser executes on control by a task in the WAP application (with CLNTc_run, as showed in the picture above). An Adapter function is only called when the function CLNTc_run is called. A few Adapter functions return data from the WAP application. The data retrieval time is assumed deterministic 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.

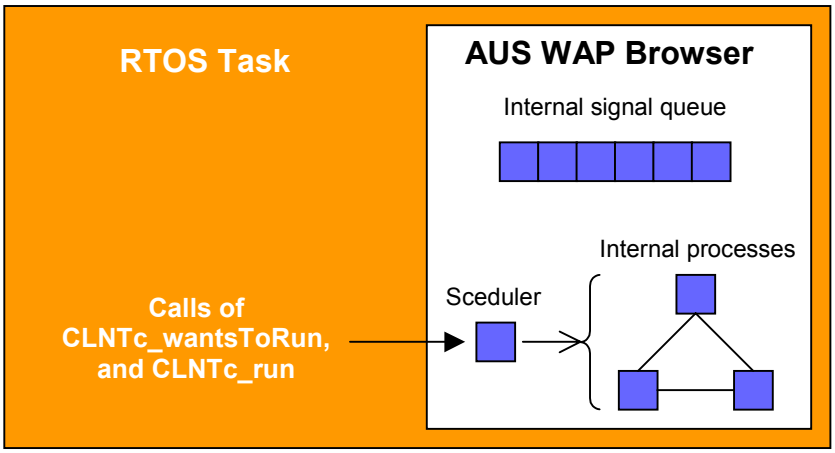
4.4 The AUS WAP Browser

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. 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.

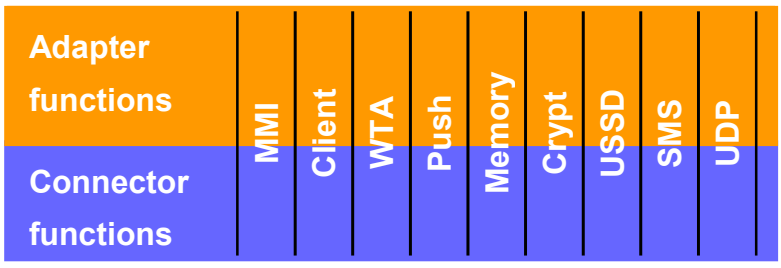
The system is signal driven. This means that if there are no signals in the signal queue, no process will be run by the scheduler. When there are signals, the scheduler processes them in FIFO order.

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 an 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 APIs is 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 to which it belongs. Either the prefix has 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

4.5.1 MMI API

For 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.

4.5.2 Client API

This API defines the general interface between the WAP application and the AUS WAP Browser component. 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://)

4.5.3 WTA API

This API hosts the telephony functionality. The AUS WAP Browser implements the public WTA functionality, as well as full WTA.

4.5.4 Push API

The Push API contains functionality, which enables the AUS WAP Browser to retrieve pushed content from a server.

4.5.5 Memory API

This API defines the interface to an optional cache memory. It can be persistent as well as volatile. It defines also the interface to the persistant memory used for the Push and WTA Services repository.

4.5.6 Crypt API

The Crypt API is the interface WTLS has in order to access crypto algorithms in the WAP application environment.

4.5.7 USSD API

This API defines the interface between the AUS WAP Browser and the USSD service in the Host Device environment.

4.5.8 SMS API

This API defines the interface between the AUS WAP Browser and the SMS service in the Host Device environment.

4.5.9 UDP API

This API defines the interface between the AUS WAP Browser and the UDP service in the Host Device environment.

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 an acceptance test program, which verifies that the AUS WAP Browser has been properly ported. It does a basic test of the AUS WAP Browser. The program has most Adapter function implemented to call 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. The developers of the WAP application can make use of these function stubs 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.

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 select in the menu 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 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, sizeOfCache);
```

sizeOfCache is assigned the amount of available cache memory in bytes.

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

5.1.3 Initialising the user agent

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

```
MMIc_startUserAgent(1, WML_USER_AGENT);
```

When the WAP application has opened a user agent, an id that identifies it is given to the AUS WAP Browser. This object id will then be referred in all calls

between the WAP application and the AUS WAP Browser. The constant WML_USER_AGENT 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 AUS WAP Browser

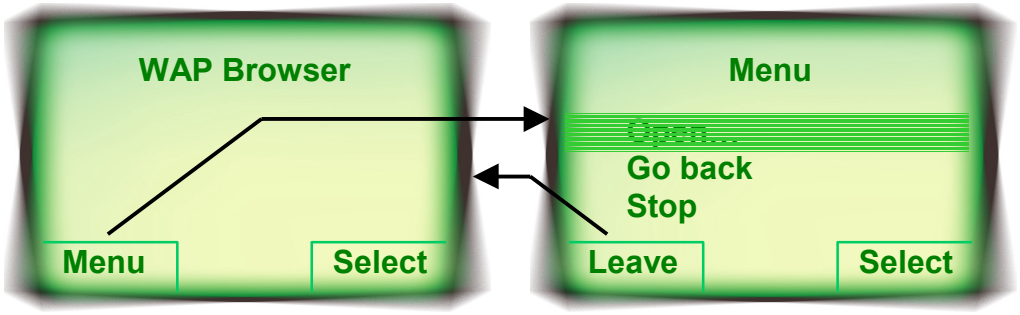
As a last step, in the initialising process, the WAP application provides the AUS WAP Browser with some essential parameters. The AUS WAP Browser needs general information about how to handle the downloaded content, how to handle images, etc. All general values have default values. If any value is not appropriate, it can be set as in the following example:

```
CLNTc_setIntConfig (ALL_USER_AGENT, configHISTORY_SIZE,
          20);
```

More specific parmeters, e.g. the gateway addresses, are set per user agent (a WML user agent, for instance). The following examples show parts of a configuration sequence:

```
CLNTc_setDCHIntConfig (1, 1, configACCESS_TYPE,
          BEARER_GSM_CSD);
CLNTc_setDCHStrConfig (1, 1,
          configUDP_IP_GW,"\xFA\xB0\x40\x20", 4);
CLNTc_setDCHIntConfig (1, 1, configSTACKMODE, CO_WTLS);
CLNTc_setDCHIntConfig (1, 2, configACCESS _TYPE,
          BEARER_BT);
CLNTc_setDCHStrConfig (1, 2,
          configUDP_IP_GW,"\x00\x00\x00\x00", 4);
CLNTc_setDCHIntConfig (1, 3, configACCESS_TYPE,
          BEARER_BT);
CLNTc_setDCHStrConfig (1, 3,
          configUDP_IP_GW,"\xE4\xA0\x44\x20", 4);
CLNTc_setDCHStrConfig (1, 2, configADD_HOST,"myCar.local",
          11);
CLNTc_setDCHStrConfig (1, 3,
          configADD_HOST,"myStereo.homeNet", 16);
CLNTc_setDCHStrConfig (1, 3,
          configADD_HOST,"myTV.homeNet", 12);
CLNTc_setDCHIntConfig (2, 1, configACCESS_TYPE,
          BEARER_CSD_SMS);
CLNTc_setDCHStrConfig (2, 1,
          configSMS_GW,"\xE4\xA0\x44\x20", 4);
CLNTc_setDCHStrConfig (2, 1,
          configSMS_C,"\x00\x00\x00\x00", 4);
CLNTc_setDCHIntConfig (2, 1, configONLINE, TRUE);
```

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.

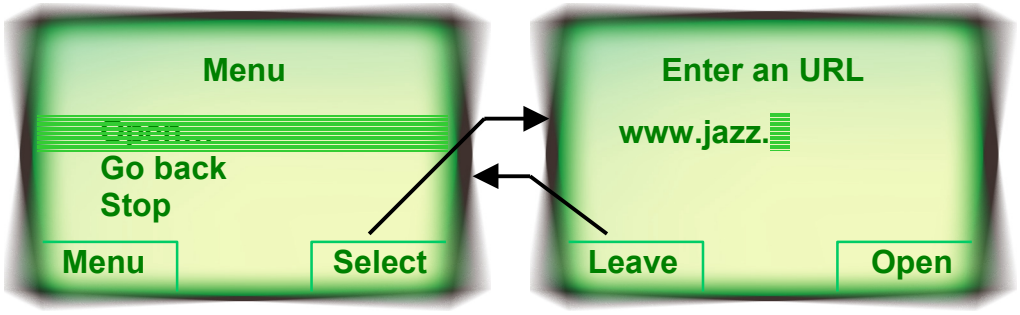
At this step, implementation is required of the following adapter function:

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_reload

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

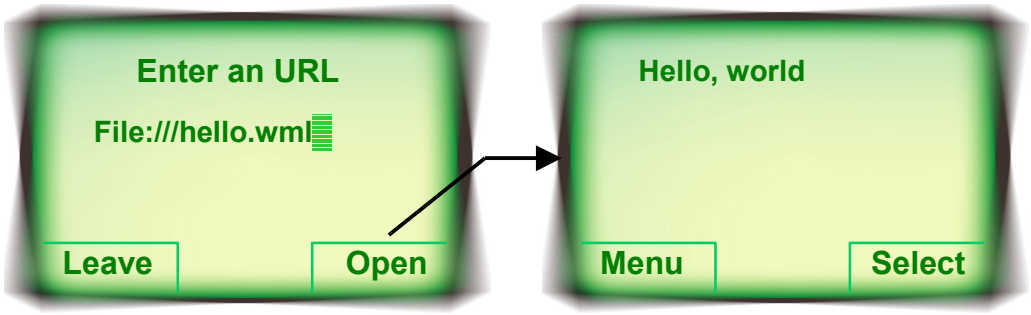
when the AUS WAP Browser currently gives draw instructions (like MMIa_newText, which is discussed later). This means that the Menu key should be accessible, always.

5.3 Hello world

After the initialisation phase has been implemented and run, and after the necessary user-interface 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>
```

The compiled byte code 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, FALSE);
MMIa_newText (1, 0, Unicode("Hello, world!"), FALSE, 0,
          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 as follows:

```
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, FALSE);
MMIa_newText (1, 0, Unicode("Left aligned text"), FALSE,
           0, TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_newParagraph (1, ALIGN_CENTER, FALSE);
MMIa_newText (1, 0, Unicode("Centred text"), FALSE, 0,
           TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_newParagraph (1, ALIGN_RIGHT, FALSE);
MMIa_newText (1, 0, Unicode("Right aligned text"), FALSE,
           0, TXT_NORMAL);
MMIa_closeParagraph (1);
```

```
MMIa_newParagraph (1, ALIGN_LEFT, FALSE);
MMIa_newText (1, 0, Unicode("Left aligned text"), FALSE,
          0, TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_newParagraph (1, ALIGN_LEFT, FALSE);
MMIa_newText (1, 0, Unicode("Left aligned text, no
          wrapping is to be performed on this very long
          line."), FALSE, 0, TXT_NORMAL);
MMIa_newBreak (1);
MMIa_newText (1, 0, Unicode("This is a new line."), FALSE,
          0, TXT_NORMAL);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

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

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 as follows:

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

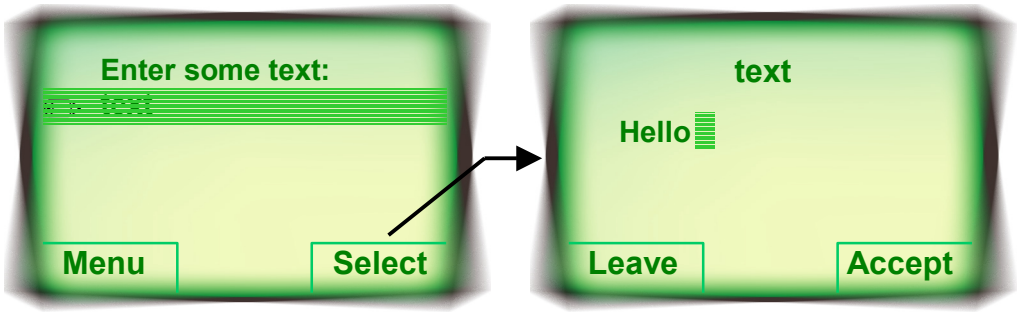
This results in the following calls:

```
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, FALSE);
MMIa_newText (1, 0, Unicode("Enter some text:"), FALSE, 0,
           TXT_NORMAL);
MMIa_newInput (1, 1, Unicode("text"), NULL, NULL, FALSE,
           TRUE, NULL, 0, 0, 0, 0);
MMIa_closeParagraph (1);
MMIa_showCard (1);
```

After 60 seconds is card number two loaded. A string should be entered, during card one is displayed, 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 as follows:

```
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, FALSE);
MMIa_newText (1, 0, Unicode("The text entered: string"),
           FALSE, 0, 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 as follows:

```
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:

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, FALSE);
MMIa_newText (1, 0, Unicode("Select an option:"), FALSE,
           0, 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 be made, during card one is displayed, 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, FALSE);
MMIa_newText (1, 0, Unicode("Menu choice: 2"), FALSE, 0,
           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);
```

This 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, 0,
              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 as follows:

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

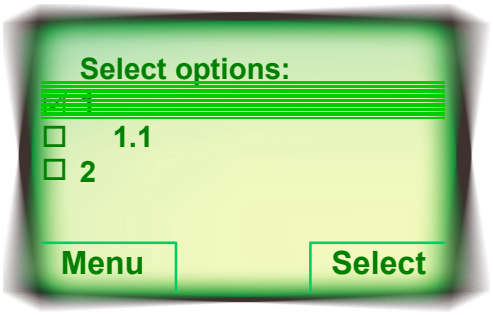
This results in the following calls:

```
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 be displayed, in accordance with the examples above, 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, FALSE);
MMIa_newText (1, 0, Unicode("Select options:"), FALSE, 0,
          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 keys. A sample WML application, which displays a single key, is opened as follows:

```
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, FALSE);
MMIa_newText (1, 0, Unicode("Card 1"), FALSE, 0,
          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, FALSE);
MMIa_newText (1, 0, Unicode("Card 2"), FALSE, 0,
          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 as follows:

```
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 as follows:

```
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:

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. A bearer might be implemented in a task, other than the WAP application task. To keep in mind then is that 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 is all functionality on place in order to be WAP conformant. However, several optional AUS WAP Browser functions have not been integrated in this build example of a WAP application. This section states the functions in matter.

Optional MMI API functions that have not been described in this build example:

MMIc_reload
MMIa_status
MMIa_unknownContent
MMIa_newImage
MMIa_completeImage
MMIc_imageSelected
MMIa_newTable
MMIa_newTableData
MMIa_newFieldSet
MMIa_closeFieldSet
MMIa_linkInfo

Optional Client API functions that have not been described in this build example:

CLNTa_nonSupportedScheme
CLNTa_content

Optional WTA API function that has not been described in this build example:

WTAIa_publicMakeCall
WTAIa_publicSendDTMF
WTAIa_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

In the AUS WAP Browser, variables 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 memory management function (malloc) 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	To be defined if the internal AUS WAP Browser memory management implementation is to be used (instead 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 when a certain level of memory usage has been reached, 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	For each image that is opened, the AUS WAP Browser must store the URL. Since URLs can be very large, this list of URLs can consume a lot of RAM. This constant sets the maximum number of simultaneously requests for images that an user agent will queue for a WML card that currently is opened. All requests that are issued after this limit are ignored.
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 an user agent (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 user agent) 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 user agent. This option is only present for future enhancements of the WAP standard. 1: Allow only execution of one script per user agent.
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.

When the AUS WAP Browser is configured to support WTA, the following variables may be changed:

Variable name	Default	Description
cfg_wae_wta_Rep_ maxcompact	3	When the WTA repository grows, compactisations may be neccesary to perform. The resources in the repository are moved, one resource at time, in order to maximize the size of the remaining free areas. Depending on the repository size and on the capacity of the host device, the number of resources to handle in each call to CLNTc_run, can be configured with this variable.

When the AUS WAP Browser is configured to support Push, the following variables may be changed:

Variable name	Default	Description
cfg_wae_push_compare_ authority	1	For incoming push indication: 0: do not perform authority comparison between push content and X-Wap-header. 1: perform authority comparison between push content and X-Wap-header.
cfg_wae_push_notify_sl	1	For incoming Service Loadings: 0: Handle them entirely within the AUS WAP Browser. No notifications are made to the WAP application. 1: Let the WAP application handle them. Notify the WAP application using the function PUSHa_newSLreceived.
cfg_wae_push_in_buffer _size	5	The number of incoming push-signals that can be received by the push handler when it is busy executing other tasks, i.e, the end-user chooses to interact on them. A medium-sized push-signal consumes RAM about 250 bytes. The recommendation to to the push content providers is to not have them exceeding 500 bytes.

cfg_wae_push_notify_change	1	For stored push messages, the AUS WAP Browser will act as follows on the two different settings of this variable: 0: Do not notify the WAP application when a message is replaced or deleted. 1: Let the WAP application know when a push message is replaced or deleted, by calling the function PUSHa_messageChange.

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.
CAN_SIGN_TEXT	Not defined	When this variable is defined, the functionality for computing digital signatures from WTLS is enabled (i.e. the functions MMIa_signText and MMIc_textSigned are added)
REP_STORAGESIZE	8000	The size of the WTA repository, when the AUS WAP Browser supports WTA
PUSH_STORAGESIZE	4500	The size of the Push repository, when the AUS WAP Browser supports Push
DATABASE_ STORAGESIZE	4000	The size of the database repository, where configuration variables are stored.
WMLS_CORRECT_ FLOAT2STRING	Not defined	In WMLS, conversion from float to string is performed with the help of sprintf, using the

		conversion specifier "g". The number of significant digits is by default 6, although trailing zeros are removed from the fractional part of the result. With this constant defined, the number of significant digits is instead 9. In the worst case, as many as 9 significant digits may be required to guarantee that a conversion from float to string and back again will yield the original float value. However, having it not defined results in more wellformed presentation.
USE_PROPRIETARY_ WMLS_LIBS	Not defined	When this constant is defined, functionality for accessing propriatary WML Script functions implemented outside the AUS WAP Browser (see CLNTa_hasWMLSLibFunc) is enabled.
LARGE_DATA_ TRANSFER_ENABLED	Defined	If defined, then the mechanisms for large data transfer are enabled.
CONTENT_UA_MAX_ MESSAGE_SIZE	261120	The maximum size of data that can be sent/retrieved using the content handler. Note that this value is negotiated with the WAP gateway. It may result in a lower value.

6.3 WAE – 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_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.
cfg_wae_wspif_ AuthenticationItems	10	Maximum number of items the AUS WAP Browser stores in the authentication list, i.e. the list with login data for the sites where the user logged in since the AUS WAP Browser started.
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 image capabilities of the WAP application 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 application capabilities.

Variable name	**Template**	**Description**
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.
WTP_SAR_GROUP_SIZE	5120	The maximum size of a group, as

		used by WTP in SAR.
WTP_SAR_SEGMENT_ SIZE	1024	The maximum size of a segment, as used by WTP in SAR.

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.

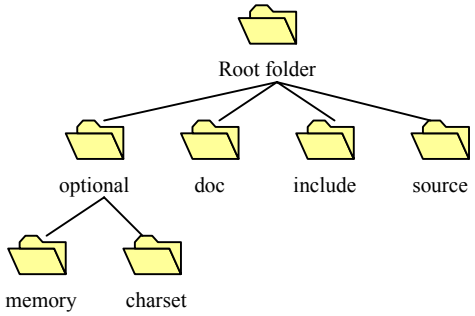
Variable name	Default	Description
MaxReassTime	3000	1/10 of seconds a SMS or USSD message segment is waited for, before it is considered lost.
LowestMaxUssdLength	70	The max length of an USSD text message may vary from country to country. It is not standardised. In order to determine when segmentation 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. 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.
aapiwd.h	Header file for Adapter functions in the WAP Device API.
aapiwta.h	Header file for Adapter functions in the WTA API.
aapipush.h	Header file for Adapter functions in the Push API.
aapicrpt.h	Header file for Adapter functions in the Crypto API.
capiclnt.h	Header file for Connector functions in the Client API.
capimem.h	Header file for Connector functions in the Memory API.
capimmi.h	Header file for Connector functions in the MMI API.
capisms.h	Header file for Connector functions in the SMS API.

capiudp.h	Header file for Connector functions in the UDP API.
capiussd.h	Header file for Connector functions in the USSD API.
capiwd.h	Header file for Connector functions in the WAP Device API.
capiwta.h	Header file for Connector functions in the WTA API.
capipush.h	Header file for Connector functions in the Push API
tapicmmn.h	Header file for the Common API.
tapiclnt.h	Header file for types used in the Client 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.
wiptrgt.h	Header files for macros to help the AUS WAP Browser distinguish between different target environments like EPOC, OSE and REX.
ansilibs.h	A header file, which includes all nessecary ANSI C library files.

\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.

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_CHARSET 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 is integrated with a WAP application, and 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 with corresponding floating point operations that conform to IEEE754. When overflow or any other floating point exception is detected in the hardware due to a call to any floating point operation from the AUS WAP Browser, the harware must not generate an exception, or any other kind of interupt. The AUS WAP Browser handles the exceptions according to ANSI C (see also the technichal specification). If no floating point type exist for the host device, map to INT32. See also the static configuration variable HAS_FLOAT, which is defined in wiptrgt.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:

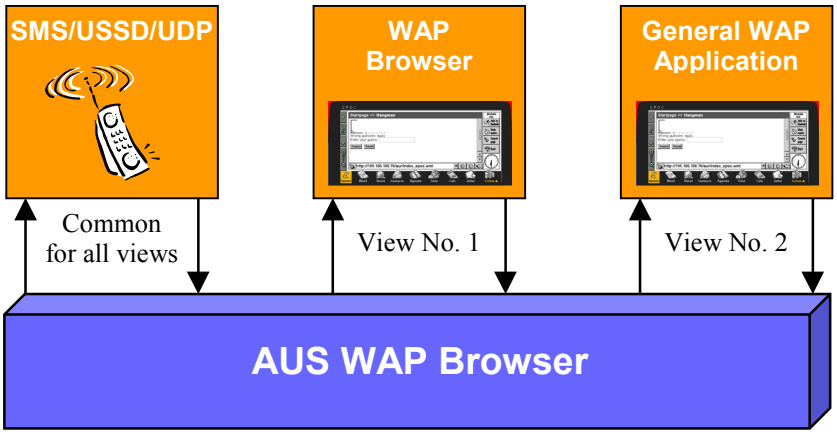
NULL	Integer value 0
TRUE	Integer value 1
FALSE	Integer value 0

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 User agents



Before a WAP application can start using the AUS WAP Browser, it must start a user agent in 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 user agents are used by the AUS WAP Browser to distinguish one WAP application from another. They will have there own set of configuration variables.

startUserAgent

This function tells the AUS WAP Browser that a new user agent is to be opened. A WAP application that only uses content retrieval (with CLNTc_getContent) can ignore this function. There is a predefined objectId constant (see table below) to be used in this case (e.g. to be used when a configuration variables shall be set).

```
VOID MMIc_startUserAgent (UINT8 objectId, INT8
uaMode)
```

`objectId`	An object id, in the range from 1 to 127, of the user agent is passed in the argument. Other values for object 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 objectId in the function CLNTa_error. The objectId argument may be used in subsequent calls to Connector functions as MMIc_loadURL. The objectId argument will then be used by the AUS WAP Browser in calls

	of Adapter functions for displaying WML cards.
`uaMode`	The user agent 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_USER_AGENT	The constant WML_USER_AGENT should be used when the application is a normal WML browser.
2	WTA_USER_AGENT	The constant WTA_ USER_AGENT should be used when the application is a WTA browser.

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

Value	Constant	Description
0	ALL_ USER_AGENT	AUS WAP Browser regards the objectId value 0 as general. (0 is used as a objectId when error and log messages do not origin from a specific user agent).
128	CONTENT_USER_ AGENT	Reserved object id for WML browsing for content retrieval (with the functions CLNTc_getContent and CLNTa_content).
129	REPOSITORY_ USER_AGENT	Reserved object id for downloading activities to the WTAI repository.
130	PUSH_USER_AGENT	Reserved object id for PUSH activities.

terminateUserAgent

This function tells the AUS WAP Browser that the object identified by objectId 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_terminateUserAgent (UINT8 objectId)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.

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 objectId, const CHAR* url,
BOOL reload)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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. A call to MMIc_loadURL resets the context of the user agent; 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 MMIc_reload does not reset the context of the user agent; i.e., all WML variables and the internal history are preserved.

```
VOID MMIc_reload (UINT8 objectId)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.

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 user agent. 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 objectId)
```

objectId	The object id received from a call to MMIc_startUserAgent.

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 before the current card in the history list.

```
VOID MMIc_goBack (UINT8 objectId)
```

objectId	The object id received from a call to MMIc_startUserAgent.

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 objectId, BOOL isWait)
```

objectId	The object id received from a call to MMIc_startUserAgent.
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 objectId, UINT8 status,
const CHAR *URL)
```

objectId	The object id received from a call to MMIc_startUserAgent.
status	Status code. All codes are described below in the section Constants.
URL	The URL for which the status is associated with. 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
ReadFromNetworkDone	11	Used at the following situations: 1. Content is recieved from a server 2. Timeout of the current read operation 3. The current read operation is aborted
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.
WTAServiceUnloading Initiated	104	Removal of an installed WTA service.
PushStarted	200	Indicates that the Push handler within the AUS WAP Browser has started. It is now ready to receive Push indications. If UDP is the bearer, the port 2948 (non-secure push) shall be opened and listened to, when this status indication is received. If WTLS is supported, 2949 must be opened and listened to, as well.
WTLSConnection Established	300	Indicates that a secure session has been set-up. This status indication is given with the general objectId (ALL_ USER_AGENT) and without a URL (set to NULL).
WTLSConnection Terminated	301	Indicates that a secure session has been terminated. This status indication is given with the general objectId (ALL_USER_AGENT) and without a URL (set to NULL).

The status codes above, given when data is downloaded and opened, 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 can 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, etc, 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 objectId, const
CHAR *data, UINT16 length, const CHAR *contentType,
const CHAR *URL)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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]. It 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. 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. A database of authentications is accessed when the AUS WAP Browser is running. The database is searched when a server requires authentication. If no item is found this dialog is used instead. The number of times this dialog is invoked is therefore kept to a minimum. If the authentications database is full, the oldest authentification items are deleted. To clear the database from the WAP Application, the function MMIc_clearAuthenticationDatabase can be called. The database is stored persistantly when the AUS WAP Browser is terminated. Read more about the database in the Memory API.

```
VOID MMIa_passwordDialog (UINT8 objectId, UINT16
dialogId, const CHAR *realm, INT8 type)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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, when the 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, this function should be called with the name and password arguments set to NULL. If the user not enters any text and accepts the dialog, this function should be called with the name and password arguments set to the empty strings.

```
VOID MMIc_passwordDialogResponse (UINT8 objectId,
UINT16 dialogId, const CHAR *name, const CHAR
*password)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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.

clearAuthenticationDatabase

This function is called by the WAP application when the database with autentification posts shall be cleared. See the function MMIa_passwordDialog.

```
VOID MMIc_clearAuthenticationDatabase (VOID)
```

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 be cancelled by calling CLNTc_stop.

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

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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 objectId,
UINT8 dialogId, const WCHAR *answer)
```

objectId	The object id received from a call to MMIc_startUserAgent.
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 the cancel operation 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 objectId, UINT8
dialogId, const WCHAR *message, const WCHAR *ok,
const WCHAR *cancel)
```

objectId	The object id received from a call to MMIc_startUserAgent.
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 object identified by objectId, (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 objectId,
UINT8 dialogId, BOOL answer)
```

objectId	The object id received from a call to MMIc_startUserAgent.
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

The function should cause the given message to be displayed 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 objectId, UINT8
dialogId, const WCHAR *message)
```

objectId	The object id received from a call to MMIc_startUserAgent.
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

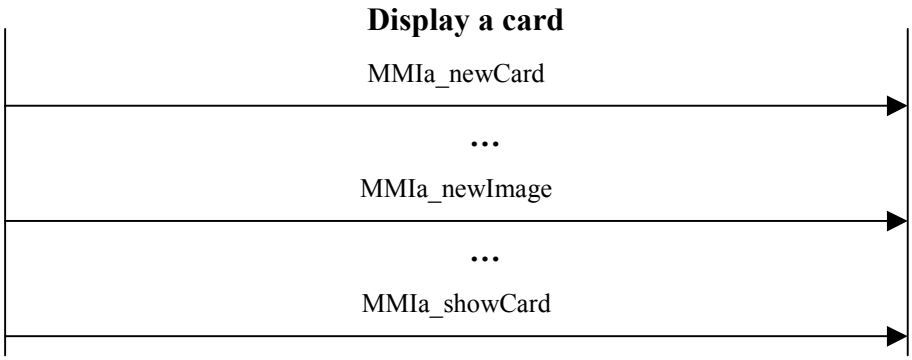
If an alert dialog has been opened (MMIa_alertDialog) a call to this function must be made when the user closes the dialog. If CLNTc_stop is called and the WML script is cancelled, this function has not to be called.

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

objectId	The object id received from a call to MMIc_startUserAgent.
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 needs not to be removed until the corresponding MMIa_showCard function is called.

