EFI C Library Internationalization Support Design

Revision 0.11(Internal)

May 29, 2000

ESG Server Software Technologies (SST)

Revision History

Date	Revision	Author	Modifications
01/24/00	0.01	TC	Initial Version
02/03/00	0.02	TC/SL	Add locale and wide character functions info
03/02/00	0.03	TC	Update the design of classification functions, and stream i/o functions
03/06/00	0.04	TC	Add wchar.h description
03/09/00	0.05	TC	Add section System I/O functions
03/10/00	0.06	TC	Update wide character functions design
03/13/00	0.07	SL	Update locale part design
03/13/00	0.08	TC/SL	Document refinement
04/10/00	0.09	TC	Add getws, putws routines
05/29/00	0.11	TC/SL	Remove section 3.2, old locale mechanism, and integrated the protocol-based locale document into this doc.

Disclaimers

Table Of Figures

1 Introduction

This document provides the specification for EFI C Library i18n support.

1.1 Scope

This Specification provides detailed design information about the EFI C Library i18n support.

1.2 Target Audience

This Specification targets individuals who wish to understand the product functionality provided and the implementation details of C Library i18n support for EFI. It is not a pure user manual, because some architecture and design information are included. The reader will also find information in this document to aid in understanding the i18n support provided by EFI C Library.

1.3 Reference Documents

The following documents were useful in preparing this specification:

- *Extensible Firmware Interface Specification*. Version 0.92, Jan 19, 2000.
- *EFI Developer's Guide*. Version 0.2, July 14, 1999.
- *EFI Application Toolkit Product Requirements Document*. Revision 0.97, Sept. 27, 1999.
- *FreeBSD ver 3*.2 source code.
- *ISO/ANSI 2000 draft specification for Programming languages – C.*

2 EFI C Library i18n Support Design

EFI C Library i18n support includes two major parts:

- Locale support
- Wide character functions support

Currently, the ASCII version of EFI libc is ported from FreeBSD libc code. In general, EFI libc i18n support will be based on currently EFI libc and FreeBSD libc code investigation.

FreeBSD v3.2 has it’s own locale mechanism, but not fully fit for the EFI environment requirement, there are some significant enhancement in libc locale mechanism.

FreeBSD doesn’t support any wide character functions, EFI Libc wide character function support will be based on the corresponding ASCII version of libc.

3 Locale support

3.1 Overview

The locale uses a set of language and cultural rules. These cover aspects such as language for messages, different character sets, lexigraphic conventions, etc. A program needs to be able to determine its locale and act accordingly to be portable to different cultures.

FreeBSD's default locale is C/POSIX locale.

There are different categories for locale information a program might need.

LC_COLLATE

Collation of strings (sort order), because in some language character order is not the same as lexigraphic order.

LC_CTYPE

Classification and conversion of characters

LC_MESSAGES

Translation of yes and no, etc.

LC_MONETARY

Format of monetary values

LC_NUMERIC

Format of non-monetary numeric values

LC_TIME

Date and time formats

LC_ALL

Sets all of the above (overrides all of them)

And in FreeBSD, there are several functions used for management of message categories, which allow standard message strings to be stored in a standard database for access via the locale mechanism. FreeBSD 3.2 seems to provide message database for only US English, French, German and Polish.

3.2 FreeBSD's supports & nonsupports

FreeBSD's locale mechanism now supports multibyte.

It now supports the following 5 types of encodings: NONE, EUC, BIG5, UTF2, MSKANJI.

For the six locale categories, FreeBSD implements LC_COLLATE, LC_CTYPE, and LC_TIME. And now does not actually implement LC_MESSAGES, LC_MONETARY and LC_NUMERIC.

Below is a figure summarizing locale category implementation.

Category	LC_CTYPE	LC_COLLATE	LC_TIME
Key Data Structure	_RuneLocale	struct __collate_st_char_pri struct __collate_st_chain_pri	struct lc_time_T

Key Vari-ables	_RuneLocale _DefaultRune : default C/POSIX locale
	
	RuneLocale *_CurrentRuneLocale: point to the current rune locale
		u_char __collate_substitute_table [UCHAR_MAX + 1][STR_LEN];
		
		struct __collate_st_char_pri __collate_char_pri_table[UCHAR_MAX + 1]
		
		struct __collate_st_chain_pri __collate_chain_pri_table [TABLE_SIZE];
			struct lc_time_T _time_localebuf; //current time locale
			
			int _time_using_locale;
			
			struct lc_time_T _C_time_locale; //default “C” time locale

Locale File De-fault Location	/usr/share/locale/locale_name/ LC_CTYPE	/usr/share/locale/locale_na me/LC_COLLATE	/usr/share/locale/locale_nam e/LC_TIME

Below is the detailed analysis of FreeBSD’s locale load mechanism. And since LC_CTYPE is the basis of the whole locale mechanism, these two parts will be discussed together, which will be helpful to understand FreeBSD’s locale mechanism much better.

