Unicoder
Reference Manual
A Portable Wide-Character Terminal Output Package for C and C++

S. Thomas Bradley

version 21.11
(for Linux and Windows)
November 2021

This manual documents how to install and use *Unicoder, A Portable Wide-Character Terminal Output Package for C and C++,* version 21.11 for Linux and Windows. *Unicoder* is the main part of the 2020 Console Video Project (COVID-20).

Table of Contents

Unicoder

for C and C++

It is the intent of *Unicoder* to be a portable software package which provides support for displaying unicode characters in a C or C++ console terminal application for Linux or Windows. That is, it provides the support needed to *easily* create console terminal applications that can output Unicode characters, such as Greek letters or scientific symbols to a C or C++ console terminal display. This software has been written in C11 to be portable to a number of compilers and the source code is distributed under the *GNU General Public License.*

1 Overview

The C and C++ languages support a type of text only based application called a console. Programming these types of applications is far easier then creating a GUI based application. Both C and C++ provide support for outputting text to the console's terminal display; referred to on the Windows platform as the *Command Prompt* and on Linux as the *GNOME-terminal.* In this document both the Linux GNOME-terminal and the Windows Command-Prompt terminal will be referred to as the Terminal. In Windows, these types of programs are run using the cmd.exe or the Powershell applications while on Linux they are ran in the Terminal window which using Ubuntu Linux can be called simply by pressing Ctrl-Alt-T.

In both C and C++ the `char` type is used for storing characters such as letters and punctuation marks. To handle these characters the computer uses a numeric code in which integer numbers represent the various characters. For example, C and C++ use the number 65 `(0x41)` as the code for the character `A` and the integer 77 `(0x4D)` is the code for the character `M`. The `char` type is typically defined as an 8-bit (1-byte) unit of memory so `char` can store up to 256 different characters. You are able to display any of these characters on the Terminal's display using the C/C++ standard library.

This limit of 256 characters can be a problem so both C and C++ support the wide character type `wchar_t`. This type can hold a larger range of values such as the Unicode character set. On the Windows platform the `wchar_t` represents a 16-bit (2-byte) unit of memory while on Linux it is 32-bits (4 bytes). Both the C and C++ libraries provide functions, such as `wprintf()` or `wcout` to support this type. However, if you use any of these functions in a Terminal application to output unicode characters to the display you will not get the result you expect, your display will be unreadable.

This is because the Terminal doesn't know that your application is using wide characters so it doesn't display them correctly. All that is usually required for a Windows or Linux Terminal app to display wide characters correctly is for the app to use the `udispOn()` (*u*nicode *disp*lay *on*) function to tell the Terminal that it will be using wide characters and unicode. This is very useful because different Windows compilers use different methods to turn on this display mode. *Unicoder* takes care of this for you by using the proper call for the compiler you are using.

Additionally, on any platforms that displays wide characters *Unicoder* provides definitions that make it easy for you to add Unicode characters for display in your Terminal app.

1.1 Copying Conditions

This software package is *free*; this means that everyone is free to use it and free to redistribute it on a free basis. The software package is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of this software package that they might get from you.

Specifically, I want to make sure that you have the right to give away copies of this software package, that you receive source code or else can get it if you want it, that you can change this software package or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, I have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of *Unicoder* you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for my own protection, I must make certain that everyone finds out that there is no warranty for *Unicoder*. If it is modified by someone else and passed on, I want their recipients to know that what they have is not what I distributed, so that any problems introduced by others will not reflect on my reputation.

The precise conditions of the license for *Unicoder* are found in the Lesser General Public License, version 3 that accompanies the source code, see 'COPYING.LIB'.

1.2 C11 Compliance

This software package is written in C and is intended to conform to the C11 Standard. It is the intent that this software package be portable to any system with a working C11 or C++ compiler.

Because it isn't possible for me to test this software package on every possible platform or compiler it will probably have some issues with various compilers/platforms, If you make improvements to this software package so it will work on your platform or compiler and you would like to contribute please pass those improvement on to me so they can be incorporated into future versions of *Unicoder*.

1.2.1. C++ and Visual Studio

Meeting the C11 standard may cause some issues if you are using C++ or Microsoft's Visual Studio.

The function `to_wstring()` is defined for both C and C++. The C version defined by *Unicoder* relies on using the C11 `_Generic` type while the. C++ version is an overload of the C++ `to_wstring` function found in `<string>`, The C++ version does not return the string length.

Additionally, Microsoft has given updating the C compiler in Visual Studio a low priority so they have not yet upgraded their C compiler to conform to the C11 standard. This means you may have some issues when you use functions that are part of the C11 standard. This includes `to_wstring()`. Therefore, it is recommended that you program in C++ so you can use the C++ version of `to_wstring()`. The macro `umodetype()` is not implemented and always returns -1 with C++.

1.3 Hearing from you

I am very much interested in hearing from you about how you are using this software package. So please feel free to send me an email about your project. Also, if you have any constructive criticism or want to report bugs please email them to me with example code that will reproduce the bug. See section 6. Bug Reporting.

2 Using Unicoder, a Portable Wide-Character Terminal Output Package

This software package does not rely on a specific platform or compiler to work. It was developed using the Windows 7 and Windows 10 platforms and the Ubuntu Linux platform. In Windows it was developed using multiple compilers. On Ubuntu Linux it was developed using the Ubuntu version of GCC. Additionally, it was tested using both C and C++ compilers. See the `README` for a list of compilers.

2.1 Adding Unicoder to your C or C++ application

Unicoder is contained in a single header file called `unicoder.h`. You will find it in the `\src` folder. You should place a copy of `unicoder.h` either in the folder with your source or in the compiler's `\include` folder. Before using any of the functions or macros in *Unicoder* you need to insert one of the following lines near the beginning of your app:

```
#include "unicoder.h"
```

or

```
#include <unicoder.h>
```

This header file will automatically include all necessary header files needed by *Unicoder.*

2.2 Unicode

The entire purpose for using *Unicoder* is to make it easy for you to include and use wide characters and Unicode in your C and C++ terminal applications. So what is Unicode?

Unicode is a set of standards that define how text is represented and encoded. In the Terminal text is displayed in a grid of *mono-spaced* cells; one space per character or symbol. Unicode has several methods of encoding; *Unicoder* uses UTF-8 and UTF-16. Both UTF-8 and UTF-16 have variable length encoding. UTF-8 requires between one and four bytes to represent all of the Unicode characters while UTF-16 requires either one or two 16-bit values to represent all of the Unicode characters.

The Windows Command Prompt Terminal was designed more then 25 years ago to represent each text character as a fixed length 16-bit value. Thus, this Windows Terminal text buffer contains 2-byte `wchar_t` per grid cell. Since the Windows platform `wchar_t` represents one 16-bit unit of memory the UTF-16 encoding fits it's needs. It should be noted that the new *Windows 10 Terminal* is Microsoft's combined replacement for the Command Prompt and Powershell applications and it does support UTE-8 encoding. On Linux `wchar_t` represents a 32-bit (4 bytes) unit so for it UTF-8 works best.

The procedure that selects the proper encoding is unique to the compiler being used. Setting the correct mode based on the platform and compiler is handled automatically by the function `udispOn()`. Because this function handles this you are free to change to different compilers and platforms without having to re-write your source code. All that is required is for you to re-compile your app.

2.2.1. Unicode character set

The Ubuntu Linux and Windows Terminals use different default fonts, however, the Unicode characters that they contain are basically the same. This means you can compile your app on both platforms and it will display correctly on both platforms without the need to change the default Terminal font.

However, these default Terminal fonts are only a small subset of the Unicode character set. *Unicoder* allows you to optionally use a larger Unicode character set which has significantly more scientific and

mathematical symbols. But you will be required to have the end user of your app change their Terminal's font in order to display the output correctly.

2.3 Getting started

Let's begin by writing a simple Terminal application that will display the equation: `f(x) = α²+ β³`. Place a copy of the header `unicoder.h` in the folder along with the source code for `myfirst.c`.

Listing 2.1: The `myfirst.c` Program

```
#include "unicoder.h"

int main(void)
{
    umode_t oldmode;

    oldmode = udispOn();                    // Turn on unicode display mode.
    wprintf(L"Unicode: f(x) = %lc%lc+ %lc%lc\n",U_alpha,U_squared,U_beta,U_cubed);
    udispOff(oldmode);

    return 0;
}
```

2.3.1. The example explained

To turn on the Unicode display for a Ubuntu Linux or Windows application you just need to use the function `udispOn()`. The previous mode will be returned as a `umode_t` type, which can be either a `int` or a pointer to a `char`, depending on the compiler. You can use the macro `umodetype()` to find out which type was returned. Once the Unicode display mode has been turned on you can use the C wide character functions like `wprintf()` to output Unicode characters to the display.

Notice that in `wprintf()` the `L` specifier appears before the first quotation mark. The `L` preceding the quotation mark tells the compiler that you want it to be stored as two-bytes. For C++ apps you use the `L` specifier like this:

```
wcout << L"Hello, World!" << endl;
```

Wide character constants and string literals are also indicated with an `L`. Examples are:

```
wchar_t wch = L'I';
wchar_t warr[20] = L"wide char!";
wchar_t * pw = L"points to a wide-character string";
wchar_t w = L'\u00E2';     // A 16-bit character code
```

Additionally, you should use `%lc` instead of `%c` (and `%ls` instead of `%s`) in your print statements to denote a wide (`long`) character. This is required for Linux but is optional for some Windows Terminals. The macro names prefixed with `U_` are *Unicoder* definitions, see tables in sections 4 and 5.

