Unicode for the PCC C99 Compiler

by Eric Olson

This document describes the implementation of UTF-8 unicode support for the pcc compiler which coded by Eric Olson during May 2014 and how to write and test UTF-8 unicode programs on a Unix, Linux or similar operating system using the pcc compiler.

Introduction

To support unicode the internal representation used by pcc is now UTF-8 throughout. This allows internal code paths to remain `char` 8-bit, which was the engineering design of UTF-8 by Ken Thompson and Rob Pike. Wide character constants are provided by UTF-8 to UTF-32 conversions which happen at compile time. Support for unicode UCN escapes `\uxxxx` and `\Uxxxxxxx` are provided in the preprocessor in the same routine that processes trigraphs. The trigraphs allow people to write C code using the EBCDIC character set and punchcards; the UCNs allow people to write unicode characters using the ASCII character set and VT100 terminals. All trigraphs and UCN escapes are converted to their corresponding UTF-8 byte sequences as they are encountered by the preprocessor. This means we don't have to clutter the compiler with code to handle trigraphs and UCNs, neither of which are needed in a modern computing environment. Note that a byte order mark `\uFEFF` should not appear in the file and will cause a syntax error if it does.

To handle unicode identifiers in C code, such as

```
double δ=1.0/M;
int tamaño=13;
long første=1;
```

the grammar for the scanner was modified to treat all 8-bit codes with the high-bit set as letters. This is how the original implementation was done by Thompson and Pike in 1992 for Plan 9. The C++11 ISO places some restrictions on the use of unicode characters in identifiers. However, these restrictions are not restrictive enough to ensure readable code. The problems of ASCII capital `I` looking like lowercase `l` and capital `O` looking like the number `0` have been compounded a hundred fold with unicode. Every project using pcc should specify which extended characters are allowed in identifiers.

The pcc compiler does not check that sequences of bytes with the high-bit set form valid UTF-8 code points. This allows a code which contains strings meant to be displayed using an 8-bit code page such as IBM PC code page 437 to function as expected. An exception to this rule is the initialisation of wide character constants and strings. In this case, it is essential that the input represent valid UTF-8 code sequences. The `wchar_t` wide data type is stated in ISO 9899:1990 section 4.1.5 to be

> *an integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.*

Since unicode consists of more than 2^{16} different characters, the only reasonable choice for `wchar_t` in a unicode environment is a 32-bit integer, or what is called UTF-32. If an

invalid UTF-8 code sequence is encountered while initialising a wide character, the unicode specified not a character `\uFFFF` is used as the initialisation.

Wide character constants and strings may be specified using the `L` prefix as follows:

```
wchar_t code = L'β';
wchar_t questw[] = L"Хочете чашечку кави?";
```

The pcc compiler translates the unicode characters in the UTF-8 source code and stores the resulting code points in each `wchar_t` location. In the example above, `questw` is an array of length 21 since the string consists of 20 unicode characters plus the terminal null character 0. This array takes a total space of $21 \cdot 4 = 84$ bytes. The UTF-8 byte sequence for the same string is 38 bytes. These facts are illustrated by the following code

```
1  #include <stdio.h>
2  #include <wchar.h>
3
4  int main(){
5      char q1[] = "Хочете чашечку кави?";
6      wchar_t q2[] = L"Хочете чашечку кави?";
7      printf("sizeof(q1)=%d\n",(int)sizeof(q1));
8      printf("sizeof(q2)=%d\n",(int)sizeof(q2));
9      return 0;
10 }
```

which produces the output

```
sizeof(q1)=38
sizeof(q2)=84
```

when compiled and run by pcc.

Unicode on Unix, Linux and compliant systems is always done using UTF-8. Out of the box `xterm` supports UTF-8 encoding. To load a unicode font and turn on the UTF-8 display option start the `xterm` using

```
$ LANG=en_GB.UTF-8 xterm -fa "Liberation Mono"
```

The terminal emulators for Gnome and KDE also display UTF-8 by default. UTF-8 files can be edited with `vim`. For example,

```
:set keymap=greek
:set nobomb
```

loads the Greek keymap, which can then be toggled on and off using `ctrl+^` while in input mode. It is important that `nobomb` is set so that `vim` doesn't write the unicode byte order sequence `\uFEFF` at the beginning of the file. The editors `gedit` or `emacs` may also be used editing UTF-8 source files.

