Reversing (Undocumented) Windows Kernel Functions

Note: The functions that were analyzed are not found in Windows XP or lower.

This post will show my analysis of some undocumented APIs in the Windows kernel. The functions that were found relate to Locale IDs.

LCIDs are identifiers used to specify localizable information. They are also known as culture identifiers in the Microsoft® .NET Framework environment.

The name of a culture consists of its [ISO-639] language code, its [ISO-3166] country/region code, and an optional [ISO-15924] script tag for the written language. For example, the name of the culture in which the language is Bosnian (as written in Latin script and used in the Bosnia and Herzegovina region) is bs-Latn-BA.

More info: [MS-LCID] (PDF warning)

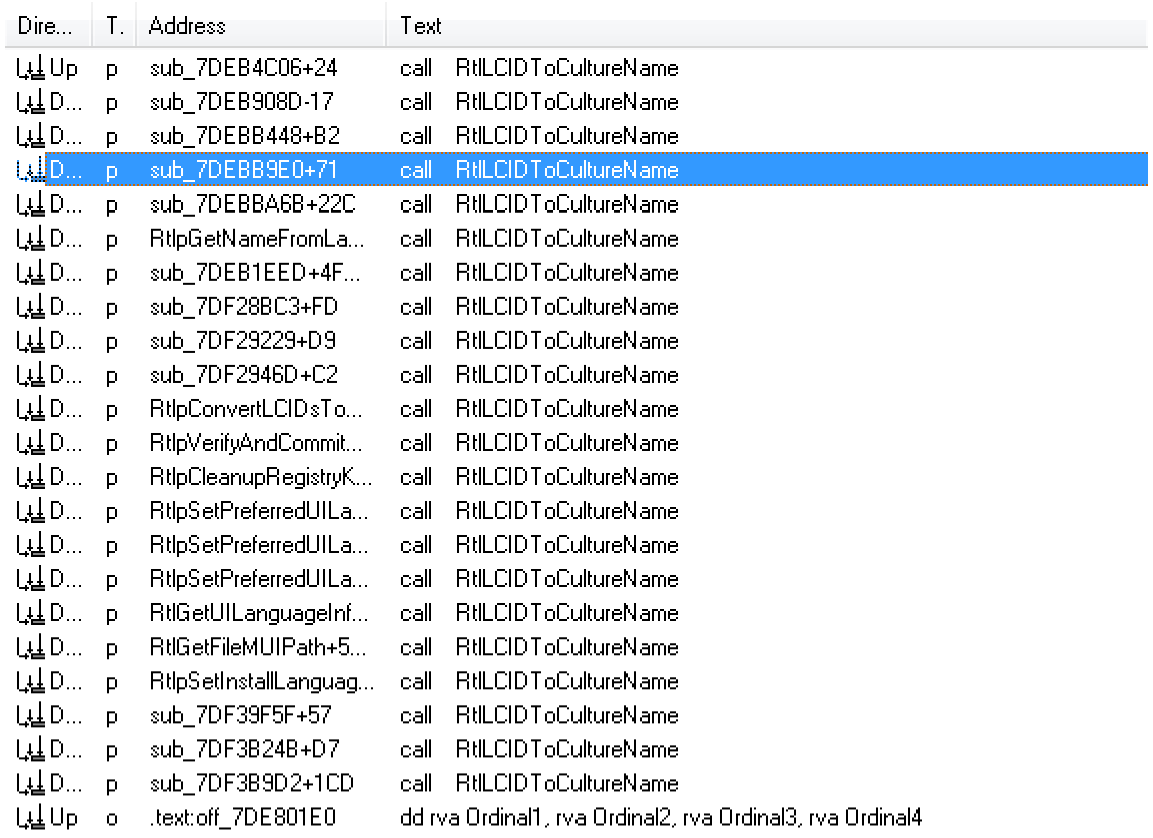
The functions that I will be analyzing are RtlLCIDToCultureName, RtlLcidToLocaleName, RtlCultureNameToLCID, RtlLocaleNameToLcid, and RtlConvertLCIDToString. The names alone give away a hint of what they probably do. The first two will be analyzed in depth and then sample code utilizing all five will be provided. There are a few additional functions with Rtlp* pertaining to LCIDs, but the generally Rtlp* functions are private functions and are meant only to be used within the native API, not to be exported and used by external programs.