The last statement `udispOff(oldmode)` returns the display to it;s previous display mode (maybe!).

2.4 Using a Unicode character set

Now that you know how to get the Terminal to display the Unicode characters correctly by changing to Unicode mode, the next step is to find out how to display any character you want. There are several ways to do this.

With *Unicoder* you can use any font set you want as long as they are *mono-spaced* unicode. The Ubuntu Linux Terminal by default uses `Ubuntu Mono` and the default Windows Terminal uses the `Consolas` font.

These two fonts contain many of the same characters so you can create a universal app using the *Unicoder* Default Character Set (section 4) that will work on both platforms without requiring any font changes. Neither of these fonts can display all of the characters defined by *Unicoder*. Both the Ubuntu Linux and the Windows Terminals have the `DejaVu Sans Mono` character set available and this expanded character set is able to display many more Unicode characters that *Unicoder* has defined. See section 2.6 for instruction on how you can include this expanded Unicode character set and 2.7 for instructions on changing the Terminal font.

2.5 Character Map

To see all of the available Unicode characters in a font you need to load them into a viewer. In Windows and Ubuntu the viewer is the Character Map utility. These utilities display all of the available characters that your Terminal application can use. For Ubuntu set the viewer to `View>By Unicode Block>Miscellaneous>Character Table` tab. In Windows set the viewer to `consolas` then using the mouse scroll to the very bottom of the font set. In either platform find the happy face character, the ☺ and select it. Notice at the lower left of the viewer it displays: `U+263A: White Smiling Face`. We are interested in the code `263A`. This is the Unicode hexadecimal number of that character, in terms of C and C++ we write this as `0x263A`. It is this number that we will send to the Terminal to display this character.

Enter the following program to see how you manually add any character, including ones not defined by *Unicoder*, to your app. Here we will also display the ace of spades character: ♠, Unicode `0x2660`.

Listing 2.2: The `happyace.c` Program

```
#include "unicoder.h"

int main(void)
{
    #define SPADE 0x2660

    udispOn();
    wprintf(L"Don't worry, be happy! \u263A\n");
    wprintf(L"An Ace of %lc\n",SPADE);

    return 0;
}
```

Besides defining a single character you can also create a wide character string like: ♠♥♣♦

```
    wchar_t suits[] = {0x2660,0x2665,0x2663,0x2666,0x0};
```

Now that you've seen the hard way to add Unicode characters to your apps this next program let;s you use *Unicoder* to simplify this process. *Unicoder* has defined a large number of Unicode characters in a number of tables that you will find in sections 4 and 5 so all you need to do is add their macro names to your programs. Here is the same program as above using *Unicoder's* tables.

Listing 2.3: The `happyace2.c` Program

```
#include "unicoder.h"

int main(void)
{
    udispOn();
    wprintf(L"Don't worry, be happy! %lc\n",U_lthappy); // Table 4.6
    wprintf(L"An Ace of %lc\n",U_spades);               // Table 4.7

    return 0;
}
```

Note that all the characters are defined using the `U_` prefix (for Unicode). The tables in section 4 and 5 do not include all the characters in the Unicode character set so for characters not included in these tables you will need to manually look up the character code and add it to your app as was done in *Listing 2.2*.

2.6 Using an expanded Unicode character set

In Windows the default character set for the Terminal is `Consolas` and in Ubuntu it is `Ubuntu Mono`. The unicode characters that have been defined in both of them are almost identical. So you can write apps using the Unicoder Default Character Set that will work on both systems without requiring the User to change the Terminal font.

If, however, you need to make use of characters not available with these default character sets, you can switch to the expanded unicode character set available with `DejaVu Sans Mono`. This font is available on both the Windows and the Ubuntu Linux Terminals. To use the expanded Unicode set simply place `#define EXTENDED_UNICODE` before the `#include <unicoder.h>` at the beginning of your code. See *Listing 2.4* for an example. You also must change the Terminal font for it to display correctly, see 2,7.

To make this change permanent just uncomment `#define EXTENDED_UNICODE` inside the `unicoder.h` header and save the changes.

Listing 2.4: The `checkmate.c` Program

```
#define EXTENDED_UNICODE
#include "unicoder.h"

int main()
{
    udispOn();
    wprintf(L"Checkmate! %lc %lc\n",U_whking,U_blkqueen);

    return 0;
}
```

Note that some characters defined with the default font `Consolas` and `Ubuntu Mono` are not defined with `DejaVu Sans Mono` so you may need to find another font that works for your app.

2.7 Changing the Terminal font

To change the Terminal font start up the Terminal. In Windows right click the icon located at the upper left corner of the Terminal window. In the menu that appears select the `Properties` menu item then select the `Font` tab. Select the font to use (`DejaVu Sans Mono`). If you are using Ubuntu Linux click on ☰ to open the menu. Select `Preferences>Profile`, then select your current profile, check the `Custom font` box and select a font (`DejaVu Sans Mono Book`).

Unicoder is not limited to only the three fonts mentioned in this document. It can use any *fixed-width* font. So that you can easily determine what character sets you can use with *Unicoder* you can compile and run the `fonttest.c` program included in the `\src` folder. This app will display all the default fonts; missing fonts will appear either as blank spaces or as rectangular squares. Change fonts until you find one that works.

2.8 Displaying 8-bit char types

In Windows if you want to display an 8-bit `char` type after you have turned on the wide character mode you will now run into problems because the Terminal is expecting to receive 16-bit characters. This does not mean you can't output 8-bit characters. The trick here is to use a capital `%S` to tell `wprintf()` that the string contains 8-bit characters. For a single 8-bit character you use a capital `%C`. For a Ubuntu Linux app you can simply use the standard lower case letters `%s` and `%c` in the `wprintf()` statement. Keep in mind that this is a difference that will not allow you to write a generic app that can be compiled on both platforms unless you include a `#ifdef _WIN32` and `#ifdef __linux__` section.

The Windows and Ubuntu Linux Terminals react differently to mixing wide characters and 8-bit character functions. For a Windows app you can use the 8-bit output functions both before turning on wide character mode using `udispOn()` and after turning it off using `udispOff()` but not in-between. With Ubuntu Linux if you start using 8-bit output functions like `printf()` the Terminal will continue to use the 8-bit mode even after you turn on the wide character mode using `udispOn()` and if you turn on the wide character mode first the Ubuntu Linux Terminal will continue to use wide characters even if you turn it off using `udispOff()`.

So it is best that your app only uses wide character functions exclusively. To convert an 8-bit character string to a wide character string use the macro `strtowstr().`

2.9 Other display functions

Unicoder also includes several macros that can be used to change how your app will display characters such as using bold characters, underlined characters and colored fonts and backgrounds. There are a number of different procedures for carrying out these functions which depend on the compiler being used. So that your source code can seamlessly be compiled using different compilers without having to be modified (i.e. be portable) *Unicoder* automatically substitutes the required compiler specific code when you use the *Unicoder* macros listed in sections 3.2 and 3.3 .

Listing 2.5: The `formats.c` Program

```
#include "unicoder.h"

int main()
{
    umode_t oldmode;

    oldmode = udispOn();

    boldOn();
    wprintf(L"This text is bright (bold)\n");
    formatOff();
    wprintf(L"This text has returned to default\r\n");
    underlineOn();
    wprintf(L"Underlined text\n");
    underlineOff();
    redfontOn();
    wprintf(L"Font is RED\n");
    greenfontOn();
    wprintf(L"%lc, %lc, %lc\n",U_alpha, U_beta,U_gamma);
    bluebkgdOn();
```

```
    wprintf(L"Hard to read this\n");
    formatOff();
    wprintf(L"Return to default\n");
    udispOff(oldmode);
    return 0;
}
```

2.10 Drawing

Unicoder provides several line drawing functions, which are listed in section 3.4 These functions will draw a line segment *n* characters wide with or without a newline (`Ln`). The function `rowDraw()` can be used to draw a row made up of any glyph or glyphs repeated as many times as needed.