The GNU C library supports the standard POSIX function calls for dealing with wide characters. The pcc implementation of wide characters is compatible with this library. Note that wide strings used in a C program must be converted to UTF-8 before they are displayed on the terminal. The following example

```
1  #include <stdio.h>
2  #include <stdlib.h>
```

```
#include <locale.h>
#include <wchar.h>

int main(){
    setlocale(LC_ALL,"");
    wchar_t sw[] = L"Et æble om dagen holder lægen væk.";
    int len=wcstombs(0,sw,0);
    char cliche[len+1];
    wcstombs(cliche,sw,len+1);
    puts(cliche);
    return 0;
}
```

with output

```
Et æble om dagen holder lægen væk.
```

illustrates that this works with pcc.

The remainder of this document describes specific details of my UTF-8 implementation for the pcc compiler and finishes with some concluding remarks.

The Preprocessor

As already mentioned, the preprocessor has been modified to accept UCN escape codes. This was done by adding the routine `chkucn` to `token.c` in the recursive descent parser. The code is straight forward and looks like

```
static int
chkucn(void)
{
    int ch, sz;

    ch = inpch();
    if(ch != '\\'){
        unch(ch);
        return 0;
    }
    ch = inpch();
    if (ch == 'u') {
        int i, v;
        sz=4;
ucnshort:
        for(i=0,v=0;i<sz;i++){
            ch=toupper(inpch());
            if(ch>='0' && ch<='9') v=(v<<4)+ch-'0';
            else if(ch>='A' && ch<='F') v=(v<<4)+ch-'A'+10;
            else {
                if(sz>2) warning(
                    "unicode escape %d digits too short",sz-i);
                unch(ch);
                break;
            }
        }
        char u8str[8],*p=cp2u8(u8str,v);
        while(--p>u8str) unch(*p);
```

```
29          return u8str[0];
30      } else if (ch == 'U') {
31          sz=8;
32          goto ucnshort;
33      }
34      unch(ch);
35      unch('\\');
36      return 0;
37  }
```

Note that a UTF-8 sequence can be at most 6 bytes, so `u8str` is actually one byte longer than absolutely necessary. The resulting UTF-8 byte sequence is pushed back onto the input stack in place of the UCN and the first byte is then returned.

Modifications were also made so that the preprocessor treats all 8-bit codes with the high bit set as letters. This allows preprocessor macros to be identified using UTF-8 unicode characters. The modifications involved adding one line of code in the routine `sloscan` to check if the high-bit is set. The results allow programs such as

```
1 #include <stdio.h>
2 #define für for
3
4 int main(){
5     für(int i=1;i<=7;i++) printf("%d%c",i,i!=7?' ':'\n');
6     exit(0);
7 }
```

to produce the output

```
1 2 3 4 5 6 7
```

This source code also compiles with clang and produces identical output; however, the usefulness of creating foriegn language synonyms for C keywords is debatable.

The C99 Compiler

Changes were made to `scan.l` to remove the legacy parse rules for the UCNs now handled by the preprocessor and to update the code for character constants. The original code, had support routines that appeared to be intended to pack bytes into words such as what might be done in UTF-7 or UTF-16. These routines depended on a fragile endian-aware support macro `makecc` and appeared broken. Moreover, neither UTF-7 or UTF-16 are useful on a Linux, Unix or similar computing environment so these routines have been completely rewritten for UTF-8 and UTF-32 as follows:

```
1  NODE *
2  wcharcon(void)
3  {
4      unsigned int val = 0, i = 0;
5      char *pp = yytext;
6
7      if (*pp == 'L')
8          pp++;
9      pp++;
10     while (*pp != '\'') {
11         if(*pp == '\\') val=esc2char(&pp);
12         else val=u82cp(&pp);
13         i++;
14     }
15
16     if (i == 0)
17         uerror("empty wide-character constant");
18     if (i > 1)
19         werror("%d too many characters in wide-character constant",i-1);
20     return xbcon(val, NULL, ctype(UNSIGNED));
21 }
```

Note that these routines are now much simpler. The endian-aware macros `makecc` have been left in `arch/*/machdefs.h` because the C++ compiler still depends on them.