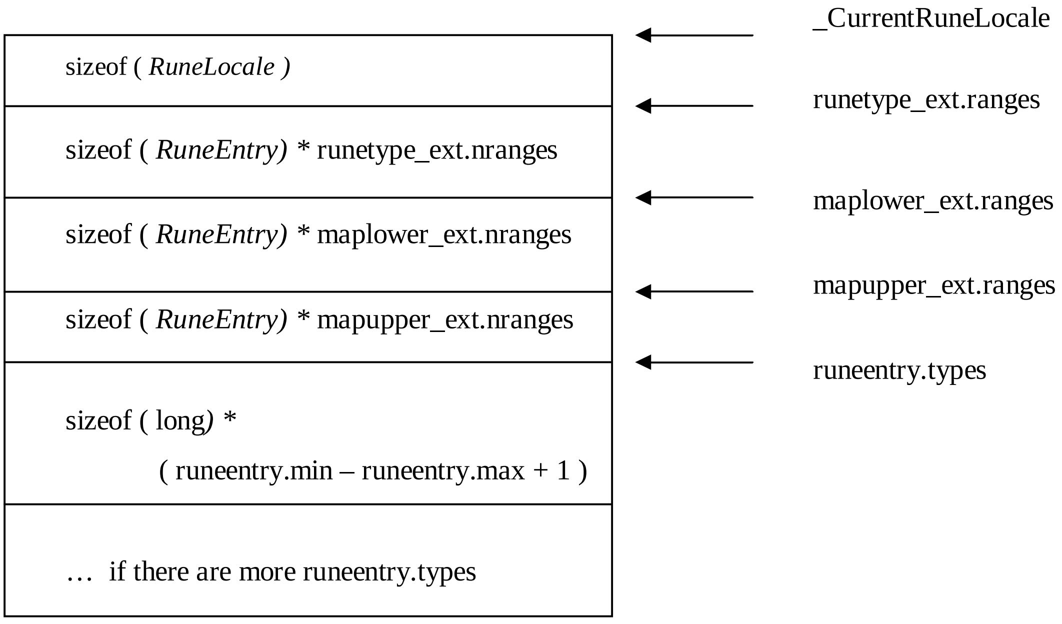
3.2.1 FreeBSD Locale Key Data Structure

3.2.1.1 _RuneLocale

```
typedef struct{
        char        magic[8];     /* Magic saying what version we are.
                                  Currently it is "RuneMagi"*/
        char        encoding[32];       /* ASCII name of this encoding.
                                  For example,UTF2, EUC, etc.*/
```

```
      /*--------------------------------------------------------------------
-
      Each encoding is specified by 3 functions:
      encoding_init
            This function sets up the requested locale.  It assigns
            the s{get,put}rune functions, sets __mb_cur_max (the
            maximum number of 8 bit characters in a multibyte sequence)

      encoding_sgetrune
      encoding_sputrune
            The encoding specific versions of these routines
      --------------------------------------------------------------------
*/
      rune_t (*sgetrune) __P((const char *, size_t, char const **));
      /* Basic function used to read a single rune from a buffer.
      Different according to different encoding. A rune's type is rune_t.
      And rune_t's ultimate definition is int */

      int (*sputrune) __P((rune_t, char *, size_t, char **));
      /* Basic function used to write a single rune from a buffer.
      Different according to different encoding */

      rune_t       invalid_rune;

      unsigned long      runetype[_CACHED_RUNES];
      rune_t       maplower[_CACHED_RUNES];
      rune_t       mapupper[_CACHED_RUNES];
      /*--------------------------------------------------------------------
-
      For the lower 256 runes we keep a simple arrays of
      information about them.
      The runetype is an array of bitfields (isprint, isalpha, etc).
      The other two arrays are the upper and lower case versions.
      --------------------------------------------------------------------
*/
      _RuneRange   runetype_ext;
      _RuneRange   maplower_ext;
      _RuneRange   mapupper_ext;
      /*--------------------------------------------------------------------
--
      The following are to deal with runes larger than _CACHED_RUNES - 1.
      Their data is actually contiguous with this structure so as to
      make it easier to read/write from/to disk. Refer to the following
```

```
        figure.
        ----------------------------------------------------------------------
*/
        void         *variable;
        /*---------------------------------------------------------------------
--
        Data which depends on the encoding. It's used only when loading
        rune locale information from file. Rune locale information can be
        divided two parts: length-fixed data and length-variant data. The
        length-fixed data is _RuneLocale. The length-variant data is those
        pointed by runetype, maplower and mapupper. The beginning of
        length-variant data is pointed by *variable when loading from file.
        But this field will never be used after loading. Refer to the
        following figure.
        ----------------------------------------------------------------------
*/      int          variable_len;      /* how long that length-variant data is
*/
} _RuneLocale;
```


sizeof (RuneLocale)
sizeof (RuneEntry) * runetype_ext.nranges
sizeof (RuneEntry) * maplower_ext.nranges
sizeof (RuneEntry) * mapupper_ext.nranges
sizeof (long) * (runeentry.min – runeentry.max + 1)
… if there are more runeentry.types
_CurrentRuneLocale
runetype_ext.ranges
maplower_ext.ranges
mapupper_ext.ranges
runeentry.types