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

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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 reloaded (the function MMIc_reload has been used). The state of the card currently viewed can be stored in order to being able to open the new card in the same position and with the same object in focus.
`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

The function should cause a created card to be displayed. The function is called when no more card content is to be added. Note that the card may be empty, and that the card in these cases shall be displayed as an empty card.

```
VOID MMIa_showCard (UINT8 objectId)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.

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. Note that MMIa_wait (FALSE) not will not be called when this function is called. Interactive elements of the card, which have been displayed up until this function was called, are not possible to use.

```
VOID MMIa_cancelCard (UINT8 objectId)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.

9.6 WML Keys

newKey

After the WAP application has received a MMIa_newCard call, calls to this function will be done for every do element defined in the WML card. The calls 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 objectId, UINT8 keyId,
const WCHAR *eventType, const WCHAR *label, BOOL
isOptional)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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 ‘*’ character is exchanged with the actual event name.
vnd.*	Vendor specific event. 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 objectId, UINT8 keyId)
```

objectId	The object id received from a call to MMIc_startUserAgent.
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 objectId, UINT8 textId,
const WCHAR *text, BOOL isLink, const WCHAR
*linkTitle, WCHAR accessKey, INT8 format)
```

objectId	The object id received from a call to MMIc_startUserAgent.
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. The id of this text is to be used as an argument in that call.
linkTitle	A link may be associated with a title that may be displayed in various ways by the WAP application.
accessKey	A link may be associated with a key on the keyboard on the terminal that hosts the WAP application. When the key is selected, a call to MMIc_textSelected is to be performed. This argument holds a character that identifies the key. A value of zero means that no key has been identified for this link. This argument is optional to support.

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 bitwise OR operator "|". Example of how the format argument is used to determine if TXT_SMALL is set:
	`if (format & TXT_SMALL)`
	`  setSmallFont();`
	Note: the WAP application must distinguish between emphasised text and non-emphasised text. Emphasised text

should be distinguished from strong emphasised text. Strong text, big text and bold text can be displayed in the same way. Emphasised text, italic text, underlined text and small text can be displayed in the same way.

textSelected

Shall be called when a text link has been selected.

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

objectId	The object id received from a call to MMIc_startUserAgent.
textId	An id retrieved from the function MMIa_newText.

9.7.2 Images

newImage

This function adds an image to a view.

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

objectId	The object id received from a call to MMIc_startUserAgent.
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 now.
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. The id of this image is to be passed as an argument to that function.
`linkTitle`	The link may contain a title. The title may be displayed in various ways by the WAP application.
`accessKey`	A link may be associated with a key on the keyboard on the terminal that hosts the WAP application. When the key is selected, a call to MMIc_ imageSelected is to be performed. This argument holds a character that identifies the key. A value of zero means that no key has been identified for this link. This argument is optional to support.
`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 the bitwise AND operator "&":
	`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 objectId, UINT8
imageId, const CHAR *imageData, UINT16 imageSize,
const CHAR *imageType)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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 objectId, UINT8
imageId)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`imageId`	An id retrieved from the function MMIa_newImage.

9.7.3 Languages

The entire WML deck or/and parts of it can specified to be in a certain language. The WML deck can be told to be in English. A piece of text inside that WML

deck can be told to be in a language that is not written from left to right, Chinese for instance. The WAP application may justify the text content according to the specified language. The direction of the words and the sentences can be adjusted, as well.

setLanguage

This function indicates what language the following text content is written in. The function may be called whenever a new language is specified. This may occur on a global level, to set the language for all forthcoming cards. It can also occur inside cards. The default language of the WAP application should be used if this function not has been called.

```
VOID MMIa_setLanguage (UINT8 objectId, const CHAR
*language)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`language`	The language in form of a text string, e.g. “english”. The complete list of all possible languages is found in [WAP-WSP]. If NULL is given, the default language for the WAP application is to be used.

9.7.4 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 objectId, INT8 align,
BOOL wrap, BOOL preformatted)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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.
`preformatted`	When this argument is set to TRUE, it indicates that all text content within this paragraph is preformatted. No white space and line breaks will be removed from such texts. This means also that the text may be displayed using a fixed pitch font and that the text not needs to be word wrapped. This argument is optional to handle.

closeParagraph

This function closes the current paragraph.

```
VOID MMIa_closeParagraph (UINT8 objectId)
```

objectId	The object id received from a call to MMIc_startUserAgent.

newBreak

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

```
VOID MMIa_newBreak (UINT8 objectId)
```

objectId	The object id received from a call to MMIc_startUserAgent.

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. Even if the field set is choosen to be ignored, the contained elements must be displayed.

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

objectId	The object id received from a call to MMIc_startUserAgent.
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 objectId)
```

objectId	The object id received from a call to MMIc_startUserAgent.

9.7.5 Constants

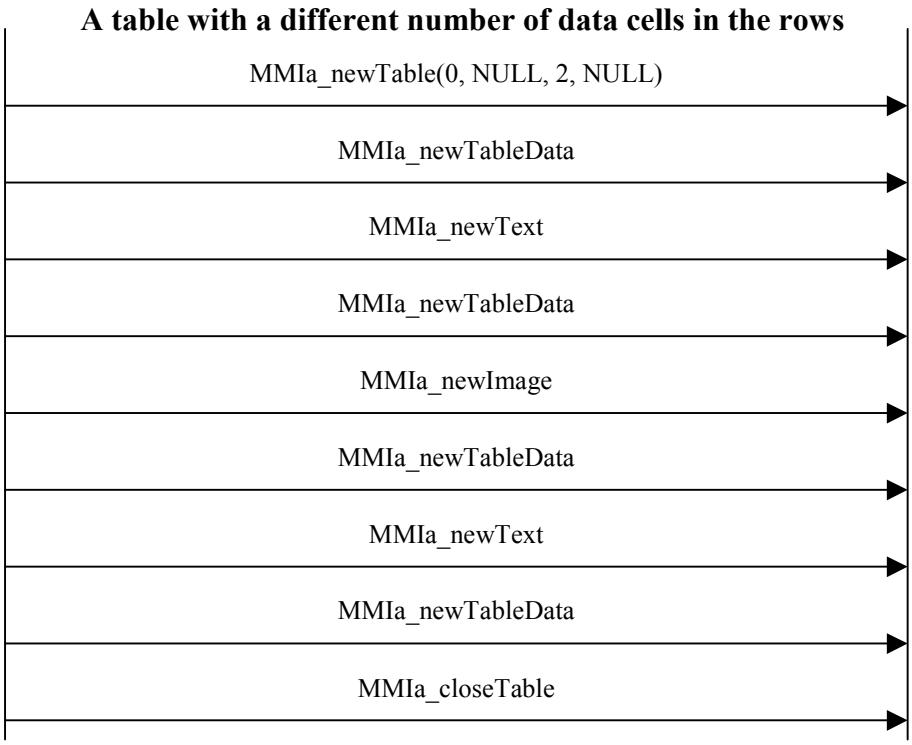
Constants that the text, image and layout functions use are (defined in tapimmi.h):

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

This function adds a new table to a view.

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

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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 objectId)
```

objectId	The object id received from a call to MMIc_startUserAgent.

closeTable

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

```
VOID MMIa_closeTable (UINT8 objectId)
```

objectId	The object id received from a call to MMIc_startUserAgent.

9.9 WML Menus

newSelect

This function 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 objectId, const WCHAR
*title, BOOL multiSelect, INT8 tabIndex)
```

objectId	The object id received from a call to MMIc_startUserAgent.
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

This function closes the selection menu.

```
VOID MMIa_closeSelect (UINT8 objectId)
```

objectId	The object id received from a call to MMIc_startUserAgent.

newOption

This function 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 objectId, UINT8
optionId, const WCHAR *label, const WCHAR *title,
BOOL isSelected)
```

objectId	The object id received from a call to MMIc_startUserAgent.
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 objectId, const
WCHAR *label)
```

objectId	The object id received from a call to MMIc_startUserAgent.
label	The option group may have a label. The label is deleted when the function returns.

closeOptionGroup

This function closes the option group initiated by the former call to MMIa_newOptionGroup.

```
VOID MMIa_closeOptionGroup (UINT8 objectId)
```

objectId	The object id received from a call to MMIc_startUserAgent.

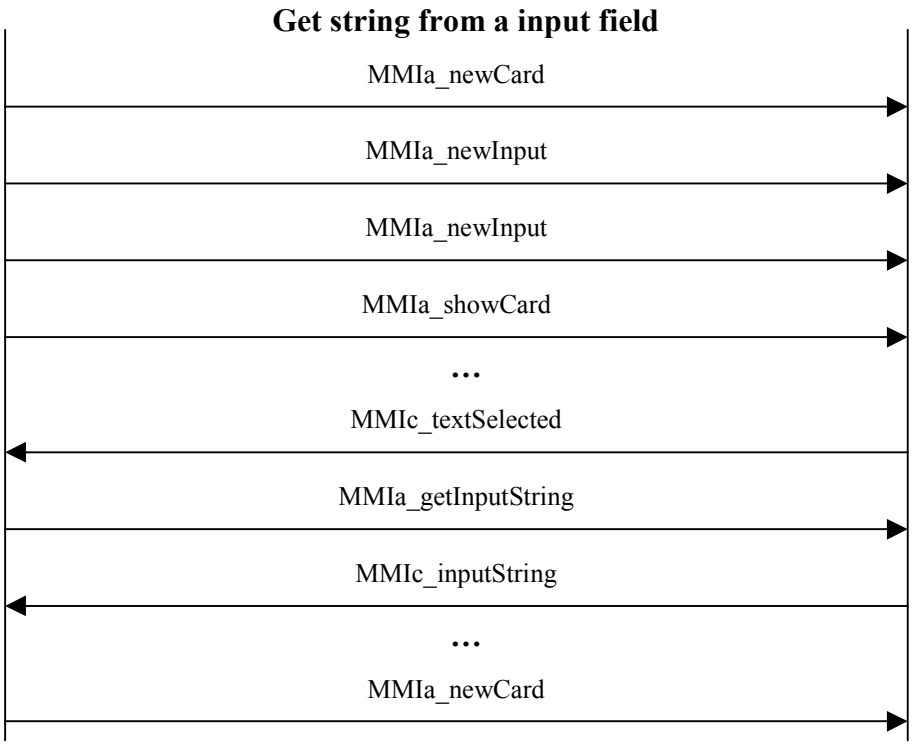
optionSelected

This function changes the state of an option in a selection menu. The function is used when options are selected or deselected. In multiple-choice, it is used in both cases. In single-choice menus, it is only used when an option is selected. It shall be called also when the menu is single-choice, and the option is already on.

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

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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 objectId, UINT8 inputId,
const WCHAR *title, const WCHAR *text, BOOL
isPassword, BOOL emptyOk, const WCHAR *format, INT8
size, INT8 nChars, INT8 tabIndex, WCHAR accessKey)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.

`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.
`accessKey`	An input field may be associated with a key on the keyboard on the terminal that hosts the WAP application. When the key is selected, the input field should be activated or put into fokus of the user. This argument holds a character that identifies the key. A value of zero means that no key has been identified for this input field. This argument is optional to support.

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 objectId, UINT8
inputId)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`inputId`	The id of the input field of which the string is requested.

inputString

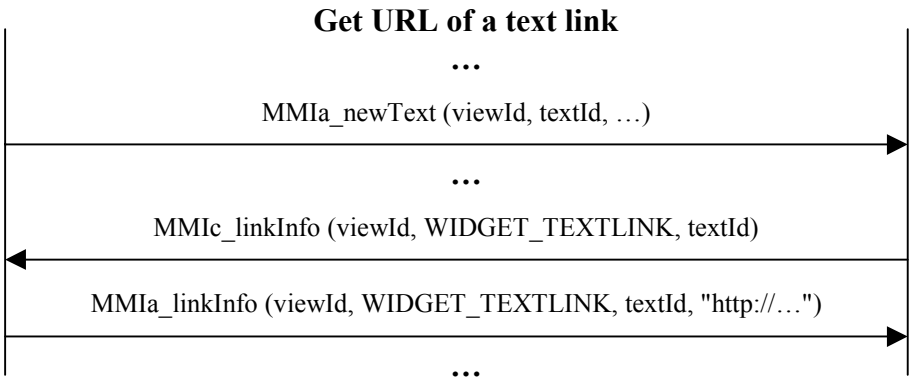
This function is called in order to return the string requested in a MMIa_getInputString call. It is an error to not respond on this function. This will block the downloading activity, when it is dependent upon the input string. However, if the answer of any reason not has been received by the AUS WAP 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 objectId, UINT8
inputId, const WCHAR *text)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`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.



linkInfo

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

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

objectId	The object id received from a call to MMIc_startUserAgent.
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 objectId, UINT8
widgetType, UINT8 widgetID, const CHAR* URL)
```

objectId	The object id received from a call to MMIc_startUserAgent.
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.

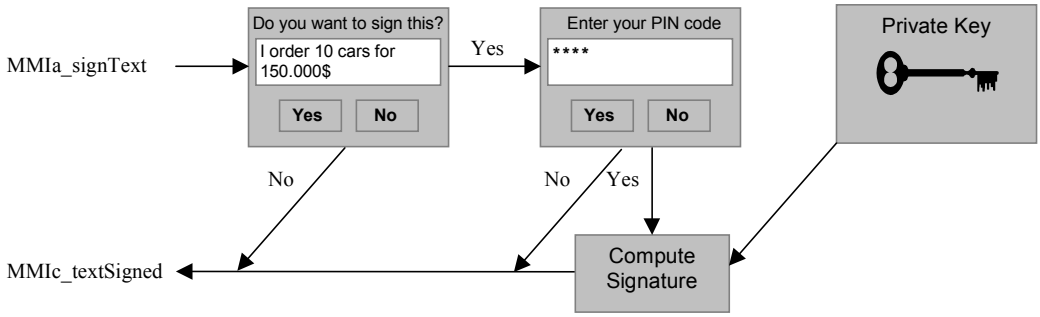
Constants

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

9.12 WMLS Library function Crypto.signText

Support for the functionality described herein is optional. To enable the signText feature, the configuration variable CAN_SIGN_TEXT (prepared in advance to be defined in confvars.h) must be set.



The WMLScript library Crypto has a function signText [WMLS-CRYPT], which when called causes the AUS WAP Browser to call the following Adapter function. The WAP application must open a dialog, display the message to sign (according to law in many countries) and wait for user confirmation. It must then prompt the user for verification information, like a PIN code. This information, in combination with the private key and data (the message, digested by the AUS WAP Browser), is used to compute a digital signature. The WAP application should use special signature keys that are distinct from authentication keys used for WTLS. The WAP application shall compute and pass the computed signature back to the AUS WAP Browser.

Note: When this functionality is enabled in the AUS WAP Browser, one or two Crypto API functions are used:

- CRYPTa_generateRandom (if not CLNTa_currentTime returns GMT)
- CRYPTa_hash (to perform an SHA-hash, i.e., MD5 is not used)

One or both of these functions must therefore be implemented if the signText functionality is enabled, and if the configuration of the AUS WAP Browser does not include WTLS.

signText

The WAP application must open a dialog, display the message to sign (according to law in many countries) and wait for the user to confirm. Then the data (which is a digest computed from the text) is encrypted using a private key and either RSA or ECDSA. The private key used must require user verification information, e.g., a PIN. The user verification information must be requested every time this function is called.

```
VOID MMIa_signText (UINT8 objectId, UINT8 signId,
const WCHAR *text, const CHAR *data, UINT16
dataLen, UINT8 keyIdType, const CHAR *keyId, UINT16
keyIdLen, UINT8 options)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`signId`	The id of this operation. The id is to be used in the corresponding Connector function call.
`text`	The message to be signed. The WAP application must open a dialog, display the message and wait for the user to confirm (according to law in many countries). The string is deleted when the function returns.
`data`	The data to be encrypted. The original text message (the text being signed) has already been digested using SHA-1. The 20-byte output and an SHA-1 algorithm identifier has been combined into an ASN.1 value of type DigestInfo [WMLS-CRYPT], which in turn has been DER-encoded [DER]. This data is to be encrypted with the signer's private key and user verification information (e.g. PIN), as described in [PKCS1] section 7, using block type 1. The resulting data string is the signature. The verification information must be retrieved from the user through a dialog. The string is deleted when the function returns.
`dataLen`	The length of the text.
`keyIdType`	Can be set to the following constants. SIGN_NO_KEY: No key identifier is supplied. Any key and certificate available may be used. SIGN_SHA_KEY: An SHA-1 hash of the public key is supplied in the next parameter. A private key corresponding to the given public key hash must be used. SIGN_SHA_CA_KEY: An SHA-1 hash of a trusted CA public key (or multiple of them) is supplied in the next parameter. A private key that is certified by the indicated CA (or some of them) must be used.
`keyId`	Identifies the private key, based on the previous parameter. The string is deleted when the function returns.
`keyIdLen`	The number of bytes in the keyId.
`options`	Contains two options joined with the bitwise OR operation The following constants can be used to extract the value of each option. SIGN_RETURN_HASHED_KEY: If this option is set, a hash of the public key corresponding to the signature key used, must be passed in the call to MMIc_textSigned. SIGN_RETURN_CERTIFICATE: If this option is set a certificate, or the URL of a certificate, must be passed in the call to MMIc_textSigned.

textSigned

Pass back the computed signature requested by a previous call to MMIa_signText.

```
VOID MMIc_textSigned (UINT8 objectId, UINT8 signId,
UINT8 algorithm, const CHAR *signature, UINT16
sigLen, const CHAR *hashedKey, UINT16 hashedKeyLen,
const CHAR *certificate, UINT16 certificateLen,
UINT8 certificateType, UINT16 err)
```

`objectId`	The object id received from a call to MMIc_startUserAgent.
`signId`	The id is to be taken from the corresponding Adapter function call.
`algorithm`	The algorithm used to compute the signature. It can be set to either SIGN_ALG_RSA or SIGN_ALG_ECDSA.
`signature`	The computed signature. The string can be deleted after the call.
`sigLen`	The number of bytes in the computed signature.
`hashedKey`	The requested signer info in form of a hashed key. This argument shall be set if the argument options (in the MMIa_signText call) is set to SIGN_RETURN_HASHED_KEY. Otherwise, it shall be set to NULL. The string can be deleted after the call.
`hashedKeyLen`	The length of the byte string hashedKey.
`certificate`	The requested signer info in form of a certificate or an URL to a certificate. Shall be set if the argument options is set to SIGN_RETURN_CERTIFICATE. Otherwise, it shall be set to NULL. The string can be deleted after the call.
`certificateLen`	The length of the byte string certificate.
`certificateType`	The type of certificate supplied. Can be set to one of the following constants: SIGN_WTLS_CERTIFICATE, SIGN_X509_CERTIFICATE, SIGN_X968_CERTIFICATE and SIGN_URL_CERTIFICATE.
`err`	Error indication, having one of the following values: SIGN_NO_ERROR: Signing succeded. SIGN_MISSING_CERTIFICATE: No certificate matching the keyId parameter in the adapter function was available. SIGN_USER_CANCELED: the user canceled the operation by not confirming one of the dialogs.

SIGN_OTHER_ERROR: Other error.

Constants

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

SIGN_NO_KEY	0
SIGN_SHA_KEY	1
SIGN_SHA_CA_KEY	2
SIGN_ALG_RSA	1
SIGN_ALG_ECDSA	2
SIGN_WTLS_CERTIFICATE	2
SIGN_X509_CERTIFICATE	3
SIGN_X968_CERTIFICATE	4
SIGN_URL_CERTIFICATE	5
SIGN_RETURN_HASHED_KEY	0x2
SIGN_RETURN_CERTIFICATE	0x4
SIGN_NO_ERROR	0
SIGN_MISSING_CERTIFICATE	1
SIGN_USER_CANCELED	2
SIGN_OTHER_ERROR	3

10 Client API

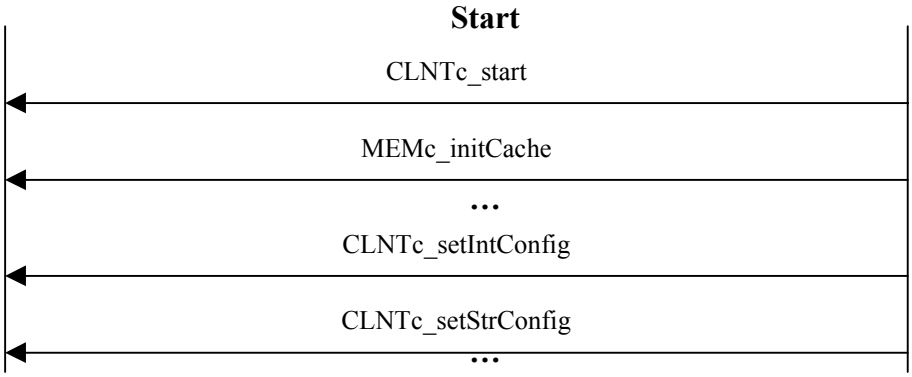
This API defines the general interface between the WAP application and the AUS WAP Browser. The functionality cover areas like:

- 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.



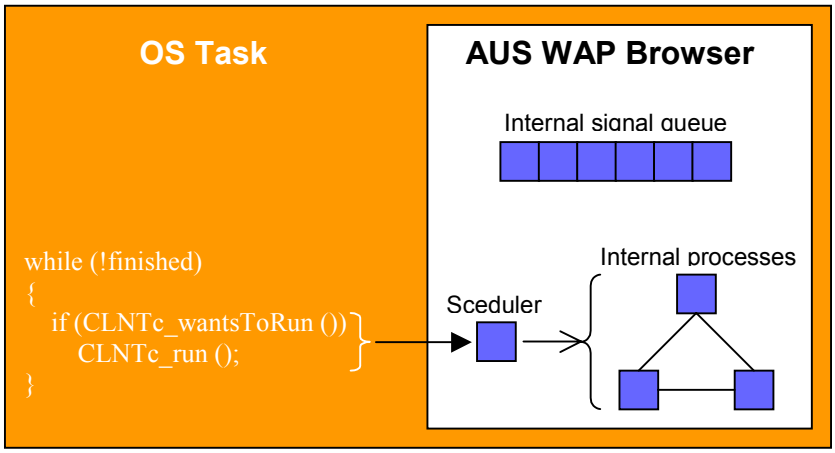
start

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.
After CLNTc_start has been called, it is free to call the initialising Connector functions, i.e., MEMc_initCache, CLNTc_setStrConfig and CLNTc_setIntConfig. Only general configurations variables may be initialised before any view has been opened. See the section “Configuration”.

```
VOID CLNTc_start (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.

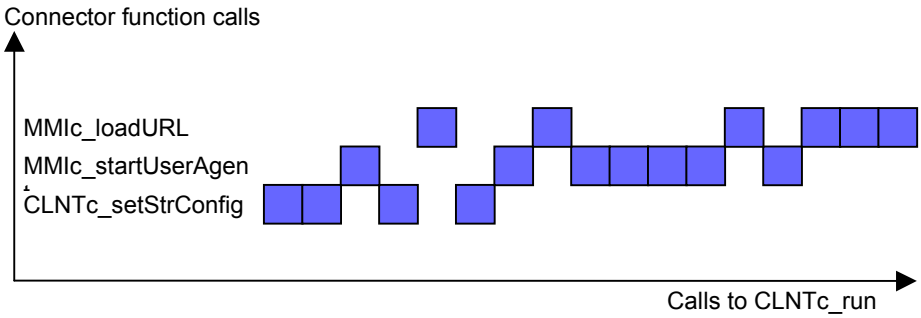


The picture above gives a very simple example of how the Connector functions might be called. The Connector function CLNTc_run processes the events (in form of internal signals) in the internal message queue. Connector functions or internal processes put the signals there.

If many Connector functions are called in a sequence, the signal queue will contains many signals. Since they are processed in order as they are put in the signal queue, and they require 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 are 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.

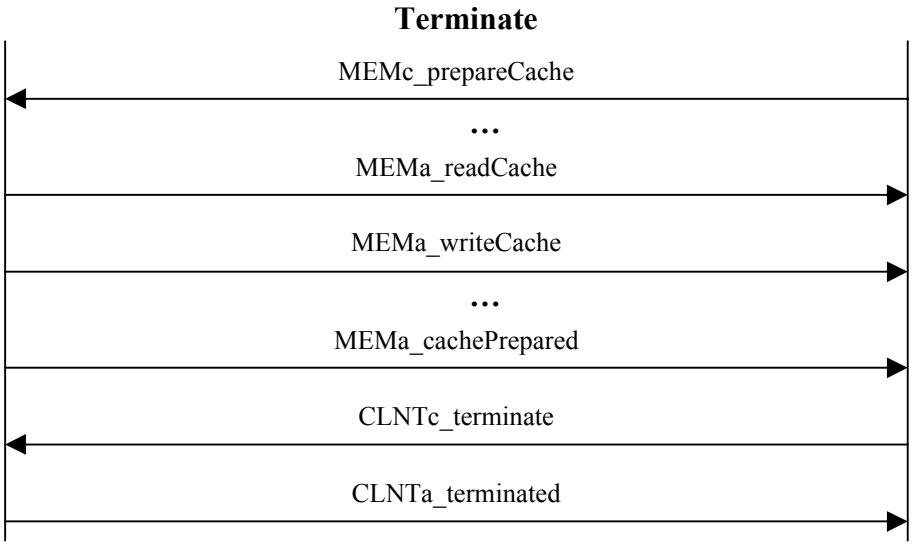
wantsToRun

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 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 and that the acknowledgement has been received (MEMa_cachePrepared). The termination is regarded as finished when the function CLNTa_terminated is called, e.g. the function CLNTc_start must not be called before the termination has finnished properly.

```
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. When this function has been called, CLNTc_run needs not to be called anymore.

```
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.

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.

When the user activates the AUS WAP Browser, by selecting a hyper link or such like, the request for a new WML deck is being sent over the network, a timer is being 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. Therefore, 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.

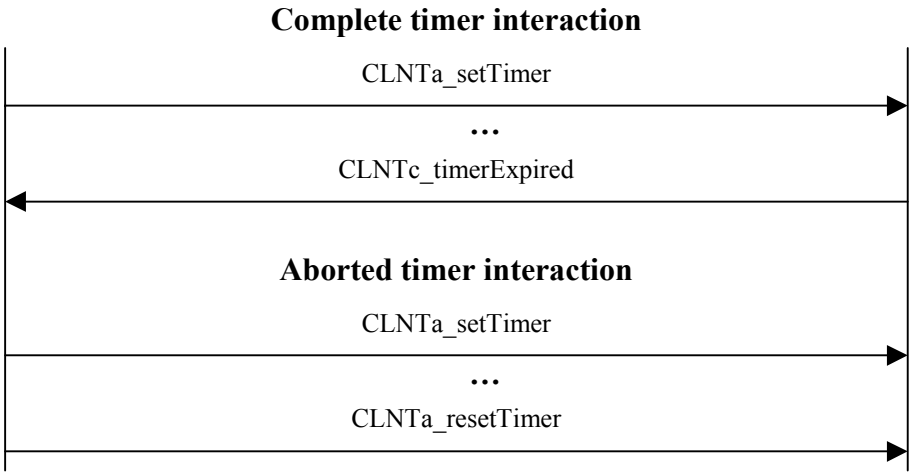
10.2 Time

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)
```

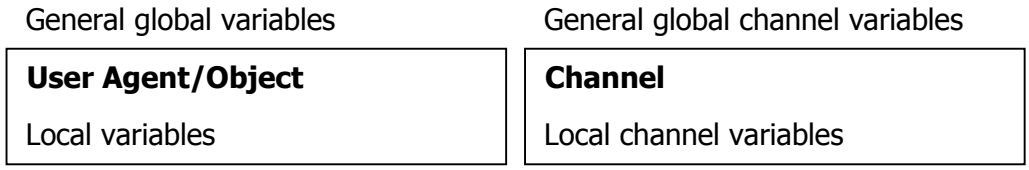
resetTimer

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 of the client

When the Generic WAP Stack is started, certain configuration variables have to be set and certain can be set. These variables can then be changed during run time. This is done through the functions described in this section.



The variables are divided into two kinds, general variables for WML user agents and WTA repository, etc, and variables for data channels. Each kind is in turn divided into global and local variables. Global general variables are set for all user agents and global channel variables are set for all channels. Local variables are set for individual user agents or channels. If a general variable is set, it can be overridden by setting it for an individual user agent or channel.

10.4.1 Configuration of general attributes

The following two functions are used to control general configuration variables of the AUS WAP Browser.

setIntConfig

This function is called by the WAP application at start up. It is also called when a configuration variable has been changed.

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

`objectId`	Each oject has its own set of general configuration variables; therefore must an object id be provided with the function. If the object ID is zero, the configuration value will be used in all object that has not this variable set. The object id has been received from a call to MMIc_startUserAgent.
`kind`	The valid kinds of configuration variables are found in the table below, and are of the type ConfigInt.
`value`	The value the variable shall take.

setStrConfig

This function is called by the WAP application at start up. It is also called when a configuration variable has been changed.

```
VOID CLNTc_setStrConfig (UINT8 objectId, ConfigStr
kind, const CHAR *value, UINT16 length)
```

`objectId`	Each object has its own set of general configuration variables; therefore must an object id be provided with the function. If the object ID is zero, the configuration value will be used in all object that has not this variable set. The object id has been received from a call to MMIc_startUserAgent.

`kind`	The valid kinds of configuration variables are found in the table below, and are of the type ConfigStr.
`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.