Changes to `gram.y` involved the removal of special processing for unicode as all the work was done either before or after the grammar. This was possible because the compiler works internally with UTF-8.

Conversion of UTF-8 to UTF-32 for the initialisation of wide character arrays happens in `init.c` just before output to the assembler. This is done with the code

```
1  static void
2  strcvt(NODE *p)
3  {
4      NODE *q = p;
5      char *s;
6      int i;
7
8  #ifdef mach_arm
9      /* XXX */
10     if (p->n_op == UMUL && p->n_left->n_op == ADDROF)
11         p = p->n_left->n_left;
12 #endif
13
```

```
14      for (s = p->n_sp->sname; *s != 0; ) {
15          if(p->n_type==ARY+WCHAR_TYPE) i=u82cp(&s);
16          else i=(unsigned char)*s++;
17          asginit(bcon(i));
18      }
19      tfree(q);
20  }
```

Please ignore the `#ifdef` as I didn't put it in, nor do I know why it is there.

The output routine for 8-bit strings also needed a few modifications. We add C style escapes to 8-bit strings as they are output for the assembler while making sure not to break long lines in the middle of a UTF-8 byte sequence. This is done in `pftn.c` in the routine `instring` which reads

```
1  void
2  instring(struct symtab *sp)
3  {
4      char *s = sp->sname;
5
6      locctr(STRNG, sp);
7      defloc(sp);
8
9      char line[70], *t = line;
10     printf("\t.ascii \"");
11     while(s[0] != 0) {
12         unsigned int c=(unsigned char)*s++;
13         if(c<' ') t+=sprintf(t,"\\%03o",c);
14         else if(c=='\\') *t++='\\', *t++='\\';
15         else if(c=='\"') *t++='\\', *t++='\"';
16         else if(c=='\'') *t++='\\', *t++='\'';
17         else {
18             *t++=c;
19             int sz=u8len(&s[-1]);
20             int i;
21             for(i=1;i<sz;i++) *t++=*s++;
22         }
23         if(t>=&line[60]) {
24             fwrite(line, 1, t-&line[0], stdout);
25             printf("\"\n\t.ascii \"");
26             t = line;
27         }
28     }
29     fwrite(line, 1, t-&line[0], stdout);
30     printf("\\0\"\n");
31 }
```

While most of the work in this routine is done to ensure that UTF-8 byte sequences are not broken between lines, this routine also has to be 8-bit clean so that all bytes are passed to the assembler without modification This is important because C character-string constants which contain arbitrary 8-bit values are useful.

The C++ Compiler

I did not work on the C++ compiler. C99 has a faster and cleaner implementation of variable length arrays as well as a native complex data type. Although operator overloading has uses, I find it difficult to get all the special cases right and don't use C++. The exact same changes that I made for unicode support in the C99 compiler should be easy to implement for the C++ complier.

New Code

I wrote two custom routines to make UTF-8 to UTF-32 conversion convenient in the compiler and preprocessor. There are contained in `unicode.c` which lives in the `mip` directory. The routines are

```
char *
cp2u8(char *p,unsigned int c)
{
    unsigned char *s=(unsigned char *)p;
    if(c>0x7F){
        if(c>0x07FF){
            if(c>0xFFFF){
                if(c>0x1FFFFF){
                    if(c>0x3FFFFFF){
                        if(c>0x7FFFFFFF){
                            u8error("invalid unicode code point");
                        } else {
                            *s++=0xF8|(c>>30);
                            *s++=0x80|((c>>24)&0x3F);
                        }
                    } else {
                        *s++=0xF8|(c>>24);
                    }
                    *s++=0x80|((c>>18)&0x3F);
                } else {
                    *s++=0xF0|(c>>18);
                }
                *s++=0x80|((c>>12)&0x3F);
            } else {
                *s++=0xE0|(c>>12);
            }
            *s++=0x80|((c>>6)&0x3F);
        } else {
            *s++=0xC0|(c>>6);
        }
        *s++=0x80|(c&0x3F);
    } else {
        *s++=c;
    }
    return (char *)s;
}
```