2.10.1. C-Style and P-Style

To use the `rowDraw()` function you pass a variable length two dimensional array of type `const long` to the function. The general form of this array is:

```
const long array[][2];
```

The function `rowDraw()` will accept two styles of the array data, a C-style where the last array element contains zeros `{0,0}` and a Pascal-style where the very first element in the array is always set to the number of glyphs `n` in the array, that is

```
array[0][1] = n
```

The other element (`array[0][0]`) must be set to `0`. For example, a P-style array can be declared like this:

```
const long MyArray[4][2] = {{0,3},{U_hearts,3},{U_spades,20},{U_diamonds,1}};
```

or alternatively

```
const long MyArray[][2] = {{0,3},{U_hearts,3},{U_spades,20},{U_diamonds,1}};
```

This example array has defined three glyphs:`U_hearts`, `U_spades` and `U_diamonds`. The first element `{0,3}` tells the function that this array is a Pascal-type array because of the zero in `MyArray[0][0]` and `MyArray[0][1]` contains the number of glyphs that are defined in the array (`3`).. The second element `{U_hearts,3}` contains the first glyph `U_hearts` and the number of times that glyph should be drawn; in this case three times. The third element contains the second glyph which should be drawn 20 times and lastly the fourth element contains the last glyph which should also be drawn one time. If you prefer you can, instead use a null-terminated C-style array which you declare like this:

```
const long MyArray[][2] = {{U_hearts,3},{U_spades,20},{U_diamonds,1},{0,0}};
```

These arrays can be of any length you need. You would then draw this line with the call:

```
rowDraw(MyArray);
```

You are not limited to only drawing glyphs, you can include characters within a line: ♥♥♥Hiiiii♥♥♥♥

```
const long Greetings[][2] = {{0,4},{{U_hearts,3},{L'H',1},{L'i',5},{U_hearts,4}};
```

2.11 Combining characters

At this time *Unicoder* does *not* support combining characters. This is mainly due to a Windows limitation. For Windows the Terminal uses legacy functions dating back to Windows NT. 3.1 (1993). Microsoft is currently updating the entire Terminal and has hinted that they *may* resolve this issue in a future release. In the mean time avoid using any combining characters.

2.12 Homoglyphs

Homoglyphs are characters (or glyphs) with a shape that appears identical or very similar. *Unicoder* does not make any distinction between homoglyphs. That is, it only defines a character once even if it has many homoglyphs defined in the unicode character set (but it may show that character in the tables several times using different macro names). So if you need a specific character and can't find it look around in other tables to see if it is defined elsewhere.

2.13 Missing characters

Unicoder doesn't define every character that may be available in a font set. Therefore, if all else fails and you can't find a particular character then manually look for the character by scrolling through the font using the `Character Map` utility (outlined in section 2.5). If the character is found add it to your app as was done in *Listing 2.2* or if you use it a lot define it in `unicoder.h`.

2.14 Version Info

Version information for *Unicoder* is contained in the wide string macro `UNICODER_VERSION`. The version consists of the two digit year followed by a decimal point and then the month of the release.

2.15 Portability

Your compiler and linker add *startup code* to set up the run-time environment. This startup code includes functions to display wide-characters, including Unicode, colored fonts and backgrounds, underlined and bold fonts within the Terminal. Since it is the compiler/linker that supply the startup code, what the Terminal is able to display and how you go about displaying it depends on how the creator of the compiler decided to implement it. However, a problem with porting your app may occur at this point if the new compiler doesn't support all the features you need. So if you want to be able to compile your code using different compilers you may need to modify your code.

Another problem with portability is that the wide-character function names are not universally recognized by all compilers. Some have not yet incorporated all of the C11 wide-character functions while others have come up with variations to functions that they want you to use (Microsoft being a big offender). So this again makes it difficult to create a portable app.

Unicoder attempts to hide the differences compilers use to perform similar tasks from you by defining it's own macros and inline functions for both displaying fonts and Unicode and for some wide-character functions. By using the *Unicoder* macros and inline function calls given in section 3 instead of the compiler specific calls or the wide-character calls, when you compile your app *Unicoder* will insert the appropriate code that is tailored to the compiler being used. Thus making it easy for you to port your app to different compilers or platforms.

However, there are instances in which *Unicoder* is not able to determine the compiler or provide definitions that work for a particular compiler. In these cases you may have to change to a different compiler. A list of compilers that have been tested and the functions that work is provided in the `README` file.

3 Unicoder Macros and Inline Functions

The header `unicoder.h` declares types, inline functions and macros for use in manipulating wide-characters. The type `umode_t` represents the unicode display mode of the Terminal. In Linux `umode_t` represents a pointer to a character (`char *`). It can be overwritten by subsequent calls to `udispOn()` and `udispOff()` so you should make a copy of it. In Windows `umode_t` represents either an integer value or a `char *`, depending on the compiler used. NOTE: Not all macros/functions work on all platforms.

3.1 Terminal mode control

prototype	description
`umode_t udispOn(`*`void`*`)`	This function can be used by both Ubuntu Linux and Windows 7/10 to turn on the correct unicode display mode automatically. In Ubuntu Linux it returns a pointer to a character (*char* *) while in Windows it returns an *int* or a *char* *.
`umode_t utf8On(`*`void`*`)`	Used to specifically turn on UTF-8 encoding by Windows 10
`umode_t udispOff(`*`umode_t`*`)`	This function tells the Terminal to switch the output mode to *umode_t* and returns the previous output mode.
`int umodetype(umode_t)`	Returns type of display mode: 0 = int, 1 = char *, -1 = ?

3.2 Terminal format control

prototype	description
`void scrnClr()`	Clears the entire Terminal screen.
`void boldOn(void)`	Turns on the bold (bright) character display.
`void boldOff(void)`	Turns off the bold (bright) character display.
`void underlineOn(void)`	Adds an underline to the characters.
`void underlineOff(void)`	Stops underlining characters.
`void reverseTextOn(void)`	Reverses the font and background colors.
`void reverseTextOff(void)`	Changes font and background back to previous colors.
`void formatOff(void)`	Turns off all character format modes including colors.

3.3 Terminal color control

prototype	description
`void blackfontOn(void)`	Font color is black.
`void redfontOn(void)`	Font color is red.
`void greenfontOn(void)`	Font color is green.
`void yellowfontOn(void)`	Font color is yellow.
`void bluefontOn(void)`	Font color is blue.
`void magentafontOn(void)`	Font color is magenta.
`void cyanfontOn(void)`	Font color is cyan.
`void whitefontOn(void)`	Font color is white.
`void blackbkgdOn(void)`	Background color is black.
`void redbkgdOn(void)`	Background color is red.
`void greenbkgdOn(void)`	Background color is green.
`void yellowbkgdOn(void)`	Background color is yellow.
`void bluebkgdOn(void)`	Background color is blue.
`void magentabkgdOn(void)`	Background color is magenta.
`void cyanbkgdOn(void)`	Background color is cyan.
`void whitebkgdOn(void)`	Background color is white.

3.4 Drawing functions

Drawing names using the Ln suffix return a newline.

prototype	description
void thinlineDraw(int *cnt*) or void thinlineDrawLn(int *cnt*)	Draws a solid unbroken thin line repeated *cnt* times.
void boldlineDraw(int *cnt*) or void boldlineDrawLn(int *cnt*)	Draws a solid bold unbroken line repeated *cnt* times.
void dashlineDraw(int cnt) or void dashlineDrawLn(int cnt)	Draws a dashedline repeated *cnt* times.
void dbllineDraw(int cnt) or void dbllineDrawLn(int cnt)	Draws a solid double unbroken line repeated *cnt* times.
void rowDraw(long arr[][2]) or void rowDrawLn(long arr[][2])	Draws glyphs defined in a two dimensional array arr[][2].

3.5 Redefined wide character macros

A few of the wide character functions have been redefined as macros wuth names that more closely resemble the C library function names you are familiar with from K&R C. You can still use the wide character function names as define by the C/C++ languages but be aware that some compilers use non-standard function calls so using these re-defined names insures cross compiler compatibility.

3.5.1. String macros

prototype	description
wchar_t *wstrcpy(wchar_t *ws, const wchar_t *ct)	copies string *ct* to string ws including '\0'; returns ws.
wchar_t *wstrncpy(wchar_t *ws, const wchar_t *ct, size_t *n*)	copies at most *n* characters of string *ct* to ws; return ws.
wchar_t *wstrcat(wchar_t *ws, const wchar_t *ct)	concatenates string *ct* to end of string ws; returns ws.
wchar_t *wstrncat(wchar_t *ws, const char *ct, size_t *n*)	concatenates at most *n* characters of string *ct* to string_ ws, terminate ws with '\0'; returns ws.
int wstrcmp(const wchar_t *ws, const char_t *ct)	compares string ws to string *ct*; returns <0 if ws<*ct*; 0 if ws==*ct* or >0 if ws>*ct*.
int wstrncmp(const wchar_t *ws, const char_t *ct, size_t *n*)	compares at most *n* characters of string ws to string *ct*; returns <0 if ws<*ct*; 0 if ws==*ct* or >0 if ws>*ct*.
wchar_t *wstrchr(wchar_t *ws, wchar_t *wc*)	returns pointer to first occurrence of wc in ws or NULL if not present.
wchar_t *wstrrchr(wchar_t *ws, wchar_t *wc*)	returns pointer to last occurrence of wc in ws or NULL if not present.
size_t wstrspn(const wchar_t *ws, const wchar_t *ct*)	returns length of prefix of ws consisting of wide charac ters in ct.

prototype	description
size_t wstrcspn(const wchar_t *ws, const wchar_t *ct)	Scans ws for first occurrence of any character found in ct
wchar_t *wstrpbrk(const wchar_t *ws, const wchar_t *ct)	return pointer to first occurrence in wide string ws of any character in string ct, or NULL if none is present.
wchar_t *wstrwstr(const wchar_t *ws, const wchar_t *ct)	returns pointer to first occurrence of wide string ws in wide string ct or NULL if not present.
size_t wstrlen(const wchar_t *ws)	returns the length of ws.
wchar_t wstrtok(wchar_t *ws, const wchar_t *ct,wchar_t **endp)	Searches ws for separator wide characters in ct. wstrtok is meant to be called multiple times to obtain successive tokens. *endp* is a used to store it's internal state.
errno_t strtowstr(wchar_t *ws,const char *s, size_t *sz)	converts a char string *s* into a wchar_t string *ws* with a terminating L'\0'. The number of wide characters is in sz.