RtlLCIDToCultureName (disassembly reproduced below)

```
.text:7DEB4C38                 public RtlLCIDToCultureName
.text:7DEB4C38 RtlLCIDToCultureName proc near          ; CODE XREF:
sub_7DEB4C06+24↑p
.text:7DEB4C38                                         ; sub_7DEB908D-17↓p
...
.text:7DEB4C38
.text:7DEB4C38 var_8           = word ptr -8
.text:7DEB4C38 var_4           = dword ptr -4
.text:7DEB4C38 arg_0           = dword ptr  8
.text:7DEB4C38 arg_4           = dword ptr  0Ch
.text:7DEB4C38
.text:7DEB4C38 ; FUNCTION CHUNK AT .text:7DEBB21D SIZE 0000006D BYTES
.text:7DEB4C38
.text:7DEB4C38                 mov     edi, edi
.text:7DEB4C3A                 push    ebp
.text:7DEB4C3B                 mov     ebp, esp
.text:7DEB4C3D                 push    ecx
.text:7DEB4C3E                 push    ecx
.text:7DEB4C3F                 push    ebx
.text:7DEB4C40                 push    esi
.text:7DEB4C41                 push    edi
.text:7DEB4C42                 mov     edi, [ebp+arg_0]
```

```
.text:7DEB4C45                 xor     ebx, ebx
.text:7DEB4C47                 cmp     edi, ebx
.text:7DEB4C49                 jz      short loc_7DEB4C88
.text:7DEB4C4B                 mov     esi, [ebp+arg_4]
.text:7DEB4C4E                 cmp     esi, ebx
.text:7DEB4C50                 jz      short loc_7DEB4C88
.text:7DEB4C52                 cmp     edi, 1000h
.text:7DEB4C58                 jz      short loc_7DEB4C88
.text:7DEB4C5A                 mov     eax, dword_7DF7208C
.text:7DEB4C5F                 cmp     eax, ebx
.text:7DEB4C61                 jz      short loc_7DEB4C77
.text:7DEB4C63                 lea     ecx, [ebp+arg_0]
.text:7DEB4C66                 push    ecx
.text:7DEB4C67                 push    ebx
.text:7DEB4C68                 push    edi
.text:7DEB4C69                 push    eax
.text:7DEB4C6A                 call    sub_7DEB4C96
.text:7DEB4C6F                 test    eax, eax
.text:7DEB4C71                 jge     loc_7DEBB21D
.text:7DEB4C77
.text:7DEB4C77 loc_7DEB4C77:                           ; CODE XREF:
RtlLCIDToCultureName+29↑j
.text:7DEB4C77                                         ;
RtlLCIDToCultureName+65FF↓j
.text:7DEB4C77                 push    0
.text:7DEB4C79                 push    2
.text:7DEB4C7B                 push    esi
.text:7DEB4C7C                 push    edi
.text:7DEB4C7D                 call    RtlLcidToLocaleName
.text:7DEB4C82                 test    eax, eax
.text:7DEB4C84                 jl      short loc_7DEB4C88
.text:7DEB4C86
.text:7DEB4C86 loc_7DEB4C86:                           ; CODE XREF:
RtlLCIDToCultureName+6646↓j
.text:7DEB4C86                 mov     bl, 1
.text:7DEB4C88
.text:7DEB4C88 loc_7DEB4C88:                           ; CODE XREF:
RtlLCIDToCultureName+11↑j
.text:7DEB4C88                                         ;
RtlLCIDToCultureName+18↑j ...
.text:7DEB4C88                 pop     edi
.text:7DEB4C89                 pop     esi
.text:7DEB4C8A                 mov     al, bl
.text:7DEB4C8C                 pop     ebx
.text:7DEB4C8D                 leave
.text:7DEB4C8E                 retn    8
.text:7DEB4C8E RtlLCIDToCultureName endp
```

Quite a function at first, but relatively easy to analyze. I started the analysis by looking at all of the cross references to this function (click to enlarge).