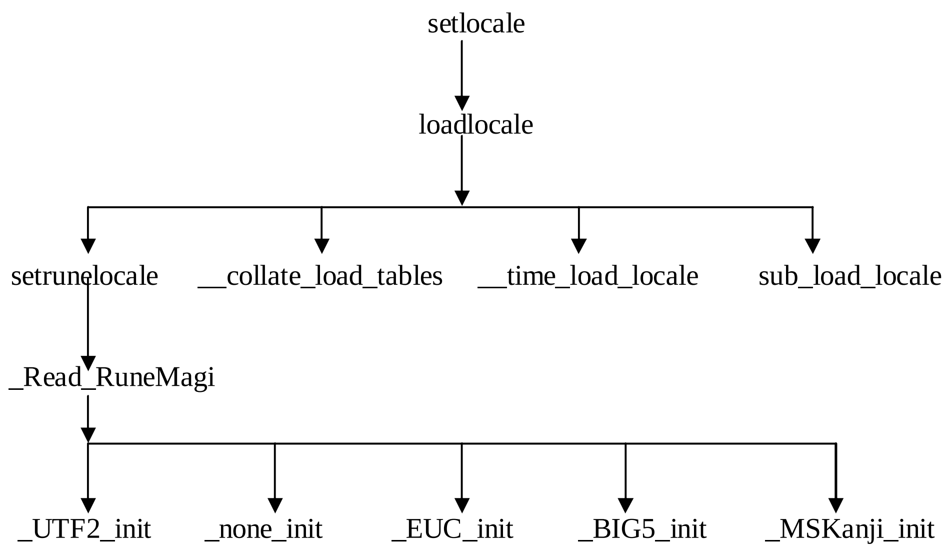
3.2.1.2 _RuneLocale

```
typedef struct {
        int          nranges;      /* Number of ranges stored */
        _RuneEntry   *ranges;      /* Pointer to the ranges */
} _RuneRange;
```

3.2.1.3 _RuneEntry

```
typedef struct {
      rune_t       min;         /* First rune of the range */
      rune_t       max;         /* Last rune (inclusive) of the range */
      rune_t       map;
      /*---------------------------------------------------------------
      map is the type of the runes in this rune range.
      If map != 0, then all the runes from min to max are same type and
      their type is map.
      If map == 0, then not all the runes from min to max are same type.
      Then each rune has a type indicator in types.
      --------------------------------------------------------------
*/

      unsigned long      *types;            /* Array of types in range */
      /*---------------------------------------------------------------
-     if map != 0 , then types = NULL;
      if map ==0, then for rune c , types[c - min] is c's type.
      ---------------------------------------------------------------
*/
} _RuneEntry;
```

3.2.2 Locale File Load Mechanism



Below is detailed explaination of key locale file load mechanism

3.2.2.1 setlocale

Key variables used:

current_categories: the locales currently used

new_categories: the new locales will be used

saved_categories: just used to save temp locales in order to restore back

Parameter:

category: category want to set, value ranging from LC_ALL to LC_MESSAGES

locale: locale name

Process:

```
Analyze category & locale, and set new_categories
If  ( category != LC_ALL)
       loadlocale(category);
else
```

```
for ( i = 1; i < _LC_LAST; ++i ) {
      loadlocale(i);
      if  failed  restore all previous;
}
```

3.2.2.2 loadlocale

Key variables:

_PathLocale:	the path to find the locale file. If environment variable PATH_LOCALE has been set, then _PathLocale = env(PATH_LOCALE); else _PathLocale = Micro _PATH_LOCALE

Parameter:

category: category want to set, value ranging from LC_COLLATE to LC_MESSAGES

Process:

```
According to the logic mentioned above, set _PathLocale;
new = new_categories(category);
// from here start, differenet category locale
// will have different load logic
if ( category == LC_TYPE )
      setrunelocale(new);

if ( category == LC_COLLATE )
      __collate_load_tables(new);

if ( category == LC_TIME )
      __time_load_locale(new);

if ( category == LC_MONETARY ||
      category == LC_MESSAGES ||
      category == LC_NUMERIC )
      stub_load_locale(new);
```

3.2.2.3 setrunlocale, __collate_load_tables, __time_load_locale, stub_load_locale

In FreeBSD, different categories locale information is stored in different files. And they have different load procedures. The above 4 procedures are used to load different category locale information from files.

stub_load_locale actually do nothing. It just checks whether such locale file exists. So that it means FreeBSD does not actually support LC_MONETARY, LC_MESSAGES and LC_NUMERIC category locale.

3.2.2.4 setrunelocale

Parameter:

Encoding: locale naming. Note: this is not the encoding as UTF2, EUC, etc.

Process:

```
According to the logic mentioned above, set _PathLocale;
_Read_RuneMagi;     // just read the rune information
                    // from file to  _RuneLocale *rl
#ifdef XPG4
      if ( rl->encoding == “” || rl->encoding == “UTF2”)
            _UTF2_init(rl);

      if ( rl->encoding == “NONE”)
            _none_init(rl);

      if ( rl->encoding == “EUC” )
            _EUC_init(rl);

      if ( rl->encoding == “BIG5” )
            _BIG5_init(rl);

      if (rl->encoding == “MSKanji”)
            _MSKanji_init(rl);
#else
      if (rl->encoding == “NONE”)
            _none_init(rl);
```

3.2.2.5 _XXXX_init

set _CurrentRuneLocale to rl and load some encoding specific procedures into rl.

Parameter:

rl: locale will be inited

Process:

```
rl->sgetrune = _XXXX_sgetrune; // one encoding specific procedure
rl->sputrune = _XXXX_sputrune; // another encoding specific
                              // procedure
_CurrentRuneLocale = rl;
__mb_cur_max = n;              // this means under current
                              // encoding, the longest rune can
                              // be n bytes long.
```

3.3 EFI C Library locale support

3.3.1 General Set/Load locale mechanism

FreeBSD's locale system now supports multibyte, but in EFI C Library i18n, it should be wide character. Since wide character is just the subset of multibyte, even without modify FreeBSD's locale code, it can be used to set/load wide character locale.

So here are two choices:

1) Use the multibyte locale system

2) Modify it to a dedicated wide character system

The two choices have their own advantages & disadvantages:

Choice 1) requires less work and are more flexible when we have multibyte encoding requirements in the future.