Constants

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

Name	Type	Description
configHISTORY_SIZE	ConfigInt	How many URLs should be held in the history? Default value is 10.
configWSP_Language	ConfigStr	A string that contains WSP codes that describe what languages the WAP application is able to handle. A content server may choose between content of different languages in order to suit the WAP browser best, if this header is supplied. The codes shall be given as encoded octets, as defined in [WAP-WSP]. This variable is only possible to set for all views.
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. The default value is 86400 hours. 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. Default value is 1.
configDISPLAY_IMAGES	ConfigInt	1 if images can be displayed, 0 otherwise. Default value is 1.
configUPDATE_IMAGES	ConfigInt	1 if images can be displayed after the card has been displayed, 0 otherwise. Default value is 0.
configUSERAGENT	ConfigStr	During runtime it is possible change the HTTP User Agent Field using this configuration variable. This variable is only possible to set for all views. Default value is "WAPPER".
configPROFILE	ConfigStr	An URL where the device's Profile is located. See [UAPROF]. This variable is only possible to set for all views.
configPROFILE_DIFF	ConfigStr	WBXML encoded content that describes the difference against the original Profile (configured with configPROFILE). The difference can be, for instance, that image handling is turned off. See [UAPROF]. This variable is only possible to set for all views.
configPUSH_SECURITY_ LEVEL	ConfigInt	There are two ways to utilise SIA – Trusted and Authenticated. It indicates whether the content is trusted and the Push Initiator is authenticated by the PPG. This following configuration variable is used to control the behaviour of the AUS WAP Browser, regarding this aspect. There are three possible push security levels: 0 - allow all initiators and untrusted content 1 – allow only authenticated initiators and trusted content 2 – do not allow any pushes This variable is only possible to set for all views.
configDEFAULT_ CHANNEL	ConfigInt	When the user agent cannot resolve what channel to use by comparing the request URL with the hosts that

		the channels have been configured with, the user agent will use the default channel. This variable identifies that channel by its channel id. If this variable is not set globally nor locally, the first created channel will be the default.

10.4.2 Configuration of network connections

Apart from the general configuration variables described in the previous section, the AUS WAP Browser has a set of specific configuration variables. The specific information is communication parameters needed for network connections. One user agent may have several connections associated. Data channels are therefore used to bundle information and characteristics about a specific connection. The following two functions are used when the specific details about a connection shall be set.

setDCHIntConfig

This function is called by the WAP application at start-up in order to initialise the AUS WAP Browser. After start-up, the function is called when a configuration variable is to be changed. This function is used when an integer value is to be configured.

```
VOID CLNTc_setDCHIntConfig (UINT8 objectId, UINT8
channelID, ConfigInt kind, UNIT32 value)
```

`objectId`	Each object has its own set of configuration variables; therefore must an object id be provided with the function. If the object ID is zero, the configuration value will be used in all object that has not this variable set. The object id has been received from a call to MMIc_startUserAgent.
`channelID`	Defines which channel the configuration should be applied on.
`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 should have.

setDCHStrConfig

This function is called by the WAP application at start-up in order to initialise the AUS WAP Browser. After start-up, the function is called when a configuration variable is to be changed. This function is used when a string value is to be configured.

```
VOID CLNTc_setDCHStrConfig (UINT8 objectId, UINT8
channelID, ConfigStr kind, const CHAR *value, UINT8
length)
```

`objectId`	Each object has its own set of general configuration variables; therefore must an object id be provided with the function. If the object ID is zero, the configuration value will be used in all object that has not this variable set. The object id has been received from a call to MMIc_startUserAgent.
`channelID`	Defines which channel the configuration should be applied on.
`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.

Constants

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

Name	Type	Description
configACCESS_TYPE	ConfigInt	Which bearer to use for a specific data channel. Currently supported bearers are BEARER_ANY_UDP = 0, BEARER_GSM_CSD = 10, BEARER_GSM_SMS = 3, BEARER_GSM_USSD = 2, BEARER_GSM_GPRS = 11, BEARER_BT = 15, BEARER_ANY = 255. All constants are defined in capiclnt.h.
configONLINE	ConfigInt	When initialising a channel the application may configure if the channel connection is to be regarded as online (TRUE) or offline (FALSE), by default. This variable must be defined when a user agent is started, before it is accessed. The functions CLNTa_setupConnection and CLNTa_closeConnection are then called if the user agent is offline. This variable must not be

		changed while a network connection is open. The default value is 0.
configCLIENT_LOCAL_PORT	ConfigInt	Defines the local “port” that the channel will use (eventually used in UDPa_sendRequest and UDPc_receivedRequest) For the AUS WAP Browser, this is basically just an ID, i.e it does not need to be the actual port number that the UDP message actually uses. It is only used as routing information. However, the two port numbers 2948 and 2949 are reserved for usage in AUS WAP Browser configurations that support PUSH. Further on, there must not be two channels with equal port numbers.
configUDP_IP_SRC	ConfigStr	IP address in network byte order (bytes ordered from left to right) for the WAP application (max 14 bytes)
configUDP_IP_GW	ConfigStr	IP address in network byte order (bytes ordered from left to right) for the WAP gateway (max 14 bytes)
configSMS_C	ConfigStr	BCD encoded msisdn number [GSM0340] of SMS center (max 14 bytes)
configSMS_GW	ConfigStr	BCD encoded msisdn number [GSM0340] for WAP gateway server (SME) (max 14 bytes)
configUSSD_C	ConfigStr	The service code for the network node, i.e. the correspondant to the SMS-C (max 14 bytes). The string is different depending on the operator of the GSM network. The service code is to be BCD encoded msisdn number [GSM0340].
configUSSD_GW	ConfigStr	Depending on the value of the configuration variable configUSSD_GW_TYPE, an IP number (bytes ordered from left to right) or a BCD encoded MSISDN number [GSM0340] for the WAP gateway (max 14 bytes)
configUSSD_GW_TYPE	ConfigInt	The kind of address the WAP gateway has.

		0 = IPv4 1 = IPv6 2 = MSISDN FF = No GW address, i.e. the configuration variable configUSSD_GW is not used. Note that WAP/1.2 requires a GW address. This is supported in order to be compliant with the WAP/1.1-standard.
configTIMEOUT	ConfigInt	The time in seconds the client can wait, when downloading has stalled, before the transaction is cancelled. Default value is 60.
configAUTH_PASS_GW	ConfigStr	The password for the WAP proxy server. If the configuration variable configAUTH_ID_GW is set, this variable has to be set.
configAUTH_ID_GW	ConfigStr	The user id for the WAP proxy server. If NULL is given, the current id, and its associated password, will be deleted.
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 and WTLS MODE_CL_WTLS_WTA = 2805 Connection mode MODE_CO_WSP = 9201 Connection mode with security layer MODE_CO_WTLS = 9203 Connection mode with WTA and WTLS MODE_CO_WTLS_WTA = 2923 All constants are defined in capiclnt.h.
configADD_HOST	ConfigStr	To add a routing host (i.e. www.xxx.yyy) for a specific channel. Wildcards (*) can be used in the host in order to have pattern matching performed. E.g. the host

		*.company.com matches the URLs: http://www.company.com http://download.company.com http://company.com https://www.company.com If two or more hosts match the URL, an arbitary host is choosen. I.e. it can not be determined what channel will be used when two channels both have hosts that match the URL.
configDELETE_HOST	ConfigStr	To delete a routing host for a specific channel.

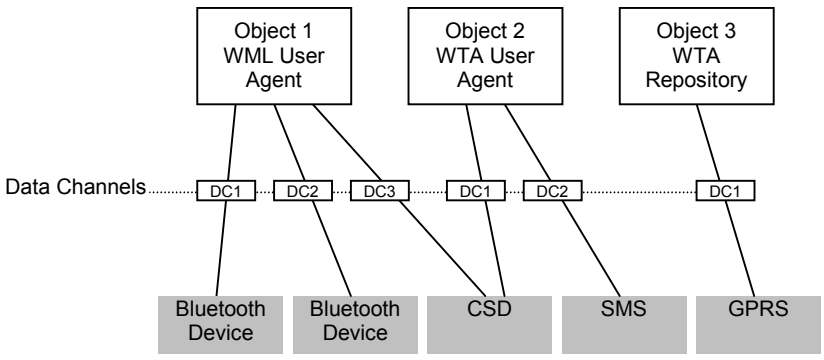
10.4.3 Type definitions

The functions in the sections above uses types that are defined in the header file capiclnt.h:

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

10.5 Data connection management

In order to manage different bearers in general (setup, teardown, etc.) a number of functions are defined together with the concept of “data channels”. The channels make it possible to enable the use of multiple concurrent bearers for one user agent (or more generally a “configurable object”). Furthermore, this is also to support the use of many access points for one single bearer, e.g. a number of Bluetooth devices accessible from a terminal.



The data channels are used to bundle information and characteristics about a specific connection. Typically the channel stores information about which bearer

(access type) to use, which gateway and stack configuration to use, gateway passwords and usernames, initial connection status (is the connection online or offline), local port number etc. The objects in the picture above could use channel configurations like this:

```
Object=1                                Object=2
        DefaultChannel=3                        DefaultChannel=1
        DataChannel=1                           DataChannel=1
                AccessType=BT                           AccessType=GSM_CSD
                Host=myCar.local                        StackMode=CO_WTLS_WTA
                StackMode=CL                            Gateway=195.100.108.76
                Gateway=0.0.0.0                         Username=Billy
                ClientLocalPort=1                       Password=Bob
        DataChannel=2                                   ClientLocalPort=4
                AccessType=BT                   DataChannel=2
                Host=myStereo.local                     AccessType=GSM_SMS
                StackMode=CL                            StackMode=CO_WTLS_WTA
                Gateway=0.0.0.0                         Gateway=4353453453
                ClientLocalPort=2                       ClientLocalPort=5
        DataChannel=3                   Object=3
                AccessType=GSM_CSD              DataChannel=1
                StackMode=CO_WTLS                       AccessType=GSM_GPRS
                Gateway=195.100.108.76                  StackMode=CO
                ClientLocalPort=3                       Gateway=195.100.108.76
                                                        ClientLocalPort=6
```

The channel numbers do only need to be unique within a user agent object.

When the WML user agent is opened, the default channel for CSD may be initialised emidiately. The following calls should occur in this case:

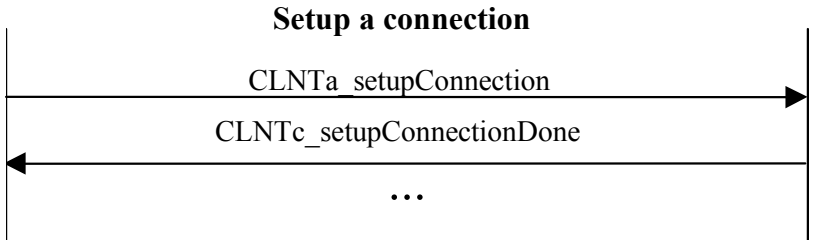
```
MMIc_startUserAgent (1, WML_USER_AGENT);
CLNTc_setIntConfig (1, configDEFAULT_CHANNEL, 3);
            …
CLNTc_setDCHIntConfig (1, 3, configACCESS _TYPE,
            BEARER_GSM_CSD);
CLNTc_setDCHIntConfig (1, 3, configSTACKMODE,
            MODE_CO_WTLS);
CLNTc_setDCHStrConfig (1, 3,
            configUDP_IP_GW,"\xC3\x64\x6C\x4C", 4);
CLNTc_setDCHIntConfig (1, 3, configCLIENT_LOCAL_PORT, 3);
            …
```

If an additional bearer, for instance a Bluetooth bearer, is detected and connected with the WAP application host device, the user agent object shall be updated with a new channel:

```
CLNTc_setDCHIntConfig (1, 2, configACCESS _TYPE,
            BEARER_BT);
CLNTc_setDCHIntConfig (1, 2, configSTACKMODE,
            MODE_CL_WSP);
CLNTc_setDCHStrConfig (1, 2,
            configUDP_IP_GW,"\x00\x00\x00\x00", 4);
CLNTc_setDCHIntConfig (1, 2, configCLIENT_LOCAL_PORT, 2);
CLNTc_setDCHStrConfig (1, 2,
            configADD_HOST,"myStereo.local", 14);
```

The channels are then controlled by the AUS WAP Browser by using the functions described in the following pages.

setupConnection



The AUS WAP Browser uses this function when a data channel is to be used by an object and the channel is not yet connected. Ex: a WML User Agent tries to load a URL and the CSD data call is not yet setup, or the GPRS connection has no active PDP-context. This function will only be called if the configuration variable configONLINE of the channel is set to FALSE.

```
VOID CLNTa_setupConnection (UINT8 objectId, UINT8
channelID)
```

`objectId`	The ID of the object that is about to make use of a network connection and requires it to be setup. (In the case of a WML browser this is the object id used in a call to MMIc_startUserAgent.)
`channelID`	The channel ID is defined by the WAP application at configuration time. Typically a device may maintain an own mapping table (with bearer, etc. associated with an arbitrary ID, for instance the channelID), and only use the channelID to identify the specific connection (See example in the UDP API).

setupConnectionDone

The function will be called from the WAP application as a response to `CLNTa_setupConnection` when the connection setup is finished and the user agent may continue sending the data request.

```
VOID CLNTc_setupConnectionDone (UINT8 objectId,
UINT8 channelID, BOOL success)
```

`objectId`	The ID of the object that requested the connection.
`channelID`	The channel ID is defined by the WAP application at configuration time.
`success`	Indicates whether the connection was succesfully setup or not (TRUE=setup ok).

closeConnection

This function is called by the AUS WAP Browser to initiate a connection shutdown. It may occur for instance when a user agent is closed and connections that the browser uses are still open. Configurations of the AUS WAP Browser that supports Push will also call this function when a Push session is terminated from the server. The network connection can be terminated when this function is called. This function will only be called if the configuration variable configONLINE of the channel is set to FALSE. The network connection can be terminated when this function is called.

```
VOID CLNTa_closeConnection (UINT8 objectId, UINT8
channelID)
```

`objectId`	The ID of the object that is about to have a network connection closed. (In the case of a WML browser this is the object id used in a call to MMIc_startUserAgent.)
`channelID`	The channel ID is defined by the WAP application at configuration time.

closeConnection

This function is called from the WAP application when the device wants to initiate a channel/connection shutdown. It may occur for instance when a Bluetooth device leaves the radio coverage range. The AUS WAP Browser needs to be notified about this in order to be able to terminate ongoing activities. Example:

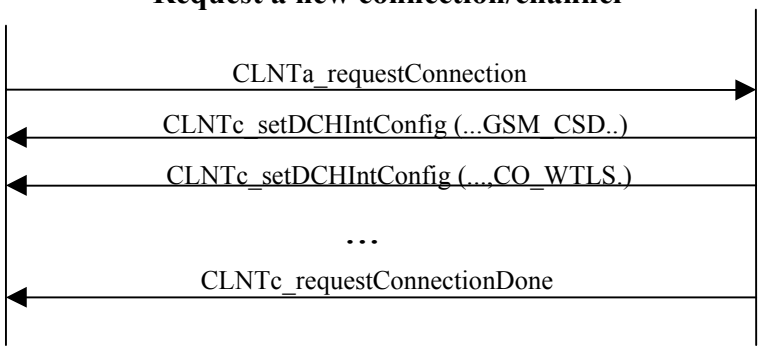
```
CLNTc_closeConnection (1, 3, TRUE);
```

```
VOID CLNTc_closeConnection (UINT8 objectId, UINT8
channelID, BOOL deleteChannel)
```

`objectId`	The ID of the object that uses the connection that will be shut down.
`channelID`	The channel ID is defined by the WAP application at configuration time.
`deleteChannel`	If deleteChannel is equal to TRUE the corresponding channel is also deleted.

requestConnection

Request a new connection/channel



When an object (e.g. WML user agent) makes a request and there is no data channel defined to serve the request this function will be called by the AUS WAP Browser. The WAP application then configures an apropriate (default) data channel to be used (by using CLNT_setDCHIntConfig and CLNT_setDCHStrConfig) and responds with CLNTc_requestConnectionDone.

```
VOID CLNTa_requestConnection (UINT8 objectId)
```

`objectId`	The ID of the object that requires a connection to be setup. (In the case of a WML browser this is the object id used in a call to MMIc_startUserAgent.)

requestConnectionDone

The function will be called from the WAP application as a response to CLNTa_requestConnection when the channel is configured. The user agent may now continue sending the data request.

```
VOID CLNTc_requestConnectionDone (UINT8 objectID,
BOOL success)
```

`objectId`	The ID of the object that requested the connection.
`success`	Indicates whether the configuration was succesfully performed or not (TRUE indicates success).

10.6 Messages

error

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

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

`objectId`	The object id received from a call to MMIc_startUserAgent. If

	the object id is equal to 0, the error concerns all user agents.
`errorNo`	The error number gives what kind of error it is. All errors are described in Appendix, error codes.
`errorType`	The error type indicates the severity of the error. All types are described in Appendix, error codes.

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.

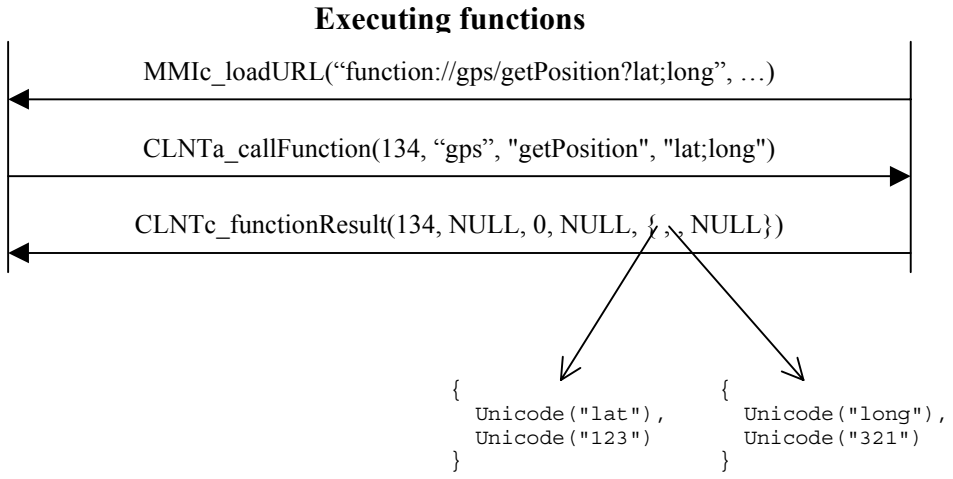
```
VOID CLNTa_log (UINT8 objectId, INT16 logNo, const
CHAR *format, …)
```

`objectId`	The object id received from a call to MMIc_startUserAgent. If the object id is equal to 0, the log concerns all user agents.
`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.

10.7 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">
```



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 function attributes, which is addressed by the URL (see the figure above for a reference). The string is deleted after the function call.

functionResult

This function is 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)

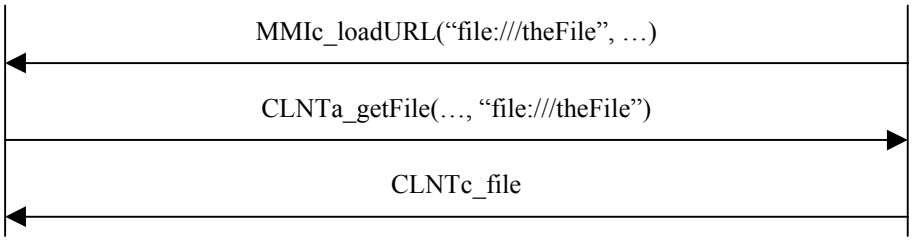
`functionId`	The functionId corresponds to the id sent in the adapter function call.

`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. The structures should be 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.

10.8 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



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

This function is called by the WAP application as a response to a CLNTa_getFile call.

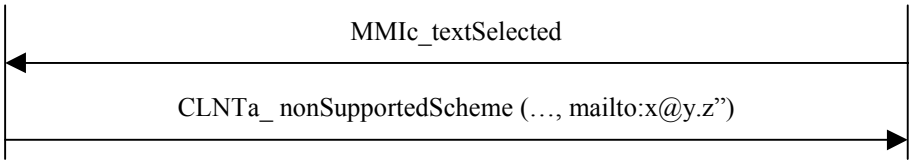
VOID CLNTc_file (UINT8 fileId, CHAR *data, UINT16 length, const CHAR *contentType);

`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.

10.9 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, and function). 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 schemes http, file, wtai, and function.

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

`objectId`	The object id received from a call to MMIc_startUserAgent.
`URL`	The non-supported URL. The URL is deleted after the function call.

10.10 Support for other applications to download arbitrary content

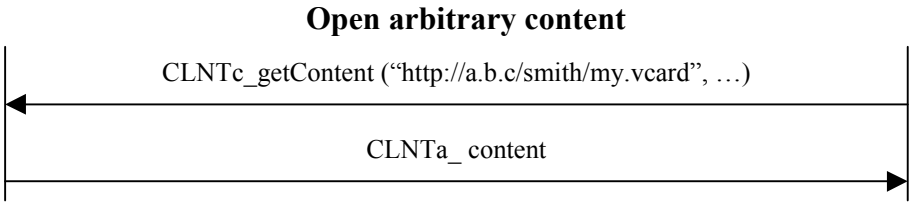
If for instance other content types than WML and WMLS should be downloaded, without going through WAE, or if other applications want to make use of the WAP protocol stack, these functions should be used. The function MMIa_unknownContent will not be called when these functions are used.

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

10.10.1 Standard functions

If the content to be retrieved/sent is no larger than the maximum PDU size, then the functions in this section can be used. For larger amounts of data, see the next section.

getContent



This function instructs the AUS WAP Browser to retrieve content (using HTTP GET). 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. A user agent must not be started, to use this function. This means that a get operation started with this function not can be stopped with the MMIc_stop function.

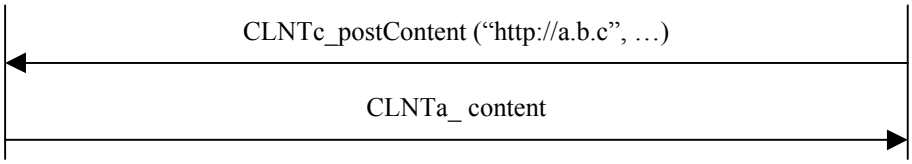
```
VOID CLNTc_getContent (const CHAR *url, UINT8 id,
BOOL reload, const CHAR *acceptHeader)
```

`url`	The URL of the content to be retrieved. The URL may be deleted after the call.
`id`	The id identifies this request in the WAP application. It is used in the corresponding Adapter function CLNTa_content. The id must differ from the ids that are currently taken by both CLNTc_postContent and CLNTc_getContent.

`reload`	When this argument is set to TRUE the request ignores any cached content and forces the client to load data from the network. Very large data that has been segmented is not stored in the cache.
`acceptHeader`	The argument specifies if any Accept-Header fields [RFC2068] are to be sent along in the request header. If this parameter is NULL, the accept-header will be ”*/*” by default, i.e., any content-type is accepted as response. The argument may be deleted after the call.

postContent

Post arbitrary content



This function instructs the AUS WAP Browser to send content (using HTTP POST). Note that this function bypasses the WAE layer of the AUS WAP Browser. A user agent must not be started, to use this function. This means that a post operation started with this function not can be stopped with the MMIc_stop function. If large data is to be transferred (see the next subsection), you will invoke CLNTc_postMoreContent to post the following segments.

```
VOID CLNTc_postContent (const CHAR *url, UINT8 id,
BOOL reload, const CHAR *acceptHeader, const CHAR
*data, UINT16 dataLen, BOOL moreData, const CHAR
*contentType, UINT8 sendMode, const CHAR
*contentDisp, UINT32 totalSize)
```

`url`	The URL of the content to be retrieved. The URL may be deleted after the call.
`id`	The id identifies this request in the WAP application. It is used in the corresponding call to the Adapter function CLNTa_content. The id must differ from the ids that are currently taken by both CLNTc_getContent and CLNTc_postContent. However, if the data posted in segments (see the argument “moreData”), the same id is to be used for all segments of the data being posted.
`reload`	When this argument is set to TRUE the request ignores any cached content and forces the client to load data from the network.
`acceptHeader`	The argument specifies if any Accept-Header fields [RFC2068] are to be sent along in the request header. If this parameter is NULL, the accept-header will be ”*/*” by

	default, i.e., any content-type is accepted as response. The argument may be deleted after the call.
`data`	The data to be sent to the server in the post operation. The data may be deleted after the call.
`dataLen`	The length of the data.
`moreData`	If large data is posted to a server, it should be segmented by the WAP application before it is sent (see the next subsection). The first segment is sent using this function and the following segments will use CLNTc_postMoreContent. This argument is set to TRUE if the data is segmented, otherwise this argument is set to FALSE. This functionality is optional to support. Note that segments of large data are not cached.
`contentType`	A zero terminated string with the content-type of the data may be defined in this argument (as a content type defined in [RFC2068]). A value of NULL indicates that no specific content-type information is to be sent in the post operation. The contentType value may be deleted after the call.
`sendMode`	This argument shall be set to: SENDMODE_URL_ENCODED - when the post shall be perfomed using the content type application/x-url-form-encoded (any non US-ASCII characters or other illegal URL characters will be automatically URL-encoded). The value given in the argument contentType is ignored. SENDMODE_MULTIPART_FORMDATA - when the data shall be posted using multipart/form-data [RFC2045]. The part in the multipart package will have its content type set the value given with the argument contentType. SENDMODE_BINARY - when the data shall be posted using the given with the argument contentType. The constants are defined in capiclnt.h.
`contentDisp`	When the argument sendMode is set to SENDMODE_MULTIPART_FORMDATA, i.e., posting of data using multipart/form-data, the contentDisp parameter may optionally be used to specify, for example, a filename of the content [RFC2045]. A NULL value indicates that no Content Disposition value is given. The contentDisp parameter may be deleted after the call.
`totalSize`	If moreData is set to TRUE, this argument indicates the total size of the large data (transferred in many segments).

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 or CLNTc_postContent. If large data is to be transferred, each call to this function (except the first), must be preceded by a call to the function CLNTc_acknowledgementContent (see the next subsection). The operation can be cancelled at any time by calling the function CLNTa_cancelContent.

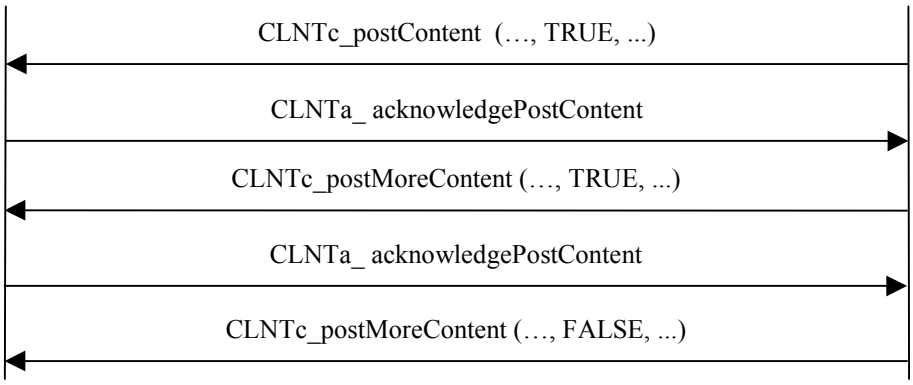
```
VOID CLNTa_content (UINT8 id, const CHAR *data,
UINT16 length, BOOL moreData, const CHAR
*contentType, UINT32 totalSize, INT16 errorNo)
```

`id`	The id retrieved in the Connector function CLNTc_getContent. If the data is segmented, the same id is used for all segments.
`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.
`moreData`	When large data is transferred from a content server to the AUS WAP Browser, the content server may segment the data before it is sent (see the next subsection). In order for the WAP application to detect if the data is segmented, this argument is set to TRUE for all segments except the last, for which it is set to FALSE. If the data is not segmented at all, the argument is set to FALSE. This functionality is optional to support.
`contentType`	The contentType is taken from the WSP header [WAP-WSP]. It gives the content type of the data. Content types are defined in [RFC2068]. The string is deleted when the function returns. Note that this argument is set to NULL when additional segments are retrieved with this function.
`totalSize`	If the data comes in one block, the totalSize is set to zero. Otherwise, if the data received is divided into segments, this argument gives the total size of all segments. This information however, may be omitted by the WAP gateway. This argument is in that case set to zero. Therefore, the WAP implementation must not depend on the total size when the data is segmented.
`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).

10.10.2 Additional functions for large data transfer

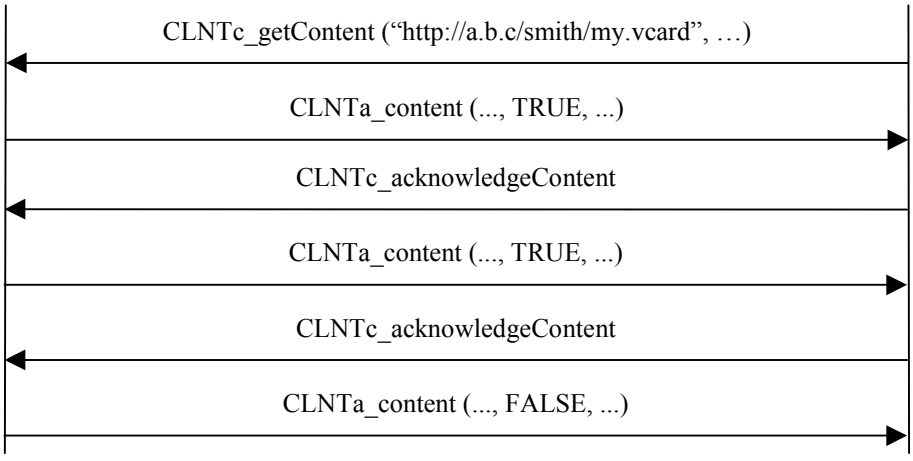
The AUS WAP Browser can handle content retrieval of sizes larger than a single PDU. This is accomplished using the Segmentation and Reassembly (SAR) feature of WTP. Flow control between the client and the server is handled by WTP, and flow control between the user level and the AUS WAP Browser is done with the help of a set of acknowledgement functions. In this way, flow control is achieved all the way from the user level to the server.

Sending large data



Sending large data works like this: Data to be sent is given to WAE and sent further down to WTP. There, the data block is split into *groups*. Each group consists of a number of *segments*. Using the SAR procedure, WTP sends one group and then waits for an acknowledgement from the server before it sends the next group. When all groups have been sent and acknowledged, the AUS WAP Browser forwards an acknowledgement to the user level. After receiving this (by CLNTa_acknowledgePostContent), the user can send the next part of the message.

Retrieving large data



On retrieving large data, a similar procedure is used: WTP receives the data from the server split into groups and segments. As soon as all the segments of a group have been assembled, WTP forwards the data upward to the user level. The user level processes the data and when it is prepared to handle the next data chunk, it

acknowledges the reception by calling the function CLNTc_acknowledgeContent. WTP then sends an acknowledgement to the server, indicating that it is ready to receive the next group.

postMoreContent

This function posts the following segments in the large data that was initiated by the call to CLNTc_postContent. Each call to this function must be preceded by a call to the function CLNTa_acknowledgementPostContent.