For your convenience the mem functions for manipulating objects as wide character arrays is given here.

prototype	description
wchar_t *wmemcpy(wchar_t *s, const wchar_t *ct, size_t n)	copy *n* wide characters from *ct* to *s* and returns *s*. Does not add a terminating '\0' character.
wchar_t *wmemmove(wchar_t *s, const wchar_t *ct, size_t n)	same as wmemcpy except that it works even if the objects overlap.
int wmemcmp(const wchar_t *cs, const wchar_t *ct, size_t n)	compares the first *n* wide characters of *cs* with *ct*, returns as with wstrcmp.
wchar_t *wmemchr(const wchar_t *cs, wchar_t c, size_t n)	returns pointer to first occurance of wide character *c* in *cs* or NULL if not present among the first *n* wchar_t.
wchar_t *wmemset(wchar_t *cs, wchar_t c, size_t n)	place wide character *c* into first *n* wide characters of *cs*, returns *cs*.

3.5.2. Utility macros

prototype	description
double watof(const wchar_t *ws)	watof converts ws to a double.
int watoi(const wchar_t *ws)	watoi converts ws to a int.
long watol(const wchar_t *ws)	watol converts ws to a signed long.

prototype	description
float wstrtof(const wchar_t *ws, wchar_t **endp)	converts the prefix of ws to float, ignoring leading white spaces; if not NULL sets *endp* to point to the first wide character after the last valid character of the wide string if any, otherwise the pointer is set to NULL.
double wstrtod(const wchar_t *ws, wchar_t **endp)	wstrtod is the same as wstrtof except that the result is double.
long wstrtol(const wchar_t *ws, wchar_t **endp, int *base*)	converts the prefix of ws to signed long, ignoring leading white spaces; sets *endp* to point to the first wide charac ter after the last valid character of the wide string if_ there is any, otherwise the pointer is set to NULL.
long long wstrtoll(const wchar_t *ws, wchar_t **endp, int *base*)	wstrtoll is the same as wstrtol except that the result is signed long long.
unsigned long wstrtoul(const wchar_t *ws, wchar_t **endp, int *base*)	wstrtoul is the same as wstrtol except that the result is unsigned long.
unsigned long long wstrtoull(const wchar_t *ws, wchar_t **endp, int *base*)	wstrtoull is the same as wstrtol except that the result is unsigned long long.
wchar_t *itowstr(*int n*, wchar_t *ws, int *base*)	converts int *n* to a null terminated wide string ws. Uses *base* as the number base to use to convert the int.
wchar_t *ltowstr(*long n*, wchar_t *ws, int *base*)	converts long *n* to a null terminated wide string ws. Uses *base* as the number base to use to convert the long int.
wchar_t *ultowstr(*unsigned long n*, wchar_t *ws, int *base*)	converts unsigned long *n* to a null terminated wide string ws. Uses *base* as the number base used to convert the un- signed long int.
wchar_t *lltowstr(*long long n*, wchar_t *ws, int *base*)	converts long long *n* to a null terminated wide string ws. Uses *base* as the number base used to convert the long long int.
wchar_t *ulltowstr(*unsigned long long n*, wchar_t *ws, int *base*)	converts unsigned long long *n* to a null terminated wide string ws. Uses *base* as the number base used to convert the unsigned long long int.
int to_wstring(const wchar_t *ws, *x*)	converts any type *x* into a wide character string, stores the string in ws. Returns the string's length. This macro is defined for C and C++. This macro is not available for non-C11 compilers.
wchar_t *to_wchar_t(wstring)	converts from C++ std::wstring to const wchar_t.

3.5.3. Character Class Tests

These functions used for testing wide characters are referenced here for your convenience.

prototype	description
`int iswalnum(wint_t wc)`	returns true if wc is an alphanumeric wide character.
`int iswalpha(wint_t wc)`	returns true if wc is an alphabetic wide charavter.
`int iswcntrl(wint_t wc)`	returns true if wc is a control character.
`int iswdigit(wint_t wc)`	returns true if wc represents a digit.
`int iswgraph(wint_t wc)`	returns true if wc is a printing chacacter except a space.
`int iswlower(wint_t wc)`	returns true if wc represents a lower-case letter.
`int iswprint(wint_t wc)`	returns true if wc represents a printable character.
`int iswpunct(wint_t wc)`	returns true if wc represents a punctuation character.
`int iswspace(wint_t wc)`	returns true if wc represents a tab, space or neline.
`int iswupper(wint_t wc)`	returns true if wc represents an upper-case character.
`int iswxdigit(wint_t wc)`	returns true if wc represents a hexadecimal digit.
`int iswblank(wint_t wc)`	returns true if wc represents a blank.

In addition, there are two functions that convert the case of letters.

`wint_t towlower(wint_t wc)`	converts wc to lowercase.
`wint_t towupper(wint_t wc)`	converts wc to uppercase.

3.5.4. Standard Output

More wide character functions listed here for your convenience.

prototype	description
`int wprintf(const wchar_t *format, ...)`	prints formatted data to stdout.

4 Unicoder Default Character Set

In the tables that follow the unicode characters have been bound to names which you can use to include them in your app as shown in *Listing 2.4*. All names are prefixed with the Unicode tag U_.

4.1 Greek

Lower case				Upper case		Lower case		Upper case	
	Letter	Superscript	Subscript		Letter		Letter		Letter
α	U_alpha	U_supalpha		Α	U_Alpha	ζ	U_zeta	Ζ	U_Zeta
β	U_beta	U_supbeta	U_subbeta	Β	U_Beta	η	U_eta	Η	U_Eta
γ	U_gamma	U_supgamma	U_subgamma	Γ	U_Gamma	κ	U_kappa	Κ	U_Kappa
δ	U_delta	U_supdelta		Δ	U_Delta	λ	U_lambda	Λ	U_Lambda
ε	U_epsilon	U_supepsilon		Ε	U_Epsilon	μ	U_mu	Μ	U_Mu
θ	U_theta	U_suptheta		Θ	U_Theta	ν	U_nu	Ν	U_Nu
ι	U_iota	U_supiota		Ι	U_Iota	ξ	U_xi	Ξ	U_Xi
ρ	U_rho		U_subrho	Ρ	U_Rho	ο	U_omicron	Ο	U_Omicron
υ	U_upsilon	U_supupsilon		Υ	U_Upsilon	π	U_pi	Π	U_Pi
φ	U_phi	U_supphi		Φ	U_Phi	σ	U_sigma	Σ	U_Sigma
χ	U_chi	U_supchi	U_subchi	Χ	U_Chi	τ	U_tau	Τ	U_Tau
ψ	U_psi	U_suppsi	U_subpsi	Ψ	U_Psi	ω	U_omega	Ω	U_Omega
ϐ	U_varbeta					ϗ	U_kai	Ϗ	U_Kai
ϰ	U_vargamma					ϙ	U_koppa	Ϙ	U_Koppa
ϵ	U_varepsilon					ϝ	U_digamma	Ϝ	U_Digamma
ϑ	U_vartheta	U_varsuptheta		ϴ	U_VarOmega	ϡ	U_sampi	ϡ	U_Sampi
ς	U_varsigma			ς	U_VarSigma	ϧ	U_heta	Ϧ	U_Heta
ϒ	U_varupsilon			ϒ	U_VarUpsilon				
φ	U_varphi	U_varsupphi		ϰ	U_varGamma	ς	U_finalsigma		
ϟ	U_varkoppa			ϟ	U_VarKoppa				

4.2 Modified Greek

Lower case				Upper case	
	2nd derv	with bar	with tilde		2nd derv
α		U_baralpha	U_alphatilde		
θ	U_theta2nd			Ө	U_Theta2nd
ι	U_iota2nd		U_iotatilde	Ї	U_Iota2nd
υ	U_upsilon2nd	U_barupsilon	U_upsilontilde	Ӳ	U_Upsilon2nd
ω		U_baromega	U_omegatilde		

4.3 Mathematical symbols

Symbol		Superscript		Subscript		Symbol			
−	U_minussign	⁻	U_supminus	₋	U_subminus	∂	U_partial	∞	U_infty
±	U_pm	⁺	U_supplus	₊	U_subplus	∑	U_sum	∅	U_emptyset
≡	U_equiv	⁼	U_supequal	₌	U_subequal	∏	U_prod	∃	U_exists
≠	U_neq	⁽	U_supintgl			∫	U_intgl	†	U_dagger
≈	U_approx	Symbol				√	U_sqrt	‡	U_dbldagr
‹	U_less	≤	U_leq	«	U_ll	Δ	U_increment	∩	U_intersect
›	U_greater	≥	U_geq	»	U_gg	ℓ	U_lnell	∀	U_forall
×	U_times	⨯	U_crossprod	⋆	U_dotprod	⅂	U_esh	⊖	U_circldot
÷	U_div	⁄	U_divslash	⋅	U_cdot	¬	U_neg	⌐	U_revneg
²	U_squared	³	U_cubed	ª	U_ordinal	∈	U_belongsto	∋	U_contains
′	U_prime	″	U_diprime	‴	U_triprime	⌠	U_intgltop	⌡	U_intglbtm
⊤	U_transpose	⌊	U_rightangle	º	U_masordin	⌜	U_lpartop	⌞	U_lparbtm
⅁	U_imaginary	℮	U_eulersnum	℮	U_estimate	⌝	U_rpartop	⌟	U_rparbtm

