User Manual

for

AutoUniConv SDK

A C/C++ software development kit to
automatically convert text encoded in various charsets to Unicode

Covers version 1.2.0

Lingua Systems
NATURAL
LANGUAGE
PROCESSING

AutoUniConv SDK User Manual, published April 11, 2014.

Lingua-Systems Software GmbH, Gerichtsstraße 42, 44649 Herne, Germany, *info@lingua-systems.com*

If you have problems using the links or get aware of any faults, feel free to give a brief hint on it via *support@lingua-systems.com*.

Contents

About this Manual

This manual addresses users with experience in C/C++ programming and at least a basic knowledge of library usage.

The manual provides a short introduction to the library, its supported character encodings (charsets) as well as instructions how to install the AutoUniConv SDK software package. Afterwards the complete API is introduced along with the possibilities of error handling. A complete usage example is attached in appendix A.

For a quickstart have a look at the function reference in the documentation of the application programming interface (chapter 4.3 on page 11).

Administrators who want to install the software get all necessary information in chapter 3, page 7.

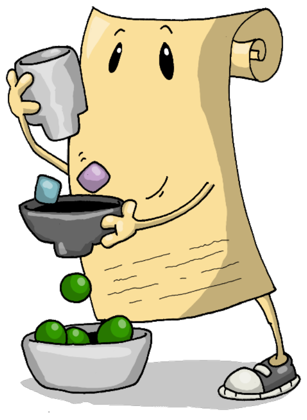
Conventions used in this Manual

At several points of this manual it is necessary to make a distinction between strings that may not contain embedded *NUL* characters and those which may – due to their special charset – potentially contain *NUL* characters. In this manual, the former are called "*Strings*" while the latter are called "*Byte Strings*".

1. Introduction

AutoUniConv SDK provides a dynamic C/C++ library that provides the functionality to automatically identify a string's charset and convert it to Unicode afterwards, if the string is encoded in a supported charset. This way, documents can be processed with all the advantages of the modern Unicode standard even if they are stored in a potentially unknown charset.

The library features fast and reliable processing and conversion. It has no software dependencies other than the standard C and thread library of the system it is deployed on and is easy to integrate. Even on dated hardware it works quiete efficiently.



2. Supported Charsets

AutoUniConv currently identifies 39 distinct charsets and converts them to one of the common Unicode Transformation Formats (UTF). The set of supported charsets covers both current and legacy charsets. This way, a maximum amount of documents can be converted automatically.

Family	Charsets
Unicode	UTF-8, -16LE, -16BE, -32LE, -32BE
ISO	ISO-8859-1, -2, -3, -4, -5, -6, -7, -15, -16
Windows	Windows-1250, -1251, -1252, -1253, -1256, -1257
IBM/DOS Code Pages	CP-720, -737, -775, -850, -852, -855, -866
Macintosh	MacArabic, -CentralEuropean, -Cyrillic, -Greek, -Roman, -Romanian, -Ukrainian
National	Big5, GB2312, KOI8-R, KOI8-U, ASCII

Figure 1: Supported Charsets (Input)

AutoUniConv supports UTF-16 and UTF-32 in both little- and big-endian byte order. As the byte order is chosen explicitly, the converted byte strings do – in conformance to official proposals – not contain any byte order mark ("BOM").

Family	Charsets
Unicode	UTF-8, -16LE, -16BE, -32LE, -32BE

Figure 2: Supported Charsets (Output)

3. Installation

3.1. Requirements

AutoUniConv SDK merely requires the system's standard C runtime environment.

3.2. What Will Be Installed

The AutoUniConv SDK contains a dynamic library (DLL/SO), its header file, the code of an example application and this manual.

The Software Development Kit for Linux contains the following files:

```
./doc:
example.c  LICENSE.txt  manual-sdk-eng.pdf

./include:
auc.h

./lib:
libauc.so@  libauc.so.1@  libauc.so.1.0.0
```

3.3. Installing the Software

AutoUniConv SDK is provided as a compressed archive, either in "Zip" or "tar.gz" form, depending on the target platform.

To install the software, just unpack the archive to a directory of your choice and add the library and header files to your project.