```
VOID CLNTc_postMoreContent (UINT8 id, const CHAR
*data, UINT16 dataLen, BOOL moreData)
```

`id`	The id identifies this request in the WAP application. The same id was used in the function CLNTc_postContent.
`data`	The data to be sent to the server in the post operation. The data may be deleted after the call.
`dataLen`	The length of the data.
`moreData`	This argument is set to TRUE until the last segment is sent, when it is set to FALSE.

acknowledgePostContent

This function is used as an acknowledgement of all calls to the functions CLNTc_postContent and CLNTc_postMoreContent where the attribute “moreData” is set to TRUE.

```
VOID CLNTa_acknowledgePostContent (UINT8 id)
```

`id`	The id used in the Connector function CLNTc_postContent.

acknowledgeContent

This function should be used as an acknowledgement of all calls to the function CLNTa_content where the attribute “moreData” is set to TRUE.

```
VOID CLNTc_acknowledgeContent (UINT8 id)
```

`id`	The id used in the Adapter function CLNTa_content.

cancelContent

This function is used when the sequence of calls to the function CLNTa_content should be cancelled. Such cancellation is possible as long the attribute “moreData” is set to TRUE.

```
VOID CLNTc_cancelContent (UINT8 id)
```

`id`	The id used in the Connector function CLNTa_content.

10.10.3 Configuration and memory requirements

To enable the functionality for large data transfer, the constant LARGE_DATA_TRANSFER_ENABLED must be defined.

The maximum size of groups and segments is determined by two configuration variables, WTP_SAR_GROUP_SIZE and WTP_SAR_SEGMENT_SIZE.

Both these constants are declared in confvars.h.

When sending large data amounts, the memory requirements to handle the data are d + min (d, m_g), where d is the size of a data block sent in via a call to CLNTc_postContent (or CLNTc_postMoreContent), and m_g is the maximum size of a group.

When receiving large data amounts, the AUS WAP Browser will use at most $2 * m_g$ bytes to handle incoming segments and groups.

Hence the setting of the configuration variable WTP_SAR_GROUP_SIZE directly affects the memory requirements of the AUS WAP Browser. However, a larger group size means more efficient data transfer.

The maximum segment size (WTP_SAR_SEGMENT_SIZE) should never be larger than the group size. In addition, it is preferable to make the segment size small enough that the risk of further segmentation at a lower level is minimised.

10.11 Support of propriatary WML Script library functions

If the browser shall be able to execute propriatary WML script function, like drawing lines on the screen, the functions, which are implemented outside the scope of the AUS WAP Browser, must be accessible from the AUS WAP Browser. The functions in this section are used for that. They are enabled when the configuration variable USE_PROPRIETARY_WMLS_LIBS (in confvars.h) is defined.

Propriatary WML Script functions must not only be implemented in the browser, they require the gateway to recognise them, as well. If the gateway not has a special compiler that recognises the propriatary functions, the scripts will be rejected. The gateway can be by-passed if the scripts are compiled by the content provider with a special compiler on forehand.

hasWMLSLibFunc

This function is called when a library function is called, from within a WMLS script, where the library ID is unknown to the interpreter. It will then call this function enabling the implementation and handling of proprietary WMLS library

functions. If the function exist, it will be executed by calling the CLNTa_callWMLSLibFunc function (see below).

```
BOOL CLNTa_hasWMLSLibFunc (UINT16 libNbr, UINT8
funcNbr, UINT8 *nbrOfParams)
```

`libNbr`	The numeric Library ID. The range 0..32767 are reserved for standard libraries. Therefore it is recomended to use numbers in the 32768..65535 range.
`funcNbr`	The numeric identifier for the function. The valid range is 0..255.
`nbrOfParams`	If the library function exists, this parameter must be set to the number of parameters that the function takes.

If there is an external implementation of the library function with libNbr and funcNbr then the return value should be TRUE, otherwise FALSE.

callWMLSLibFunc

This function will be called if a call to CLNTa_hasWMLSLibFunc returned TRUE. This function must result in the desired behaviour for the library function. Since the call to CLNTa_hasWMLSLibFunc gave the number of arguments, the call to this function will pass this number of arguments to this function in the params parameter. After the desired actions have been performed, this function must return a result in the form of a pointer to a WMLSvar struct (see description at the end of the section).
This function must not block the execution (e.g. hang while waiting for a network/user event). If this function is blocked the whole AUS WAP Browser will be blocked (CLNTc_run). When a proprietary library function can not return a result directly without being blocking, the pSeparateResponse parameter (see below) must be set to TRUE. The result must then be given at a later point in time by calling the connector function CLNTc_WMLSLibFuncResponse. This function requires an identifier so that the script interpreter knows what script is to be handed the separate response. This ID is given in the parameter invokeId. To use the CLNTc_WMLSLibFuncResponse function, the invokeId parameter that was given in the call to this function (CLNTa_callWMLSLibFunc) must be remembered and used when calling CLNTc_WMLSLibFuncResponse.

```
WMLSvar* CLNTa_callWMLSLibFunc (UINT16 invokeId,
UINT8 libNbr, UINT8 funcNbr, const WMLSvar * const
*params, BOOL *pSeparateResponse)
```

`invokeId`	This ID identifies this request. If the library function can not return directly, this ID must be used when the return value is passed back to the AUS WAP Browser by calling the CLNTc_WMLSLibFuncResponse function.
`libNbr`	The numeric Library ID. The range 0..32767 are reserved for standard libraries. Therefore it is

	recomended to use numbers in the 32768..65535 range.
`funcNbr`	The numeric identifier for the function. The valid range is 0..255.
`nbrOfParams`	If the library function exists, this parameter must be set to the number of parameters that the function takes.
`params`	An array with the parameter values with which the library function has been called. This parameter is a null-terminated array of pointers to WMLSvar structs. The array will contain the number of parameters that the function CLNTa_hasWMLSLibFunc returned in its nbrOfParams parameter. Note that when the in-parameter is of string type, the character encoding will always be UCS-2 (stringIANAcharset == 1000).
`pSeperateResponse`	When the library function returns emidiately and the return value of that function can be returned directly fom this function, this parameter must be set to FALSE. Otherwise, if the library function cannot return a return value directly, this parameter shall be set to TRUE. The return value will in that case be supplied with the CLNTc_WMLSLibFuncResponse function at a later point in time.

The function returns a pointer to a WMLSvar if pSeparateResponse is set to FALSE. If a fatal error occurs, NULL must be returned. A fatal error will abort the running WMLS script. If pSeparateResponse is TRUE, NULL must be returned. The responsibility to free the memory of the return value is the AUS WAP Browser's. Therefore must wip_malloc be used if USE_WIP_MALLOC if defined. Otherwise, malloc must be used.

WMLSLibFuncResponse

This function must be called if a call to CLNTa_callWMLSLibFunc was received and the parameter pSeparateResponse was set to TRUE. The purpose of this function is to asynchronously return a result from a proprietary library function. This is the case for any type of blocking proprietary library functions (e.g. a function waiting for a network/user event).

```
VOID CLNTc_WMLSLibFuncResponse (UINT16 invokeId,
const WMLSvar *resultVar)
```

`invokeId`	This ID was supplied from the initial CLNTa_callWMLSLibFunc function call when a function result could not directly be returned.
`resultVar`	A pointer to a structure with the result of the external library

function. Every library function shall return a value. If this parameter is NULL, a fatal library error is assumed to have occurred; the WMLS script will be aborted.

The WMLSvar struct

The WMLS library functions are assumed to return values of a certain type, WMLSvar. The return type field of the structure indicates what other field that is in use in the structure. The WMLSvar structure is composes as follows:

Type	Variable name	Description
UINT8	type	The value in this variable indicates what type of return value this structure holds: 0 = integer 1 = float 2 = string 3 = boolean 4 = invalid
INT32	intVal	If the type indicates integer, the value is stored here
BOOL	boolVal	If the type indicates boolean, the value is stored here: 0 = FALSE 1 = TRUE
FLOAT32	floatVal	If the type indicates float, the value is stored here. The value is to be a 32 bit floating point value following the ANSI/IEEE Standard 754
INT16	stringIANAcharset	If the type indicates string, this value is the MIBenum IANA code for the character encoding used in the string (stored in stringVal). For instance 1000 = UCS-2 (Unicode) and 36 = KSC5610. The character set must be supported by the AUS WAP Browser (see INIT_ACCEPTCHARSET in confvars.h).
UINT32	stringLengthInBytes	If the type indicates string, this is the length (in number of bytes) of the string stored in stringVal. The length should not count a terminating NULL, if there is one.
CHAR *	stringVal	If the type indicates string, this is the string value. It is encoded in the character encoding indicated by stringIANAcharset. The length of the string is indicated by stringLengthInBytes. It is not necessary to

		terminate the string with a NULL character. The stringVal must be a NULL pointer if the type indicates another value than string.

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

The function 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 the argument 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

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.

11.1 Overview

11.1.1 Design principles

When calling a WTAI function, i.e. when the AUS WAP Browser executes a WTAI function, the mobile phone implementation is to be designed so that control is returned to the AUS WAP Browser as soon as possibly. In order to support this design, every Adapter function has a corresponding connector function which purpose is to return the outcome of the adapter function back to the AUS WAP Browser. The WMLS application is idling between the calls of the Adapter and the corresponding Connector functions.

11.1.2 Return values of Adapter function calls

In order to control the operation of the mobile phone, a series of Adapter functions are to be used. They are to be implemented in a way so that control is given if possible. When it is not possible to control the mobile phone, for instance if an error occurs, an error code is to be returned as a result of the Adapter function that failed. The result of the execution of the Adapter function is returned through a corresponding Connector function. The result contains on errors the error code. The following table states general result values. Codes that are more specific are given with each Adapter function.

Value	Constant	Description
0	WTA_SUCCESS	Function successful
-128	WTA_INVALID	Function failure, invocation error.

11.2 Public WTAI

The functions in this section describe public WTAI telephony functionality. It is an optional API, which may be implemented, or not. If not needed, an Adapter function call should result in a corresponding Connector function call indicating a function failure. 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. The *number* parameter must be displayed to the user. If a data call currently is established in order to supply the IP network, it must be closed down before this operation is performed. The call must be terminated using the standard MMI.

```
VOID WTAIa_publicMakeCall (UINT8 objectId, const
CHAR *number)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`number`	This parameter holds the telephone number. The string is deleted after the function has returned.

publicMakeCallResponse

Call this function to indicate the establishing of a voice call or function failure arising from a *WTAIa_publicMakeCall* call.

```
VOID WTAIc_publicMakeCallResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	See table below.

The *result* parameter can take the following values:

Value	Constant name	Description
0	WTA_SUCCESS	Function successful. Voice call established.
-1	WTA_UNSPECIFIED_ERROR	Unspecified error.
-105	WTA_PUB_BUSY	Called party busy.
-106	WTA_PUB NO_NETWORK	Network is not available.
-107	WTA_PUB_NO_ANSWER	Called party did not answer.
-128	WTA_INVALID	Function failure, invocation error.

publicSendDTMF

WTAI Lib/Func ID: 512.1

This function is called when a DTMF sequence should be sent through the voice call most recently created using the *publicMakeCall* function. 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 (UINT8 objectId, const
CHAR *dtmf)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`dtmf`	The *dtmf* parameter specifies the DTMF sequence to be sent. The string is deleted after the function has returned.

publicSendDTMFResponse

Call this function to indicate a sent DTMF sequence or function failure arising from a *WTAIa_publicDTMFSend* call.

```
VOID WTAIc_publicSendDTMFResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	See table below.

The *result* parameter can take the following values:

Value	Constant name	Description
0	WTA_SUCCESS	Function successful. DTMF sequence was sent.
-1	WTA_UNSPECIFIED_ERROR	Unspecified error.
-108	WTA_PUB_NO_CONNECTION	No active voice connection.
-128	WTA_INVALID	Function failure, invocation error.

publicAddPBEntry

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. The *name* and *number* parameters must be displayed to the user.

```
VOID WTAIa_publicAddPBEntry (UINT8 objectId, const
CHAR *number, const WCHAR *name)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`number`	This parameter holds the telephone number. The string is deleted

after the function has returned.

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

publicAddPBEntryResponse

Call this function to indicate writing to the phone book or a function failure arising from a *WTAIa_publicAddPBEntry* call.

```
VOID WTAIc_publicAddPBEntryResponse (UINT8
objectId, INT8 result)
```

`objectId` The object id received by the corresponding adapter function.

`result` See table below.

The *result* parameter can take the following values:

Value	Constant name	Description
0	WTA_SUCCESS	Function successful. Phone book entry added.
-1	WTA_UNSPECIFIED_ERROR	Unspecified error.
-100	WTA_PB_NAME_INVALID	*Name* parameter contains unacceptable characters or is too long.
-101	WTA_PB_NUMBER_INVALID	*Number* parameter does not contain a valid phone number.
-102	WTA_PB_NUMBER_TOO_LONG	*Number* parameter is too long.
-103	WTA_PB_ENTRY_NOT_WRITTEN	Entry could not be written.
-104	WTA_PB_FULL	Phone book is full.
-128	WTA_INVALID	Function failure, invocation error.

11.3 Non-public WTAI

The functionality described in this section and the coming sections of this chapter is included in AUS WAP Browser configurations that include full WTA.

WTAI comprises a set of functions that control the mobile phone and retrieve information from it. The main control remains in the mobile phone, which decides if a WTAI operation can be performed, or not.

11.3.1 WTAI event handling by the AUS WAP Browser

To retrieve events from the mobile phone, e.g. if it is ringing; certain Connector functions are to be called from the mobile phone. The AUS WAP Browser gives acknowledgement of each such event by calling the function WTAa_processedByAService. An argument of it is assigned the value TRUE if the AUS WAP Browser has processed the event, FALSE if not. If FALSE is given, the mobile phone should act on the event as it does when the WTA browser is not active.

11.4 WTAI - Voice Calls

This section describes the events and functions that are used for call control. Voice calls can be placed, received, and terminated by WTAI services.

11.4.1 Call-handle

The WTAI functions make use of call-handles to identify voice and multiparty calls. They are generated and managed by the mobile phone. A call-handle must be unique within a context and may be given any value between 0 and 255.

11.4.2 WTA Events

incomingCall

Network Event: cc/ic

Call this function to indicate an incoming voice call.

```
VOID WTAIc_incomingCall (INT8 callHandle, const
CHAR *callerId)
```

`callHandle`	The call-handle of the incoming voice call.
`callerId`	The telephone number to which the call is connected. The AUS WAP Browser copies the string. The empty string is given if no number is available.

callCleared

Network Event: cc/cl

Call this function to indicate that a voice call has been cleared.

```
VOID WTAIc_callCleared (INT8 callHandle, INT8
result)
```

`callHandle`	The call-handle of the cleared voice call.
`result`	A description of why the call was cleared. See table below.

The *result* parameter can take the following values:

Value	Constant name	Description
0	WTA_CC_CL_NORMAL	Normal termination of a voice call, e.g., the near or far end released the voice call.
1	WTA_CC_CL_UNSPECIFIED	Unspecified, no details avilable.
2	WTA_CC_CL_NETWORK_ SPECIFIC	Network-specific reason.
3	WTA_CC_CL_DROPPED	Voice call was dropped, e.g., because a loss of signal.
4	WTA_CC_CL_BUSY	Called party was busy.
5	WTA_CC_CL_NO_ NETWORK	Network not available.
6	WTA_CC_CL_NO_ANSWER	Called party did not answer.
100	WTA_CC_CL_MULTI_OK	Normal termination of a GSM multiparty call, e.g., the near or far end released the voice call.
101	WTA_CC_CL_MULTI_ UNSPECIFIED	Termination of GSM multiparty call – unspecified, i.e. no details available.

callConnected

Network Event: cc/co

Call this function to indicate that an incoming or outgoing voice call has been established.

```
VOID WTAIc_callConnected (INT8 callHandle, const
CHAR *callerId)
```

`callHandle`	The call-handle of the voice call.
`callerId`	The telephone number to which the call is connected. The AUS WAP Browser copies the string. The empty string is given if no number is available..

outgoingCall

Network Event: cc/oc

Call this function to indicate that an outgoing voice call is being set up. It is not yet ringing at the called party.

```
VOID WTAIc_outgoingCall (INT8 callHandle, const
CHAR *callerId)
```

callHandle	The call-handle of the voice call.
callerId	The telephone number of the called party. The AUS WAP Browser copies the string.

callAlerting

Network Event: cc/cc

Call this function to indicate that an outgoing voice call is ringing at the called party.

```
VOID WTAIc_callAlerting (INT8 callHandle)
```

callHandle	The call-handle of the voice call.

DTMFSent

Network Event: cc/dtmf

Call this function to indicate that a DTMF sequence has been sent on a voice call. Multiple call of *WTAIa_voiceCallSendDTMF* will generate a matching sequense of *DTMFSent* events. The order shall be preserved.

```
VOID WTAIc_DTMFSent (INT8 callHandle, const WCHAR
*result)
```

callHandle	The call-handle of the voice call.
result	The parameter *result* holds the DTMF sequence that was sent. The AUS WAP Browser copies the string.

11.4.3 WMLScript functions

voiceCallSetup

WTAI Lib/Func ID: 513.0

This function is called to initialize a mobile-originated call.

```
VOID WTAIa_voiceCallSetup (UINT8 objectId, const
WCHAR *wtaChannelId, const CHAR *number)
```

objectId	This parameter is passed on to the corresponding connector function.
wtaChannelId	Identifies the entity calling this function. Should be used to

	check user permission. The string is deleted after the function has returned.
`number`	Holds the called telephone number. The string is deleted after the function has returned.

voiceCallSetupResponse

Call this function to return the result of the *WTAIa_voiceCallSetup* function.

```
VOID WTAIc_voiceCallSetupResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	The call-handle of the new voice call if successful or WTA_INVALID to indicate a function failure.

voiceCallAccept

WTAI Lib/Func ID: 513.1

This function is called to accept an incoming voice call that is waiting to be answered. The mobile phone should respond with a call to *WTAIc_callActive* when the call has been activated.

```
VOID WTAIa_voiceCallAccept (UINT8 objectId, const
WCHAR *wtaChannelId, INT8 callHandle)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`callHandle`	The call-handle of the voice call previously received with *WTAIc_incomingCall*.

voiceCallAcceptResponse

Call this function to return the result of the *WTAIa_voiceCallAccept* function.

```
VOID WTAIc_voiceCallAcceptResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID

voiceCallRelease

WTAI Lib/Func ID: 513.2

This function is called to release a voice call or a multiparty call. If this function is invoked to release a multiparty call, the multiparty call and all voice calls that were part of the multiparty call must be released. All releases of the voice calls and the multiparty call must be signalled by a CallCleared event each.

```
VOID WTAIa_voiceCallRelease (UINT8 objectId, const
WCHAR *wtaChannelId, INT8 callHandle)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`callHandle`	The call-handle of the voice call or multiparty call to be released.

voiceCallReleaseResponse

Call this function to return the result of the *WTAIa_ voiceCallRelease* function.

```
VOID WTAIc_voiceCallReleaseResponse (UINT8
objectId, INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID.

voiceCallSendDTMF

WTAI Lib/Func ID: 513.3

This function is called to send a DTMF sequence over the active call.

```
VOID WTAIa_voiceCallSendDTMF (UINT8 objectId, const
WCHAR *wtaChannelId, INT8 callHandle, const CHAR
*dtmf)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`callHandle`	The call-handle of the voice call over which the DTMF sequense will be sent.

`dtmf`	The DTMF sequence to be sent. The string is deleted after the function has returned.

voiceCallSendDTMFResponse

Call this function to return the result of the *WTAIa_voiceCallSendDTMF* function.

```
VOID WTAIc_voiceCallSendDTMFResponse (UINT8
objectId, INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID.

voiceCallCallStatus

WTAI Lib/Func ID: 513.4

This function is called when the call status is required from the running service.

```
VOID WTAIa_voiceCallCallStatus (UINT8 objectId,
const WCHAR *wtaChannelId, INT8 callHandle, INT8
field)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`callHandle`	The call-handle of the voice call to retrieve status about.
`field`	The parameter field indicates which field that should be used. Valid values are provided as constants declared in aapiwta.h. See table below.

The *field* parameter can take the following values:

Value	Constant name	Description
0	WTA_VC_CS_NUMBER	A telephone number will be returned.
1	WTA_VC_CS_STATUS	The state of the call. Must return one of the following values: "pending", "initiating", "wait_ring", "wait_answer", "in_call" or "end".
2	WTA_VC_CS_MODE	The mode. Must have one of the following values: "true" (equals keep-mode) or "false" (equals drop-

		mode).
3	WTA_VC_CS_NAME	The name or if not available the number.
4	WTA_VC_CS_DURATION	The duration of the call in seconds.

voiceCallCallStatusResponse

Call this function to return the result of the *WTAIa_voiceCallCallStatus* function.

```
VOID WTAIc_voiceCallCallStatusResponse (UINT8
objectId, INT8 result, const WCHAR *fieldValue)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID.
`fieldValue`	The *fieldValue* is a string associated with the requested *field* (see *WTAIa_voiceCallCallStatus*). If the *field* parameter is unrecognized, unsupported, or otherwise unavailable, an empty string is returned.

If the AUS WAP Browser internal memory allocator (see USE_WIP_MALLOC) is used, the string must be created with wip_malloc. Otherwise, it should be created with malloc. The AUS WAP Browser frees it with either wip_free or free.

voiceCallList

WTAI Lib/Func ID: 513.5

This function is called when a List Call WTAI service is required.

```
VOID WTAIa_voiceCallList (UINT8 objectId, const
WCHAR *wtaChannelId, BOOL returnFirst)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`returnFirst`	TRUE = return oldest, FALSE = return second oldest.

voiceCallListResponse

Call this function to return the result of the *WTAIa_voiceCallList* function.

```
VOID WTAIc_voiceCallListResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	Call-handle or WTA_INVALID.

11.5 WTAI - Network Messages

This section describes the events and functions used for network messaging. Network messages can be sent and received and information can be obtained by using these WTAI services. USSD messaging is described in the GSM section of this chapter.

11.5.1 Message-handle

The WTAI functions make use of message-handles to identify network messages. They are generated and managed by the mobile phone. A message-handle must be unique within a context and may be given any value between 0 and 65535.

11.5.2 WTA Events

messageSendStatus

Network Event: nt/st

Call this function to indicate a change in the status of an outgoing network message. This function shuold be called when a text message is sent, for instance if *WTAIa_netTextSend* has been called.

```
VOID WTAIc_messageSendStatus (INT16 msgHandle, INT8
result)
```

`msgHandle`	The message-handle.
`result`	See table below.

The *result* parameter can take the following values:

Value	Constant name	Description
0	WTA_NT_ST_MESSAGE_SENT	Message has been sent.
1	WTA_NT_ST_UNSPECIFIED	Message has been abandoned for unspecified reason.
2	WTA_NT_ST_NO_NETWORK	Message has been abandoned because network is not available.
3	WTA_NT_ST_NO_RESOURCE	Message has been abandoned because of insufficient resources.

incomingMessage

Network Event: nt/it

Call this function to indicate that an incoming network text is received.

```
VOID WTAIc_incomingMessage (INT16 msgHandle, const
WCHAR *sender)
```

`msgHandle`	The message-handle.
`sender`	The parameter sender contains address information about the sender. If no information of the sender can be provided, it is set to NULL. The AUS WAP Browser copies the string.

11.5.3 WMLScript functions

netTextSend

WTAI Lib/Func ID: 514.0

This function is called when a text message is to be sent.

```
VOID WTAIa_netTextSend (UINT8 objectId, const WCHAR
*wtaChannelId, const CHAR *number, const WCHAR
*text)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`number`	The destination telephones number. The string is deleted after the function call.
`text`	The actual text to be sent. The string is deleted after the function call.

netTextSendResponse

Call this function to return the result of the *WTAIa_netTextSend* function.

```
VOID WTAIc_netTextSendResponse (UINT8 objectId,
INT16 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	Message-handle or error code from table below.

The *result* parameter can take the following values:

Value	Constant name	Description
-1	WTA_UNSPECIFIED_ERROR	Unspecified error.
-100	WTA_NET_TEXT_TOO_LONG	*Text* parameter is too long.
-128	WTA_INVALID	Function failure, invocation error.

netTextList

WTAI Lib/Func ID: 514.1

This function is called to retrieve the message-handler of an existing message.

```
VOID WTAIa_netTextList (UINT8 objectId, const WCHAR
*wtaChannelId, BOOL returnFirst, INT8 messageType)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`returnFirst`	TRUE = return oldest message-handler, FALSE = return "next" message-handler.
`messageType`	See table below. Ignored if *returnFirst* is FALSE.

The *messageType* parameter can take the following values:

Value	Constant name	Description
0	WTA_NT_LIST_ALL	Include all read, unread and unsent messages.
1	WTA_NT_LIST_ONLY_UNREAD	Include only message, which are unread.
2	WTA_NT_LIST_ONLY_READ	Include only message, which are read.
3	WTA_NT_LIST_ONLY_UNSENT	Include only message, which are unsent.

netTextListResponse

Call this function to return the result of the *WTAIa_ netTextList* function.

```
VOID WTAIc_netTextListResponse (UINT8 objectId,
INT16 result)
```

objectId	The object id received by the corresponding adapter function.
result	Message-handle or WTA_INVALID.

netTextRemove

WTAI Lib/Func ID: 514.2

This function is called to permanently remove an incoming or outgoing network message from the device.

```
VOID WTAIa_netTextRemove (UINT8 objectId, const
WCHAR *wtaChannelId, INT16 msgHandle)
```

objectId	This parameter is passed on to the corresponding connector function.
wtaChannelId	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
msgHandle	The message-handle of the message to be removed.

netTextRemoveResponse

Call this function to return the result of the *WTAIa_ netTextRemove* function.

```
VOID WTAIc_netTextRemoveResponse (UINT8 objectId,
INT8 result)
```

objectId	The object id received by the corresponding adapter function.
result	See table below.

The *result* parameter can take the following values:

Value	Constant name	Description
0	WTA_SUCCESS	Function successful.
-1	WTA_UNSPECIFIED_ERROR	Unspecified error.
-101	WTA_NET_NO_REMOVE	Could not remove message.
-128	WTA_INVALID	Function failure, invocation error.

netTextGetFieldValue

WTAI Lib/Func ID: 514.3

This function is called when the content of a specific text message is to be read.

```
VOID WTAIa_netTextGetFieldValue (UINT8 objectId,
const WCHAR *wtaChannelId, INT16 msgHandle, INT8
field)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`msgHandle`	The message-handle of the specific message.
`field`	The parameter field indicates which entry of the text message that should be retrieved. Valid values are provided as constants declared in aapiwta.h. See table below.