Dire...	T.	Address	Text
Up	p	sub_7DEB4C06+24	call RtlLCIDToCultureName
D...	p	sub_7DEB908D-17	call RtlLCIDToCultureName
D...	p	sub_7DEBB448+B2	call RtlLCIDToCultureName
D...	p	sub_7DEBB9E0+71	call RtlLCIDToCultureName
D...	p	sub_7DEBBA6B+22C	call RtlLCIDToCultureName
D...	p	RtlpGetNameFromLa...	call RtlLCIDToCultureName
D...	p	sub_7DEB1EED+4F...	call RtlLCIDToCultureName
D...	p	sub_7DF28BC3+FD	call RtlLCIDToCultureName
D...	p	sub_7DF29229+D9	call RtlLCIDToCultureName
D...	p	sub_7DF2946D+C2	call RtlLCIDToCultureName
D...	p	RtlpConvertLCIDsTo...	call RtlLCIDToCultureName
D...	p	RtlpVerifyAndCommit...	call RtlLCIDToCultureName
D...	p	RtlpCleanupRegistryK...	call RtlLCIDToCultureName
D...	p	RtlpSetPreferredUILa...	call RtlLCIDToCultureName
D...	p	RtlpSetPreferredUILa...	call RtlLCIDToCultureName
D...	p	RtlpSetPreferredUILa...	call RtlLCIDToCultureName
D...	p	RtlGetUILanguageInf...	call RtlLCIDToCultureName
D...	p	RtlGetFileMUIPath+5...	call RtlLCIDToCultureName
D...	p	RtlpSetInstallLanguag...	call RtlLCIDToCultureName
D...	p	sub_7DF39F5F+57	call RtlLCIDToCultureName
D...	p	sub_7DF3B24B+D7	call RtlLCIDToCultureName
D...	p	sub_7DF3B9D2+1CD	call RtlLCIDToCultureName
Up	o	.text:off_7DE801E0	dd rva Ordinal1, rva Ordinal2, rva Ordinal3, rva Ordinal4

This might lead to some hints as to what the types of the two parameters this function takes are. I found an interesting hint at

```
.text:7DEBBA51                  call    RtlLCIDToCultureName
```

By scrolling up and looking at the call in context, I found

```
.text:7DEBBA3C                  push    eax
.text:7DEBBA3D                  push    esi
.text:7DEBBA3E                  call    RtlInitUnicodeString
.text:7DEBBA43
.text:7DEBBA43 loc_7DEBBA43:                              ; CODE XREF:
sub_7DEBB9E0+78↓j
.text:7DEBBA43                                            ; sub_7DEBB9E0+40E8D↓j
.text:7DEBBA43                  mov     eax, edi
.text:7DEBBA45
.text:7DEBBA45 loc_7DEBBA45:                              ; CODE XREF:
sub_7DEBB9E0+84↓j
.text:7DEBBA45                  pop     edi
.text:7DEBBA46                  pop     esi
.text:7DEBBA47                  pop     ebp
.text:7DEBBA48                  retn    0Ch
.text:7DEBBA4B ; ---------------------------------------------------------------
---------------
.text:7DEBBA4B
.text:7DEBBA4B loc_7DEBBA4B:                              ; CODE XREF:
sub_7DEBB9E0+22↑j
.text:7DEBBA4B                  movsx   eax, word ptr [eax+4]
```

```
.text:7DEBBA4F
.text:7DEBBA4F loc_7DEBBA4F:                                   ; CODE XREF:
sub_7DEBB9E0+40E83↑j
.text:7DEBBA4F                 push    esi
.text:7DEBBA50                 push    eax
.text:7DEBBA51                 call    RtlLCIDToCultureName
```

Looking at how ESI is used in this function (the entirety is not reproduced here, it begins at .text:7DEBB9E0) shows that its value is set to the second parameter. There is a test for 0 and a branch to a function that moves an error code into EAX. Thus, since ESI was not reused anywhere in this function, it can safely be determined that it will have the same type as an argument to RtlLCIDToCultureName as it does to RtlInitUnicodeString.

```
VOID WINAPI RtlInitUnicodeString(
__inout PUNICODE_STRING DestinationString,
__in_opt PCWSTR SourceString
);
```