Choice 2) is more fit for current requirements.

Here selecting choice 1).

3.3.2 Export library interface

2 locale functions are exported for user to call:

setlocale: used to set & load locale

localeconv: used to get lconv information

3.3.3 Environment variables

One environment variable are defined:

- LANGUAGE

LANGUAGE denotes the current locale name. Its naming convention follows ISO 639:1988. I will explain just a little bit here. About the detail, please refer to ISO 639:1988.

LANGUAGE'S naming convention is as the format lang_LOCALE.
For example, for traditional Chinese it should be zh_TW. For US English, it should be en_US.

For example if user want to use a traditional Chinese locale, then he just need to type 'set LANGUAGE zh_TW' in EFI shell. Then afterwards, applications will all follow the updated locale accordingly.

In EFI global environment variable list, there is already a variable Lang, which denotes the language code that the system is configured for. But we can't use Lang in the locale mechanism. Below are the reasons:

1) Lang's value change will only take effect after the next boot. That is not fit with our requirement.
2) Lang's naming convention follows IS0-639-2. For example, for English, there is only one value "eng". It's unable to show the differences among the English of different countries, such as American English, British English, etc.
3) In current EFI core, during initialization, Lang is always set to "eng".

3.3.4 Locale initialization

When application starting up, it need to initialize a locale. And since all locale-dependent functions are in libc, locale initialization need to be in libc initialization.

What should change to InitializeLibC():
Just add a call as setlocale(LC_ALL,"") to initialize locale.

setlocale(LC_ALL,"") will automatically set a locale according to environment variable LANGUAGE. If LANGUAGE is not set or the value is wrong, then will use the old locale, or the default C/POSIX locale.

3.3.5 Protocol-based Locale Mechanism

In the original FreeBSD locale implementation, there is a large block of data (namely, the Unicode map table) that is statically linked with the user application, which makes the user application very large in size. Other locale specific data are stored in some files (LC_CTYPE, LC_COLLATE, LC_TIME, etc), which will be read into memory and handled during application run time. These files make directory structures unnecessarily complicated.

To solve these problems, a new mechanism is established. That mechanism wraps locale specific data and functions into a protocol and lets user applications dynamically load and make use of the protocol during run time. This new protocol called ‘EFI_LOCALE protocol’ (‘EFI_LOCALE protocol’ is sometimes simply called ‘locale protocol’ hereafter). Locale specific data are no longer linked into user applications or stored in files, they are now stored in locale protocol that is able to be dynamically loaded and used by multiple applications. To be more precise, all the data that are specific to a certain locale are wrapped into one implementation of the locale protocol (that is, each locale is represented by one EFI image). Note that all the implementations of the locale protocol share the same protocol GUID.

This protocol-based mechanism has two advantages:

1) The size of user application is greatly reduced;

2) There are no unnecessary resource files left. Each locale is represented by a single EFI image.

Once a locale protocol image is loaded into memory, it stays resident. For each locale, one copy of loaded image is enough because the data are read-only to applications. Unload mechanism is not provided here. All the locale protocol images are unloaded when *ExitBootService()* are called to terminate EFI boot services.

More details regarding this locale protocol will be discussed in the next two chapters.

4 Protocol-based Locale Mechanism

In this section, we will discuss the detailed design of this protocol-based locale mechanism.

4.1 EFI_LOCALE Protocol

4.1.1 Overview

A new protocol called EFI_LOCALE protocol is introduced. For convenience, sometimes it's simply called 'locale protocol'. In locale protocol, there are locale specific data and functions. These data and functions are wrapped by protocol interface and accessed through protocol services exposed by the interface.

Namely, there are four types of data that are wrapped:

1) Unicode map table
2) Ctype data
3) Collate data
4) Time data

And there are also several functions that are wrapped.

4.1.2 Definition

GUID

```
// {0D58ED19-DAE9-401d-A2DC-02EC387E36D6}
#define EFI_LOCALE_PROTOCOL \
  { 0x0d58ed19, 0xdae9, 0x401d, 0xa2, 0xdc, 0x2, 0xec, 0x38, 0x7e, 0x36, 0xd6 }
```