The *field* parameter can take the following values:

Value	Constant name	Description
0	WTA_NT_GET_TEXT	The actual text message. This field is mandatory to support.
1	WTA_NT_GET_TSTAMP	The timestamp field received with the message for an incoming message, or an empty string for an outgoing message. This field is mandatory to support.
2	WTA_NT_GET_ ADDRESS	The originating address for an incoming message or the destination address for an outgoing message. This field is mandatory to support.
3	WTA_NT_GET_STATUS	Indicating the state of the message. Must be one of the following values: "unsent", "received" or "end". This field is mandatory to support.
4	WTA_NT_GET_READ	Indicating whether the message is marked as read or unread. Must be "true" or "false". This field is mandatory to support.
5	WTA_NT_GET_TSTAMP_	Indicates the offset of "tstamp" field from Coordinated Universal Time

	OFF	(UTC) in minutes. This field is optional to support.
6	WTA_NT_GET_TSTAMP_DEVICE	The date and time of the message as determinated by the device in [ISO8601] format. This field is optional to support.

netTextGetFieldValueResponse

Call this function to return the result of the *WTAIa_netTextGetFieldValue* function.

```
VOID WTAIc_netTextGetFieldValueResponse (UINT8
objectId, INT8 result, const WCHAR *fieldValue)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID
`fieldValue`	The *fieldValue* is a string associated with the requested *field* (see *WTAIa_netTextGetFieldValue*). If the *field* parameter is unrecognized, unsupported, or otherwise unavailable, an empty string is returned.

If the AUS WAP Browser internal memory allocator (see USE_WIP_MALLOC) is used, the string must be created with wip_malloc. Otherwise, it should be created with malloc. The AUS WAP Browser frees it with either wip_free or free.

netTextMarkAsRead

WTAI Lib/Func ID: 514.4

This function is called to mark a message as read.

```
VOID WTAIa_netTextMarkAsRead (UINT8 objectId, const
WCHAR *wtaChannelId, INT16 msgHandle)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`msgHandle`	The message-handle.

netTextMarkAsReadResponse

Call this function to return the result of the *WTAIa_ netTextMarkAsRead* function.

```
VOID WTAIc_netTextMarkAsReadResponse (UINT8
objectId, INT8 result)
```

objectId	The object id received by the corresponding adapter function.
result	WTA_SUCCESS or WTA_INVALID

11.6 WTAI - Phone Book

This section describes the events and functions used for phone book handling. The device's phone book can be accessed and modified by using these WTAI services.

11.6.1 Phone book index

The AUS WAP Browser considers the phone book to be an array of entries. To access an individual phone book entry an index between 1 and 65535 is to be used.

11.6.2 WMLScript functions

phoneBookWrite

WTAI Lib/Func ID: 515.0

This function is called when a phone book entry is to be added, overwriting any existing entry.

```
VOID WTAIa_phoneBookWrite (UINT8 objectId, const
WCHAR *wtaChannelId, INT16 index, const CHAR
*number, const WCHAR *name)
```

objectId	This parameter is passed on to the corresponding connector function.
wtaChannelId	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
index	The index parameter specifies the desired location of the entry within the phone book. Any previous phone book entry at the same index shall be overwritten. If the index is equal to 0, the next available phone book entry shall be used.
number	The parameter number holds the actual telephone number. The string is deleted when the function returns.
name	The parameter name holds the name associated with the number. The string is deleted when the function returns.

phoneBookWriteResponse

Call this function to return the result of the *WTAIa_phoneBookWrite* function.

```
VOID WTAIc_phoneBookWriteResponse (UINT8 objectId,
INT16 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	Phone book entry index, or error code from table below.

The *result* parameter can take the following values:

Value	Constant name	Description
-1	WTA_UNSPECIFIED_ERROR	Unspecified error.
-100	WTA_PB_NAME_INVALID	*Name* parameter contains unacceptable characters or is too long.
-101	WTA_PB_NUMBER_INVALID	*Number* parameter does not contain a valid phone-number.
-102	WTA_PB_NUMBER_TOO_LONG	*Number* parameter is too long.
-103	WTA_PB_ENTRY_NOT_WRITTEN	Entry could not be written.
-104	WTA_PB_FULL	Phone book is full.
-128	WTA_INVALID	Function failure, invocation error.

phoneBookSearch

WTAI Lib/Func ID: 515.1

This function is called to search the phone book for a specific item.

This function must first be invoked with a WTA_PB_NAME or WTA_PB_NUMBER as *field* parameter.

To continue a search, this function must be invoked with WTA_PB_CONTINUE for *field* parameter and an empty string for *value* parameter.

Note: This function is intended for use within a loop:

- with either WTA_PB_NAME or WTA_PB_NUMBER value of the *field* parameter on the first invocation

- with WTA_PB_CONTINUE as *field* value and empty string for *value* parameters of each subsequent time through the loop
- with the loop exiting when WTA_INVALID is returned

If the mobile phone do not supports ordering of strings in alphabetic order, the entries should be sorted in the order that they have in the phone book.

```
VOID WTAIa_phoneBookSearch (UINT8 objectId, const
WCHAR *wtaChannelId, INT8 field, const WCHAR
*value)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`field`	The parameter *field* defines what field that should be used. Valid values are provided as constants declared in aapiwta.h. See table below.
`value`	The parameter *value* is the actual data that is used when searching for a phone book entry, e.g. a name string if the field parameter is set to name. The string is deleted when the function returns.

The *field* parameter can take the following values:

Value	Constant name	Description
0	WTA_PB_SEARCH_ CONTINUE	Continue previous search.
1	WTA_PB_SEARCH_ NAME	The name in the phone book entry.
2	WTA_PB_SEARCH_ NUMBER	The telephone number in the phone book entry.

phoneBookSearchResponse

Call this function to return the result of the *WTAIa_phoneBookSearch* function.

```
VOID WTAIc_phoneBookSearchResponse (UINT8 objectId,
INT16 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	Phone book entry index, if successful. WTA_INVALID to indicate function failure or no match.

phoneBookRemove

WTAI Lib/Func ID: 515.2

This function is called when an entry at a certain position is to be removed. The index of the entry has for instance been retrieved as the result of a call to WTAIa_phoneBookSearch.

```
VOID WTAIa_phoneBookRemove (UINT8 objectId, const
WCHAR *wtaChannelId, INT16 index)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`index`	The parameter holds the index of the entry to be removed.

phoneBookRemoveResponse

Call this function to return the result of the *WTAIa_phoneBookRemove* function.

```
VOID WTAIc_phoneBookRemoveResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	See table below.

The *result* parameter can take the following values:

Value	Constant name	Description
0	WTA_SUCCESS	Function successful.
-1	WTA_UNSPECIFIED_ERROR	Unspecified error.
-105	WTA_PB_ENTRY_NOT_REMOVABLE	Could not remove phone book entry.
-128	WTA_INVALID	Function failure, invocation error.

phoneBookGetFieldValue

WTAI Lib/Func ID: 515.3

This function is called to retrieve a field value from a specific phone book entry.

```
VOID WTAIa_phoneBookGetFieldValue (UINT8 objectId,
const WCHAR *wtaChannelId, INT16 index, INT8 field)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`index`	The parameter index identifies a phone book entry.
`field`	The parameter field indicates which part of the phone book entry that should be retrieved. Valid values are provided as constants declared in aapiwta.h. See table below.

The *field* parameter can take the following values:

Value	Constant name	Description
0	WTA_PB_GET_NAME	The name in the phone book entry.
1	WTA_PB_GET_NUMBER	The telephone number in the phone book entry.

phoneBookGetFieldValueResponse

Call this function to return the result of the *WTAIa_phoneBookGetFieldValue* function.

```
VOID WTAIc_phoneBookGetFieldValueResponse (UINT8
objectId, INT8 result, const WCHAR *fieldValue)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID
`fieldValue`	A string associated with the requested field (see *WTAIa_phoneBookGetFieldValue*). If the field parameter is unrecognized, unsupported, or otherwise unavailable, an empty string is returned.

If the AUS WAP Browser internal memory allocator (see USE_WIP_MALLOC) is used, the string must be created with wip_malloc. Otherwise, it should be created with malloc. The AUS WAP Browser frees it with either wip_free or free.

phoneBookChange

WTAI Lib/Func ID: 515.4

This function is called if a certain field of a specific phone book entry is to be changed.

```
VOID WTAIa_phoneBookChange (UINT8 objectId, const
WCHAR *wtaChannelId, INT16 index, INT8 field, const
WCHAR *newValue)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`index`	The parameter index identifies a phone book entry.
`field`	The parameter field indicates which part of the phone book entry that should be changed. Valid values are provided as constants declared in aapiwta.h. See table below.
`newValue`	The new value is provided through this parameter. The string is deleted when the function returns.

The *field* parameter can take the following values:

Value	Constant name	Description
0	WTA_PB_CHANGE_NAME	The name in the phone book entry.
1	WTA_PB_CHANGE_ NUMBER	The telephone number in the phone book entry.

phoneBookChangeResponse

Call this function to return the result of the *WTAIa_phoneBookChange* function.

```
VOID WTAIc_phoneBookChangeResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	See table below.

The *result* parameter can take the following values:

Value	Constant name	Description
0	WTA_SUCCESS	Function successful.
-1	WTA_UNSPECIFIED_ERROR	Unspecified error.
-100	WTA_PB_NAME_INVALID	*Field* parameter is *name* and *newValue* contains an unacceptable character or is too long.
-101	WTA_PB_NUMBER_INVALID	*Field* parameter is *number*

		and *newValue* is not a phone-number.
-102	WTA_PB_NUMBER_TOO_LONG	*Field* parameter is *number* and *newValue* is too long.
-103	WTA_PB_ENTRY_NOT_WRITTEN	Entry could not be changed.
-104	WTA_PB_FULL	Phone book is full.
-106	WTA_PB_FIELD_NOT_SUPPORTED	*Field* parameter is not supported.
-107	WTA_PB_VALUE_INVALID	*Field* parameter is not *name* or *number* and *newValue* is unacceptable or is too long.
-128	WTA_INVALID	Function failure, invocation error.

11.7 WTAI - Call Logs

11.7.1 Log-handle

The WTAI functions make use of log-handles to identify call log entries. They are generated and managed by the mobile phone. A log-handle must be unique within a context and may be given any value between 0 and 65535.

11.7.2 WMLScript functions

callLogDialled

WTAI Lib/Func ID: 519.0

This function is called when the running service requires a handle to a call in the list of last called numbers.

```
VOID WTAIa_callLogDialled (UINT8 objectId, const
WCHAR *wtaChannelId, BOOL returnFirst)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`returnFirst`	TRUE = the most recent call log entry is returned, FALSE = the next most recent.

callLogDialledResponse

Call this function to return the result of the *WTAIa_callLogDialled* function.

```
VOID WTAIc_callLogDialledResponse (UINT8 objectId,
INT16 result)
```

objectId	The object id received by the corresponding adapter function.
result	Log-handle if successful or WTA_INVALID if the end of the list was reached.

callLogMissed

WTAI Lib/Func ID: 519.1

This function is called when the running service requires a handle to a call in the list of missed calls.

```
VOID WTAIa_callLogMissed (UINT8 objectId, const
WCHAR *wtaChannelId, BOOL returnFirst)
```

objectId	This parameter is passed on to the corresponding connector function.
wtaChannelId	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
returnFirst	TRUE = the most recent missed call log entry is returned, FALSE = the next most recent.

callLogMissedResponse

Call this function to return the result of the *WTAIa_callLogMissed* function.

```
VOID WTAIc_callLogMissedResponse (UINT8 objectId,
INT16 result)
```

objectId	The object id received by the corresponding adapter function.
result	Log-handle if successful or WTA_INVALID if the end of the list was reached.

callLogReceived

WTAI Lib/Func ID: 519.2

This function is called when the running service requires a handle to a call in the list of received called.

```
VOID WTAIa_callLogReceived (UINT8 objectId, const
WCHAR *wtaChannelId, BOOL returnFirst)
```

objectId	This parameter is passed on to the corresponding connector function.
wtaChannelId	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
returnFirst	TRUE = the most recent received call log entry is returned, FALSE = the next most recent.

callLogReceivedResponse

Call this function to return the result of the *WTAIa_callLogReceived* function.

```
VOID WTAIc_callLogReceivedResponse (UINT8 objectId,
INT16 result)
```

objectId	The object id received by the corresponding adapter function.
result	Log-handle if successful or WTA_INVALID if the end of the list was reached.

callLogGetFieldValue

WTAI Lib/Func ID: 519.3

This function is called when the value from a specific entry of a structure, previously retrieved from any of the functions *WTAIa_callLogDialled*, *WTAIa_callLogMissed*, *WTAIa_callLogReceived*.

```
VOID WTAIa_callLogGetFieldValue (UINT8 objectId,
const WCHAR *wtaChannelId, INT16 logHandle, INT8
field)
```

objectId	This parameter is passed on to the corresponding connector function.
wtaChannelId	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
logHandle	A log-handle previously retrieved from any of the functions *WTAIa_callLogDialled*, *WTAIa_callLogMissed*, *WTAIa_callLogReceived*.
field	An entry in the structure to retrieve. See table below.

The *field* parameter can take the following values:

Value	Constant name	Description
0	WTA_CL_GET_NUMBER	The telephone number of the call.
1	WTA_CL_GET_TSTAMP	Date and time in [ISO8601] format.
2	WTA_CL_GET_EXPLANATION	The reason why the phone number is not available or an empty string.

callLogGetFieldValueResponse

Call this function to return the result of the *WTAIa_callLogGetFieldValue* function.

```
VOID WTAIc_callLogGetFieldValueResponse (UINT8
objectId, INT8 result, const WCHAR *fieldValue)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID
`fieldValue`	A string associated with the requested field (see *WTAIa_callLogGetFieldValue*). If the *field* parameter is unrecognized, unsupported, or otherwise unavailable, an empty string is returned.

If the AUS WAP Browser internal memory allocator (see USE_WIP_MALLOC) is used, the string must be created with wip_malloc. Otherwise, it should be created with malloc. The AUS WAP Browser frees it with either wip_free or free.

11.8 WTAI - Miscellaneous

11.8.1 WTA Events

networkStatus

Network Event: ms/ns

Call this function to indicate that the network status has changed.

```
VOID WTAIc_networkStatus (INT8 inService, const
WCHAR *networkName, INT8 explanation)
```

`inService`	The parameter *inService* if the device is in service and can place or receive calls. 0 = the device is not in service 1 = the device is in service

networkName	The parameter *networkName* is the name of the network. The AUS WAP Browser copies the string.
explanation	The *explanation* parameter contains the reason why the device is not in service. 0 = no explanation given 1 = no network were found 2 = only forbidden networks were found

11.8.2 WMLScript functions

miscSetIndicator

WTAI Lib/Func ID: 516.0

This function is called to modify the state of a logical indicator in the mobile phone.

```
VOID WTAIa_miscSetIndicator (UINT8 objectId, const
WCHAR *wtaChannelId, INT8 type, INT8 newState)
```

objectId	This parameter is passed on to the corresponding connector function.
wtaChannelId	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
type	The parameter type holds a value describing what indication to be turned on or off. Valid values are provided as constants declared in aapiwta.h. See table below.
newState	This parameter specifies the desired state of the indicator.

The *type* parameter can take the following values:

Value	Constant name	State	Description
0	WTA_MISC_INCOMING_SPEECH	*	Incoming Speech call.
1	WTA_MISC_INCOMING_DATA	*	Incoming Data call.
2	WTA_MISC_INCOMING_FAX	*	Incoming Fax Call.
3	WTA_MISC_CALL_WAITING	*	Call Waiting.
4	WTA_MISC_TEXT_MESSAGE	**	Text Message.
5	WTA_MISC_VOICE_MAIL	**	Voice Mail Message.
6	WTA_MISC_FAX_MESSAGE	**	Fax Message.

7	WTA_MISC_EMAIL_MESSAGE	**	Email Message.

*) Zero signifies the indicator is “off”, positiv values signify it is “on”, and negative values are not allowed.

**) Zero indicates there are no messages, positive values indicate the number of messages, and negative values are not allowed.

miscSetIndicatorResponse

Call this function to return the result of the *WTAIa_miscSetIndicator* function.

```
VOID WTAIc_miscSetIndicatorResponse (UINT8
objectId, INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID.

11.9 WTAI - GSM

A GSM mobile phone has an extended set of WTAI functionality to support. The functions in this section are structured in the same subsections as the standard WTAI functionality is structured.

After a call over a GSM network has been set-up, it can be controlled further. The calls can be held and retaken, additional calls can be joined with current calls.

11.9.1 WTA Events

callHeld

Network Event: gsm/ch

Call this function when a call held indication is to be given to the AUS WAP Browser as a response upon a *WTAIa_GSMcallHold* call, as well as upon a *WTAIa_GSMretrieveFromMultiparty* function call.

```
VOID WTAIc_callHeld (INT8 callHandle)
```

`callHandle`	The call-handle of the voice call or multiparty call put on hold.

callActive

Network Event: gsm/ca

Call this function when a call active indication is received. It should be called as a response upon a WTAIa_voiceCallAccept function call, as well as upon a *WTAIa_GSMjoinMultiparty* function call.

```
VOID WTAIc_callActive (INT8 callHandle)
```

`callHandle`	The call-handle of the activated voice call or multiparty call.

USSDReceived

Network Event: gsm/ru

Call this function when an incoming USSD string is detected.

```
VOID WTAIc_USSDReceived (const WCHAR *message,
const WCHAR *codingScheme, INT8 type, const WCHAR
*transactionId)
```

`message`	The incoming string. Contents of the incoming USSD string. This may include any of the USSD characters permitted by GSM 02.90. For type 3 messages, this parameter is NULL. The AUS WAP Browser copies the string.
`codingScheme`	The argument dataCodingScheme can take values according to GSM 02.90. For type 3 messages, this parameter is NULL. The AUS WAP Browser copies the string.
`type`	The parameter type is the type of USSD operation. Valid values are provided as constants declared in aapiwta.h. See table below.
`transactionId`	This parameter (specified in GSM 04.07 §11) should, in the case of a response to a network initiated USSD (i.e. a type 1 or 2 message), be set to the value of the transaction id of the corresponding network initiated message. In case of a type 0 message, the parameter id should be set to -1. The AUS WAP Browser copies the string.

The *type* parameter can take the following values:

Value	Constant name	Description
0	WTA_USSD_RECEIVED_ RESULT	Result to a ProcessUnstructuredSS-Request operation.
1	WTA_USSD_RECEIVED_ REQUEST	UnstructuredSS-Request operation.
2	WTA_USSD_RECEIVED_	UnstructuredSS-Notify operation.

	NOTIFY	
3	WTA_USSD_RECEIVED_ERROR	Error to a ProcessUnstructuredSS-Request operation.

11.9.2 WMLScript functions

GSMHold

WTAI Lib/Func ID: 518.0

This function is called to put an active GSM voice or multiparty call on hold. A callHeld event occurs when the call is put on hold.

```
VOID WTAIa_GSMHold (UINT8 objectId, const WCHAR
*wtaChannelId, INT8 callHandle)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`callHandle`	The call-handle of the voice call or multiparty call to be put on hold.

GSMHoldResponse

Call this function to return the result of the *WTAIa_GSMHold* function.

```
VOID WTAIc_GSMHoldResponse (UINT8 objectId, INT8
result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID.

GSMRetrieve

WTAI Lib/Func ID: 518.1

This function is called to make a held GSM voice or multiparty call active. A callActive event occurs when the call is retrieved from hold.

```
VOID WTAIa_GSMRetrieve (UINT8 objectId, const WCHAR
*wtaChannelId, INT8 callHandle)
```

`objectId`	This parameter is passed on to the corresponding connector function.

`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`callHandle`	The call-handle of the voice call or multiparty call to be retrieved from hold.

GSMRetrieveResponse

Call this function to return the result of the *WTAIa_GSMRetrieve* function.

```
VOID WTAIc_GSMRetrieveResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID.

GSMTransfer

WTAI Lib/Func ID: 518.2

This functino is called to transfers a GSM call to another party. A call cleared event occurs for each of the GSM calls B and C when they are no longer available to the WTA user agent.

```
VOID WTAIa_GSMTransfer (UINT8 objectId, const WCHAR
*wtaChannelId, INT8 callHandleB, INT8 callHandleC)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`callHandleB`	The call-handle of the incoming call.
`callHandleC`	The call-handle of the ongoing call.

GSMTransferResponse

Call this function to return the result of the *WTAIa_GSMTransfer* function.

```
VOID WTAIc_GSMTransferResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID.

GSMDeflect

WTAI Lib/Func ID: 518.3

This functin is called to deflect an unanswered incoming GSM voice call to another number. A CallCleared event for the incoming call occurs once the call has ended.

```
VOID WTAIa_GSMDeflect (UINT8 objectId, const WCHAR
*wtaChannelId, INT8 callHandle, const CHAR *number)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`callHandle`	The call-handle of the call to deflect.
`number`	The destination (any valid telephone number) to deflect the call to. The string is deleted when the function returns.

GSMDeflectResponse

Call this function to return the result of the *WTAIa_GSMDeflect* function.

```
VOID WTAIc_GSMDeflectResponse (UINT8 objectId, INT8
result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID.

GSMMultiparty

WTAI Lib/Func ID: 518.4

This function is called to establishe a GSM multiparty call. It is also called when to add an additional GSM voice call to an already established GSM multiparty call. If the function is successfully invoked, a callActive event occurs for the previously held call. If a multiparty call is created as a result of this function, CallConnected event also occurs for the newly created GSM multiparty call.

```
VOID WTAIa_GSMMultiparty (UINT8 objectId, const
WCHAR *wtaChannelId)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to

check user permission. The string is deleted after the function has returned.

GSMMultipartyResponse

Call this function to return the result of the *WTAIa_GSMMultiparty* function.

```
VOID WTAIc_GSMMultipartyResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	Call-handle or WTA_INVALID.

GSMSeparate

WTAI Lib/Func ID: 518.5

This function is called to separate a specific GSM voice call from a GSM multiparty call.

```
VOID WTAIa_GSMSeparate (UINT8 objectId, const WCHAR
*wtaChannelId, INT8 callHandle)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`callHandle`	The call-handle of the call to sepatate.

GSMSeparateResponse

Call this function to return the result of the *WTAIa_GSMSeparate* function.

```
VOID WTAIc_GSMSeparateResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	WTA_SUCCESS or WTA_INVALID.

GSMSendUSSD

WTAI Lib/Func ID: 518.6

This function is called to make the mobile phone send a USSD message.

```
VOID WTAIa_GSMSendUSSD (UINT8 objectId, const WCHAR
*wtaChannelId, const WCHAR *message, const WCHAR
*codingScheme, INT8 type, const CHAR
*transactionId)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`message`	The string to be sent. This may include any of the USSD characters permitted by GSM 02.90. The string is deleted when the function returns.
`codingScheme`	The argument dataCodingScheme can take values according to GSM 02.90. The string is deleted when the function returns.
`type`	The parameter type is the type of USSD operation. Valid values are provided as constants declared in aapiwta.h. See table below.
`transactionId`	This parameter (specified in GSM 04.07 §11), should in the case of a response to a network initiated USSD (i.e. a type 1 or 2 message), be set to the value of the transaction id of the corresponding network initiated message. In case of a type 0 message, the parameter id should be set to -1. The string is deleted when the function returns.

The *type* parameter can take the following values:

Value	Constant name	Description
0	WTA_GSM_SEND_REQUEST	The ProcessUnstructuredSS-Request operation.
1	WTA_GSM_SEND_REQUEST_RESULT	Result to an UnstructuredSS-Request operation.
2	WTA_GSM_SEND_NOTIFY_RESULT	Result to an UnstructuredSS-Notify operation.

GSMSendUSSDResponse

Call this function to return the result of the *WTAIa_GSMSendUSSD* function.

```
VOID WTAIc_GSMSendUSSDResponse (UINT8 objectId,
INT8 result)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	Transaction id (specified in GSM 04.07 §11) of the USSD message, or error code from table below.

The *result* parameter can take the following values:

Value	Constant name	Description
-100	WTA_GSM_USSD_IN_PROGRESS	USSD dialogue in progress
-101	WTA_GSM_ILLEGAL_CHAR	illegal characters or too many characters
-106	WTA_GSM_NO_NETWORK	network in not available
-128	WTA_INVALID	function failure, invocation error

GSMNetinfo

WTAI Lib/Func ID: 518.7

This function is called to provide the current network information of the GSM terminal.

```
VOID WTAIa_GSMNetinfo (UINT8 objectId, const WCHAR
*wtaChannelId, INT8 type)
```

`objectId`	This parameter is passed on to the corresponding connector function.
`wtaChannelId`	Identifies the entity calling this function. Should be used to check user permission. The string is deleted after the function has returned.
`type`	Amount of network measurement results to return, see table below.

The *type* parameter can take the following values:

Value	Constant name	Description
0	WTA_GSM_NETINFO_NO	Return no network measurement results.
1	WTA_GSM_NETINFO_SIX_BEST	Return network measurment results for the six “best” surrounding nodes.

2	WTA_GSM_NETINFO_ALL	Return network measurment results for all possible surrounding nodes.

GSMNetinfoResponse

Call this function to return the result of the *WTAIa_GSMNetinfo* function.

```
VOID WTAIc_GSMNetinfoResponse (UINT8 objectId, INT8
result, const WCHAR *location)
```

`objectId`	The object id received by the corresponding adapter function.
`result`	This parameter is set to WTA_SUCCESS if location information is available, or WTA_INVALID if not available.
`location`	GSM location information, or empty string indicating information not available.

The *location* parameter contains a value in a byte string with the eight octets of GSM location information in hexadecimal representation:

Octets	**Content coded as in GSM 04.08**
1 – 3	Mobile Country & Network Codes (MCC & MNC)
4 – 5	Location Area Code (LAC)
6 – 7	Cell Identity Value (Cell ID)
8	Timing Advance
9	Status
10 - 25	Network Measurement Results

If the AUS WAP Browser internal memory allocator (see USE_WIP_MALLOC) is used, the string must be created with wip_malloc. Otherwise, it should be created with malloc. The AUS WAP Browser frees it with either wip_free or free.

11.10 Services

The repository stores the WTAI services, which contains resources (WML and WMLS files). It stores also bindings from WTAI events to the services.

11.10.1 Installation of services

confirmInstallation

Called by the AUS WAP Browser when a service that has been downloaded, and is about to be installed.

```
VOID WTAa_confirmInstallation (INT8 installId,
const WCHAR *wtaChannelId, const WCHAR *title,
const WCHAR *abstract)
```

installId	The id of the installation process.
wtaChannelId	The id of the channel. The string is deleted after the function has returned.
title	The title of the service about to be installed. The string is deleted when the function returns.
abstract	A description of the service about to be installed. The string is deleted when the function returns.

confirmInstallation

This function is used to confirm or decline an installation procedure initiated by the function WTAa_ confirmInstallation.

```
VOID WTAc_confirmInstallation (INT8 installId, BOOL
install)
```

installId	The id of the installation process that was provided from the function WTAa_confirmInstallation.
install	Set to TRUE if the installation shall proceed, FALSE otherwise.

retryGetInstallationResult

Called by the AUS WAP Browser when the success URL not could be retrieved. The Connector function WTAc_retryGetInstallationResult is to be used for the answer.

```
VOID WTAa_retryGetInstallationResult (INT8
installId)
```

installId	The id of the installation process that was provided from the function WTAa_confirmInstallation.

retryGetInstallationResult

This function is used to confirm or decline if the success result should be opened again (initiated by the function WTAa_retryGetInstallationResult.

```
VOID WTAc_retryGetInstallationResult (INT8
installId, BOOL retry)
```

installId	The id of the installation process that was provided from the

	function WTAa_confirmInstallation.
`retry`	Set to TRUE if the success URL should be loaded again or FALSE if not.

showInstallationResult

Called by the AUS WAP Browser when the installation is finished and the result may be viewed. The answer is to be given by a call of the Connector function WTAc_showInstallationResult.

```
VOID WTAa_showInstallationResult (INT8 installId,
const CHAR *url)
```

`installId`	The id of the installation process that was provided from the function WTAa_confirmInstallation.
`url`	This argument holds the URL for the installation result. It can be used to view the result after the function WTAc_showInstallationResult has been called with a value of FALSE. The string is deleted when the function returns.

showInstallationResult

This function is used to confirm or decline if the result of the installation procedure shall be viewed. This does not affect the installation procedure, which has been finished or failed.

```
VOID WTAc_showInstallationResult (INT8 installId,
BOOL show)
```

`installId`	The id of the installation process that was provided from the function WTAa_confirmInstallation.
`show`	The argument is set to TRUE if the result shall be displayed at once, otherwise set to FALSE. If the value is set to FALSE, the AUS WAP Browser discards the result content. It can be retrieved again by using the URL provided in the Connector function WTAa_showInstallationResult.

abortInstallation

This function aborts an ongoing installation in the AUS WAP Browser.

```
VOID WTAc_abortInstallation (INT8 installId)
```

`installId`	The parameter installId is provided with the Adapter function WTAc_confirmInstallation.

11.10.2 Accessing services

getServices

This function is called when the services in the repository are needed.

```
VOID WTAc_getServices (VOID)
```

services

This function is called after a WTAc_getServices Connector has been called.

```
VOID WTAa_services (const ServiceType * const
*services)
```

services	This parameter holds on success a pointer to a NULL terminated list of pointers to structures containing information about the services installed in the repository. If the execution not was successful, this argument is set to NULL. The list and its content are deleted after the function returns.

The ServiceType struct is defined in aapiwta.h and contains the following variables:

Type	Name	Description
WCHAR *	title	The name of the service.
WCHAR *	abstract	A description of the service.
WCHAR *	wtaChannelId	The id of the channel.

deleteService

This function is called when a specific service in the repository is to be deleted.

```
VOID WTAc_deleteService (const WCHAR *wtaChannelId)
```

eventId	The id of the channel. The AUS WAP Browser copies the string.

deleteService

This function is called to indicate a deletion of a service in the repository.