ESI is passed as the first argument into RtlInitUnicodeString and as the second argument of RtlLCIDToCultureName concluding that ESI is a PUNICODE_STRING structure. Also, looking at .text:7DEBBA4B shows that the first argument passed into RtlLCIDToCultureName appears to be a 16-bit integer that got sign extended into EAX. Now the two arguments are known and the RtlLCIDToCultureName function itself can be analyzed. It appears that loc_7DEB4C88 is where the function jumps to if an error occured where it subsequently will return 0. If dword_7DF7208C is 0 then the function branches a bit further down, otherwise it jumps into a huge function chunk. The assumption is that short branches are good and branches to function chunks result from an unwanted condition. Following this assumption, a good branch would go to .text:7DEB4C77 where two constants, ESI, and EDI are pushed followed by RtlLcidToLocaleName being called. The return value is then tested and if all goes well, RtlLCIDToCultureName returns a 1 (otherwise 0). Given all of this information, the following can be inferred

```
NTSTATUS RtlLCIDToCultureName(ULONG unknown1, PUNICODE_STRING unknown2);
```

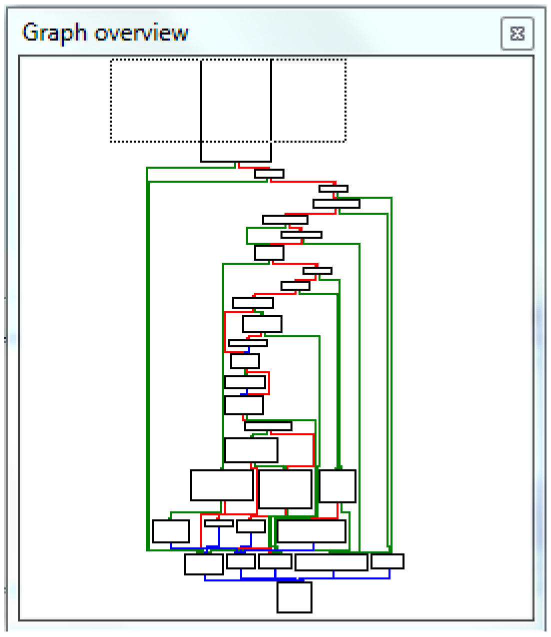
With some logic and testing, it can be guessed and tested that unknown1 is the numerical LCID which will be converted and stored in unknown2. However, analyzing RtlLcidToLocaleName in depth will confirm this.