3.4. Deinstalling the Software

To deinstall the software, just remove the directory you unpacked AutoUniConv SDK to.

4. Application Programming Interface

The AutoUniConv C/C++ library provides an API that is intuitive to use and allows integration into applications easily. All functions and data structures are prefixed "*auc_*" to avoid confusions and collisions with other third party library functions and are defined in the header file *auc.h*.

AutoUniConv provides two main functions that allow to automatically convert text encoded in a supported charset to Unicode:

→ *auc_conv()* – use a character string as input source (`const char *`)

→ *auc_nconv()* – use a byte string of a given length as input source (`const char *`)

Both functions return the same data structure, a pointer to an `auc_bytes_t` structure. This data structure provides the converted bytes of the input and its length. For convenience, the structure also contains information on the Unicode Transformation Format used to encode the bytes (see chapter 4.2.1 on page 10).

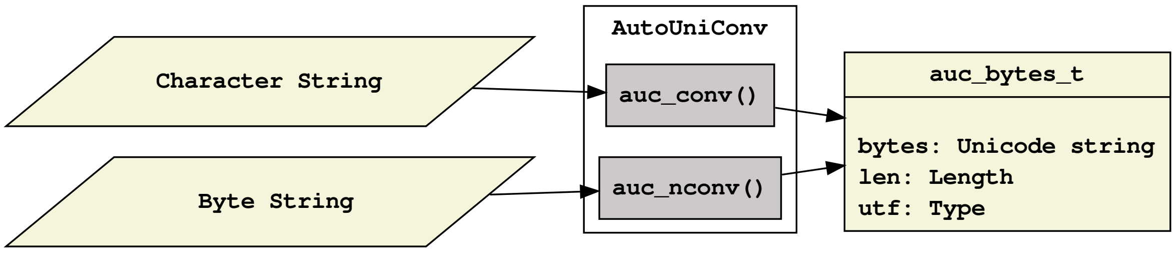


Figure 3: Flowchart of the main AutoUniConv functions

In order to fulfill its purpose, AutoUniConv has to identify the charset of the input string internally. To ensure optimal identification results, an input string should not be too short, consist of different words and thus provide some degree of variance (see chapter 6.1 on page 15).

If the results are no longer needed, you should utilize the *auc_free_bytes_t()* function to free all memory used by the result's data structure.

To handle errors in applications that make use of AutoUniConv, the library provides *auc_strerror()*, that allows to obtain a natural language description of error codes stored in the pseudo-variable *auc_errno* (see chapter 4.3.4 on page 12 and chapter 5.2 on page 14).

All functions provided by the AutoUniConv library are thread-safe and can therefore be used by more than one thread simultaneously.

4.1. A Minimal Application

The following application gives a first overview on the usage of the AutoUniConv library. Every provided function and the `auc_bytes_t` data structure is described in detail in the subsequent chapters.

```
#include <stdio.h>
#include <stdlib.h>
#include <auc.h>

int main(int argc, char *argv[])
{
    int i = 0;

    if (argc < 2)
    {
        fprintf(stderr, "usage: %s string(s)\n", argv[0]);
        return EXIT_FAILURE;
    }

    for (i = 1; i < argc; i++)
    {
        auc_bytes_t *b = NULL;

        b = auc_conv(argv[i], AUC_UTF32BE, AUC_DEFAULT);

        if (b)
        {
            printf("String %d: %u %s bytes\n",
                   i, (unsigned int) b->len,
                   auc_utf_t_to_name(b->utf));
            auc_free_bytes_t(b);
        }
        else
        {
            fprintf(stderr, "auc_conv: %s\n",
                    auc_strerror(auc_errno));
            return EXIT_FAILURE;
        }
    }

    return EXIT_SUCCESS;
}
```

The example application makes use of the function *auc_conv()* to automatically convert all argument strings to UTF-32BE and print their length. In case an error occurs during processing, the application prints an appropriate error message and terminates.

```
$ ./auc-mini \
"Alle Menschen sind frei und gleich an Würde und Rechten geboren." \
"Alla människor äro födda fria och lika i värde och rättigheter."
String 1: 256 UTF-32BE bytes
String 2: 252 UTF-32BE bytes
```

4.2. Important Data Structures

4.2.1. auc_bytes_t

AutoUniConv's main functions return a pointer to an `auc_bytes_t` data structure as a result. The data structure comprises the byte string converted to the requested Unicode Transformation Format, its length and the type it has been converted to.