Revision Number

```
#define EFI_LOCALE_INTERFACE_REVISION   0x00010000
```

Protocol Interface Structure

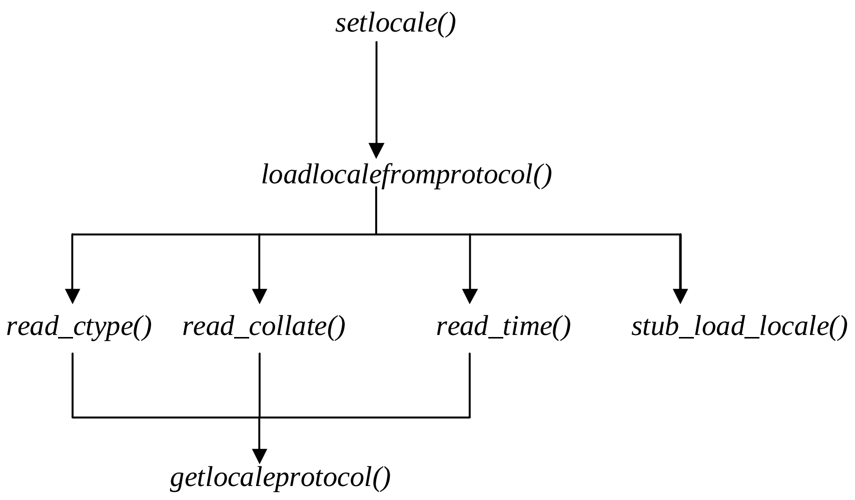
```
/*
 *  locale protocol interface
 */
typedef struct _EFI_LOCALE {
    UINT64                          Revision;

// Return locale name
    EFI_GETLOCALE                   getlocale;

// Read a single rune from a buffer
```

```
    EFI_SGETRUNE                    sgetrune;

// Write a single rune to a buffer
    EFI_SPUTRUNE                    sputrune;

// Return longest size of a rune
    EFI_GETMBCURMAX                 getmbcurmax;

// Return pointer to Unicode map table
    EFI_GETUNICODEMAP               getunicodemap;

// Return Unicode map table's entry count
    EFI_GETUNICODEMAPENTRYCOUNT     getunicodemapentrycount;

// Return pointer to Ctype data
    EFI_GETCTYPEDATA                getctypedata;

// Return size of Ctype data
    EFI_GETCTYPEDATASIZE            getctypedatasize;

// Return pointer to Collate data
    EFI_GETCOLLATEDATA              getcollatedata;

// Return size of Collate data
    EFI_GETCOLLATEDATASIZE          getcollatedatasize;

// Return pointer to Time data
    EFI_GETTIMEDATA                 gettimedata;

// Return size of Time data
    EFI_GETTIMEDATASIZE             gettimedatasize;

// Set invalid rune value
    EFI_SETINVALIDRUNE              setinvalidrune;
} EFI_LOCALE_INTERFACE;
```

4.2 Locale Setting Process In Lib C

The users of the locale protocol are the applications that make use of Lib C. In the Lib C initialization process, there is a function call to *setlocale()* (which is an exposed Lib C function itself) which sets current locale to the language indicated by the environment variable LANGUAGE. After Lib C initialization, applications can still make calls to *setlocale()* to set current locale to languages other than that indicated by LANGUAGE.

Following is the Lib C locale setting process function call hierarchy:



In Lib C implementation, function *setlocale()* makes internal calls to *loadlocalefromprotocol()*, and *loadlocalefromprotocol()* sees which category is being set and calls one of the following functions: *read_ctype()*, *read_collate()*, *read_time()* or *stub_load_locale()* . In each of these functions except *stub_load_locale()*, there is a call to *getlocaleprotocol()* which load and start the right protocol image if needed and return the interface pointer. Once the interface pointer of the protocol is acquired, *read_ctype()*, *read_collate()* and *read_time()* functions then get the references to the locale specific data provided by the protocol through services exposed by the protocol interface, and set locale related global variables in Lib C to correct values according to what is learned from those data.

The above sequence of function calls makes obsolete the FreeBSD sequence of function calls which involves *loadlocale()*, *setrunelocale()*, *__collate_load_tables()*, *__time_load_locale()* and *_Read_RuneMagi()*.

Once all the locale related global variables are set, the locale setting process is completed and locale information is available for use.

Following is a one by one explanation of the functions involved in locale setting process. Please refer to *lib\libc\locale\setlocale.c* for detailed information.

4.2.1 setlocale()

This function analyzes category & locale, and set new categories.

Pseudo code

```
if  ( category != LC_ALL)

       loadlocalefromprotocol(category);

else
```

```
    for ( i = 1; i < _LC_LAST; ++i ) {
            loadlocalefromprotocol(i);
            if failed restore all previous;
    }
```

4.2.2 loadlocalefromprotocol()

This function sets locale specific data for specific category.

Pseudo code

```
if (category == LC_CTYPE) {
       read_ctype(newlocale);
       if failed read_ctype(oldlocale);
}
else if (category == LC_COLLATE) {
       read_collate(newlocale);
       if failed read_collate(oldlocale);
}
else if (category == LC_TIME) {
       read_time(newlocale);
       if failed read_time(oldlocale);
}
else if (category == LC_MONETARY ||
        category == LC_MESSAGES ||
        category == LC_NUMERIC) {
       stub_load_locale(newlocale);
       if failed stub_load_locale(oldlocale);
}
```

4.2.3 read_ctype(), read_collate(), read_time() and stub_load_locale()

These four functions perform actual global variable setting tasks for a certain category. The *stub_load_locale()* function actually does nothing real work. In each of the other three functions, a call to *getlocaleprotocol()* is made to obtain an interface pointer of the locale protocol so as to get locale specific information from the protocol and set related global variables.

4.2.4 getlocaleprotocol()

This function gets the locale protocol interface pointer for a certain locale.

Pseudo code

```
Find expected locale protocol in current loaded protocols
```

```
if expected protocol is not found {
      Load protocol image of the expected locale;
      Start this protocol image;
      Locate expected protocol again;
      if still not found, return NULL;
}
Expected protocol is found, return the locale protocol interface pointer;
```

5 Locale Protocol Implementation

This section explains what should be done to add a certain locale support to Lib C.

The naming convention for locale protocol name is: ‘language_LOCALE’ (such as ‘zh_CN’, ‘en_US’, etc).

The following explanation is based on the locale zh_TW (with the encoding BIG5) as an example (For each locale, there is an encoding). Please refer to the source code files of zh_TW locale protocol implementation located in *protocols\locale\zh_tw* for more information. The result is an EFI image named ‘*zh_tw.efi*’ built from these source code files.

5.1 Code Structure

The definition of type EFI_LOCALE_INTERFACE is located in *include\bsd\locale_protocol.h*.