RtlLcidToLocaleName (disassembly reproduced below)

```
.text:7DEB454F                 public RtlLcidToLocaleName
.text:7DEB454F RtlLcidToLocaleName proc near           ; CODE XREF:
RtlLCIDToCultureName+45↑p
.text:7DEB454F                                         ; sub_7DECC39F+58↑p
...
.text:7DEB454F
.text:7DEB454F var_BC          = word ptr -0BCh
.text:7DEB454F var_BA          = word ptr -0BAh
.text:7DEB454F var_B8          = dword ptr -0B8h
.text:7DEB454F var_B4          = dword ptr -0B4h
.text:7DEB454F var_B0          = byte ptr -0B0h
```

```
.text:7DEB454F var_4           = dword ptr -4
.text:7DEB454F arg_0           = dword ptr  8
.text:7DEB454F arg_4           = dword ptr  0Ch
.text:7DEB454F arg_8           = dword ptr  10h
.text:7DEB454F arg_C           = dword ptr  14h
.text:7DEB454F
.text:7DEB454F ; FUNCTION CHUNK AT .text:7DEFFB8C SIZE 000000CC BYTES
.text:7DEB454F
.text:7DEB454F                 mov     edi, edi
.text:7DEB4551                 push    ebp
.text:7DEB4552                 mov     ebp, esp
.text:7DEB4554                 sub     esp, 0BCh
.text:7DEB455A                 mov     eax, dword_7DF72088
.text:7DEB455F                 xor     eax, ebp
.text:7DEB4561                 mov     [ebp+var_4], eax
.text:7DEB4564                 push    ebx
.text:7DEB4565                 mov     ebx, [ebp+arg_4]
.text:7DEB4568                 push    esi
.text:7DEB4569                 push    edi
.text:7DEB456A                 mov     edi, [ebp+arg_0]
.text:7DEB456D                 mov     [ebp+var_B4], 55h
.text:7DEB4577                 test    edi, edi
.text:7DEB4579                 jz      loc_7DEB4649
.text:7DEB457F                 cmp     edi, 1000h
.text:7DEB4585                 jz      loc_7DEB4649
.text:7DEB458B                 test    ebx, ebx
.text:7DEB458D                 jz      loc_7DEFFB8C
.text:7DEB4593                 test    [ebp+arg_8], 0FFFFFFFDh
.text:7DEB459A                 jnz     loc_7DEFFB96
.text:7DEB45A0                 cmp     byte ptr [ebp+arg_C], 0
.text:7DEB45A4                 jnz     short loc_7DEB45B0
.text:7DEB45A6                 cmp     dword ptr [ebx+4], 0
.text:7DEB45AA                 jz      loc_7DEFFB8C
.text:7DEB45B0
.text:7DEB45B0 loc_7DEB45B0:                           ; CODE XREF:
RtlLcidToLocaleName+55↑j
.text:7DEB45B0                 cmp     edi, 1400h
.text:7DEB45B6                 jz      loc_7DEFFBA0
.text:7DEB45BC                 cmp     edi, 0C00h
.text:7DEB45C2                 jz      loc_7DEFFC16
.text:7DEB45C8                 cmp     edi, 400h
.text:7DEB45CE                 jz      loc_7DEFFC16
.text:7DEB45D4                 mov     esi, dword_7DF72028
.text:7DEB45DA                 test    esi, esi
.text:7DEB45DC                 jz      loc_7DEFFBD6
.text:7DEB45E2
.text:7DEB45E2 loc_7DEB45E2:                           ; CODE XREF:
RtlLcidToLocaleName+4B696↓j
.text:7DEB45E2                 cmp     edi, 800h
.text:7DEB45E8                 jz      loc_7DEFFBEA
.text:7DEB45EE
.text:7DEB45EE loc_7DEB45EE:                           ; CODE XREF:
RtlLcidToLocaleName+4B6A1↓j
.text:7DEB45EE                 push    edi
.text:7DEB45EF                 call    sub_7DEB4510
.text:7DEB45F4                 test    eax, eax
.text:7DEB45F6                 jl      short loc_7DEB4649
```

```
.text:7DEB45F8                 test    byte ptr [ebp+arg_8], 2
.text:7DEB45FC                 jz      loc_7DEFFBF5
.text:7DEB4602
.text:7DEB4602 loc_7DEB4602:                           ; CODE XREF:
RtlLcidToLocaleName+4B6BC↓j
.text:7DEB4602                 mov     ecx, [esi+14h]
.text:7DEB4605                 movzx   eax, word ptr [ecx+eax*8+6]
.text:7DEB460A                 mov     ecx, [esi+1Ch]
.text:7DEB460D                 lea     esi, [ecx+eax*2+2]
.text:7DEB4611                 lea     eax, [ebp+var_B4]
.text:7DEB4617                 push    eax
.text:7DEB4618                 push    54h
.text:7DEB461A                 push    esi
.text:7DEB461B                 call    sub_7DEB44CC
.text:7DEB4620                 test    eax, eax
.text:7DEB4622                 jl      loc_7DEFFBB9
.text:7DEB4628                 push    ebx
.text:7DEB4629                 push    [ebp+var_B4]
.text:7DEB462F                 push    esi
.text:7DEB4630
.text:7DEB4630 loc_7DEB4630:                           ; CODE XREF:
RtlLcidToLocaleName+4B682↓j
.text:7DEB4630                                         ;
RtlLcidToLocaleName+4B704↓j
.text:7DEB4630                 push    [ebp+arg_C]
.text:7DEB4633                 call    sub_7DEB4655
.text:7DEB4638
.text:7DEB4638 loc_7DEB4638:                           ; CODE XREF:
RtlLcidToLocaleName+FF↓j
.text:7DEB4638                                         ;
RtlLcidToLocaleName+4B642↓j ...
.text:7DEB4638                 mov     ecx, [ebp+var_4]
.text:7DEB463B                 pop     edi
.text:7DEB463C                 pop     esi
.text:7DEB463D                 xor     ecx, ebp
.text:7DEB463F                 pop     ebx
.text:7DEB4640                 call    sub_7DE9DF74
.text:7DEB4645                 leave
.text:7DEB4646                 retn    10h
.text:7DEB4649 ; -----------------------------------------------------------
---------------
.text:7DEB4649
.text:7DEB4649 loc_7DEB4649:                           ; CODE XREF:
RtlLcidToLocaleName+2A↑j
.text:7DEB4649                                         ;
RtlLcidToLocaleName+36↑j ...
.text:7DEB4649                 mov     eax, 0C00000EFh
.text:7DEB464E                 jmp     short loc_7DEB4638
.text:7DEB464E RtlLcidToLocaleName endp
```