4.4 Numbers

	Superscript	Subscript	Lite Circled	Dark Circled		Lite Circled	Dark Circled
0	U_sup0	U_sub0	U_ltcircle0	U_dkcircle0	10	U_ltcircle10	U_dkcircle10
1	U_sup1	U_sub1	U_ltcircle1	U_dkcircle1	11	U_ltcircle11	U_dkcircle11
2	U_sup2	U_sub2	U_ltcircle2	U_dkcircle2	12	U_ltcircle12	U_dkcircle12
3	U_sup3	U_sub3	U_ltcircle3	U_dkcircle3	13	U_ltcircle13	U_dkcircle13
4	U_sup4	U_sub4	U_ltcircle4	U_dkcircle4	14	U_ltcircle14	U_dkcircle14
5	U_sup5	U_sub5	U_ltcircle5	U_dkcircle5	15	U_ltcircle15	U_dkcircle15
6	U_sup6	U_sub6	U_ltcircle6	U_dkcircle6	16	U_ltcircle16	U_dkcircle16
7	U_sup7	U_sub7	U_ltcircle7	U_dkcircle7	17	U_ltcircle17	U_dkcircle17
8	U_sup8	U_sub8	U_ltcircle8	U_dkcircle8	18	U_ltcircle18	U_dkcircle18
9	U_sup9	U_sub9	U_ltcircle9	U_dkcircle9	19	U_ltcircle19	U_dkcircle19
					20	U_ltcircle20	U_dkcircle20

4.5 Fractions

⅛	U_oneeighth	⅙	U_onesixth	⅕	U_onefifth	¼	U_onefourth	⅓	U_onethird	½	U_onehalf
⅜	U_threeeighths			⅖	U_twofifths			⅔	U_twothirds		
⅝	U_fiveeighths	⅚	U_fivesixths	⅗	U_threefifths	¾	U_threequarter				
⅞	U_seveneighths			⅘	U_fourfifths						

4.6 Symbols

Units		Symbols		Arrows & Arrowheads								Glyphs	
Å	U_Angstrom	℅	U_careof	←	U_Warrow	↓̲	U_subleftarw	<	U_lefthead	☺	U_lthappy
°	U_Degree	©	U_copyright	→	U_Earrow	⎯→	U_combsubrgtarw	>	U_rgthead	☻	U_dkhappy
µ	U_Micro	№	U_numero	↑	U_Narrow	↑̲	U_supNarrow	ˆ	U_subuphead	✶	U_dkstar
℧	U_Mhos	‰	U_permile	↓	U_Sarrow	↓̲	U_supSarrow	ˇ	U_subdownhead	♀	U_female
Ω	U_Ohms	®	U_registered	↔	U_WEarrow	↔̲	U_combosublrarw	‹	U_sublefthead	♂	U_male
ƛ	U_Wavelen	℗	U_soundrecd	↕	U_NSarrow			›	U_subrgthead	☼	U_sunshine
ħ	U_RedPlanck	™	U_trademark	^	U_uphead	v	U_downhead			⌂	U_house

4.7 Geometric shapes

Black Objects				White Objects				Cards	
■	U_blacksqr	▪	U_blacksqrsm	□	U_box	▫	U_boxsm	♠	U_spades
●	U_blackcircle	◉	U_inversecircle	○	U_circle	◌	U_litecircle	♣	U_clubs
▬	U_blacksqrlow	■	U_blacksqrhigh	֍	U_swirl	◊	U_lozenge	♦	U_diamonds
▉	U_blackblock	▓	U_darkblock	▒	U_mediumblock	▒	U_liteblock	♥	U_hearts
▲	U_blktriangle	▴	U_blktrianglesm					Music	
▼	U_blkdwnarw	▾	U_blkdwnarwsm					♪	U_8thnote
►	U_blkrgtarw	▸	U_U_blkrgtarwsm					♬	U_beamed8th
◄	U_blklftarw	◂	U_blklftarwsm					♯	U_sharp

4.8 Characters

Typographical		Dots		Punctuation							
§	U section	̣	U lowdot	“	U leftquot	‘	U leftsnglquot	(	U supopen	₍	U subopen
¶	U_paragraph	·	U_highdot	”	U_rgtquot	’	U_rgtsnglquot	)	U_supclose	₎	U_subclose
‿	U tie	̇	U 2vtdots	„	U lowquot	‚	U lowsnglquot	~	U suptilde	~	U subtilde
•	U_bullet	⋮	U_4vtdots	“	U_reversed	‛	U_revsnglquot	¡	U_supinvtexclm	!	U_supexclm
◦	U litebullet	¨	U diaeresis	ˇ	U caron	´	U acute	¡	U invertexclm	‼	U dblexclm
		…	U_ellipsis		U_cedilla	⸗	U_obliquehyphen	¿	U_invertquestn	¡	U_exclmsm

4.9 Latin Modifiers Lower case

	Super-script	Sub-script		Super-script	Sub-script		Super-script	Sub-script		Super-script	Sub-script		Super-script	Sub-script		Dot-less
a	U_supa	U_suba	f	U_supf		k	U_supk		p	U_supp		v	U_supv	U_subv	ɪ	U_dli
b	U_supb		g	U_supg		l	U_supl		r	U_supr	U_subr	w	U_supw		ʝ	U_dlj
c	U_supc		h	U_suph		m	U_supm		s	U_sups		x	U_supx	U_subx		
d	U_supd		i	U_supi	U_subi	n	U_supn		t	U_supt		y	U_supy			
e	U_supe	U_sube	j	U_supj	U_subj	o	U_supo	U_subo	u	U_supu	U_subu	z	U_supz			

4.10 Latin Modifiers Upper case

Superscript									
A	U_supA	G	U_supG	K	U_supK	O	U_supO	U	U_supU
B	U supB	H	U supH	L	U supL	P	U supP	V	U supV
D	U supD	I	U supI	M	U supM	R	U supR	W	U supW
E	U_supE	J	U_supJ	N	U_supN	T	U_supT		

4.11 Latin Modified

Lower case						Upper case					
	1st derv	2nd derv	w/hat	w/bar	w/tilde		1st derv	2nd derv	w/hat	w/bar	w/tilde
a	U a1st	U a2nd	U ahat	U abar	U atilde	A	U A1ST	U A2ND	U Ahat	U Abar	U Atilde
b	U b1st					B	U B1ST				
c	U c1st		U chat			C	U C1ST		U Chat		
d	U d1st					D	U D1ST				
e	U e1st	U e2nd	U ehat	U ebar	U etilde	E	U E1ST	U E2ND	U Ehat	U Ebar	U Etilde
f	U f1st					F	U F1ST				
g	U g1st		U ghat	U gbar		G	U G1ST		U Ghat	U Gbar	
h	U h1st	U h2nd	U hhat			H	U H1ST	U H2ND	U Hhat		
i	U i1st	U i2nd	U ihat	U ibar	U itilde	I	U I1ST	U I2ND	U Ihat	U Ibar	U Itilde
j	U j1st		U jhat			J			U Jhat		
m	U m1st					M	U M1ST				
n	U n1st				U ntilde	N	U N1ST				U Ntilde
o	U o1st	U o2nd	U ohat	U obar	U otilde	O	U O1ST	U O2ND	U Ohat	U Obar	U Otilde
p	U p1st					P	U P1ST				
r	U r1st					R	U R1ST				
s	U s1st		U shat			S	U S1ST		U Shat		
t	U t1st	U t2nd				T	U T1ST				
u		U u2nd	U uhat	U ubar	U utilde	U		U U2ND	U Uhat	U Ubar	U Utilde
v					U vtilde	V					U Vtilde
w	U w1st	U w2nd	U what			W	U W1ST	U W2ND	U What		
x	U x1st	U x2nd				X	U X1ST	U X2ND			
y	U y1st	U y2nd	U yhat	U ybar	U ytilde	Y	U Y1ST	U Y2ND	U Yhat	U Ybar	U Ytilde
z	U z1st		U zhat			Z	U Z1ST		U Zhat		

4.12 Currency

₳	U_austral	₫	U_dong	₴	U_hryvnia	₥	U_mill	£	U_pound	₸	U_tenge
฿	U_baht	₯	U_drachma	₹	U_indiarupee	₦	U_naria	₽	U_ruble	₮	U_tugrik
₿	U_bitcoin	€	U_euro	₭	U_kip	₧	U_peseta	₨	U_rupee	₺	U_turkeylira
¢	U_cent	₣	U_franc	£	U_lira	₱	U_peso	₪	U_shekel	₩	U_won
¤	U_currency	₲	U_guarani	₶	U_livre	℔	U_pfennig	₷	U_spesmilo	¥	U_yen