The rest of the source files are located in *protocols\locale\zh_tw*. These files constitute the implementation of zh_TW locale protocol. Following functions are to be found in these files.

5.1.1 External Functions

Following are external functions.

The *EfiLocaleEntry()* function is the entry point of the locale protocol image. And there are several wrapper functions that implement the EFI_LOCALE protocol interface functions. These wrapper functions calls internal *big5_XXXX* functions or refer to locale specific data.

5.1.1.1 EfiLocaleEntry()

This function is the entry point of the zh_TW locale protocol image. It installs the protocol interface for zh_TW locale protocol.

5.1.1.2 EFI_LOCALE.getlocale()

This function returns the locale name.

In zh_TW, the function is implemented as follows:

```
static char*
getlocale (void)
{
	return "zh_TW";
}
```

5.1.1.3 EFI_LOCALE.sgetrune()

This function returns a single rune from a buffer.

In zh_TW, the function is implemented as follows:

```
static rune_t
sgetrune (
        IN              const char   *string,
        IN              size_t       n,
        IN OUT          char const   **result
        )
{
        return big5_sgetrune(string, n, result);
}
```

5.1.1.4 EFI_LOCALE.sputrune()

This function writes a single rune to a buffer.

In zh_TW, the function is implemented as follows:

```
static int
sputrune (
        IN              rune_t       c,
        IN              char         *string,
        IN              size_t       n,
        IN OUT          char         **result
        )
{
        return big5_sputrune(c, string, n, result);
}
```

5.1.1.5 EFI_LOCALE.getmbcurmax()

This function returns the longest size of a rune under current encoding.

In zh_TW, the function is implemented as follows:

```
static int
getmbcurmax (void)
{
        return big5_getmbcurmax();
}
```

5.1.1.6 EFI_LOCALE.getunicodemap()

This function returns pointer to Unicode map table.

In zh_TW, the function is implemented as follows:

```
static _ToUnicodeMap*
getunicodemap (void)
{
        return big5_getunicodemap();
}
```

5.1.1.7 EFI_LOCALE.getunicodemapentrycount()

This function returns Unicode map table's entrycount.

In zh_TW, the function is implemented as follows:

```
static int
getunicodemapentrycount (void)
{
        return big5_getunicodemapentrycount();
}
```

5.1.1.8 EFI_LOCALE.getctypedata()

This function returns pointer to Ctype data.

In zh_TW, the function is implemented as follows:

```
static UINT8*
getctypedata (void)
{
        return CtypeData;
}
```

5.1.1.9 EFI_LOCALE.getctypedatasize()

This function returns size of Ctype data.

In zh_TW, the function is implemented as follows:

```
static int
getctypedatasize (void)
{
        return CtypeDataSize;
}
```

5.1.1.10 EFI_LOCALE.getcollatedata()

This function returns pointer to Collate data.

In zh_TW, the function is implemented as follows:

```
static UINT8*
getcollatedata (void)
{
        return CollateData;
}
```

5.1.1.11 EFI_LOCALE.getcollatedatasize()

This function returns size of Collate data.

In zh_TW, the function is implemented as follows:

```
static int
getcollatedatasize (void)
{
      return CollateDataSize;
}
```

5.1.1.12 EFI_LOCALE.gettimedata()

This function returns pointer to Time data.

In zh_TW, the function is implemented as follows:

```
static UINT8*
gettimedata (void)
{
      return TimeData;
}
```

5.1.1.13 EFI_LOCALE.gettimedatasize()

This function returns size of Time data.

In zh_TW, the function is implemented as follows:

```
static int
gettimedatasize (void)
{
      return TimeDataSize;
}
```

5.1.1.14 EFI_LOCALE.setinvalidrune()

This function sets invalid rune value.

In zh_TW, the function is implemented as follows:

```
/*
*The value of 'invalid_rune' is only relevant to the machine byte order
*and the ctype data provided by this protocol,
*so multiple protocol users (always in one machine) will set the same value
*to 'invalid_rune', which has an equivalent result as a read-only
*'invalid_rune' once it is set by certain client.
*/
static EFI_STATUS
setinvalidrune( rune_t val )
{
    invalid_rune = val;
    return (EFI_SUCCESS);
}
```

5.1.2 Internal Functions and Data

The following are the locale specific functions and data that are referred by the wrapper functions.

5.1.2.1 big5_sgetrune()

This is the basic function used to read a single rune from a buffer.

```
rune_t
big5_sgetrune(string, n, result)
        const char *string;
        size_t n;
        char const **result;
{
        rune_t rune = 0;
        int len;

        if (n < 1 || (len = _big5_check(*string)) > (int)n) {
                if (result)
                        *result = string;
                return (_INVALID_RUNE);
        }
        while (--len >= 0)
                rune = (rune << 8) | ((u_int)(*string++) & 0xff);
        if (result)
                *result = string;
return rune;
}
static  int
_big5_check(c)
        u_int c;
{
        c &= 0xff;
        return ((c >= 0xa1 && c <= 0xfe) ? 2 : 1);
}
```

5.1.2.2 big5_sputrune()

This is the basic function used to write a single rune to a buffer.

```
int
big5_sputrune(c, string, n, result)
        rune_t c;
        char *string, **result;
        size_t n;
```

```
{
	if (c & 0x8000) {
		if (n >= 2) {
			string[0] = (c >> 8) & 0xff;
			string[1] = c & 0xff;
			if (result)
				*result = string + 2;
			return (2);
		}
	}
	else {
		if (n >= 1) {
			*string = c & 0xff;
			if (result)
				*result = string + 1;
			return (1);
		}
	}
	if (result)
		*result = string;
	return (0);
}
```