Although slightly larger, the analysis is relatively straightforward. The flow graph shows a ton of error or value checking, but a stepwise path to the return point. It will be taken as an assumption for now that the branches result from error conditions from invalid or uninitialized values. The main interesting parts are

```
.text:7DEB4593                 test    [ebp+arg_8], 0FFFFFFFDh
.text:7DEB459A                 jnz     loc_7DEFFB96
.text:7DEB45A0                 cmp     byte ptr [ebp+arg_C], 0
.text:7DEB45A4                 jnz     short loc_7DEB45B0
.text:7DEB45A6                 cmp     dword ptr [ebx+4], 0
.text:7DEB45AA                 jz      loc_7DEFFB8C
```

The third argument, originally passed in as 2d, is compared against 0FFFFFFFDh (11111111111111111111111111111101b). The two values where these functions do not branch are 00b or 10b, or 0d and 2d. The many branches make the function annoying to analyze, but working through it and following a few branches (sub_7DEB4510 specifically) shows that dword_7DF72028 plays an important role. As a static analysis, not much can be determined. A lot of arithmetic follows loading dword_7DF72028 into a register, so it can be guessed that it retrieves a value based on the given key. Cross references to it also seem to suggest the same thing. Since the key to dword_7DF72028 depends on the EDI register, which is in turn, the first argument from RtlLCIDToCultureName, the guess that the first argument is the LCID has much more credibility. Further analysis also shows that the third parameter, passed in as a 2, is a flag to determine what the LCID is if additional flags are present. Using all of the knowledge gathered, it is now possible to call these functions from a normal program.