```
VOID WTAa_deleteService (const WCHAR *wtaChannelId)
```

eventId	The id of the channel. The AUS WAP Browser copies the string.

executeService

This function is called when a specific service in the repository is to be started. Note that only non-event services can be executed with this function.

```
VOID WTAc_executeService (const WCHAR *eventId)
```

`eventId`	The parameter eventId is the event id of the service. The AUS WAP Browser copies the string.

terminateService

This function is used when an executing service is to be stopped.

```
VOID WTAIc_terminateService (VOID)
```

clearServices

This function removes all services from the repository and aborts all ongoing installations.

```
VOID WTAc_clearServices (VOID)
```

11.10.3 Events

processedByAService

This function is called after a WTAI event Connector function has been called. If the event not was processed by a service, the mobile phone should act as it does when the WTA browser is inactive.

```
VOID WTAa_processedByAService (BOOL processed)
```

`processed`	If the repository contains a service, which has processed the previously received event, this argument is set to TRUE. If not, it is set to FALSE.

12 Push API

Push techniques enables the content server to send data to the client, without a prior request. Such notifications include for example new e-mails, changes in stock prices, news headlines, advertising, reminders of low pre-paid balance, etc. Two content types are handled by this API in order to support these techniques:

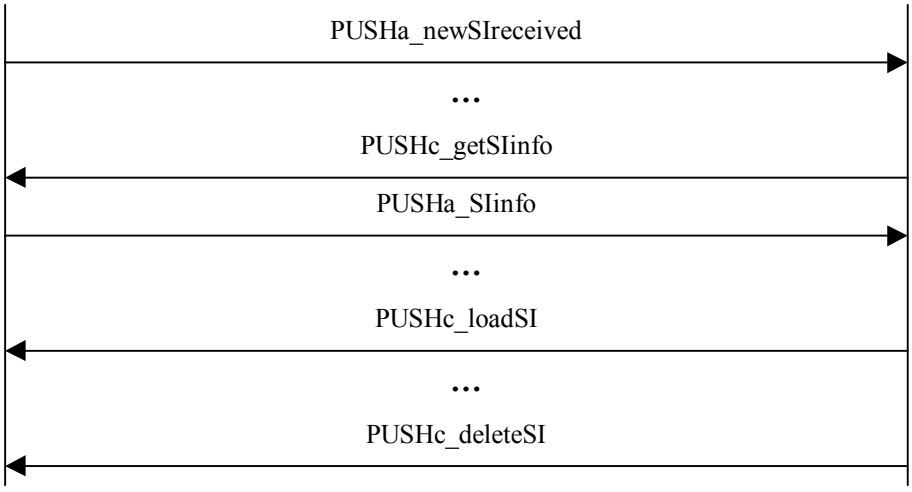
- Service Indication (SI), an indication of a new resource (WML or WMLS content) that can be downloaded and opened, if the user wants.
- Service Loading (SL), an indication of a new resource (WML or WMLS content) that is to be downloaded and opened. The user is not to be notified.

To be able to support these push techniques in a controlled and secure manner the configuration variable configPUSH_SECURITY_LEVEL are to be set to an, for the WAP application, appropriate value.

12.1 Handle Service Indications

Service indications are handled asynchronously in the WAP application. The AUS WAP Browser notifies the WAP application that a new SI has been received and that the end-user may respond upon that, when appropriate. The WAP application must provide the end-user a mean to select a SI from a list of all received SIs and then perform actions on it.

Typical Service Indication scenario



newSIreceived

This function is to be called when a new SI is received. When a SI is received, the priority attribute indicates its priority. According to the specification, it is recommended that the client can associate different levels of priority with various logical indicators. This function is not connected to a specific user agent.

```
VOID PUSHa_newSIreceived (INT16 id, UINT8 priority)
```

`id`	An id number that identifies the SI. The id shall be used in the response upon this indication. All id numbers are positive.
`priority`	The priority parameter may be used by the WAP application to map the SI to different logical indicators. The following constants may be used to interpret the priority: PUSH_LOW_PRIO, PUSH_MEDIUM_PRIO and PUSH_HIGH_PRIO.

loadSI

This function is called by the WAP application in order to load and open the service indicated by the SI.

```
VOID PUSHc_loadSI (UINT8 objectId, INT16 id)
```

`objectId`	The objectId parameter indicates the user agent in which the pushed content is to be presented if load of service is requested. The id shall be a positive integer.
`id`	An id number that identifies the SI to be loaded. The id has previously been retrieved with the function PUSHa_newSIreceived.

deleteSI

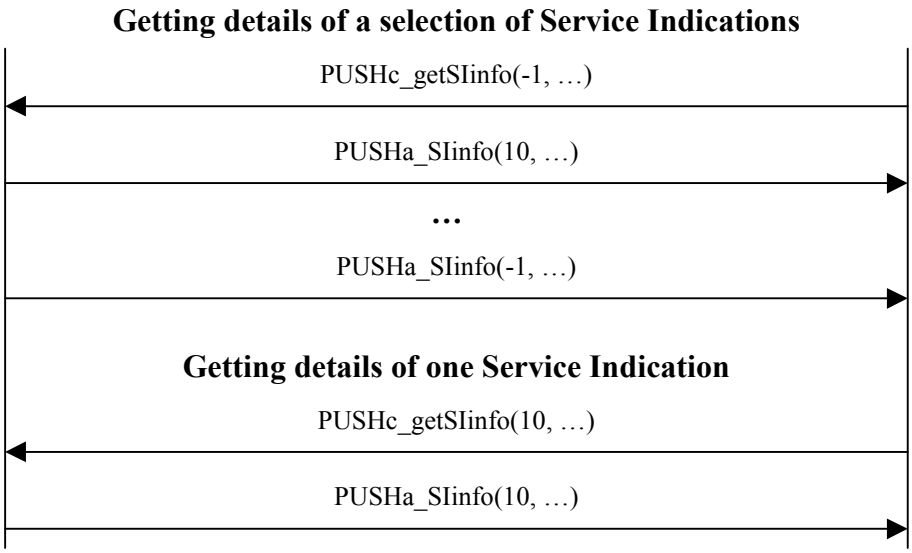
This function deletes a previously received SI. If the delete operation fails, the CLNTa_error function is called indicating ERR_WAE_PUSH_DELETE_FAILED.

```
VOID PUSHc_deleteSI (INT16 id, UINT8 selection)
```

`id`	An id number that identifies the SI to be deleted, if it is one individual SI that is the target of the operation. The id has previously been retrieved with the function PUSHa_newSIreceived. If the operation not targets an individual SI, the id shall be set to –1. The selection argument must in this case be set to any of the constants below.
`selection`	If id is equal to –1, this parameter gives the selection of Service Indications that shall be deleted. The following constants may be used: PUSH_DEL_ALL, PUSH_DEL_EXP, PUSH_DEL_NON_EXP, PUSH_DEL_LOADED and PUSH_DEL_NON_LOADED

12.2 Get details of Service Indications

This section describes the functionality needed to support WAP application functionality that provides the end-user with information about the individual SIs.



getSIinfo

The WAP application calls this function in order to get information about one or several SIs. The AUS WAP Browser will then return the information for each SI, one by one using the function PUSHa_SIinfo.

```
VOID PUSHc_getSIinfo (INT16 id, UINT8 selection)
```

`id`	An id number that identifies the SI, if it is one individual SI that is the target of the operation. The id has previously been retrieved with the function PUSHa_newSIreceived. If the operation not targets an individual SI, the id shall be set to –1. The selection argument must in this case be set to any of the constants below.
`selection`	If id is equal to –1, this parameter gives the selection of Service Indications, for which the details shall be returned. The following constants may be used: PUSH_SHOW_ALL, PUSH_SHOW_EXP, PUSH_SHOW_NON_EXP, PUSH_SHOW_LOADED and PUSH_SHOW_NON_LOADED

SIinfo

This function provides information about a SI. This is the reply to the PUSHc_getSIinfo function call.

```
VOID PUSHa_SIinfo (INT16 id, UINT8 status, UINT32
created, UINT32 expires, const WCHAR *message, BOOL
```

```
expired, const CHAR *url, UINT8 priority, const
CHAR *initURL)
```

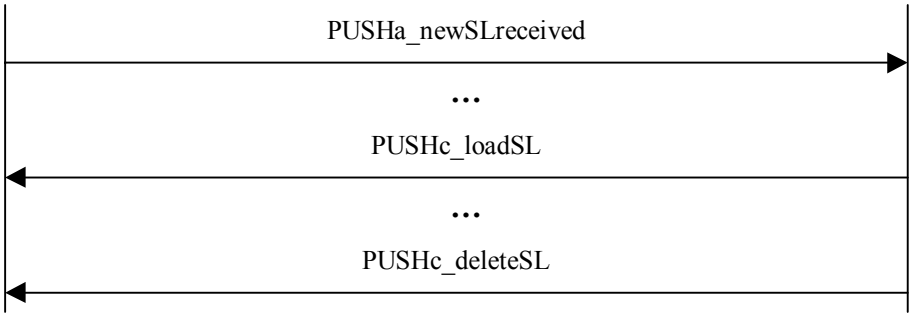
`id`	An id number that identifies the SI. The id has previously been retrieved with the function PUSHa_newSIreceived. If the function PUSHc_getSIinfo was called with the id argument set to –1, i.e. a selection of SIs are to be iterated. The id in this function will be set to –1 when there are no more SI in the selection.
`status`	This parameter indicates the status of the SI. The following constants may be used: PUSH_STATUS_NON_LOADED and PUSH_STATUS_LOADED.
`created`	This parameter gives the date and time of when the SI was created, represented in the UTC format (see example in [SI]). If this data is not provided in the push message, this parameter will be set to 0.
`expires`	This parameter gives the date and time of when the SI expires, represented in the UTC format (see example in [SI]). This parameter needs not to be read if the expired parameter is set to FALSE. If this data is not provided in the push message, this parameter will be set to 0.
`message`	A description of the received SI that shall be presented to the end-user. The AUS WAP Browser is responsible for the de-allocation of the memory. If no message is provided in the push message, this parameter will be set to NULL.
`expired`	This parameter indicates TRUE if the SI has expired and FALSE otherwise.
`url`	The URL that points to the service to retrieve. The AUS WAP Browser is responsible for the de-allocation of the memory. If the URL is not provided in the push message (e.g. if it is a notification message), this parameter will be set to NULL.
`priority`	The priority parameter may be used by the WAP application to map the SI to different logical indicators. The following constants may be used to interpret the priority: PUSH_LOW_PRIO, PUSH_MEDIUM_PRIO and PUSH_HIGH_PRIO.
`initURL`	This URL identifies the push initiator. This variable is NULL if the initiators URL is not provided in the push message. The AUS WAP Browser is responsible for the de-allocation of the memory.

12.3 Handling Service Loadings

The Service Loading (SL) content type is similar to the SI content type. The main difference is that the SL is designed, depending on its priority, to be handled

without questioning the user. Of that reason, there is no need for an API to this functionality in the AUS WAP Browser. However, the API may be desirable to have anyhow. The configuration variable cfg_wae_push_notify_sl is to be set in order to enable the functions in this section.

Typical Service Loading scenario



newSLreceived

This function is called when a new SL is received (and when the configuration variable cfg_wae_push_notify_sl is set to the value 1). This function is not connected to a specific user agent.

```
VOID PUSHa_newSLreceived (INT16 id, UINT8 priority)
```

`id`	An id number that identifies the SL. The id shall be used in the response upon this indication. All id numbers are positive.
`priority`	The priority parameter may be used by the WAP application to map the SL to different logical indicators. The constants PUSH_LOW_PRIO, PUSH_HIGH_PRIO and PUSH_CACHE_PRIO can be used to determine the priority.

loadSL

This function is called by the WAP application in order to load the service indicated by the Service Loading. Note that this function can be called only when the configuration variable cfg_wae_push_notify_sl is set to the value 1.

```
VOID PUSHc_loadSL (UINT8 objectId, INT16 id)
```

`objectId`	The objectId parameter indicates the user agent in which the pushed content is to be presented. The id shall be a positive integer.
`id`	An id number that identifies the SL to be loaded. The id has previously been retrieved with the function PUSHa_newSLreceived.

deleteSL

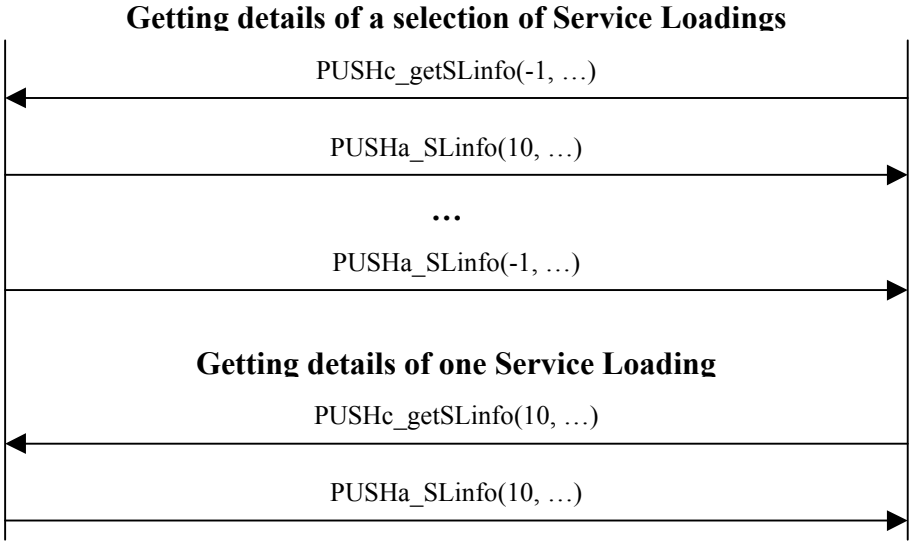
This function deletes a previously received SL. If the delete operation fails, the CLNTa_error function is called indicating ERR_WAE_PUSH_DELETE_FAILED. Note that this function can be called only when the configuration variable cfg_wae_push_notify_sl is set to the value 1.

```
VOID PUSHc_deleteSL (INT16 id, UINT8 selection)
```

`id`	An id number that identifies the SL to be deleted. The id has previously been retrieved with the function PUSHa_newSLreceived.
`selection`	If the id is equal to –1, this parameter gives the selection of Service Loadings that shall be deleted. The following constants may be used: PUSH_DEL_ALL, PUSH_DEL_LOADED and PUSH_DEL_NON_LOADED

12.4 Getting details of Service Loadings

This section describes the functionality needed to support WAP application functionality that provides the end-user with information about the individual SLs.



getSLinfo

The WAP application calls this function in order to get information about one or several SLs. The AUS WAP Browser will then return the information for each SL, one by one using the function PUSHa_SLinfo.

```
VOID PUSHc_getSLinfo (INT16 id, UINT8 selection)
```

`id`	An id number that identifies the SL, if it is one individual SL that is the target of the operation. The id has previously been retrieved with the function PUSHa_newSLreceived.

	If the operation not targets an individual SL, the id shall be set to –1. The selection argument must in this case be set to any of the constants below.
`selection`	If id is equal to –1, this parameter gives the selection of Service Loadings, for which the details shall be returned. The following constants may be used: PUSH_SHOW_ALL, PUSH_SHOW_LOADED and PUSH_SHOW_NON_LOADED

SLinfo

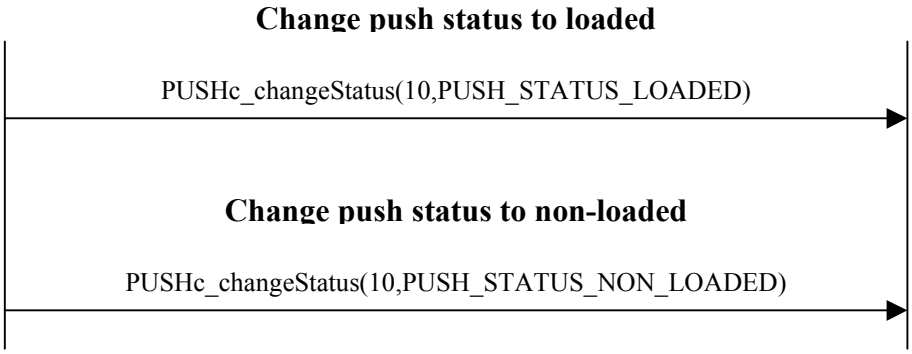
This function provides information about a SL. This is the reply to the PUSHc_getSLinfo function call.

```
VOID PUSHa_SLinfo (INT16 id, UINT8 status, const
CHAR *url, UINT8 priority, const CHAR *initURL)
```

`id`	An id number that identifies the SL. The id has previously been retrieved with the function PUSHa_newSLreceived. If the function PUSHc_getSLinfo was called with the id argument set to –1, a selection of SLs are to be iterated. The id in this function will be set to –1 when there are no more SL in the selection.
`status`	This parameter indicates the status of the SL. The following constants may be used: PUSH_STATUS_NON_LOADED and PUSH_STATUS_LOADED.
`url`	The URL that points to the service to retrieve. The AUS WAP Browser is responsible for the de-allocation of the memory.
`priority`	The priority parameter may be used by the WAP application to map the SL to different logical indicators. The constants PUSH_LOW_PRIO, PUSH_HIGH_PRIO and PUSH_CACHE_PRIO can be used to determine the priority.
`initURL`	This URL identifies the push initiator. This variable is NULL if the initiators URL is not provided in the push message. The AUS WAP Browser is responsible for the de-allocation of the memory.

12.5 Changing Status

changeStatus



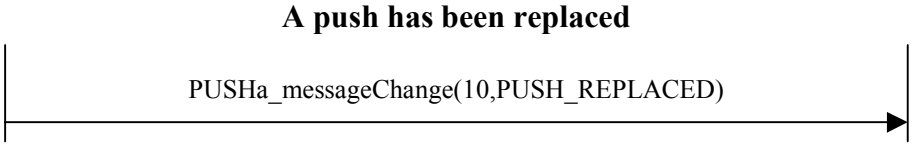
This function provides a mean to change the push status (for both SI and SL) to PUSH_STATUS_LOADED or PUSH_STATUS_UNLOADED, without loading the push. It can be used to allow the user to administrate the push inbox.

```
VOID PUSHc_changeStatus (INT16 id, UINT8 status)
```

`id`	An id number that identifies the SI or SL, if it is one individual that is the target of the operation. The id has previously been retrieved with the function PUSHa_newSIreceived or PUSHa_newSLreceived. If the operation not targets an individual SL or SI, the id shall be set to –1. In this case, all pushes will be set to the status indicated by the status parameter.
`status`	This parameter indicates the status of the SI or SL. The following constants may be used: PUSH_STATUS_NON_LOADED and PUSH_STATUS_LOADED.

12.6 Message Change

messageChange



This adaptor function is called when a push message has been replaced or deleted by an incoming push message (and when the configuration variable cfg_wae_push_notify_change is set to the value 1). It can be used to synchronise the content in a push inbox with for instance a messaging application (if push messages are to be shown there as well).

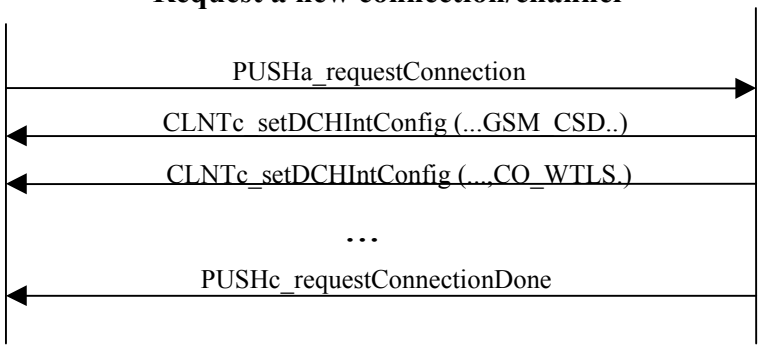
```
VOID PUSHa_messageChange (INT16 id, UINT8 change)
```

`id`	An id number that identifies the SI or SL, if it is one individual that is the target of the operation. The id has previously been retrieved with the function PUSHa_newSIreceived or PUSHa_newSLreceived.
`change`	This parameter indicates the change that has occurred. The following constants may be used: PUSH_REPLACED and PUSH_DELETED.

12.7 Network connections

requestConnection

Request a new connection/channel



When a PUSH user agent recieves a SIA, a data channel needs to be defined in order to set-up the PUSH session. The WAP application should configure the data channel (using CLNT_setDCHIntConfig and CLNT_setDCHStrConfig) and should respond with PUSHc_requestConnectionDone (see figure above).

```
VOID PUSHa_requestConnection (UINT8 siaId, UINT16
stackMode, UINT8 accessType, const CHAR *address,
UINT8 addresslen)
```

`siaId`	The ID of the PUSH SIA that requires a connection to be setup.
`stackMode`	The configuration variable configSTACK_MODE, of the created channel, shall be configured with this value. If this attribute is equal to MODE_ANY, the WAP application must decide what stack mode to use. All possible constants declared in capiclnt.h.
`accessType`	The configuration variable configACCESS_TYPE, of the created channel, shall be configured with this value. If this attribute is equal to BEARER_ANY, the WAP application must decide what access type to use. All possible constants declared in capiclnt.h
`address`	The gateway address. The data channel, to be used for the PUSH session, shall be configured with this

	gateway address. The string is not zero-terminated. The AUS WAP Browser is responsible for the de-allocation of the memory.
`addressLen`	The gateway address string length.

requestConnectionDone

The function will be called from the WAP application as a response to PUSHa_ requestConnection when the channel has been configured. The PUSH user agent may now set-up the PUSH session.

```
VOID PUSHc_requestConnectionDone (UINT8 siaID,
UINT8 channelId, BOOL success)
```

`siaId`	The ID of the PUSH SIA that requires a connection to be setup.
`channelId`	The id of the channel to be used for this PUSH session by the PUSH user agent.
`success`	Indicates whether the configuration was succesfully performed or not (TRUE indicates success).

12.8 Constants

The following constants are, or can be, used together with the functions in this API. They are all defined in aapipush.h.

Value	Constant name	Description
1	PUSH_LOW_PRIO	Handle the SI or SL with low priority
2	PUSH_MEDIUM_PRIO	Handle the SI with medium priority
3	PUSH_HIGH_PRIO	Handle the SI or SL with high priority
4	PUSH_CACHE_PRIO	Handle the SL with cache priority
4	PUSH_STATUS_ NON_LOADED	Indicates that the SI or SL has not been loaded by the end-user
5	PUSH_STATUS_ LOADED	Indicates that the SI or SL has been loaded at least once by the end-user
1	PUSH_DELETED	Indicate a push has been deleted
2	PUSH_REPLACED	Inidicate a push has been replaced
1	PUSH_DEL_ALL	Deletes all SIs
2	PUSH_DEL_EXP	Deletes only the expired SIs
3	PUSH_DEL_NON_EXP	Deletes the non-expired SIs
4	PUSH_DEL_LOADED	Deletes the loaded SIs

5	PUSH_DEL_NON_ LOADED	Deletes the non-loaded SIs
1	PUSH_SHOW_ALL	Returns all SIs
2	PUSH_SHOW_EXP	Returns only the expired SIs
3	PUSH_SHOW_NON_EXP	Returns the non-expired SIs
4	PUSH_SHOW_LOADED	Returns the loaded SIs
5	PUSH_SHOW_NON_ LOADED	Returns the non-loaded SIs

13 Memory API

The Memory API defines functionality to access four 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 AUS WAP Browser considers the cache non-existent.

The second 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 include full WTA.

Third, 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 temporarily to be viewed later. This storage type is only necessary in configurations of the AUS WAP Browser that include Push functionality.

The fourth kind of memory is the database where the AUS WAP Browser stores runtime configuration variables.

13.1 Memory or File based operation

The functions in this API are designed for a memory-based storage model. If the persistant memory in the target platform has a file based model, the File API could be used in co-operation with this API. A configuration variable in confvars.h has to be set in order to switch the AUS WAP Browser to use the File API, as well. Functions in this API that not needs to be implemented when the File API is present are:

- MEMa_readCache
- MEMa_writeCache
- MEMa_readPushRepository
- MEMa_writePushRepository
- MEMa_readServiceRepository
- MEMa_writeServiceRepository
- MEMa_readDatabase
- MEMa_writeDatabase

The operation of these functions is taken care of by the File API functions instead.

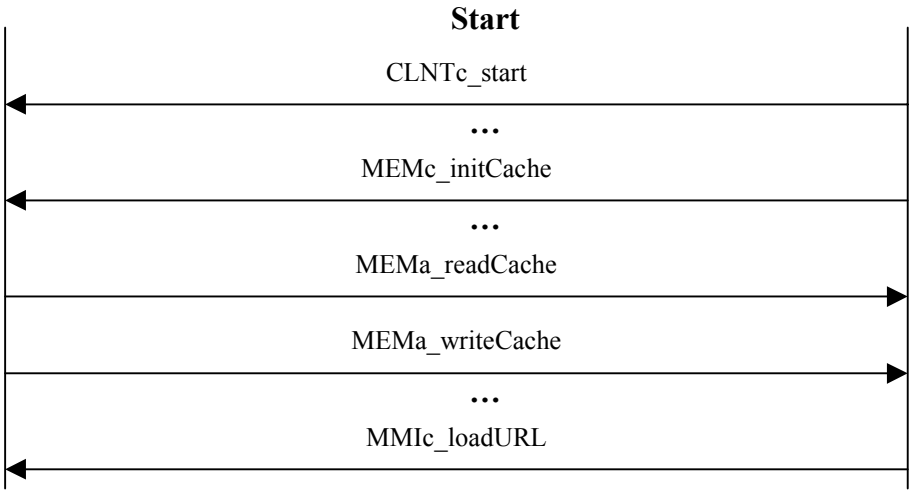
13.2 Initialising or resizing the cache

The AUS WAP Browser must be notified of how much cache memory has been made available. This should take place:

- after the AUS WAP Browser has been started (CLNTc_start)
- after the cache memory is ready to be accessed

- before any content is downloaded (MMIc_loadURL)

The AUS WAP Browser also needs this notification during runtime when the cache has been resized. Read more about this below.



initCache

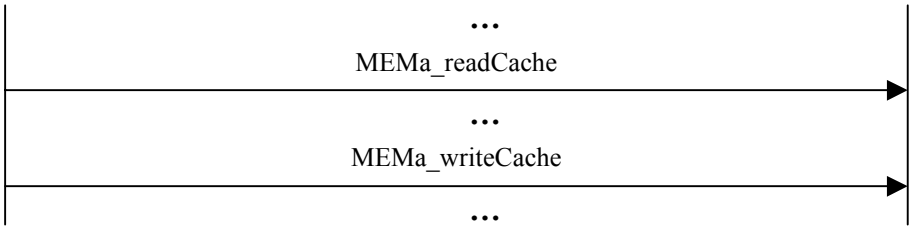
The WAP client calls this function when the cache memory has been restored (when the AUS WAP Browser has started or when the cache has been resized). If the cache memory is new, the four leading bytes of the memory must be set to zero. If the four leading bytes are not set to zero, the AUS WAP Browser 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 (UINT32 cacheSize)
```

`cacheSize`	The available size of cache memory is given in this argument. If this size is less than the size indicated in the last call to MEMc_prepareCache, then arbitrary content of the cache may be lost.

13.3 Accessing the cached content repository

The cached content repository is accessed with dedicated functions. There is 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 buffer where the bytes read should be stored.

The function returns the actual number of bytes read.

writeCache

Write a number of bytes to the cache content repository.

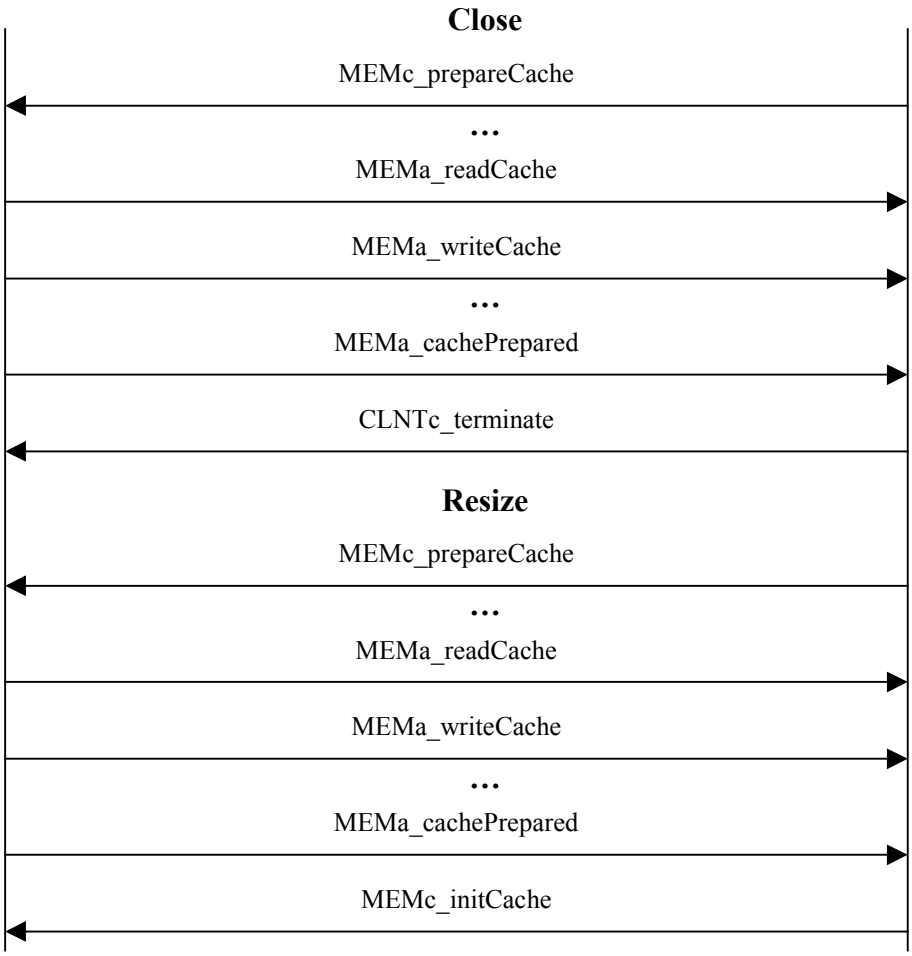
```
UINT32 MEMa_writeCache (UINT32 pos, UINT32 size,
const CHAR *buffer)
```

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

The function returns the actual number of bytes written.