This way, `auc_bytes_t` provides all information necessary to process string data in a robust and effective manner – no matter which Unicode Transformation Format it has been converted to.

Member	Type	Description	Example
`bytes`	`char *`	UTF byte string	"Chaîne de caractères"
`len`	`size_t`	Length of the string (in bytes)	22
`utf`	`auc_utf_t`	Information on the used UTF	AUC_UTF8

Figure 4: `auc_bytes_t` Members

4.2.2. auc_utf_t

The `auc_utf_t` data structure provides named constants for all Unicode Transformation Formats supported by AutoUniConv:

Named Constant	Description
`AUC_UTF8`	UTF-8
`AUC_UTF16LE`	UTF-16 (Little-Endian)
`AUC_UTF16BE`	UTF-16 (Big-Endian)
`AUC_UTF32LE`	UTF-32 (Little-Endian)
`AUC_UTF32BE`	UTF-32 (Big-Endian)

Figure 5: `auc_utf_t` Named Constants

These named constants are particularly useful to request a special Unicode Transformation Format from one of AutoUniConv's main functions.

4.2.3. auc_flag_t

The `auc_flag_t` data structure provides a set of named constants that allow to suite the mode of operation of AutoUniConv's functions *auc_conv()* and *auc_nconv()*:

Named Constant	Description
`AUC_DEFAULT`	Default behaviour
`AUC_STRICT`	Abort on the first decoding error
`AUC_WARN`	Print a warning on each decoding error

Figure 6: `auc_flag_t` Named Constants

In default mode, both *auc_conv()* and *auc_nconv()* attempt to replace decoding errors with a common placeholder, the tilde character ("~") wherever possible. Processing will not be interrupted in these cases and no warnings will be issued. Otherwise an error will be thrown.

If this behaviour is not desired for a special application of yours, you can suite the function's behaviour to your needs using the flags described above. Please note that for maximum flexibility the flags can be combined by simply adding them to another (i.e. "`AUC_STRICT + AUC_WARN`").

4.3. Function Reference

All of AutoUniConv's functions and data structures are defined within the header file *auc.h*. The header has to be included in all applications that make use of the following functions.

4.3.1. auc_conv()

```
auc_bytes_t * auc_conv (const char *str,
                        auc_utf_t   utf,
                        auc_flag_t  flags);
```

The function takes a pointer to a string (`const char *`), a specification of the desired Unicode Transformation Format (`auc_utf_t`) and (a combination of) flags (`auc_flag_t`) that may be used to suite the function's behaviour to special needs.

The function returns a pointer to an `auc_bytes_t` structure as a result (see chapter 4.2.1, page 10).

Any call of the function resets the value stored to `auc_errno`.

If an error occurs, the function returns a pointer to `NULL` and sets the pseudo-variable `auc_errno` to an appropriate value that indicates the error (see chapter 5, page 13).

For detailed explanation on the data structures `auc_utf_t` and `auc_flag_t`, refer to the chapters 4.2.2 and 4.2.3.

As soon as the results are not needed any longer, you should free the memory allocated by the *auc_bytes_t* structure using *auc_free_bytes_t()*.

Whenever a string may contain *NUL* bytes (for example in case of an UTF-16 or UTF-32 charset), use *auc_nconv()* instead of *auc_conv()*.

4.3.2. auc_nconv()

```
auc_bytes_t * auc_nconv (const char *bstr,
                         size_t      blen,
                         auc_utf_t   utf,
                         auc_flag_t  flags);
```

The function takes a pointer to a byte string (`const char *`), its length, a specification of the desired Unicode Transformation Format (`auc_utf_t`) and (a combination of) flags (`auc_flag_t`) that may be used to suite the function's behaviour to special needs.

The function returns a pointer to an `auc_bytes_t` structure as a result (see chapter 4.2.1, page 10).