RtlCultureNameToLCID and RtlLocaleNameToLcid are more or less the same functions, except in reverse. Their analysis isn't going to be shown here. The only significant difference is that RtlLocaleNameToLcid takes a PWSTR as its first parameter instead of a PUNICODE_STRING. There is also an ambiguity in the third parameter of RtlLocaleNameToLcid. All cross references call it with a 3h as the third argument, but the function only compares the third argument against a value of 2h or 0FFFFFFFCh. The condition for when the third argument matches 2h branches to code that manipulates another static address (dword_7DF72028). Since this was done as a static analysis, not too much can be inferred from what this code means. It is possible to gain more knowledge by reversing the program below and tracing through what those static addresses hold and what happens when the "magic" value is set to 2h. All of the functions and how to invoke them are shown in the sample program below.

```
#include <Windows.h>
#include <assert.h>
#include <stdio.h>

#define DECLARE_UNICODE_STRING(_var, _string) \
const WCHAR _var ## _buffer[] = _string; \
UNICODE_STRING _var = { sizeof(_string) - sizeof(WCHAR), sizeof(_string),
(PWCH) _var ## _buffer }

typedef struct _UNICODE_STRING {
  USHORT  Length;
  USHORT  MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;

typedef NTSTATUS (__stdcall* pRtlLCIDToCultureName)(ULONG lcid,
PUNICODE_STRING name);
typedef NTSTATUS (__stdcall* pRtlLcidToLocaleName)(ULONG lcid,
PUNICODE_STRING name,
        DWORD reserved, BYTE encoded);
typedef NTSTATUS (__stdcall* pRtlCultureNameToLCID)(PUNICODE_STRING name,
PULONG lcid);
typedef NTSTATUS (__stdcall* pRtlLocaleNameToLcid)(PWSTR name, PULONG lcid,
BYTE magic);
typedef NTSTATUS (__stdcall* pRtlConvertLCIDToString)(ULONG lcid, ULONG
value, ULONG precision,
        PWSTR wstr_lcid, ULONG length);

int main(int argc, char* argv[])
{
        HMODULE hmodule = LoadLibrary(L"ntdll.dll");
        pRtlLCIDToCultureName LCIDToCulture =
                (pRtlLCIDToCultureName)GetProcAddress(hmodule,
"RtlLCIDToCultureName");
        pRtlLcidToLocaleName LcidToLocale =
                (pRtlLcidToLocaleName)GetProcAddress(hmodule,
"RtlLcidToLocaleName");
        pRtlCultureNameToLCID CultureToLCID =
                (pRtlCultureNameToLCID)GetProcAddress(hmodule,
"RtlCultureNameToLCID");
        pRtlLocaleNameToLcid LocaleToLcid =
                (pRtlLocaleNameToLcid)GetProcAddress(hmodule,
"RtlLocaleNameToLcid");
        pRtlConvertLCIDToString LCIDToString =
                (pRtlConvertLCIDToString)GetProcAddress(hmodule,
"RtlConvertLCIDToString");
        NTSTATUS ret = 0;
        DECLARE_UNICODE_STRING(name, L"dummystringinitializer");

        if(LCIDToCulture && LcidToLocale && CultureToLCID && LocaleToLcid &&
LCIDToString)
        {
                //Test RtlLCIDToCultureName and LCIDCultureNameToLCID
                ULONG lcid_enus = 0x0409;
                ULONG lcid_ruru = 0x0419;
                ret = LCIDToCulture(lcid_enus, &name);
                wprintf(L"[%i] LCID: %i  Culture/Locale name: %s\n", ret,
lcid_enus, name.Buffer);
                ret = LCIDToCulture(lcid_ruru, &name);
                wprintf(L"[%i] LCID: %i  Culture name: %s\n", ret, lcid_ruru,
name.Buffer);

                ULONG ret_enus = 0x0;
                ULONG ret_ruru = 0x0;
                DECLARE_UNICODE_STRING(wstr_enus, L"en-US");
                DECLARE_UNICODE_STRING(wstr_ruru, L"ru-RU");
                ret = CultureToLCID(&wstr_enus, &ret_enus);
                wprintf(L"[%i] LCID: %i  Culture/Locale name: %s\n", ret,
lcid_enus, wstr_enus.Buffer);
                ret = CultureToLCID(&wstr_ruru, &ret_ruru);
                wprintf(L"[%i] LCID: %i  Culture/Locale name: %s\n\n", ret,
lcid_ruru, wstr_ruru.Buffer);

                assert(ret_enus == lcid_enus);
                assert(ret_ruru == lcid_ruru);
                //End RtlLCIDToCultureName and LCIDCultureNameToLCID

                //Test RtlLcidToLocaleName and RtlLocaleNameToLcid
                ULONG lcid_dedephoneb = 0x10407; //Phone book sorting flag
added
                ULONG lcid_esve = 0x200A;
                ret = LcidToLocale(lcid_dedephoneb, &name, 0x00, 0x02);
                wprintf(L"[%i] LCID: %i  Culture/Locale name: %s\n", ret,
lcid_dedephoneb, name.Buffer);
                ret = LcidToLocale(lcid_esve, &name, 0x00, 0x00);
                wprintf(L"[%i] LCID: %i  Culture/Locale name: %s\n", ret,
lcid_esve, name.Buffer);

                ULONG ret_dedephoneb = 0x0;
                ULONG ret_esve = 0x0;
                DECLARE_UNICODE_STRING(wstr_uzlatnuz, L"de-DE_phoneb");
                DECLARE_UNICODE_STRING(wstr_esve, L"es-VE");
                ret = LocaleToLcid(wstr_uzlatnuz.Buffer, &ret_dedephoneb, 3);
                wprintf(L"[%i] LCID: %i  Culture/Locale name: %s\n", ret,
ret_dedephoneb, wstr_uzlatnuz.Buffer);
                ret = LocaleToLcid(wstr_esve.Buffer, &ret_esve, 3);
                wprintf(L"[%i] LCID: %i  Culture/Locale name: %s\n\n", ret,
ret_esve, wstr_esve.Buffer);

                assert(lcid_dedephoneb == ret_dedephoneb);
                assert(lcid_esve == ret_esve);
                //End RtlLcidToLocaleName and RtlLocaleNameToLcid

                //Test ConvertLCIDToString
                ULONG lcid_is = 0x040F;
                ret = LCIDToString(lcid_is, 0x10, 0x4, name.Buffer,
name.MaximumLength);
                wprintf(L"[%i] LCID: %i  Culture/Locale name: %s\n", ret,
lcid_is, name.Buffer);
                ret = LCIDToString(lcid_is, 0x10, 0x8, name.Buffer,
name.MaximumLength);
                wprintf(L"[%i] LCID: %i  Culture/Locale name: %s\n", ret,
lcid_is, name.Buffer);
                //End ConvertLCIDToString
        }
        else
                wprintf(L"One or more functions could not be exported.\n");

        return 0;
}
```