which converts unicode code points to UTF-8 byte sequences, and

```
unsigned int
```

```
u82cp(char **q)
{
    unsigned char *t=(unsigned char *)*q;
    unsigned int c=*t;
    unsigned int r;
    if(c>0x7F){
        int sz;
        if((c&0xE0)==0xC0){
            sz=2;
            r=c&0x1F;
        } else if((c&0xF0)==0xE0){
            sz=3;
            r=c&0x0F;
        } else if((c&0xF8)==0xF0){
            sz=4;
            r=c&0x07;
        } else if((c&0xFC)==0xF8){
            sz=5;
            r=c&0x03;
        } else if((c&0xFE)==0xFC){
            sz=6;
            r=c&0x01;
        } else {
            u8error("invalid utf-8 prefix");
            (*q)++;
            return 0xFFFF;
        }
        t++;
        int i;
        for(i=1;i<sz;i++){
            if((*t&0xC0)==0x80){
                r=(r<<6)+(*t++&0x3F);
            } else {
                u8error("utf-8 encoding %d bytes too short",sz-i);
                (*q)++;
                return 0xFFFF;
            }
        }
    } else {
        r=*t++;
    }
    *q=(char *)t;
    return r;
}
```

which goes the other direction. There is also a routine for checking the length of the next UTF-8 byte sequence.

```
1  int
2  u8len(char *t)
3  {
4      unsigned int c=(unsigned char)*t;
5      if(c>0x7F){
6          int sz;
7          if((c&0xE0)==0xC0) sz=2;
8          else if((c&0xF0)==0xE0) sz=3;
9          else if((c&0xF8)==0xF0) sz=4;
10         else if((c&0xFC)==0xF8) sz=5;
11         else if((c&0xFE)==0xFC) sz=6;
12         else return 1;
13         int i;
14         for(i=1;i<sz;i++){
15             c=(unsigned char)*++t;
16             if((c&0xC0)!=0x80) return 1;
17         }
18         return sz;
19     }
20     return 1;
21 }
```

This is used in the `instring` output routine mentioned about to make sure we don't insert line breaks in the middle of a UTF-8 byte sequence as it is output to the assembler.

These routines are used both by the preprocessor and the compiler. Unfortunately, parse errors in these programs are reported using functions with different names. I have introduced the error reporting function `u8error`, which is a wrapper for `werror` in the compiler and `warning` in the preprocessor.

Conclusions

My implementation of unicode for pcc took about a week. With my patches, cpp accepts UTF-8 encoded input files and generates executables which produce output exactly the same as clang. For some reason gcc doesn't support UTF-8 input. However, it is possible to translate all UTF-8 sequences to UCN escapes thereby creating an ASCII file that gcc will compile. The shell script

```
#!/bin/bash
case $# in
1)
    ;;
*)
    echo Wrong number of arguments!
    exit 1;
    ;;
esac
i=$1
o=${1%.*}-ucn.c
echo Converting $i to $o...
cat $i | perl -pe 'BEGIN {
    binmode STDIN,":utf8";
} s/(.)/ord($1)<128 ? $1 : sprintf("\\U%08x",ord($1))/ge;
' > $o
echo ...done
```

takes a program such as

```
#include <stdio.h>
double ε=0.1;
int main(){
    printf("ε=%g\n",ε);
    return 0;
}
```

and translates it to

```
#include <stdio.h>
double \U000003b5=0.1;
int main(){
    printf("\U000003b5=%g\n",\U000003b5);
    return 0;
}
```

Note that UCN escapes have replaced the UTF-8 byte sequences to create a valid ASCII file. In order to compile this with gcc use the command

```
gcc -std=gnu99 -fextended-identifiers example3-ucn.c
```

This file also compiles with pcc. When the resulting pcc executable is run, it produces output identical to the output produced by the executable generated by gcc.

```
ε=0.1
```

No matter how this example program is stored or which compiler it is compiled with, the resulting output of this program is as shown above.

