Windows Programming

Contents

With the Windows version of flat assembler comes the package of standard includes designed to help in writing the programs for Windows environment.

The includes package contains the headers for 32-bit and 64-bit Windows programming in the root folder and the specialized includes in the subfolders. In general, the headers include the required specialized files for you, though sometimes you might prefer to include some of the macroinstruction packages yourself (since few of them are not included by some or even all of the headers).

There are six headers for 32-bit Windows that you can choose from, with names starting with `win32` followed by either a letter `a` for using the ASCII encoding, or a letter `w` for the WideChar encoding. The `win32a.inc` and `win32w.inc` are the basic headers, the `win32ax.inc` and `win32wx.inc` are the extended headers, they provide more advanced macroinstructions, those extensions will be discussed separately. Finally, the `win32axp.inc` and `win32wxp.inc` are the same extended headers with enabled feature of checking the count of parameters in procedure calls.

There are analogous six packages for the 64-bit Windows, with names starting with `win64`. They provide in general the same functionality as the ones for 32-bit Windows, with just a few differences explained later.

You can include the headers any way you prefer, by providing the full path or using the custom environment variable, but the simplest method is to define the `INCLUDE` environment variable properly pointing to the directory containing headers and then include them just like:

```
include 'win32a.inc'
```

It's important to note that all macroinstructions, as opposed to internal directives of flat assembler, are case sensitive and the lower case is used for the most of them. If you'd prefer to use the other case than default, you should do the appropriate adjustments with `fix` directive.

1. Basic headers

The basic headers `win32a.inc`, `win32w.inc`, `win64a.inc` and `win64w.inc` include declarations of Windows equates and structures and provide the standard set of macroinstructions.

1.1 Structures

All headers enable the `struct` macroinstruction, which allows to define structures in a way more similar to other assemblers than the `struc` directive. The definition of structure should be started with `struct` macroinstruction followed by the name, and ended with `ends` macroinstruction. In lines between only data definition directives are allowed, with labels being the pure names for the fields of structure:

```
struct POINT
  x dd ?
  y dd ?
ends
```

With such definition this line:

```
point1 POINT
```

will declare the `point1` structure with the `point1.x` and `point1.y` fields, giving them the default values - the same ones as provided in the definition of structure (in this case the defaults are both uninitialized values). But declaration of structure also accepts the parameters, in the same count as the number of fields in the structure, and those parameters, when specified, override the default values for fields. For example:

```
point2 POINT 10,20
```

initializes the `point2.x` field with value 10, and the `point2.y` with value 20.

The `struct` macro not only enables to declare the structures of given type, but also defines labels for offsets of fields inside the structure and constants for sized of every field and the whole structure. For example the above definition of `POINT` structure defines the `POINT.x` and `POINT.y` labels to be the offsets of fields inside the structure, and `sizeof.POINT.x`, `sizeof.POINT.y` and `sizeof.POINT` as sizes of the corresponding fields and of the whole structure. The offset labels may be used for accessing the structures addressed indirectly, like:

```
mov eax,[ebx+POINT.x]
```

when the `ebx` register contains the pointer to `POINT` structure. Note that field size checking will be performed with such accessing as well.

The structures themselves are also allowed inside the structure definitions, so the structures may have some other structures as a fields:

```
struct LINE
  start POINT
  end   POINT
ends
```

When no default values for substructure fields are specified, as in this example, the defaults from the definition of the type of substructure apply.

Since value for each field is a single parameter in the declaration of the structure, to initialize the substructures with custom values the parameters for each substructure must be grouped into a single parameter for the structure:

```
line1 LINE <0,0>,<100,100>
```

The above declaration initializes each of the `line1.start.x` and `line1.start.y` fields with 0, and each of the `line1.end.x` and `line1.end.y` with 100.

When the size of data defined by some value passed to the declaration structure is smaller than the size of corresponding field, it is padded to that size with undefined bytes (and when it is larger, the error happens). For example:

```
struct FOO
  data db 256 dup (?)
ends

some FOO <"ABC",0>
```

fills the first four bytes of `some.data` with defined values and reserves the rest.

Inside the structures also unions and unnamed substructures can be defined. The definition of union should start with `union` and end with `ends`, like in this example:

```
struct BAR
  field_1 dd ?
  union
    field_2  dd ?
    field_2b db ?
  ends
ends
```

Each of the fields defined inside union has the same offset and they share the same memory. Only the first field of union is initialized with given value, the values for the rest of fields are ignored (however if one of the other fields requires more memory than the first one, the union is padded to the required size with undefined bytes). The whole union is initialized by the single parameter given in structure declaration, and this parameter gives value to the first field of union.