Any call of the function resets the value stored to `auc_errno`.

If an error occurs, the function returns a pointer to `NULL` and sets the pseudo-variable `auc_errno` to an appropriate value that indicates the error (see chapter 5, page 13).

For detailed explanation on the data structures `auc_utf_t` and `auc_flag_t`, refer to the chapters 4.2.2 and 4.2.3.

As soon as the results are not needed any longer, you should free the memory allocated by the *auc_bytes_t* structure using *auc_free_bytes_t()*.

As *auc_nconv()* handles byte strings appropriate, even if they contain "NUL" characters (ASCII 0x00), this function should be utilized instead of *auc_conv()* whenever the length of the input is already known.

The parameter blen has to be set to a value that is lower or equal to the length of the byte string excluding string termination characters. Due to the technical properties of a byte string, *auc_nconv()* cannot determine the correct length on its own. Severe exceptions may occur whenever blen is set to a value that exceeds the bounds of bstr, so setting this value deserves special care and attention.

4.3.3. auc_free_bytes_t()

```
void auc_free_bytes_t (auc_bytes_t *bs);
```

The function takes a pointer to an auc_bytes_t structure as an argument.

Like the *free(3)* function provided by the standard C library, *auc_free_bytes_t()* has no return value.

The memory allocated by bs is freed completely and will be available for the application again.

4.3.4. auc_strerror()

```
const char * auc_strerror (auc_errno_t errno);
```

The function takes an error number (auc_errno_t) as an argument and returns a pointer to a read-only string (const char *) containing the natural language error message.

If an error occurs, you should pass the value of auc_errno to this function in order to obtain the natural language error message associated with the error. A detailed explanation on error handling, error messages and predefined named constants can be found in chapter 5.2 on page 14.

The memory pointed to by the returned pointer must not be freed.

4.3.5. auc_utf_t_to_name()

```
const char * auc_utf_t_to_name(auc_utf_t utf);
```

The function takes a numeric representation corresponding to a Unicode Transformation Format as defined by auc_utf_t and returns a pointer to a read-only string (const char *) that contains the name of the Unicode Transformation Format referenced by utf.

The memory pointed to by the returned pointer must not be freed.

4.3.6. auc_version()

```
int auc_version();
```

The function does not take an argument and returns a numeric representation of AutoUniConv's version.

4.3.7. auc_version_string()

```
const char * auc_version_string();
```

The function does not take an argument and returns a pointer to a read-only string containing AutoUniConv's version (const char *), for example "1.2.0".

The memory pointed to by the returned pointer must not be freed.

5. Error Handling

In case an error occurs within one of the main functions of AutoUniConv, a pointer to `NULL` is returned and `auc_errno` is net to an appropriate value indicating the occurred error ($\neq$ `AUC_OK`).

The general error handling policy should be implemented as follows:

1. Return value does not equal `NULL`? $\rightarrow$ No error
2. Return value equals `NULL`? $\rightarrow$ An error occurred
   a) Evaluate `auc_errno`, handle the error and eventually
   b) utilize *auc_strerror()* to obtain a natural language error message describing the occurred error