To conclude that pcc, gcc and clang are consistent with their unicode implementations. It is also important to note that the global symbols from either compiler are the same. This can be seen in the assembler output. From pcc we have

```
    .data
    .align 8
    .globl ε
    .type ε,@object
    .size ε,8
ε:
    .long    0x9999999a,0x3fb99999
    .section .rodata
.L230:
    .ascii "ε=%g\012\0"
    .text
    .align 4
    .globl main
    .type main,@function
main:
    pushq %rbp
    movq %rsp,%rbp
    subq $16,%rsp
.L227:
.L229:
    movabsq $.L230,%rax
    movq %rax,-8(%rbp)
    movsd ε(%rip),%xmm0
    movq -8(%rbp),%rdi
    movl $1,%eax
    call printf
    movl $0,-12(%rbp)
    jmp .L228
.L228:
    movl -12(%rbp),%eax
    leave
    ret
    .size main,.-main
    .ident "PCC: pcc 1.1.0.ejo 20140512 for x86_64-unknown-linux-gnu"
    .end
```

whereas the gcc output is

```
    .file    "example3-ucn.c"
    .globl    ε
    .data
    .align 8
    .type    ε, @object
    .size    ε, 8
ε:
    .long    2576980378
    .long    1069128089
    .section    .rodata
.LC0:
    .string    "\316\265=%g\n"
    .text
```

```
14      .globl    main
15      .type    main, @function
16  main:
17  .LFB0:
18      .cfi_startproc
19      pushq    %rbp
20      .cfi_def_cfa_offset 16
21      .cfi_offset 6, -16
22      movq    %rsp, %rbp
23      .cfi_def_cfa_register 6
24      subq    $16, %rsp
25      movq    ε(%rip), %rax
26      movq    %rax, -8(%rbp)
27      movsd    -8(%rbp), %xmm0
28      movl    $.LC0, %edi
29      movl    $1, %eax
30      call    printf
31      movl    $0, %eax
32      leave
33      .cfi_def_cfa 7, 8
34      ret
35      .cfi_endproc
36  .LFE0:
37      .size    main, .-main
38      .ident    "GCC: (Debian 4.7.2-5) 4.7.2"
39      .section    .note.GNU-stack,"",@progbits
```

Although gcc doesn't accept UTF-8 encoded source files, it does generate UTF-8 encoded assembler output. In particular, the global variable `ε` is coded using UTF-8 in the same way for pcc, gcc and clang compiler. Thus, it is possible to link libraries written using unicode identifiers that were compiled with a mixture of these compilers.

Currently 80 percent of web pages available on the internet use UTF-8 encoding. While many of those pages contain only the ASCII subset of UTF-8, this the backward compatibility and popularity make it reasonable to support unicode using UTF-8 in pcc. As this document describes, that has now been done.

In the near universal adoption of UTF-8 there is a notable exception. Microsoft uses UTF-16. The reason is not intentional incompatibility, but likely historical: Microsoft planned Windows NT before UTF-8 existed and implemented a two-byte format called UCS-2 which was later extended to UTF-16 when the number of unicode characters exceeded 2^{16}. As a result, Notepad and the Microsoft Visual C editors store unicode files using UTF-16. These source files must be translated to UTF-8 before being compiled by pcc. Note that the Microsoft compiler also declares `wchar_t` as a 16-bit integer. This must also be taken into account when moving programs between operating systems.

UTF-16 wide string can be obtained in a pcc program by converting from a UTF-8 or UTF-32 encoded string constant at run time. Again, the only need for UTF-16 is for making Microsoft Windows system calls or talking to Microsoft file servers and databases at a systems level. When pcc is used in a Windows environment, suitable wrappers could perform UTF-8 to UTF-16 conversions for the Windows system calls.