13.4 Closing or resizing 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. After a call to this function the cache will be in an uninitialised state, and cannot be accessed until a call to MEMc_initCache has been made.

```
VOID MEMc_prepareCache (UINT32
availablePersistentMemory)
```

`availablePersistentMemory`	The size in bytes available in the device to store the cache permanently. If this size is less than the size indicated in the last call to MEMc_initCache, then cache records will be discarded in a first-in-first-out order.

cachePrepared

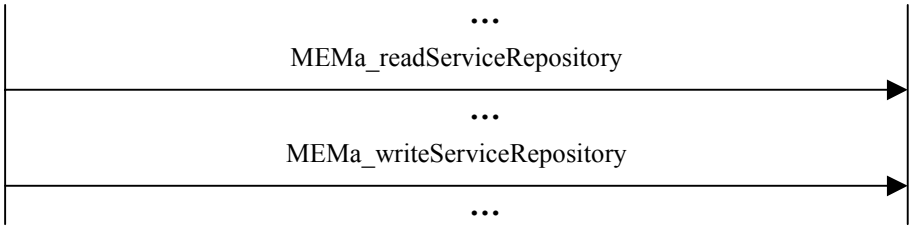
This function is called when the AUS WAP Browser has prepared the cache and the WAP client is allowed to finally store the cache. Note that a call to this function is always preceded by a call to MEMc_prepareCache.

```
VOID MEMa_cachePrepared (VOID)
```

13.5 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 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.



The size of the WTA services repository is set with the configuration variable REP_STORAGESIZE. It is by default set to the recommended size, 8 kBytes, which is based on the assumption that it has room for:

- 10 medium sized decks of 500 bytes each
- Two images of 1 kByte each
- Five WML scripts of 200 bytes each

This push storage must be on persistent memory, otherwise the stored data may be lost if the client is terminated, e.g., runs out of battery.

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 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)
```

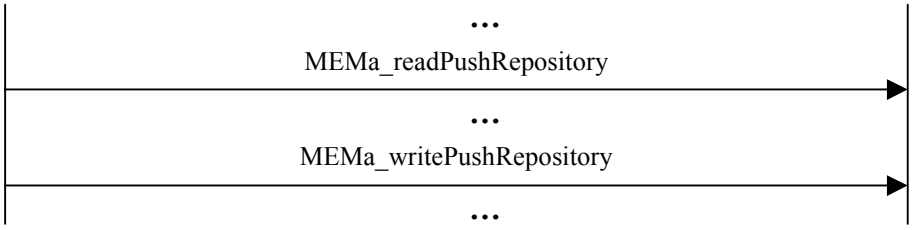
`pos`	The position in the WTAI services repository where to write.
`size`	The number of bytes to write.
`buffer`	The bytes to write.

The function returns the actual number bytes written.

13.6 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 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.



The size of the Push services repository is set with the configuration variable PUSH_STORAGESIZE. It is by default set to the recommended size, 4.5 kBytes, which is based on the assumption that it has room for:

- Five medium sized Service Indications of 250 bytes each
- 10 postponed medium sized Service Indications of 250 bytes each
- Five medium sized Service Loadings of 150 bytes each

This push storage must be on persistent memory, otherwise the stored data may be lost if the client is terminated, e.g., runs out of battery.

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 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.

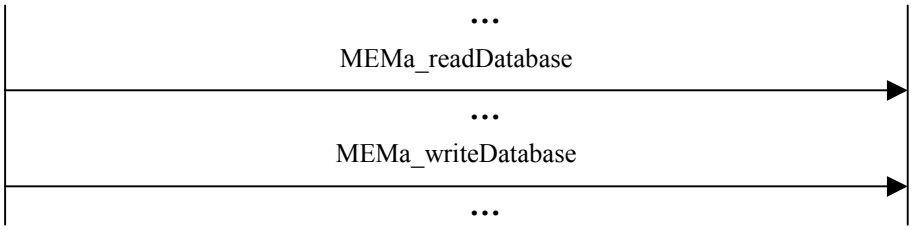
```
UINT32 MEMa_writePushRepository (UINT32 pos, UINT32
size, const CHAR *buffer)
```

`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.

13.7 Accessing the database of runtime data

There are functions to access the database 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.



The size of the database repository is set with the configuration variable DATABASE_STORAGESIZE. It is by default set to the recommended size, 4.0 kBytes, which is based on the assumption that it has room for:

- 10 host authorization records of 150 bytes each
- 10 proxy authorization records of 150 bytes each

This storage can be put on persistent memory. In that case, the stored data will not be lost if the client is terminated, e.g., if the host device runs out of battery.

readDatabase

Read a number of bytes from the database content repository.

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

`pos`	The position in the database repository from where to read.
`size`	The number of bytes to read.
`buffer`	A buffer to store the read bytes in.

The function returns the actual number bytes read.

writeDatabase

Write a number of bytes to the database content repository.

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

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

The function returns the actual number bytes written.

14 File API

The Memory API defines functionality for management of Cache, Push Repository, Configuration Database and the WTA Repository. By default, all of these use a memory-based storage model. Setting a configuration variable in confvars.h, switches the AUS WAP Browser to use the functions in this API, in co-operation with the functions in the Memory API.

An integrator that considers using the File API should bear in mind that all functions are adaptor functions only. This means that *if* a file access requires long processing time, this *may* interfere with the scheduling in the system. In that case, the memory-based storage may be preferable.

14.1 Platform requirements

File systems may have very different properties on different platforms. For example, there may be different restrictions on how filenames should look: is there a maximum length, do file names have an extension, are lower-case and upper-case filenames equal, etc. Also, some platforms have a hierarchical directory structure, whereas other have no such structure at all. The File API has been designed to minimise the set of requirements placed on the underlying system, in order to make it possible to implement on a wide variety of platforms.

Some important points to note are:

- *No file names*: the File API uses only integer IDs to identify files. It is up to the implementation to map these to whatever type of file names are used on the local platform.
- *No directory structure assumed*: the File API does not use any directories from the underlying system.
- *Limited required semantics on updates*: the File API includes a flush operation that should force any buffered data to be committed to the secondary storage. An implementation that has support for buffering and commit operations may use this feature. A file system that has write-through semantics will implement the flush operation as a no-op.

14.2 The API

In order for the adaptation to be able to separate files belonging to the Cache and the Push Repository, for example, each file is identified using a *category* as well as an *integer ID*. All files used by the Cache will have the same category value, and hence can be stored in a common directory by the adaptation. Files belonging to the Cache will have category 'C', files in the Push Repository will have category 'P', files in the WTA Repository will have category 'W' and files in the Configuration Database will have category 'D'.

The file IDs are unsigned integers, and the only requirement is that the file creation routine should allocate a presently unused value. One value is special, though, namely the value 0. This may be used by the AUS WAP Browser to store a special index file. Hence, a file with ID 0 is assumed to always exist within each category.

In the absence of a directory structure, the AUS WAP Browser will need some other way of finding out which files exist. This is accomplished by using the function FILEa_getFileIds, which retrieves a list of all current file IDs within a given category.

An adaptation might want to set an upper limit on the amount of secondary storage used by the cache, for example. This is handled by the usual routines (MEMc_initCache and MEMc_prepareCache, for example) and is not part of the File API.

create

This function creates a new file within the indicated category setting *file_id to a new unique integer identifying the file.

```
INT16 FILEa_create (CHAR category, UINT32 *file_id)
```

`category`	The category the new file should belong to.
`file_id`	An unsigned integer that uniquely identifies the new file.

Returns -1 in case of error, e.g., if the maximum number of files allowed within the indicated category has been exceeded. Otherwise, returns 0.

delete

Delete a file. If the file does not exist, this is a no-op, i.e. it can be implemented empty.

```
VOID FILEa_delete (CHAR category, UINT32 file_id)
```

`category`	The category the file belongs to.
`file_id`	An unsigned integer that uniquely identifies the file.

read

Fetch “count” bytes from position “pos” in the indicated file, and store in “buf”.

```
INT16 FILEa_read (CHAR category, UINT32 file_id,
void *buf, UINT32 pos, UINT32 count)
```

`category`	The category the file belongs to.
`file_id`	An unsigned integer that uniquely identifies the file.

buf	The location where the data should be placed.
pos	The position in the file where reading should start.
count	The number of bytes to read.

Returns -1 in case of error. Otherwise, returns the number of bytes actually read; this may be less than the requested amount in case the length of the file is less than pos+count.

write

Write “count” bytes at position “pos” in the indicated file, fetching the data from “buf”.

```
INT16 FILEa_write (CHAR category, UINT32 file_id,
void *buf, UINT32 pos, UINT32 count)
```

category	The category the file belongs to.
file_id	An unsigned integer that uniquely identifies the file.
buf	The location where the data should fetched.
pos	The position in the file where writing should start.
count	The number of bytes to written.

Returns -1 in case of error. Otherwise, returns the number of bytes actually written; this may be less than the requested amount in case the file system is full, for example.

getSize

Return the size in bytes of the indicated file.

```
INT16 FILEa_getSize (CHAR category, UINT32 file_id)
```

category	The category the file belongs to.
file_id	An unsigned integer that uniquely identifies the file.

Returns -1 in case of error, e.g., if the file does not exist.

flush

Force the buffered contents to be written to disk. An implementation that uses internal buffering should write all modified content to disk for the indicated file.

```
VOID FILEa_flush (CHAR category, UINT32 file_id)
```

category	The category the file belongs to.
file_id	An unsigned integer that uniquely identifies the file.

getFileIds

Retrieve a list of all file ids currently in use within the indicated category. If file_ids != NULL, then it points to a buffer large enough to hold at least maxcount file ids. If file_ids == NULL, then this routine should just return the number of file_ids.

```
INT16 FILEa_getFileIds (CHAR category, UINT32
*file_ids, UINT16 maxcount)
```

`category`	The category the file belongs to.
`file_ids`	An array where the file IDs can be stored.
`maxcount`	The maximum number of elements that the file_ids array can hold.

Returns the number of file ids, or -1 on error. Note that this number should always be > 0, since the file with file_id 0 is assumed to always exist, and should be included in this count.

15 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 and Connector functions. This chapter describes the cryptographic functions that the AUS WAP Browser makes use of.

15.1 Overview

15.1.1 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; any operation that involves the WIM might also be relatively slow. 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 3, i.e, it handles server authentication as well as client authentication, through the use of certificates. To verify a server certificate, the client needs one or more root certificates. The management of root and 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.

15.1.2 How the functions are used

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

Initialisation

```
CRYPTa_initialise
CRYPTa_sessionInit
```

Handshake, i.e., establishing a connection

```
CRYPTa_getMethods
CRYPTa_generateRandom
CRYPTa_peerLookup
CRYPTa_sessionFetch
CRYPTa_verifyCertificateChain
CRYPTa_keyExchange
CRYPTa_sessionInvalidate
CRYPTa_peerDeleteLinks
CRYPTa_getClientCertificate
CRYPTa_computeSignature
CRYPTa_hashInit
CRYPTa_hashUpdate
CRYPTa_hashFinal
CRYPTa_PRF
CRYPTa_sessionUpdate
CRYPTa_peerLinkToSession
CRYPTa_sessionActive
```

Computing encryption keys

```
CRYPTa_PRF
CRYPTa_hash
```

Encrypting and decrypting data

```
CRYPTa_encrypt
CRYPTa_decrypt
CRYPTa_hashInit
CRYPTa_hashUpdate
CRYPTa_hashFinal
```

Termination:

```
CRYPTa_sessionClose
CRYPTa_terminate
```

15.1.3 Function return values

Most functions in this crypto library return an integer value, either directly or in a parameter of its associated connector function. 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
CRV_MISSING_CERTIFICATE	19	A requested certificate is not available.
CRV_CONFIG_ERROR	20	A problem with the configuration of the cryptographic parameters prevents the initialisation from completing successfully.
CRV_NOT_FOUND	21	A peer or session that was requested was not found in the session store.

15.2 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.

terminate

Terminate the use of the crypto library. This function is called when WTLS is being shut down. Current handles that are held by the user (e.g., HashHandle and MasterSecretID) are no longer valid after calling this function.

```
INT16 CRYPTa_terminate (VOID)
```

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

getMethods

Retrieve lists of crypto parameters that the crypto library supports. This includes cipher methods, key exchange methods, and trusted certificates. The response should be delivered by a call to the connector function CRYPTc_getMethodsResponse. The parameter, "id", should be passed back in this response.

```
VOID CRYPTa_getMethods (UINT16 id)
```

`id`	A number that identifies this call. The id must be used in the response call.

getMethodsResponse

Deliver the result from the CRYPTa_getMethods function

```
VOID CRYPTc_getMethodsResponse (UINT16 id, INT16
result, const BYTE *cipherMethods, UINT16
cipherMethodLen, const BYTE *keyExchangeIds, UINT16
keyExchangeIdLen, const BYTE *trustedKeyIds, UINT16
trustedKeyIdLen)
```

`id`	The id number that was passed in the call to the corresponding adapter function, CRYPTa_getMethods.
`result`	The return value, as defined above.
`cipherMethods`	A byte-encoded sequence of elements of type CipherMethod, defined below. The caller may delete the string after the call.
`cipherMethodLen`	The number of bytes in cipherMethods.
`keyExchangeIds`	A byte-encoded sequence of elements of type KeyExchangeId, defined below. The value of this field will be included in the Client Hello message as client_key_ids (see WTLS specification). The caller may delete the string after the call.
`keyExchangeIdLen`	The number of bytes in keyExchangeIds.
`trustedKeyIds`	A byte-encoded sequence of elements of type KeyExchangeIds, defined below, representing the trusted certificates known to the client. The value of this field will be included in the Client Hello message as trusted_key_ids (see WTLS specification). The caller may delete the string after the call.

`trustedKeyIdLen`	The number of bytes in trustedKeyIds.

15.3 Bulk encryption algorithms

encrypt

Encrypt data.

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

`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 will 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. The AUS WAP Browser is responsible for the de-allocation of the data.

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

decrypt

Decrypt data.

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

`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. The AUS WAP Browser is responsible for the de-allocation of the data.

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

15.4 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)
```

`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. The AUS WAP Browser is responsible for the de-allocation of the data.

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.

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

`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. The AUS WAP Browser is responsible for the de-allocation of the data.

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

15.5 Key Exchange and Key Generation

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 *randval)
```

`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.
`randval`	32 bytes of random data, used to generate the master secret.

keyExchangeResponse

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

```
VOID CRYPTc_keyExchangeResponse (UINT16 id, INT16
result, UINT8 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 the public value.

PRF

Calculate the Pseudo-Random Function defined in section 11.3.2 in the WTLS spec. If the master secret is to be used as first parameter, then "secret" is NULL, and "masterSecretID" indicates which master secret to use. Otherwise, "secret" is provided explicitly. The response should be delivered by a call to the connector function CRYPTc_PRFResponse.

```
VOID CRYPTa_PRF (UINT16 id, HashAlgorithm method,
UINT8 masterSecretID, BYTE *secret, UINT16 seclen,
BYTE *label, BYTE *seed, UINT16 seedLen, UINT16
outputLen)
```

`id`	To identify this call to the adapter function.
`method`	The secure hash algorithm to use.
`masterSecretId`	If "secret" is NULL, the master secret should be used as first parameter to the PRF function. The 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.
`secret`	The secret value to be used as first parameter to the PRF function. The AUS WAP Browser is responsible for the de-allocation of the data.
`seclen`	The length of the secret value.
`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.
`outputLen`	The desired length of the output.

PRFResponse

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

```
VOID CRYPTc_PRFresponse (UINT16 id, INT16 result,
const BYTE *buf, UINT16 bufLen)
```

`id`	The identifier that was supplied in the call to CRYPTa_PRF.
`result`	One of the return codes, as specified above.
`buf`	The computed value. The caller may delete the string after the call.
`bufLen`	The length of the output data.

15.6 Certificates and Signatures

getClientCertificate

Retrieve a client certificate signed by one of the certificate authorities supplied. If the list of certificate authorities is empty, any certificate can be returned. The result should be delivered by a call to the connector function CRYPTc_getClientCertificateResponse.

```
VOID CRYPTa_getClientCertificate (UINT16 id, const
BYTE *buf, UINT16 bufLen)
```

`id`	To identify this call to the adapter function.
`buf`	A list of acceptable certificate authorities, as byte encoded KeyExchangeIds. See section 10.5.4 in the WTLS specification.
`bufLen`	The length of the data in buf.

getClientCertificateResponse

Deliver the result from the CRYPTa_getClientCertificate function. If no certificate was available, "result" should be set to CRV_MISSING_CERTIFICATE.

```
VOID CRYPTc_getClientCertificateResponse (UINT16
id, INT16 result, const BYTE *keyId, UINT16
keyIdLen, const BYTE *cert, UINT16 certLen)
```

`id`	The identifier that was supplied in the call to CRYPTa_getClientCertificate.
`result`	One of the return codes, as specified above.
`keyId`	A byte-encoded Identifier value that can be used to identify the client private key associated with the certificate(s) contained in the buffer "cert". This may be used in a subsequent call to CRYPTa_computeSignature.
`keyIdLen`	The length of keyId.
`cert`	One or more certificates, or NULL if no certificate was available.
`certLen`	The length of cert.

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.

computeSignature

Compute a digital signature. The result should be delivered by a call to the connector function CRYPTc_computeSignatureResponse.

```
VOID CRYPTa_computeSignature (UINT16 id, const BYTE
*keyId, UINT16 keyIdLen, const BYTE *buf, UINT16
bufLen)
```

id	An identifier that is to be passed back in the call to CRYPTc_computeSignatureResponse.
keyId	A byte-encoded Identifier. This latter value is either fetched from the Key Exchange Ids and passed back in CRYPTc_getMethodsResponse, or is the value passed back in CRYPTc_getClientCertificateResponse (if this function has been used). The keyId value indicates which key must be used for signing.

keyIdLen	The length of keyId.
buf	The data to be signed.
bufLen	The length of the data in buf.

computeSignatureResponse

Deliver the result from the CRYPTa_computeSignature function.

```
VOID CRYPTc_computeSignatureResponse (UINT16 id,
INT16 result, const BYTE *sig, UINT16 sigLen)
```

id	The identifier that was supplied in the call to CRYPTa_computeSignature.
result	One of the return codes, as specified above. In particular, if the requested private key was not available. The result should be set to CRV_MISSING_KEY.
sig	The computed digital signature.
sigLen	The length of the signature.

15.7 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.

15.8 Session cache

Each time a secure connection is established, the AUS WAP Browser saves information about the connection. This cached information is called the Session Cache, and is used to enable abbreviated handshakes, i.e., setting up a secure connection that re-uses a master secret previously agreed upon. Entries in the session cache should not become very old, but it may be of interest to have the session cache persist between different activations of the AUS WAP Browser.

The WIM specification has included provisions for having the session cache be stored on the WIM. This could be convenient, since the master secrets will also be on the WIM. An implementation that does not have access to a WIM might still want to have a session cache that is more persistent than just a RAM implementation. Hence, the needs for these functions to access a session cache that is external to the AUS WAP Browser.

The API below has been designed to allow a simple matching of routines to what is available on the WIM. Conceptually, the session cache consists of two different tables, the 'peers' table and the 'sessions' table. The sessions table contains records that are indexed by the corresponding master secret index. A record in the sessions store has at least the following fields:

Type	Name	Description
UINT8	session_options	A bitwise OR of zero or more of the values described below.
UINT8	session_id_length	The length of the Session ID (max 8).
BYTE[8]	session_id	The Session ID.
UINT8	mac_alg	The MAC algorithm.
UINT8	cipher_alg	The bulk encryption algorithm.
UINT8	compression_alg	The compression algorithm.
BYTE[4]	private_key_id	(Presently not used. See WIM spec, section 9.4.11.)
UINT32	creation_time	The time that this session record was created.

The peers table is used to match (address, portnumber)-pairs to sessions. The division into two tables of this kind allows for several different (address, portnumber)-pairs to refer to the same session slot. Each record in the peers table has at least the following fields:

Type	Name	Description
UINT8	session_idx	Index into the sessions table.
UINT16	portnum	The port number of the connection.
BYTE[18]	address	The Gateway address.

Besides the peers and sessions tables, it is also necessary to save the master secrets, of course. The master secret store should be maintained in parallell with the session store. This means that the same indexes or slot numbers are used in both the master secret store and the session store. New slots in the master secret store are selected by the key exchange routine. If no empty slots are available, an

old slot to be evicted should be selected using the following strategy. Here, an *active session* means that there is a current WTLS connection that uses that session. Select a slot where the session is

1. NON-resumable and NOT active
2. resumable and NOT active
3. NON-resumable and active
4. resumable and active

Preferably, the size of the store should be large enough that cases 3 and 4 are never used. This can be accomplished by giving the store one slot more than the maximum number of simultaneous WTLS connections. If there are two or more candidates from the same category, they should be ranked according to creation time. That is, the older slot will be reused first.

After the key exchange routine has delivered a new master secret index (mid) to WTLS, it is the responsibility of the AUS WAP Browser to call the routines CRYPTa_sessionInvalidate (mid) and CRYPTa_peerDeleteLinks (mid).

sessionInit

Initialize the session cache.

```
VOID CRYPTa_sessionInit (VOID)
```

sessionClose

Close the session cache.

```
VOID CRYPTa_sessionClose (VOID)
```

peerLookup

Find a peer with matching address and port number. If none exists, try matching just the address. The result, a master secret index, is returned in CRYPTc_peerLookupResponse.

```
VOID CRYPTa_peerLookup (UINT16 id, BYTE *address,
UINT16 addrlen, UINT16 portnum)
```

`id`	An identifier that is to be passed back in the call to CRYPTc_peerLookupResponse.
`address`	The Gateway address.
`addrlen`	The length of the address.

portnum	The port number of the connection.

peerLookupResponse

Deliver the result from the CRYPTa_peerLookup function.

```
VOID CRYPTc_peerLookupResponse (UINT16 id, INT16
result, UINT8 masterSecretIndex)
```

id	The identifier that was supplied in the call to CRYPTa_peerLookup.
result	One of the return codes, as specified above. In particular, if the requested peer was not found, the result should be set to CRV_NOT_FOUND.
masterSecretIndex	The index of the session slot that is linked to this peer, also the index of the corresponding master secret.

peerLinkToSession

Add a peer entry that links to the given master secret. If such an entry already exists, overwite it.

```
VOID CRYPTa_peerLinkToSession (BYTE *address,
UINT16 addrlen, UINT16 portnum, UINT8
masterSecretIndex)
```

address	The Gateway address.
addrlen	The length of the address.
portnum	The port number of the connection.
masterSecretIndex	The index of the corresponding session slot.

peerDeleteLinks

Delete all peer entries that link to the indicated master secret.

```
VOID CRYPTa_peerDeleteLinks (UINT8
masterSecretIndex)
```

masterSecretIndex	The index of the corresponding session slot.

sessionActive

If "isActive" not is equal to zero, then mark the indicated session slot as being "active". Otherwise, mark it as not active. A session slot that is "active" SHOULD NOT be reused. The AUS WAP Browser will keep a session marked as "active" as long as any WTLS connection that is based upon that session is up and running.

```
VOID CRYPTa_sessionActive (UINT8 masterSecretIndex,
UINT8 isActive)
```

`masterSecretIndex`	The index of the corresponding session slot.
`isActive`	Whether the active flag should be turned on (!= 0) or off (= 0).

sessionInvalidate

Mark a session entry as non-resumable.

```
VOID CRYPTa_sessionInvalidate (UINT8
masterSecretIndex)
```

`masterSecretIndex`	The index of the corresponding session slot.

sessionClear

Mark all entries in the session store as non-resumable.

```
VOID CRYPTa_sessionClear (VOID)
```

sessionFetch

Fetch the contents of the indicated session entry. The result is returned in CRYPTc_sessionFetchResponse.

```
VOID CRYPTa_sessionFetch (UINT16 id, UINT8
masterSecretIndex)
```

`id`	An identifier that is to be passed back in the call to CRYPTc_sessionFetchResponse.
`masterSecretIndex`	The index of the corresponding session slot.

sessionFetchResponse

Deliver the result from the CRYPTa_sessionFetch function.

```
VOID CRYPTc_sessionFetchResponse (UINT16 id, INT16
result, UINT8 sessionOptions, BYTE *sessionId,
```

```
UINT8 sessionIdLen, UINT8 macAlg, UINT8 cipherAlg,
UINT8 compressionAlg, BYTE *privateKeyId, UINT32
creationTime)
```

id	The identifier that was supplied in the call to CRYPTa_sessionFetch.
result	One of the return codes, as specified above. In particular, if the requested slot was not found (or empty), the result should be set to CRV_NOT_FOUND.
sessionOptions	The session options. See below.
sessionId	The session ID.
sessionIdLen	The length of the session ID.
macAlg	The MAC algorithm.
cipherAlg	The bulk encryption algorithm.
compressionAlg	The compression algorithm.
privateKeyId	The private Key Id (not used, see WIM spec. section 9.4.11).
creationTime	The time that this session record was created.

sessionUpdate

Store new values for a session entry.

```
VOID CRYPTa_sessionUpdate (UINT8 masterSecretIndex,
UINT8 sessionOptions, BYTE *sessionId, UINT8
sessionIdLen, UINT8 macAlg, UINT8 cipherAlg, UINT8
compressionAlg, BYTE *privateKeyId, UINT32
creationTime)
```

masterSecretIndex	The master secret index.
sessionOptions	The session options. See below.
sessionId	The session ID.
sessionIdLen	The length of the session ID.
macAlg	The MAC algorithm.
cipherAlg	The bulk encryption algorithm.
compressionAlg	The compression algorithm.
privateKeyId	The private Key Id (not used, see WIM spec. section 9.4.11).
creationTime	The time that this session record was created.

15.9 Types and constants

KeyExchangeParameters structure

This structure is used by the function CRYPTa_keyExchange.

Type	Name	Description
KeyExchangeSuite	keyExchangeSuite	A constant from the table KeyExchangeSuite
union	_u	C union of the structures KeyParam, Certificates and SecretKey defined below. The variant used depends on the value of the keyExchangeSuite field. If keyExchangeSuite is KEY_EXCH_SHARED_SECRET, then the SecretKey variant is used. If keyExchangeSuite is an anonymous method, then the KeyParam variant is used. If keyExchangeSuite is any other method different from NULL, then the Certificates variant is used.

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 CRYPTc_getMethodsResponse, 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 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
CIPHER_RC5_CBC_64	10
CIPHER_IDEA_CBC_64	11

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 and CRYPTa_decrypt use 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

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.

Session options

Constant	Value

SESSION_OPTIONS_RESUMABLE	0x80
SESSION_OPTIONS_SERVER_AUTH	0x20
SESSION_OPTIONS_CLIENT_AUTH	0x10

16 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 eight most significant bits) for network originated operations is operator specific. However, 1110 is used for WAP. The eight 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).