5.1.2.3 big5_getmbcurmax()

This function returns the longest size of a rune under current encoding.

```
int
big5_getmbcurmax(void)
{
	return 2;
}
```

5.1.2.4 big5_getunicodemap()

This function returns reference to the to_unicode_map table

```
_ToUnicodeMap*
big5_getunicodemap(void)
{
	return to_unicode_map;
}
```

5.1.2.5 big5_getunicodemapentrycount()

This function returns to_unicode_map_entrycount

```
int
big5_getunicodemapentrycount(void)
{
       return to_unicode_map_entrycount;
}
```

5.1.2.6 to_unicodemap

These variables store the Unicode map table and its entrycount.

```
_ToUnicodeMap to_unicode_map[];
int to_unicode_map_entrycount;
```

5.1.2.7 CtypeData

These variables store Ctype data and its size.

```
UINT8   CtypeData[];
int     CtypeDataSize;
```

5.1.2.8 CollateData

These variables store Collate data and its size.

```
UINT8   CollateData[];
int     CollateDataSize;
```

5.1.2.9 TimeData

These variables store Time data and its size.

```
UINT8   TimeData[];
int     TimeDataSize;
```

5.2 Set Locale Specific Data

There are four types of locale specific data that has to be set in source files. They are:

1) **Unicode map table**

   Variables: *to_unicode_map*, *to_unicode_map_entrycount*,

   In file: *tounicodemap_big5.h*

2) **Ctype data**

   Variables: *CtypeData*, *CtypeDataSize*,

   In file: *ctype_big5.h*

3) **Collate data**

   Variables: *CollateData*, *CollateDataSize*,

In file: *collate_big5.h*

4) Time data

Variables: *TimeData, TimeDataSize,*

In file: *time_big5.h*

Among them, Ctype data, Collate data and Time data are translated from resource source files:

- The path for resource source file for Ctype data is: *lib\libc\tools\mklocale\data\ctype\zh_TW.BIG5.src*
- The path for resource source file for Collate data is: *lib\libc\tools\mklocale\data\collate\zh_TW.BIG5.src*
- The path for resource source file for Time data is: *lib\libc\tools\mklocale\data\time\zh_TW.BIG5.src*

To translate these source files, there are three tools: **makectype, makecollate** and **maketime** that can be used to generate from resource source files three new files *LC_CTYPE*, *LC_COLLATE* and *LC_TIME*, respectively.

About the formats of resource source files, please refer to the manual page of **mkctype, mkcollate, mktime** respectively (in *doc\man\html*). And you can also find some pre-existing templates in the FreeBSD's distribution.

In FreeBSD 3.2, you can find those files in:

LC_CTYPE's resource file*:*	src\usr.bin\mklocale\data
LC_COLLATE's resource file*:*	src\usr.bin\colldef\data
*LC*_TIME's resource file*:*	src\usr.bin\timedef\data

All those resource files have extension .src.

Then use **bin2hex** tool to generate from *LC_CTYPE*, *LC_COLLATE* and *LC_TIME* three new files: *LC_CTYPE.txt*, *LC_COLLATE.txt* and *LC_TIME.txt*, respectively. Each of these three text files contains a block of text in hexadecimal format which can be inserted into the related source file. The inserted part serves as global variable initialization data.

- The text in *LC_CTYPE.txt* is copied to *ctype_big5.h* to initialize *CtypeData* and *CtypeDataSize*.
- The text in *LC_COLLATE.txt* is copied to *collate_big5.h* to initialize *CollateData* and *CollateDataSize.*
- The text in *LC_TIME.txt* is copied to *collate_big5.h* to initialize *TimeData* and *TimeDataSize.*

The tools mentioned above are located in *lib\libc\tools\mklocale\bin* and *lib\libc\tools\bin2hex\bin*.

6 Wide character functions support

6.1 wchar.h

Basically, all the wide character functions except classification functions are listed in wchar.h. To avoid redefinition problems, some structures listed in stdio.h such as `FILE` should be redefined in wchar.h.

```
#ifndef _FILE_DEFINED
typedef      _BSD_OFF_T_  fpos_t;
#define      _FSTDIO
struct __sbuf {
      ……
};
typedef      struct __sFILE {
      ……
} FILE;
#define _FILE_DEFINED
#endif  /* _FILE_DEFINED */
```

6.2 Formatted wide character input/output functions

Totally there are 12 functions, which can be separated into two categories:

- `fwprintf, vfwprintf`
- `swprintf, vswprintf`
- `wprintf, vwprintf`

and

- `fwscanf, vfwscanf`
- `swscanf, vswscanf`
- `wscanf, vwscanf`

There are 2 basic functions need to be implemented: `vfwprintf` and `vfwscanf`. All other functions are based on these 2 functions. These 2 basic functions can be implemented according to the corresponding ASCII version functions in FreeBSD.

Implementation example:

To implement `vfwprintf`, start with a copy of the corresponding ASCII implementation of the function `vfprintf`, and create a new wide character version of the function by converting char and char * arguments, variables, and return values to wchar_t and wchar_t *, respectively. Also need to implement some internal helper functions. There some wide character classification functions used in the implementation, so they should be ready first.