4.13 Latin Small Capital Letters

Capital											
A	U_smallcapA	E	U_smallcapE	J	U_smallcapJ	N	U_smallcapN	T	U_smallcapT	X	U_smallcapX
B	U_smallcapB	G	U_smallcapG	K	U_smallcapK	O	U_smallcapO	U	U_smallcapU	Y	U_smallcapY
C	U_smallcapC	H	U_smallcapH	L	U_smallcapL	P	U_smallcapP	V	U_smallcapV	Z	U_smallcapZ
D	U_smallcapD	I	U_smallcapI	M	U_smallcapM	R	U_smallcapR	W	U_smallcapW		

4.14 Line Segments

Single segments				Frames						Edges					
vertical		horizontal		left		middle		right		left		middle		right	
┃	U_vtdash	╌	U_hzdash	┏	U_tla	┳	U_tma	┓	U_tra	┯	U_tmh			┯	U_tmj
╻	U_vtb	╴	U_hzb	┣	U_mla	╋	U_mma	┫	U_mra	┯	U_tmi			┯	U_tmk
│	U_vtsolid	─	U_hzsolid	┗	U_bla	┻	U_bma	┛	U_bra						
┆	U_vtdashs	┄	U_hzdashs							┷	U_bmh			┷	U_bmj
┇	U_vte	┅	U_hze	┏	U_tlb	┳	U_tmb	┓	U_trb	┷	U_bmi			┷	U_bmk
┆	U_vtf	┈	U_hzdots	┣	U_mlb	╋	U_mmb	┫	U_mrb						
╽	U_vtg	╼	U_hzg	┗	U_blb	┻	U_bmb	┛	U_brb	┠	U_mlh			┨	U_mrh
┃	U_vth	━	U_hzh							┠	U_mli			┨	U_mri
╿	U_vti	╾	U_hzi	┏	U_tlc	┳	U_tmc	┓	U_trc	┠	U_mlj			┨	U_mrj
╻	U_vtj	╸	U_hzj	┣	U_mlc	╋	U_mmc	┫	U_mrc	┠	U_mlk			┨	U_mrk
┃	U_vtbold	━	U_hzbold	┗	U_blc	┻	U_bmc	┛	U_brc	Centers					
┇	U_vtl	┅	U_hzl							┼	U_mmh	┼	U_mml	┼	U_mmp
┇	U_vtm	┅	U_hzm	┏	U_tld	┳	U_tmd	┓	U_trd	┼	U_mmi	┼	U_mmm	┼	U_mmq
┇	U_vtn	┅	U_hzn	┣	U_mld	╋	U_mmd	┫	U_mrd	┼	U_mmj	┼	U_mmn	┼	U_mmr
‖	U_vtdbl	═	U_hzdbl	┗	U_bld	┻	U_bmd	┛	U_brd	┼	U_mmk	┼	U_mmo	┼	U_mms
										Miscellaneous					
╱	U_fwd			╒	U_tle	╤	U_tme	╕	U_tre						
╲	U_bwd			╘	U_mle	╪	U_mme	╡	U_mre						
╳	U_cross			╘	U_ble	╧	U_bme	╛	U_bre						
				╓	U_tlf	╥	U_tmf	╖	U_trf						
				╟	U_mlf	╫	U_mmf	╢	U_mrf						
				╙	U_blf	╨	U_bmf	╜	U_brf						
				╔	U_tldbl	╦	U_tmdbl	╗	U_trdbl						
				╠	U_mldbl	╬	U_mmdbl	╣	U_mrdbl						
				╚	U_bldbl	╩	U_bmdbl	╝	U_brdbl						

5 Unicoder Expanded Character Set

In addition to the characters defined above in section 4 additional unicode characters have been defined within this expanded *Unicoder* character set. Some of the tables from the above section have been copied into this section and expanded while other tables contain only newly defined characters. To include this expanded set you must declare `#define EXTENDED_UNICODE` before adding the `#include <unicoder.h>` at the beginning of your code.

Note that depending on the character set you choose some characters may not be defined.

5.1 Relational Operators

less		not less		greater		not greater	
‹	U_less	≮	U_notlessthan	›	U_greater	≯	U_notgreater
≤	U_leq	≰	U_notlessoreq	≥	U_geq	≱	U_notgreateroreq
				⩾̈	U_greaterdiaersis		
⩽	U_lessorequal	⪇	U_lessbutnoteq	⩾	U_greateroreq	⪈	U_greaterbutneq
⋚	U_eqorless			⋛	U_eqorgreater		
≲	U_lessoreqival	≴	U_notlessorequiv	≳	U_greateroreqiv	≵	U_notgreatoreqiv
⋦	U_lessornoteqival			⋧	U_greaterornoteqiv		
≶	U_lessorgreater	≸	U_notlessorgreater	≷	U_greaterorless	≹	U_notgreaterorless
⪋	U_lesseqgreater			⋛̸	U_greatereqless		
≺	U_precedes	⊀	U_notprecedes	≻	U_succeeds	⊁	U_notsucceed
≼	U_precedesoreq	⋠	U_notprecedesoreq	≽	U_succeedsoreq	⋡	U_notsucceedsoreq
⪯	U_eqorprecedes			⪰	U_eqorsucceeds		
≾	U_preceedsorequiv	⋨	U_preceedsnorequiv	≿	U_succeedsorequiv	⋩	U_succeedsnorequiv
equates		**equates**		**equivalent**		**not equivalent**	
≂	U_minustilde	≃	U_asymptoticeq	≒	U_reversetildeeq	≠	U_notasymptoteq
≒	U_approaches	≓	U_geoequalto	≈	U_approx	≉	U_notapprox
≒	U_approximageof	≓	U_imageofapprox	≍	U_equivalent	≭	U_notequivalent
≔	U_definition	≕	U_equalcolon	≅	U_approxequal	≇	U_approxnoteq
≋	U_almosteq	≋	U_tripletilde	≌	U_allequalto	≉	U_notapproxoreq
≣	U_ringinequal	≗	U_ringequalto	≡	U_equiv	≢	U_notequiv
≙	U_estimates	≚	U_equiangularto	≡	U_strictlyequiv		
≛	U_starequals	≙	U_deltaequalto				
≎	U_geoequivalent	≏	U_difference	≙	U_correspondsto		
≝	U_equalrobydef	≞	U_measuredby	≒	U_questionedeq	≠	U_neq

5.2 Games & Music

Chess				Cards		Dice		Music		Gylphs	
♔	U_whking	♚	U_blkking	♠	U_spades	⚀	U_ace	♩	U_quarternote	♨	U hotsprings
♕	U_whqueen	♛	U_blkqueen	♣	U_clubs	⚁	U_deuce	♪	U_8thnote	☘	U_shamrock
♖	U_whrook	♜	U_blkrook	♦	U_diamonds	⚂	U_trey	♫	U_beamed8th	☕	U_hotcoffee
♗	U_whbishop	♝	U_blkbishop	♥	U_hearts	⚃	U_cater	♬	U_beamed16th	☎	U_blkphone
♘	U_whknight	♞	U_blkknight	♤	U_ltspades	⚄	U_cinque	♭	U_flat	☏	U_whitephone
♙	U_whpawn	♟	U_blkpawn	♧	U_ltclubs	⚅	U_sice	♮	U_natural	☁	U_blkcloud
				♢	U_ltdiamonds			♯	U_sharp	☂	U_umbrella
Shogi				♡	U_lthearts					☔	U_rainumbrella
☖	U_whshogi	☗	U_blkshogi							☃	U_snowman

5.3 Characters

Punctuation				Prime		Superscript		Number sets	
⁋	U_revparagraph	⸮	U_supquestion	‵	U_revprime	ᵕ	U_supmhos	ℂ	U_complex
⁝	U_trianglecolon	⁇	U_dblquestion	‶	U_revdiprime	a‿	U_continuous	ℕ	U_naturalnum
		⁈	U_questionexclm	‷	U_revtriprime	aa	U_discontinue	ℚ	U_rationalnum
¨	U_diaeresis	⁉	U_exclmquestion	Quill		⁎	U_emphasis	ℝ	U_realnum
…	U_ellipsis	⁏	U_semicolonbar	⁅	lftquill			ℤ	U_integernum
⋯	U_midellipsis	⸏	U_quotebar	⁆	rgtquill			ℍ	U_dblstruckH
								ℙ	U_dblstruckP

5.4 Modified Greek & Latin

Greek						Latin	
Ā	U_barAlpha			⍺	U_alphabar	ƀ	U_bbar
				ϵ	U_epsilonbar	đ	U_dbar
Ī	U_barIoto	ī	U_bariota	ɩ	U_iotabar		
Ȳ	U_barUpsilon			⍵	U_omegabar		

5.5 Latin Modifiers Lower case

	Super-script	Sub-script		Super-script	Sub-script		Super-script	Sub-script		Super-script	Sub-script		Super-script	Sub-script	Dot-less	
a	U_supa	U_suba	f	U_supf		k	U_supk	U_subk	p	U_supp	U_subp	v	U_supv	U_subv	ɪ	U_dli
b	U_supb		g	U_supg		l	U_supl	U_subl	r	U_supr	U_subr	w	U_supw		ɉ	U_dlj
c	U_supc		h	U_suph	U_subh	m	U_supm	U_subm	s	U_sups	U_subs	x	U_supx	U_subx	vary	
d	U_supd		i	U_supi	U_subi	n	U_supn	U_subn	t	U_supt	U_subt	y	U_supy		ɡ	U_varg
e	U_supe	U_sube	j	U_supj	U_subj	o	U_supo	U_subo	u	U_supu	U_subu	z	U_supz			