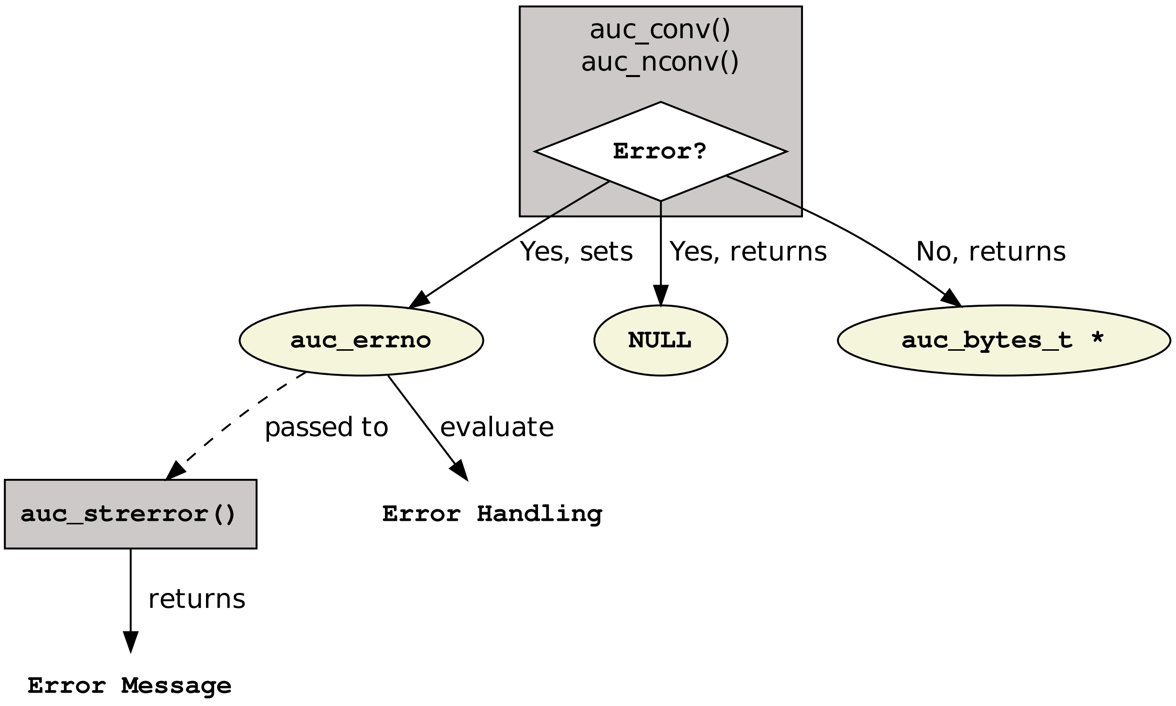


Figure 7: Flowchart of the AutoUniConv error handling

AutoUniConv's error handling takes allocation of memory into account and frees all allocated memory in every known error path.

5.1. Pseudo-Variable auc_errno

`auc_errno` may be used by many threads simultaneously, because it is not implemented as a global variable. The memory necessary for `auc_errno` is allocated on a per-thread basis using Thread-Local Storage (TLS). This way each thread is able to utilize its own `auc_errno` variable.

Nevertheless `auc_errno` can be used as if it was a common global variable[1].

If an error occurs, `auc_errno` is set to a value that discriminates the error. On any call of one of AutoUniConv's main functions, the value of `auc_errno` is reset to `AUC_OK`.

[1]Each occurrence of `auc_errno` is replaced with a call to the function *auc_errno_location()*, which returns the address of the thread-local variable, by the C preprocessor. As a result, `auc_errno` can be used as if it was a global variable, although it is not. Therefore we call it a pseudo-variable.

5.2. auc_errno_t Named Error Constants

AutoUniConv uses the type `auc_errno_t` to provide named error constants for all error cases. There is a constant defined for any runtime error.

If an error occurs, `auc_errno` is set to a value of type `auc_errno_t` that indicates the error. This way, case dependent error handling can easily be implemented in any application using AutoUniConv.

The named error constant may, as well as `auc_errno`, be used to obtain a natural language error message describing the numeric error code (see chapter 4.3.4, page 12).

The following table comprises all named error constants used in AutoUniConv version 1.2.0, accompanied by the error messages returned if passed to *auc_strerror()*.

Constant	Error Message
`AUC_OK`	No error
`AUC_ENOMEM`	Memory allocation failed
`AUC_EARG`	Invalid argument
`AUC_ESHORT`	Insufficient input length
`AUC_EIDENT`	Identifying charset failed
`AUC_ENODEC`	No decoder available for charset
`AUC_EDEC`	Decoding failed
`AUC_EENC`	Encoding failed
`AUC_EBINARY`	Binary input data
`AUC_EEMBNUL`	Embedded NUL character

Figure 8: `auc_errno_t` Named Constants

6. Hints on Application Development

6.1. Achieving Optimal Conversion

The identification of a charset may be a very complex process, because most charsets do not contain any metadata that discriminates them. As a result, AutoUniConv relies on a set of algorithms to obtain this information from the text itself. Most of these algorithms, however, gain better results the more input is available for identification. We recommend an input length of at least 25 characters.