6.3 Wide character input/output functions

- `fgetwc, getwc, getwchar`

The wide character version of underneath routines `__sgetwc()`, `__srgetw()` should be implemented according to the corresponding ASCII version routines in FreeBSD.

- `fputwc, putwc, putwchar`

The wide character version of underneath routines `__sputwc()`, `__swwbuf()` should be implemented according to the corresponding ASCII version routines in FreeBSD.

- `ungetwc`

This function can be implemented according to ASCII version `ungetc()` in FreeBSD.

- `fgetws, fputws, getws, putws`

These functions can be written by the corresponding ASCII functions `fgets()` and `fputs()`. Since they all access the buffer directly, programmer only need to care the counts of the characters instead of bytes.

- `fwide`

Currently, just write a skeleton here.

6.4 Wide character system input/output functions

- `wmkdir, wrmdir`
- `wopen, wfopen`
- `wlstat, wstat`

Since EFI console is Unicode ready, and the ASCII version of Libc does some specific handling for EFI Console by doing ASCII/Unicode conversion internally, these wide character version functions can be implemented by first calling `wcstombs()` for the parameters that the type is wide character and then calling corresponding ASCII version functions, which will do `mbstowcs()` internally.

6.5 General wide string utilities

6.5.1 Wide string numeric conversion functions

- `wcstod, wcstol, wcstoul, wcstoq, wcstouq`

The above functions can be implemented according to the corresponding ASCII version functions in FreeBSD.

6.5.2 Wide string copying functions

- `wcscpy, wcsncpy`

These functions have already been implemented in phase 1.

6.5.3 Wide string concatenation functions

- `wcscat, wcsncat`

These functions have already been implemented in phase 1.

6.5.4 Wide string comparison functions

- `wcscmp, wcsncmp`

These functions have already been implemented in phase 1.

- `wcscoll`

This functions requires some internal collation functions:
`__collate_load_error, __collate_substitute, __collate_lookup`

So need to provide wide character version of these internal collation functions.

- `wcsxfrm`

This function requires some internal collation functions mentioned above, to make it locale aware.

6.5.5 Wide string search functions

- `wcschr, wcscspn, wcslen, wcspbrk, wcsrchr, wcsspn, wcsstr, wcstok`

These functions have already been implemented in phase 1.

6.5.6 Wide character array functions

- `wmemchr, wmemcmp, wmemcpy, wmemmove, wmemset`

These functions have already been implemented in phase 1.

6.6 Wide character time conversion functions

- `wcsftime`

This function can be implemented according to the corresponding ASCII source in FreeBSD.

- `wasctime`

This function can be implemented according to the corresponding ASCII source in FreeBSD.

- `wctime`

This function can be implemented according to the corresponding ASCII source in FreeBSD.

6.7 Extended multibyte and wide character conversion utilities

6.7.1 Single byte wide character conversion functions

- `btowc, wctob, mbsinit`

These routines have not been implemented.

6.7.2 Restartable multibyte/wide character conversion functions

- `mbrlen, mbrtowc, wcrtomb`

These routines have not been implemented.

6.7.3 Restartable multibyte/wide string conversion functions

- `mbsrtowcs, wcsrtombs`

These routines have not been implemented.

6.8 Wide character classification and mapping utilities <wctype.h>

6.8.1 Wide character classification functions

- `iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit`

These functions can be implemented according to the corresponding ASCII version functions. The ASCII version functions have fully considered the locale system by reading the information from the external variable `_CurrentRuneLocale`, `_DefaultRuneLocale`. Since the member `runetype` in structure `_RuneLocale` supports wide character, and the parameter type of these ASCII version functions is `int`, the wide character version functions can directly use the implementation of these ASCII version functions.

6.8.2 Extensible wide character classification functions

- `iswctype, wctype`

These functions provide extensible wide-character classification as well as testing equivalent to that performed by the functions described in 4.6.1. They needs an internal static table to list all the classification functions that libc supports.

```
static const struct _wctypetab {

        const char *s;

        int (*p)();

        wctype_t val;
        } wctypetab[] = {
        {"alnum", iswalnum, 1},
        {"alpha", iswalpha, 2},
        {"cntrl", iswcntrl, 3},
        {"digit", iswdigit, 4},
        {"graph", iswgraph, 5},
        {"lower", iswlower, 6},
        {"print", iswprint, 7},
        {"punct", iswpunct, 8},
        {"space", iswspace, 9},
        {"upper", iswupper, 10},
        {"xdigit", iswxdigit, 11},
        {(const char *)0, (int)0, 0}};
```

6.9 Wide character mapping utilities

6.9.1 Wide character case mapping functions

- `towlower, towupper`

These functions can be implemented according to the corresponding ASCII version functions. The ASCII version functions have fully considered the locale system by reading the information from the external variable `_CurrentRuneLocale`, `_DefaultRuneLocale`. Since the member `mapupper` and `maplower` in structure `_RuneLocale` supports wide character, and the parameter type of these ASCII version functions is `int`, the wide character version functions can directly use the implementation of these ASCII version functions.

6.9.2 Extensible wide character case mapping functions

- `towctrans, wctrans`

These functions provide extensible wide-character mapping as well as case mapping equivalent to that performed by the functions described in 4.7.1. They needs an internal static table to list all the classification functions that libc supports.

```
static const struct _wctranstab {
        const char *s;
        wint_t (*p)();
        wctype_t val;
        } wctranstab[] = {
        {"tolower", towlower, 1},
        {"toupper", towupper, 2},
      {(const char *)0, (wint_t)0, 0}};
```