5.6 Mathematical symbols

Symbols								Units	
√	U_sqrt	∫	U_intgl	∁	U_complement	ϴ	U_zilde	ℎ	U_Planck
∛	U_cbrt	∬	U_dblintgl	⊤	U_verum	⊥	U_falsum	ℏ	U_RedPlanck
∜	U_4thrt	∭	U_tripleintgl	Δ	U_increment	∇	U_nabla	K	U_Kelvin
∓	U_mp	±	U_pm	×	U_multiplication	×	U_crossproduct	Å	U_Angstrom
∶	U_ratio	∷	U_proportion	·	U_dotop			‰	U_perthousand
∹	U_excess	∺	U_geoproportion	⋆	U_starop	⍣	U_stardiaeresis	⅟	U_oneover
−	U_minussign	∸	U_dotminus	⧺	U_tripleplus	⧺	U_doubleplus	Geometry	
~	U_tildeop	∽	U_reversetilde	⇝	U_nottilde	⨲	U_stiletilde	∥	U_parallel
÷	U_homothetic	∺	U_tildediaersis	⟅	U_lftbagdelimit	⟆	U_rgtbagdelimit	⊥	U_perp
Braces and Brackets								∠	U_angle
⎡	U_lftbracketto	⎤	U_rgtbrackettop	⌈	U_lftceiling	⌉	U_rgtceiling	∟	U_rightangle
⎢	U_lftbracketex	⎥	U_rgtbracketext	⌊	U_lftfloor	⌋	U_rgtfloor	⌒	U_arc
⎣	U_lftbracketbt	⎦	U_rgtbracketbtm	⌈	U_intgltop	⌋	U_intglbtm	⌓	U_segment
⎛	U_lftparenttop	⎞	U_rgtparenttop	∫	U_intgl	⍀	U_esh	⌔	U_sector
⎜	U_lftparentext	⎟	U_rgtparentext	⟨	U_lftanglebrkt	⟩	U_rgtanglebrkt	⌷	U_rectangle
⎝	U_lftparentbtm	⎠	U_rgtparentbtm	⦇	U_lfttortoise	⦈	U_rgttortoise	⌑	U_quad
⎧	U_lftcurlytop	⎫	U_rgtcurlytop	{	U_lftboldbrckt	}	U_rgtboldbrckt	▭	U_rectanglesm
⎪	U_lftcurlyext	⎪	U_rgtcurlyext	⟨	U_lftmedmangle	⟩	U_rgtmedmangle	△	U_triangle
⎨	U_lftcurlymid	⎬	U_rgtcurlymid	❰	U_lftboldangle	❱	U_rgtboldangle	▵	U_smtriangle
⎩	U_lftcurlybtm	⎭	U_rgtcurlybtm	(	U_lftmedmpar	)	U_rgtmedmpar	▱	U_parallelogram
Florette				❨	U_lftboldpar	❩	U_rgtboldpar		
∗	U_asterisk	✥	U_clubsdrop	Logic Statements					
✱	U_boldasterisk	✢	U_4ptteardrop	∨	U_logicalor	∧	U_logicaland	⊕	U_xor
✲	U_openasterisk	✣	U_balloondrop	∃	U_exists	∄	U_notexists	∀	U_forall
✳	U_spikeasterisk	✤	U_boldballoon	≡	U_equiv	≔	U_definition	˜	U_not
✴	U_blkspikeaster	❁	U_8petalled	¬	U_neg	⌙	U_turnednot	⇒	U_implies
✽	U_boldflorette	✻	U_6ptteardrop	↔	U_proportional	⇔	U_materialediv	◦	U_litebullit
✾	U_florette	✼	U_openteardrop	«	U_ll	»	U_gg	◡	U_countersink
❀	U_whflorette	✿	U_blkflorette	∴	U_therefore	∵	U_because		

5.7 Arrows

West		West/East		East		West		East	
⇜	U_Warrowopen	⇿	U_WEarrowopen	⇾	U_Earrowopen	⇝	U_Warrowtail	⇝	U_Earrowtail
←	U_Warrow	↔	U_WEarrow	→	U_Earrow	⇠	U_Warrowdash	⇢	U_Earrowdash
↚	U_Warrowslash	↮	U_WEarrowslash	↛	U_Earrowslash	↬	U_Warrowloop	↬	U_Earrowloop
↞	U_Warrowstroke	↮	U_WEarrowstroke	↠	U_Earrowstroke	↩	U_Warrowhook	↪	U_Earrowhook
⇷	U_Warrowdistroke	⇹	U_WEarrowdistroke	⇻	U_Earrowdistroke	↞	U_Wdiarrow	↠	U_Ediarrow
⇤	U_Warrowtobar	⇆	U_WbyEarrowstobar	⇥	U_Earrowtobar	↤	U_Warrowbar	↦	U_Earrowbar
↜	U_Warrowwavy	↭	U_WEwavyarrow	↝	U_Earrowwavy	⇜	U_Warrowwaves	⇝	U_Earrowwaves
⇇	U_WbyWarrows	⇆	U_WbyEarrows	⇉	U_EbyEarrows	↵	U_Warrowturn	↻	U_Earrowcircle
		⇄	U_EbyWarrows	⇶	U_Etriarrows	↲	U_Warrowleft	↳	U_Earrowright
←	U_Warrowbarb	⇋	U_WbyEarrowsbarb	→	U_Earrowbarb	↰	U_Warrowbent	↱	U_Earrowbent
↼	U_Warrowharpon	⇌	U_EbyWarrowsbarb	⇀	U_Earrowharpon	⇦	U_Wwhitearrow	⇨	U_Ewhitearrow
⇐	U_Wdblarrow	⇔	U_WEdblarrow	⇒	U_Edblarrow	⇚	U_Wtriarrow	⇛	U_Etriarrow
⇍	U_Wdblarrowslash	⇎	U_WEdblarrowslash	⇏	U_Edblarrowslash	⏎	U_return	⇨	U_Ewhitearrowped
				⇒	U_Eopenarrow				
North		North/South		South		North		South	
↑	U_Narrow	↕	U_NSarrow	↓	U_Sarrow	↺	U_Narrowhook	↶	U_Sarrowhook
↥	U_Narrowbar	↕	U_NSarrowbar	↧	U_Sarrowbar	↻	U_Narrowloop	↷	U_Sarrowloop
⇈	U_NbyNarrows	⇅	U_NbySarrows	⇊	U_SbySarrows	⇡	U_Narrowdash	⇣	U_Sarrowdash
		⇵	U_SbyNarrows			↿	U_Narrowbarb	↽	U_Sarrowbarb
⇑	U_Ndblarrow	⇕	U_NSdblarrow	⇓	U_Sdblarrow	↿	U_Narrowharpon	↽	U_Sarrowharpon
⇧	U_Nwhitearrow	⇳	U_NSwhitearrow	⇩	U_Swhitearrow	↟	U_Ndiarrow	↡	U_Sdiarrow
⇫	U_Nwhitepedbar					⇞	U_Narrowbars	⇟	U_Sarrowbars
⇮	U_Nwhitepedline					↑	U_supNarrow	↓	U_supSarrow
⇮	U_Nwhitediarrow					⇪	U_Nwhitearrowbar	↴	U_Sarrowbent
⇫	U_Nwhitediarwped					⇫	U_Nwhitearrowped	↯	U_Sarrowzigzag
Northeast		Northwest		Southeast		Southwest		Arrowheads	
↗	U_NEarrow	↖	U_NWarrow	↘	U_SEarrow	↙	U_SWarrow	^	U_uparrowhead
⇗	U_NEdblarrow	⇖	U_NWdblarrow	⇘	U_SEdblarrow	⇙	U_SWdblarrow	⩛	U_projective
⇗	U_NEbowarrow	⇱	U_NWarrowcorner	⇲	U_SEarrowcorner			⩜	U_perspective
↗	U_SWbold	↖	U_NWarrowedge	↘	U_SEbold			⌄	U_downarrowhead
				⇘	U_SEelbowarrow			⏚	U_insertion
				⇱	U_thunderstorm				

5.8 Sets

Set Operators								Set notation	
∩	U_intersect			∪	U_union			∅	U_emptyset
⊃	U_supersetof	⊅	U_notsuperset	⊂	U_subsetof	⊄	U_notsubsetof	∈	U_belongsto
⊇	U_superset	⊉	U_notsupersetoreq	⊆	U_subset	⊈	U_notsubsetoreq	∉	U_notbelong
		⊋	U_supersetnoteq			⊊	U_subsetnoteq	∋	U_contains
⊐	U_squareoriginal	⋡	U_notsquareoriginal	⊏	U_squareimageof	⋢	U_notsquareimageof	⋣	U_notcontain
⊒	U_sqroriginaloreq	⋣	U_sqroriginalnoteq	⊑	U_sqrimqgeoreq	⋤	U_squareimagenoteq		