RtlConvertLCIDToString (usage shown above) is the last function to be discussed. It is a rather mysterious function which is only cross referenced once (shown below)

```
.text:7DF298E6                  push    20h
.text:7DF298E8                  push    esi
.text:7DF298E9                  push    4
.text:7DF298EB                  push    10h
.text:7DF298ED                  push    [ebp+ebx*4+var_24]
.text:7DF298F1                  call    RtlConvertLCIDToString
```

It seems to take three known integer values as its second, third, and fifth arguments. It calls the RtlIntegerToUnicodeString function passing in it's first two arguments. Looking this up on MSDN yields

```
NTSTATUS
RtlIntegerToUnicodeString(
IN ULONG Value,
IN ULONG Base OPTIONAL,
IN OUT PUNICODE_STRING String
);
```

So the first parameter into RtlConvertLCIDToString is a ULONG value to convert — the LCID; the second one is the base of the value. This seems to make sense when looking back at the function call. The value stored in [ebp+ebx*4+var_24] will be converted to a string representation of a base 16 number. The fourth parameter can easily be deduced as the output string. The code handles it as a PWSTR instead of a UNICODE_STRING. The fifth parameter is compared against 200h (512d) and an error code is returned if it is greater or equal to. If the value is fine, the code proceeds normally and a UNICODE_STRING struct is initialized in EAX for use in RtlIntegerToUnicodeString. Looking at how the UNICODE_STRING struct is initialized, it is possible to see that the Length, and MaximumLength members are initialized depending on the value of the fifth argument. Therefore, the fifth argument to RtlConvertLCIDToString has to be the length of the PWSTR passed in the fourth argument (since Length <= MaximumLength). This just leaves the third argument to be examined. The first (and only) time that the third argument is used is at

```
.text:7DF29439                   cmp     eax, [ebp+arg_8]
```

What happens afterwards is pretty interesting. Depending on certain conditions, the code will either continue regularly, or is will loop and hit

```
.text:7DF2942F                   push    30h
.text:7DF29431                   pop     edx
.text:7DF29432                   mov     [ecx], dx
```

This block is interesting because 30h is put into [ecx], which in that context holds the PWSTR to receive the LCID as a string. The 30h is interesting because converted it is the value of a “0″. What is happening is that the PWSTR is being padded with zeroes while this loop is ongoing and there is nothing left to copy. This means that the third argument is the precision of the number.

That concludes everything for this post. A downloadable PDF is available here.