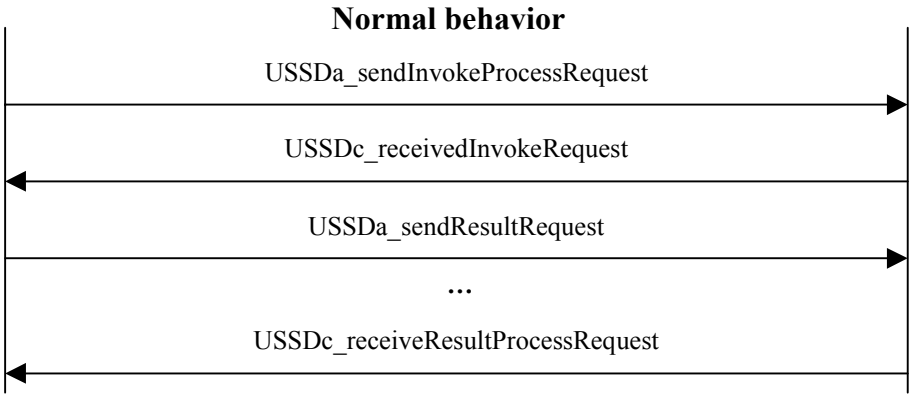
In WAP 1.2 is both address parts mandatory. In WAP 1.1 is the second part, the IP number or the MSISDN number that identifies the WAP gateway, optional. The AUS WAP Browser supports both modes. If a gateway, which is accessed with both addresses, fails, the AUS WAP Browser tryes again, this time using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

16.1 USSD dialogue scenarios

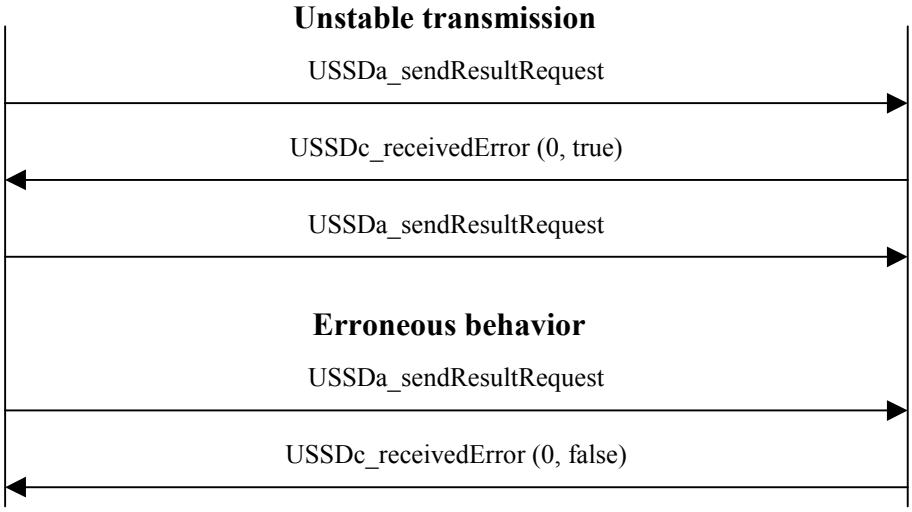
The functions in this API will be used differently dependent on whether the dialogue is mobile or network initiated.

16.1.1 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.

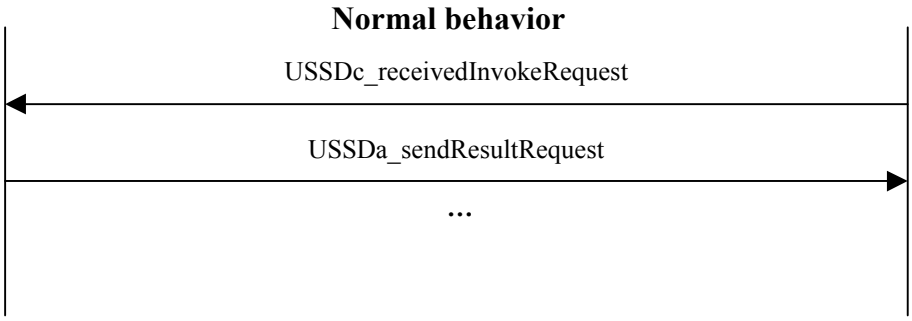


If an error occurs in the Host Device Environment (the mobile) or if the transmission is unstable, an error message must be propagated to the AUS WAP Browser. The dialogue is assumed terminated when the AUS WAP Browser receives the error message and isKept is false.



16.1.2 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.

16.2 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.

16.3 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)
```

17 SMS API

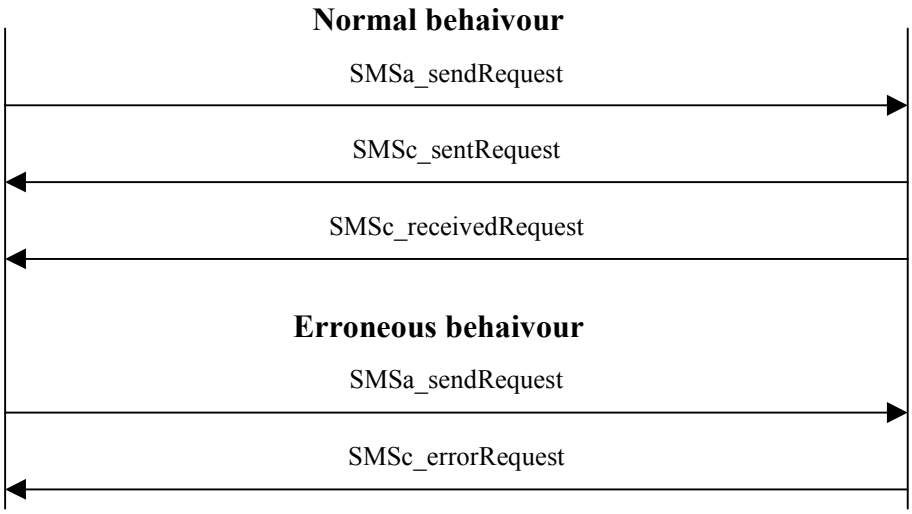
This API defines the interface between the AUS WAP Browser and the SMS service in the Host Device environment. The AUS WAP Browser is implemented to send and receive GSM phase 2 SMS with binary User Data Headers (UDH). When a SMS is sent from the AUS WAP Browser, the WAP application must add the remaining 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)
```

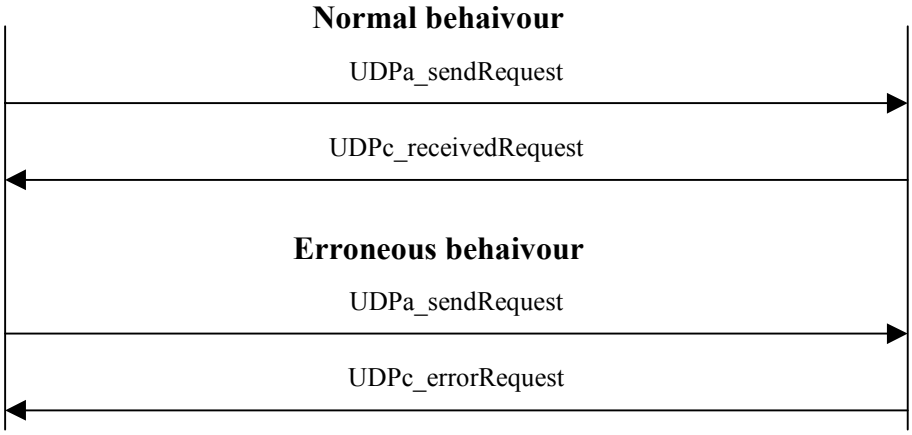
`message`	The error message can be any numeric code that is appropriate for a particular device.

18 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.

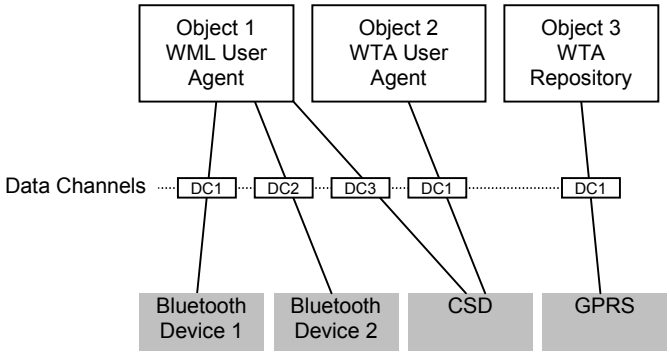
If the configuration of the AUS WAP Browser supports Push, certain ports must be opened and listened to. The actual ports that need to be opened depend on whether the configuration of the AUS WAP Browser supports WTLS, or not. The port 2948 (non-secure push) must always be opened and listened to. If WTLS is supported, 2949 must be opened and listened to, as well.

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. The actuall bearer to use is to be determined through the object id and the channel id. It is nessecary to have a table to choose from if several different bearers exist. See picture below:



Obj	Ch	Bearer
1	1	BT 1
1	2	BT 2
1	3	CSD
2	1	CSD
3	1	GPRS

Bearer	Online
BT 1	True
BT 2	True
CSD	False
GPRS	True

A table of bearers contains all predefined bearers. It is updated when new bearers occur or disapear. The default connection status of the bearer is set with the

configuration variable configONLINE. It is to be updated when bearers are connected or disconnected (after calls to CLNTa_setupConnection or CLNTa_closeConnection).

```
VOID UDPa_sendRequest (const CHAR *data, UINT16
dataLength, const CHAR *destination, UINT8
destLength, const CHAR *source, UINT8 sourceLength,
UINT16 destinationPort, UINT16 sourcePort, UINT8
objectId, UINT8 channelId)
```

`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 configUDP _IP_GW, a configuration variable . 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 configUDP _IP_SRC, a configuration variable . 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 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 UDPa_sendRequest, with different source port numbers, must also use two different physical port numbers.
`objectId`	The ID of the object (e.g. a user agent) that this request originates from. (In the case of a WML browser this is the object id used in a call to MMIc_startUserAgent.)
`channelID`	The channel, the network information has been taken from, was given to the AUS WAP Browser when the user agent was configured (CLNTc_setDCHIntConfig and/or CLNTc_setDCHStrConfig). The WAP application should use this information in order to find out the actuall bearer to use for the transmission. It could be CSD, GPRS, as well as Bluetooth. Read about channels (network configuration configuration variables) in the Client API.

receivedRequest

This function is used when an UDP datagram has been received. It is used for both requested data and pushed data.

```
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`	For requested data, the destination port is the same logical port number as was given as source port in the UDPa_sendRequest call. For non-requested data, i.e. pushed content, the destination port shall be the physical port where the data arrived, e.g. the port 2948 for non-secure push or the port 2949 for secure push.
`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. Since 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.

19 Optional source code

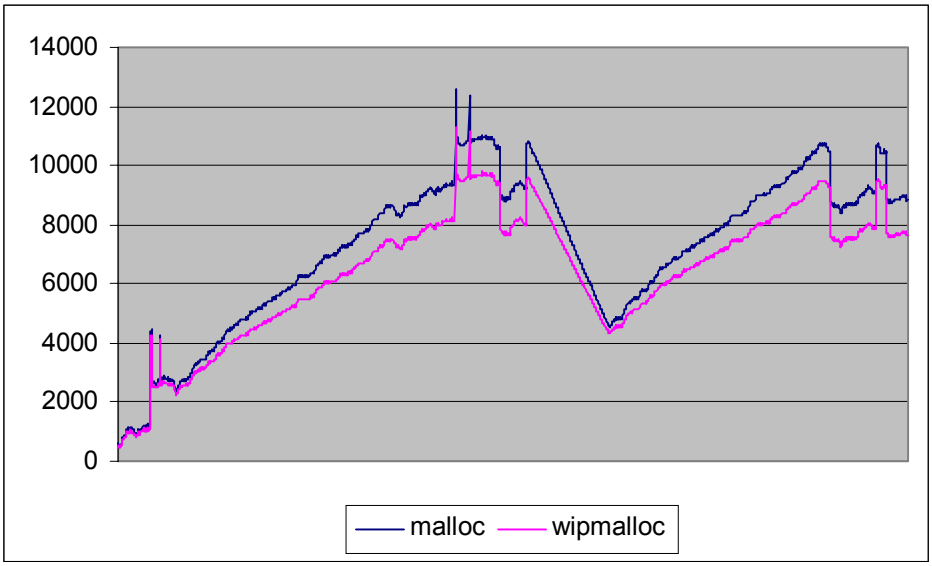
The AUS WAP Browser is delivered with optional source code that can be compiled and linked with the AUS WAP Browser. The AUS WAP Browser generally uses the optional source code when certain C macros have been set.

19.1 Memory

The environment, for which the WAP application using the AUS WAP Browser, may:

- not have a memory allocator
- have a memory allocator that returns fixed sized memory blocks, which generally are much larger than the requested size
- have a memory allocator and there is not much RAM memory in general.

An optional memory allocator can be used in these cases. It allocates the requested memory with an overhead of four extra bytes plus alignment to nearest larger multiple of four. It never allocates a chunk less than eight bytes. Comparing the ANSI C memory allocator *malloc* with the AUS WAP Browser memory allocator, *malloc* consumes eight bytes more per allocation due to extra overhead. This means that the AUS WAP Browser memory allocator consumes about two Kbytes less than malloc when large WML decks are opened. See diagram below.



The diagram illustrates the total memory consumption when two, in sequence equally sized WML decks are 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 30 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.

Incorrect usage of memory allocation and deallocation might lead to errors that are hard to track down. For example, deallocating the same memory area twice is an error whose consequences may not become evident until much later in the program. To aid in detecting such errors, one can define the symbol DEBUG_WIP_MALLOC. With this symbol defined, an internal consistency check routine will be called upon each allocation and deallocation. If an error is detected, the program is halted with an error message (using the assert macro, defined in ANSI C).

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.

19.2 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 Adapter function CLNTa_error, which is defined like this:

```
CLNTa_error (UINT8 objectId, 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. The types are described below.
Message: The associated, more general error message that can be used when this error occurs.
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. Too many user agents 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.

General error messages

Most error codes (errorNo) has messages that can be grouped into a more general message to be presented to the end-user. The following texts are examples of error messages that can be used in a WAP application:

Message #	Message
1	Requested page not found.
2	Invalid location. Check that the location has been entered correctly.
3	Internet server is not responding. Try again later.
4	Access denied. (errorNo)
5	The WAP page contains an error. (errorNo)
6	The WML script contains an error. (errorNo)
7	An error occurred in communication with the server. (errorNo)
8	An error occurred in communication with the gateway. (errorNo)
9	Unexpected error. (errorNo)
10	An unsupported character set was encountered.
11	An unexpected WBXML version was encountered.
12	An unexpected WML version was encountered.
13	A fatal error occurred. The browser will be reset.
14	The gateway is busy. Try again later.
15	The gateway does not respond. Check settings.
16	Handshake failure. A secure connection could not be established. (errorNo)

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
Message: 8
Description: WTP Abort reason according to WTP Specification: A generic error code indicating an unexpected error.

1

C enum: ERR_WTP_PROTOERR
Type: I
Message: 8
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
Message: 8

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
Message: 8
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
Message: 8
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
Message: 8
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
Message: 8
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
Message: 8
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
Message: 8
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
Message: 8
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
Message: 7

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
Message: 7
Description: See BadRequest

66

C enum: HTTPPaymentRequired
Type: I
Message: 7
Description: See BadRequest

67

C enum: HTTPForbidden
Type: I
Message: 7
Description: See BadRequest

68

C enum: HTTPFileNotFound
Type: I
Message: 1
Description: See BadRequest

69

C enum: HTTPMethodNotAllowed
Type: I
Message: 7
Description: See BadRequest

70

C enum: HTTPNotAcceptable
Type: I
Message: 7
Description: See BadRequest

71

C enum: HTTPProxyAuthenticationRequired
Type: I
Message: 7
Description: See BadRequest

72

C enum: HTTPRequestTimeout
Type: I
Message: 7
Description: See BadRequest

73

C enum: HTTPConflict
Type: I
Message: 7
Description: See BadRequest

74

C enum: HTTPGone
Type: I

Message: 7
Description: See BadRequest

75
C enum: HTTPLengthRequired
Type: I
Message: 7
Description: See BadRequest

76
C enum: HTTPPreconditionFailed
Type: I
Message: 7
Description: See BadRequest

77
C enum: HTTPRequestedEntityTooLarge
Type: I
Message: 7
Description: See BadRequest

78
C enum: HTTPRequestURITooLarge
Type: I
Message: 7
Description: See BadRequest

79
C enum: HTTPUnsupportedMediaType
Type: I
Message: 7
Description: See BadRequest

96
C enum: HTTPInternalServerError
Type: I
Message: 7
Description: See BadRequest

97
C enum: HTTPNotImplemented
Type: I
Message: 7
Description: See BadRequest

98
C enum: HTTPBadGateway
Type: I
Message: 7
Description: See BadRequest

99
C enum: HTTPServiceUnavailable
Type: I
Message: 7
Description: See BadRequest

100
C enum: HTTPGatewayTimeout
Type: I

Message: 7
Description: See BadRequest

101

C enum: HTTPVerNotSupported
Type: I
Message: 7
Description: See BadRequest

224

C enum: ERR_WSP_PROTOERR
Type: I
Message: 8
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
Message: 8
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
Message: 8
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
Message: 8
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
Message: 14
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
Message: 8
Description: WSP Abort reason according to WSP Specification: An error prevented session creation.

230

C enum: ERR_WSP_MRUEXCEEDED
Type: I
Message: 8
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
Message: 8
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
Message: 8
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
Message: 8
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
Message: 8
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
Message: 9
Description: Invalid object id was used

1004 **C enum:** ERR_WAE_UA_MAX_EXCEEDED
Type: C
Message: 9
Description: Too many active user agents

1005 **C enum:** ERR_WAE_UA_PARSE
Type: C
Message: 5
Description: Error in WML code

1006 **C enum:** ERR_WAE_UA_DISPLAY_ERROR
Type: I
Message: 5
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
Message: 5
Description: No proper WML response body, possibly empty

1008 **C enum:** ERR_WAE_UA_URL_INVALID
Type: I
Message: 2
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
Message: 3
Description: URL request timed out.

1010 **C enum:** ERR_WAE_UA_WSP_RESPONSE_INVALID
Type: I
Message: 8
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
Message: 4
Description: WML deck contained access restrictions not fulfilled by the user agent.

1012 **C enum:** ERR_WAE_UA_URL_NONSUPPORTED_SCHEME
Type: I
Message: 9
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
Message: 7
Description: A request has been redirected more than 5 times

1014 **C enum:** ERR_WAE_UA_SESSION_NOT_CONNECTED
Type: I
Message: 15
Description: Set-up of WSP session failed.

1015 **C enum:** ERR_WAE_UA_MAX_NR_OF_SESSIONS_REACHED
Type: I
Message: 9
Description: Too many WSP sessions.

1016 **C enum:** ERR_WAE_UA_INVALID_STACKMODE
Type: I
Message: 9
Description: Invalid and/or inconsistent usage of stack mode and user agent mode (e.g. WTA without WTLS).

1017 **C enum:** ERR_WAE_UA_WTA_ACCESS_DENIED
Type: I
Message: 4
Description: A non-WTA user agent is trying to run WTA content.

1018 **C enum:** ERR_WAE_UA_TOO_LARGE_DATA_TRANSFER
Type: I
Message: 8
Description: The total size of the data segments that has been downloaded, or is being sent has reached the limit (CONTENT_UA_MAX_MESSAGE_SIZE). The CONTENT_UA_MAX_MESSAGE_SIZE has been negotiated with the WAP gateway and, when the error occurs, may be a lower value. The download operation is aborted.

1019 **C enum:** ERR_WAE_UA_LARGE_DATA_TRANSFER_DISABLED
Type: I
Message: 9
Description: If the configuration variable LARGE_DATA_TRANFER_ENABLED is not defined and the function CLNTc_postContent is called with the argement moreData set to true, this error message is generated.

1101 **C enum:** ERR_WAE_WML_INSTREAM_FAULT
Type: I
Message: 5
Description: Felet uppkommer om WBXML-dokumentet är kodat så att parsern försöker läsa förbi sista byten i indatan. Detta kan till exempel uppkomma om datan har blivit trunkerad av någon anledning eller om dokumentet är kodat felaktigt.

1102 **C enum:** ERR_WAE_WML_CONTENT_CHARSET_ERROR
Type: I
Message: 5
Description: Error in the character set coding. This can be originated to the content server or to the proxy server if it performs transcoding.

1103 **C enum:** ERR_WAE_WML_CONTENT_CHARSET_NOT_SUPPORTED
Type: I
Message: 10
Description: Character set coding not supported

1104 **C enum:** ERR_WAE_WML_UNKNOWN_TOKEN
Type: I
Message: 9
Description: Illegal WML token

1105 **C enum:** ERR_WAE_WML_WML_ERROR
Type: I
Message: 5
Description: Illegal WML, e.g. unexpected end of file

1106 **C enum:** ERR_WAE_WBXML_CONTENT_VERSION_WARNING
Type: I
Message: 11
Description: If the version number of the WBXML specification is different from 01 or 02, a warning is issued. Parsing is not cancelled.

1108 **C enum:** ERR_WAE_WBXML_CONTENT_PUBLIC_ID_ERROR
Type: I
Message: 5
Description: The public id is used to identify a well-known document type contained within the WBXML entity. Parsing is not proceeded if this message is issued. This error message is issued if the following criterions for an acceptable public identifier fail:
Public id = 00
and the public identifier string is: "-//WAPFORUM//DTD WML 1.1//EN"
or the public identifier string is: "-//WAPFORUM//DTD WML 1.2//EN"
or the content-type is set to application/vnd.wap.wmlc
and the WSP parameter "level", if present, is specified to 1.1
or the WSP parameter "level", if present, is specified to 1.2
or the WSP parameter "level" is not present (WAP 1.2 is then assumed)
Public id = 04 (WAP 1.1)
Public id = 09 (WAP 1.2)

1202 **C enum:** ERR_WAE_WMLS_VERIFICATION
Type: I
Message: 6
Description: Verification failed, not proper byte-code

1203 **C enum:** ERR_WAE_WMLS_LIB
Type: I
Message: 6
Description: Fatal library function error

1204 **C enum:** ERR_WAE_WMLS_FUNC_ARGS
Type: I
Message: 6
Description: Invalid function arguments

1205 **C enum:** ERR_WAE_WMLS_EXTERNAL
Type: I
Message: 6
Description: External function not found

1206 **C enum:** ERR_WAE_WMLS_LOAD
Type: I
Message: 6
Description: Unable to load compilation unit

1207 **C enum:** ERR_WAE_WMLS_ACCESS
Type: I

Message: 4
Description: Access violation

1208
C enum: ERR_WAE_WMLS_STACK_UNDERFLOW
Type: I
Message: 6
Description: Stack underflow

1209
C enum: ERR_WAE_WMLS_ABORT
Type: I
Message: 6
Description: Programmed abort

1210
C enum: ERR_WAE_WMLS_STACK_OVRFLW
Type: I
Message: 6
Description: Stack overflow

1211
C enum: ERR_WAE_WMLS_USER_ABORT
Type: I
Message: 6
Description: User initiated abort

1212
C enum: ERR_WAE_WMLS_SYSTEM_ABORT
Type: I
Message: 6
Description: System initiated

1213
C enum: ERR_WAE_WMLS_NULL
Type: I
Message: 6
Description: Some component was inaccessible

1301
C enum: ERR_WAE_REP_SERVICE_INSTALL_FAILED
Type: I
Message: 9
Description: Service installation failed

1302
C enum: ERR_WAE_REP_MEM_ACCESS_FAILED
Type: I
Message: 9
Description: Memory access failed when global binding exists

1401
C enum: ERR_WAE_PUSH_ACTIVATE_FAILED
Type: I
Message: 9
Description: Pushed content failed to be activated

1402
C enum: ERR_WAE_PUSH_DELETE_FAILED
Type: I

Message: 9
Description: Push content failed to be deleted

1404
C enum: ERR_WAE_PUSH_STORE_FAULT
Type: I
Message: 9
Description: Pushed content failed to be stored

2001
C enum: ERR_WSPCL_ErrorInAddressFromWAE
Type: I
Message: 9
Description: Address received from WAE incorrect

2002
C enum: ERR_WSPCL_ErrorExtractReplyPDU
Type: I
Message: 9
Description: Received reply PDU incorrect or no memory available for extraction

2004
C enum: ERR_WSPCL_ErrorNoBuffersAvailable
Type: I
Message: 9
Description: No free memory available

2005
C enum: ERR_WSPCL_ErrorMethodNotSupported
Type: I
Message: 9
Description: Requested method not supported.

3001
C enum: ERR_WSPCM_ErrorNoMemoryAvailableForPDUPacking
Type: I
Message: 9
Description: No memory available for PDU construction

3002
C enum: ERR_WSPCM_ErrorNoMemoryAvailableForPDUUnPacking
Type: I
Message: 9
Description: No memory available for PDU extraction.

3003
C enum: ERR_WSPCM_ErrorInDataFromWAE
Type: I
Message: 9
Description: Data from WAE erroneous.

3004
C enum: ERR_WSPCM_ErrorInReplyFromServer
Type: I
Message: 9
Description: Data from server erroneous.

3005
C enum: ERR_WSPCM_ErrorMaxSessionsAlreadyReached
Type: I

Message: 9
Description: Max number of active sessions already reached.

3006

C enum: ERR_WSPCM_ErrorMOMAlreadyReached
Type: I
Message: 9
Description: Number of outstanding methods already reached.

3007

C enum: ERR_WSPCM_WAEErrorNoPIdFoundMatchingSession
Type: I
Message: 9
Description: The request does not conform to an existing session..

3008

C enum: ERR_WSPCM_ErrorNoPIdFoundMatchingMethod
Type: I
Message: 9
Description: Non requested data received, or a second answer of an old request.

3009

C enum: ERR_WSPCM_ErrorNoPIdFoundMatchingPush
Type: I
Message: 9
Description: Non requested Push acknowledgement received.

3010

C enum: ERR_WSPCM_ErrorStoreOMInfoFailed
Type: I
Message: 9
Description: Session data could not be stored.

3011

C enum: ERR_WSPCM_ErrorStoreHandleFailed
Type: I
Message: 9
Description: Handle could not be stored.

3012

C enum: ERR_WSPCM_ErrorMethodNotSupported
Type: I
Message: 9
Description: Requested method not supported.

3013

C enum: ERR_WSPCM_ErrorSameAQUsed
Type: I
Message: 9
Description: Session disconnected because a new one is started using the same WAP gateway.

3014

C enum: ERR_WSPCM_WAEErrorNoPIdFoundMatchingMethod
Type: I
Message: 9
Description: WAE has given acknowledgement of a non-received request.

4001
C enum: ERR_WTP_ErrorNORESPONSE
Type: I
Message: 8
Description: No response PDU has been received, and no more re-transmissions to do.

4002
C enum: ERR_WTP_ErrorNOFREEBUFF
Type: I
Message: 9
Description: Out of memory, cannot proceed processing of transaction.

4003
C enum: ERR_WTP_ErrorINVALID_BEARER
Type: I
Message: 9
Description: Invalid bearer.

4004
C enum: ERR_WTP_ErrorINVALID_CLASS
Type: I
Message: 9
Description: Invalid transaction class.

4005
C enum: ERR_WTP_ErrorINVALID_ACKTYPE
Type: I
Message: 9
Description: Invalid user acknowledgement type.

5001-5099
C enum: None
Type: I
Message: 9
Description: General WTLS error. The WTLS connection is in most cases dropped.

5100-5199
C enum: None
Type: I
Message: 16
Description: Handshake failure. A WTLS connection could not be established.

5200-5299
C enum: None
Type: I
Message: 9
Description: Error in crypty library. In most cases is the WTLS connection dropped.

5300-5399
C enum: None
Type: I
Message: 8
Description: WTLS error message from WAP gateway. WTLS connection is dropped.

5400-5499
C enum: None
Type: I
Message: 9
Description: Internal WTLS error. WTLS connection is dropped.

6001
C enum: ERR_WDP_ErrorInDatafromWSP
Type: I
Message: 9
Description: The data received from WSP was faulty

6002
C enum: ERR_WDP_ErrorInDatafromWTP
Type: I
Message: 9
Description: The data received from WTP was faulty

6003
C enum: ERR_WDP_ErrorBearerNotSupported
Type: I
Message: 9
Description: The bearer is not supported

6005
C enum: ERR_WDP_UDPErrorInd
Type: I
Message: 9
Description: UDP error

6006
C enum: ERR_WDP_UDPBigBuffer
Type: I
Message: 9
Description: Received UDP datagram too big

6007
C enum: ERR_WDP_SMSErrorInd
Type: I
Message: 9
Description: SMS error

6008
C enum: ERR_WDP_SMSBigBuffer
Type: I
Message: 9
Description: Received SMS datagram too big

6009
C enum: ERR_WCMP_PortUnreachable
Type: I
Message: 9
Description: Erroneous port number used by AUS WAP Browser. If this error code is received, the WAP application should indicate that to the AUS WAP Browser by calling the function CLNTc_closePort.

6010
C enum: ERR_WDP_WCMP_MessageTooBig
Type: I

Message: 9
Description: Datagram sent by AUS WAP Browser too big.

7001
C enum: ERR_UDCP_UNKNOWN
Type: I
Message: 9
Description: Error Code from Error PDU

7002
C enum: ERR_UDCP_PROTOERR
Type: I
Message: 9
Description: Error Code from Error PDU

7003
C enum: ERR_UDCP_UDCPVERSIONZERO
Type: I
Message: 9
Description: Error Code from Error PDU

7004
C enum: ERR_UDCP_EXTADDRNOTSUPP
Type: I
Message: 9
Description: Error Code from Error PDU

7005
C enum: ERR_UDCP_NETERROR
Type: I
Message: 9
Description: Error from USSDErrorInd

7006
C enum: ERR_UDCP_QUEUEFULL
Type: I
Message: 9
Description: Error when sendqueue is full

7007
C enum: REL_UDCP_UNKNOWN
Type: I
Message: 9
Description: Release Code from RD PDU

7008
C enum: REL_UDCP_UTIMEOUT
Type: I
Message: 9
Description: Release Code from RD PDU

7009
C enum: REL_UDCP_UIDLE
Type: I
Message: 9
Description: Release Code from RD PDU

7010
C enum: REL_UDCP_USER
Type: I

Message: 9
Description: Release Code from RD PDU

7011

C enum: REL_UDCP_NETRELEASE
Type: I
Message: 9
Description: Release from USSDReleaseInd

8001

C enum: ERR_MEMORY_WARNING
Type: C
Message: 9
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
Message: 13
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