5.9 Geometric Shapes

Quadralaterial				Circles			
⧀	U_quadless	⧁	U_quadgreater	○	U_whcircle	⊕	U_circleplus	⊛	U_circleaster
⩔	U_quaddown	⩓	U_quadup	⦵	U_circlebar	⊖	U_circleminus	⊝	U_circledash
⬱	U_quadleftarrw	⬲	U_quadrgtarrw	⌾	U_quadcircle	⊗	U_circletimes	⊛	U_circlestar
⟟	U_quaduparrw	⟱	U_quaddownarrw	⌽	U_circlestile	⊙	U_circledot	✪	U_blkcircledstar
⁝	U_quadcolon	⧸	U_quadslash	⊘	U_circleslash	⦶	U_circlenotch		
⨪	U_quaddivide	⧹	U_quadbackslash			⦸	U_circlenwarrw	⦼	U_circlebackslash
⸘	U_quadquestion	⸌	U_quadquote	⦿	U_circlergtdot	⦶	U_circletwodot	Ö	U_circlediaersis
⍜	U_jotbar	⍤	U_quadjot	◎	U_circlering	◎	U_circlejot	⍥	U_jotdiaersis
⩦	U_quadequal	≠	U_quadnotequal	⊜	U_circleequal	◍	U_cirlcebars	●	U_blkcircle
Diamonds		⩛	U_quaddel	⦲	U_circledel	◉	U_whblkcircle		
◆	U_blkdiamond	⧋	U_quadtriangle	●	U_blkcircledot	◍	U_blkcircledots	◕	U_mostlyblkcircle
◈	U_whblkdiamond	◘	U_inversebullet	◒	U_btmblkcircle	◓	U_topblkcircle	◔	U_toprgtblkcircle
◇	U_whdiamond	Tacks		◐	U_lftblkcircle	◑	U_rgtblkcircle	◴	U_toplftwhcircle
⟐	U_diamondbar	⊢	U_righttack	◖	U_lfthalfcircle	◗	U_rgthalfcircle	◵	U_btmlftwhcircle
⧈	U_quaddiamond	⊣	U_lefttack	◒	U_btmhalfcircle	◓	U_tophalfcircle	◶	U_btmrgtwhcircle
⬖	U_lfthalfdiam	⊤	U_downtack	☹	U_sadface	↻	U_circlearrow	◷	U_toprgtwhcircle
⬗	U_rgthalfdiam	⊥	U_uptack	☺	U_lthappy	☻	U_dkhappy	✇	U_tapedrive
⬘	U_tophalfdiam			☮	U_peace	☯	U_yinyang	✆	U_phone
⬙	U_halfdiam			Arcs					
⧫	U_blkdiamond			⌒	U_toparc	◜	U_toplftquarter	◝	U_toprgtquarter
⟠	U_divideddiam			◡	U_btmarc	◟	U_btmlftquarter	◞	U_btmrgtquarter

5.10 Polygons

Black Objects				Boxes							
	U_blkbargraph1		U_blkbar1		U_whitebox		U_boxhfill		U_blackbox
	U_blkbargraph2		U_blkbar2		U_boxsquare		U_boxvfill		U_whbisectbox
	U_blkbargraph3		U_blkbar3		U_boxlftdiag		U_boxrgtdiag		U_mtballotbox
	U_blkbargraph4		U_blkbar4		U_boxcage		U_boxdiag		U_chkballotbox
	U_blkbargraph5		U_blkbar5		U_boxplus		U_boxtimes		U_xedballotbox
	U_blkbargraph6		U_blkbar6		U_boxminus		U_boxdot		U_tophalfbox
	U_blkbargraph7		U_blkbar7		U_lfthalfbox		U_rgthalfbox		U_btmhalfbox
	U_blkbargraph8		U_blkbar8		U_sqrupperlft		U_sqrlowergt		U_sqrupperrgt
	U_blkbargraph9		U_blkbar9		U_whupperlftbx		U_whlowerlftbx		U_sqrlowerlft
	U_blkparallel		U_blkbar10		U_whupperrgtbx		U_whlowerrgtbx		U_dottedbox
	U_rectangleA1		U_rectangleB1		U_roundedwhbox		U_smwhbox		U_smblkbox
	U_rectangleA2		U_rectangleB2		U_whbar10		U_sqrlozenge		
	U_rectangleA3		U_rectangleB3		U_rectanglesm		U_openedbox		U_openbox
	U_rectangleA4		U_rectangleB4		U_topleft		U_topright		
	U_rectangleB12		U_rectangleB34		U_btmleft		U_btmright		
				Triangles					
					U_blkrgttriang		U_whrgttriangle		U_whtriangledot
					U_blkrgtarwsm		U_whrgtarwsm		U_lfthalftri
					U_blkrgtarw		U_whrgtarw		U_rgthalftri
					U_blkdwnarw		U_whdwnarw		U_blklowerrgt
					U_blkdwnarwsm		U_whdwnarwsm		U_blklowerlft
					U_blklfttriang		U_blkrgttriangle		U_blkupperlft
					U_blklftarw		U_whlftarw		U_blkupperrgt
					U_blklftarwsm		U_whlftarwsm		U_whupperlft
					U_delstile		U_deltilde		U_whupperrgt
					U_deltastile		U_deltabar		U_whlowerlft
									U_whlowerrgt

5.11 Symbols

Zodiac		Astronomy				Recycle		Plastics	
♈	U aries	☉	U sun	☊	U ascendnode	♲	U recylce1	♳	U plastic
♉	U_taurus	☿	U_mercury	☋	U_descendnode	♻	U_invrecycle	♴	U_plastic1
♊	U_gemini	♀	U_venus	☍	U_opposition	♼	U_recyclepaper	♵	U_plastic2
♋	U cancer	♁	U earth	☌	U conjunction	♽	U_partialrecycle	♶	U plastic3
♌	U leo	♂	U mars	☄	U comet	♾	U paper	♷	U plastic4
♍	U virgo	♃	U jupiter	☀	U blksunshine	Hazards		♸	U plastic5
♎	U_libre	♄	U_saturn	☆	U_whstar	☠	U_poison	♹	U_plastic6
♏	U scorpius	♅	U uranus	★	U blkstar	☢	U radioactive	♺	U plastic7
♐	U sagittarius	♆	U neptune	☆	U stresswhstar	☣	U biohazard	Currency	
♑	U_capricorn	♇	U_pluto	☆	U_openblkstar	☡	U_caution	₢	U_cruzeiro
♒	U aquarius	☽	U 1stquarter	✭	U blkcenterstar	⚠	U warning	keys	
♓	U pisces	☾	U lastquarter	✯	U outlinestar	Objects		⌨	U keyboard
Symbols				☆	U stardot	☸	U helm	⏎	U return
⚛	U_atomic	☧	U_chirho	✰	U_5ptpinwheel	⚓	U_anchor	⌦	U_erase
♿	U handycap	☥	U Ankh	☆	U shadowstar	⚙	U gear	⌧	U delete
⚕	U medical	☦	U orthodox	✶	U blk6ptstar			⌫	U backspace
⚕	U_aesculapius	✝	U_lorraine	✦	U_blk4pointstar	⚰	U_coffin	⌘	U_macintosh
☤	U hermes	✝	U latincross	✧	U wh4pointstar	⚱	U urn	⌥	U option
⚜	U fleurdelis	✞	U outlinecross	✵	U 8ptpinwheel	✉	U envelope	⏏	U eject
✎	U pencil1	✟	U outlinecross2	✷	U blk8ptstar	⚐	U ltflag	Miscellaneous	
✏	U_pencil2	☩	U_jerusalem	✸	U_bold8ptstar	⚑	U_dkflag	⌕	U_recorder
✐	U pencil3	☨	U westsyrian	✹	U blk12ptstar	✁	U scissors1	☓	U saltire
✑	U whnib	☨	U eastsyrian	✺	U blk16ptstar	✂	U scissors2	⚡	U highvoltage
✒	U_blknib	✠	U_maltese	Hearts		✃	U_scissors3	⌁	U_electrical
		✙	U_greekcross	❣	U_exclaimheart	✄	U_scissors4		
		✚	U boldgreekcross	❤	U boldblkheart	☭	U hammersickle	Marks	
		✛	U centrecross	❥	U rotatedheart	⚒	U hammerpick	✓	U checkmark
		✜	U_boldcentre	❦	U_floralheart	⚔	U_crossedswords	✔	U_boldcheckmark
Hands				❧	U_rotatefloral	✈	U_airplane	✖	U_boldx
✌	U_victory	✍	U_writing	❧	U revheart	⚖	U scales	✗	U ballotx
☚	U blkpointlft	☛	U blkpointrgt			⚗	U alembic	✘	U boldballotx
☜	U_whpointlft	☞	U_whpointrgt			⚘	U_flower		
☝	U_whpointup	☟	U_whpointdwn						

6 Reporting Bugs

I have attempted to debug *Unicoder* using multiple platforms and multiple compilers. However, there probably will still be some bugs, especially in the definitions. Therefore, I am relying on you to bring these bugs to my attention so I can fix them and make *Unicoder* a reliable software package for programming using wide-characters and Unicode.

Bugs and general comments or info about your project(s) using *Unicoder,* A Wide-Character Terminal Output Library for C and C++ should be sent via

sthomasbradley<at>gmail<dot>com