Besides that we recommend to pass the text to be processed by your application to AutoUniConv as early and as complete as possible, for example right before tokenization.

This way your application will not only be optimized in respect to the results' quality, but also with respect to runtime performance, because this approach helps saving overhead and a lot of (unnecessary) function calls.

6.2. Determining AutoUniConv's Version

After including the *auc.h* header, the macro `AUC_VERSION_STRING` is available and replaced by a character string containing AutoUniConv's version by the C preprocessor *at compile time*.

To determine AutoUniConv's version at runtime, use *auc_version_string()* (see chapter 4.3.7, page 12).

A. Example Application

```c
/*
 * An example application introducing AutoUniConv's auc_nconv().
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <auc.h>

#define BUFSIZE 2048

int main(int argc, char *argv[])
{
    FILE        *fp = NULL;
    auc_bytes_t *b  = NULL;
    auc_utf_t    utf;
    char         buf[BUFSIZE + 1];
    size_t       read;

    if (argc != 3)
    {
        fprintf(stderr, "usage: %s UTF file\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* map requested UTF string to auc_utf_t */
    if (strcmp(argv[1], "UTF-8") == 0)
    {
        utf = AUC_UTF8;
    }
    else if (strcmp(argv[1], "UTF-16LE") == 0)
    {
        utf = AUC_UTF16LE;
    }
    else if (strcmp(argv[1], "UTF-16BE") == 0)
    {
        utf = AUC_UTF16BE;
    }
    else if (strcmp(argv[1], "UTF-32LE") == 0)
    {
        utf = AUC_UTF32LE;
    }
    else if (strcmp(argv[1], "UTF-32BE") == 0)
    {
        utf = AUC_UTF32BE;
    }
    else
    {
        fprintf(stderr, "unhandled UTF value: %s\n", argv[1]);
        return EXIT_FAILURE;
    }
    /* read the first BUFSIZE (or less) bytes from file */
    if ((fp = fopen(argv[2], "rb")) == NULL)
    {
        fprintf(stderr, "%s: failed to open\n", argv[2]);
        return EXIT_FAILURE;
    }

    read = fread(buf, 1, BUFSIZE, fp);
    buf[read] = 0x00;
    fclose(fp);

    printf("%s: %u bytes\n", argv[2], (unsigned int) read);

    if (! read)
    {
        fprintf(stderr, "%s: no bytes read\n", argv[2]);
        return EXIT_FAILURE;
    }

    /* automatically convert the read bytes to the requested UTF */
    b = auc_nconv(buf, read, utf, AUC_WARN);

    if (b) /* not NULL -> success */
    {
        printf("auc_nconv returned %u %s bytes\n",
               (unsigned int) b->len, auc_utf_t_to_name(b->utf));

        if (b->utf == AUC_UTF8) /* do not print UTF-16 & -32 */
        {
            printf("\n%s", b->bytes);
        }

        auc_free_bytes_t(b);
    }
    else /* NULL -> error */
    {
        fprintf(stderr, "auc_nconv failed: %s\n",
                auc_strerror(auc_errno));
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

The following output shows an example execution of the application:

```
$ cat /tmp/french_iso-8859-1.txt
Tous les ?tres humains naissent libres et ?gaux en dignit? et
en droits.
$ ./auc-example UTF-8 /tmp/french_iso-8859-1.txt
/tmp/french_iso-8859-1.txt: 73 bytes
auc_nconv returned 76 UTF-8 bytes

Tous les êtres humains naissent libres et égaux en dignité et
en droits.
```

B. References

→ Lingua-Systems' AutoUniConv SDK product website,
http://www.lingua-systems.com/unicode-converter/

→ The Unicode Standard,
http://unicode.org/

→ RFC 2781: "UTF-16, an encoding of ISO 10646",
http://www.ietf.org/rfc/rfc2781.txt

→ RFC 2279: "UTF-8, a transformation format of ISO 10646",
http://www.ietf.org/rfc/rfc2279.txt

http://www.lingua-systems.com/unicode-converter/

Index