The unnamed substructure is defined in a similar way to the union, only starts with the `struct` line instead of `union`, like:

```
struct WBB
  word dw ?
  struct
    byte1 db ?
    byte2 db ?
  ends
ends
```

Such substructure only takes one parameter in the declaration of whole structure to define its values, and this parameter can itself be the group of parameters defining each field of the substructure. So the above type of structure may get declared like:

```
my WBB 1,<2,3>
```

The fields inside unions and unnamed substructures are accessed just as if they were directly the fields of the parent structure. For example, with above declaration `my.byte1` and `my.byte2` are correct labels for the substructure fields.

The substructures and unions can be nested with no limits for the nesting depth:

```
struct LINE
  union
    start POINT
    struct
      x1  dd ?
      y1  dd ?
    ends
  ends
  union
    end   POINT
    struct
      x2  dd ?
      y2  dd ?
    ends
  ends
ends
```

The definition of structure may also be based on some of the already defined structure types and it inherits all the fields from that structure, for example:

```
struct CPOINT POINT
  color dd ?
ends
```

defines the same structure as:

```
struct CPOINT
  x     dd ?
  y     dd ?
  color dd ?
ends
```

All headers define the `CHAR` data type, which can be used to define character strings in the data structures.

1.2 Imports

The import macroinstructions help to build the import data for PE file (usually put in the separate section). There are two macroinstructions for this purpose. The first one is called `library`, must be placed directly in the beginning of the import data and it defines from what libraries the functions will be imported. It should be followed by any amount of the pairs of parameters, each pair being the label for the table of imports from the given library, and the quoted string defining the name of the library. For example:

```
library kernel32,'KERNEL32.DLL',\
        user32,'USER32.DLL'
```

declares to import from the two libraries. For each of libraries, the table of imports must be then declared somewhere inside the import data. This is done with `import` macroinstruction, which needs first parameter to define the label for the table (the same as declared earlier to the `library` macro), and

then the pairs of parameters each containing the label for imported pointer and the quoted string defining the name of function exactly as exported by library. For example, the above `library` declaration may be completed with following `import` declarations:

```
import kernel32,\
       ExitProcess,'ExitProcess'

import user32,\
       MessageBeep,'MessageBeep',\
       MessageBox,'MessageBoxA'
```

The labels defined by first parameters in each pair passed to the `import` macro address the double word pointers, which after loading the PE are filled with the addresses to exported procedures.

Instead of quoted string for the name of procedure to import, the number may be given to define import by ordinal, like:

```
import custom,\
       ByName,'FunctionName',\
       ByOrdinal,17
```

The import macros optimize the import data, so only imports for functions that are used somewhere in program are placed in the import tables, and if some import table would be empty this way, the whole library is not referenced at all. For this reason, it's handy to have the complete import table for each library - the package contains such tables for some of the standard libraries, they are stored in the `APIA` and `APIW` subdirectories and import the ASCII and WideChar variants of the API functions. Each file contains one import table, with lowercase label the same as the name of the file. So, the complete tables for importing from the `KERNEL32.DLL` and `USER32.DLL` libraries can be defined this way (assuming your `INCLUDE` environment variable points to the directory containing the includes package):

```
library kernel32,'KERNEL32.DLL',\
        user32,'USER32.DLL'

include 'apia\kernel32.inc'
include 'apiw\user32.inc'
```

1.3 Procedures (32-bit)

Headers for 32-bit Windows provide four macroinstructions for calling procedures with parameters passed on stack. The `stdcall` calls directly the procedure specified by the first argument using the STDCALL calling convention. The rest of arguments passed to macro define the parameters to procedure and are stored on the stack in reverse order. The `invoke` macro does the same, however it calls the procedure indirectly, through the pointer labelled by the first argument. Thus, `invoke` can be used to call the procedures through pointers defined in the import tables. This line:

```
invoke MessageBox,0,szText,szCaption,MB_OK
```

is equivalent to:

```
stdcall [MessageBox],0,szText,szCaption,MB_OK
```

and they both generate this code:

```
push MB_OK
push szCaption
push szText
push 0
```

```
call [MessageBox]
```

The `ccall` and `cinvoke` are analogous to the `stdcall` and `invoke`, but they should be used to call the procedures that use the C calling convention, where the stack frame has to be restored by the caller.

To define the procedure that uses the stack for parameters and local variables, you should use the `proc` macroinstruction. In its simplest form it has to be followed by the name for the procedure and then names for the all the parameters it takes, like:

```
proc WindowProc,hwnd,wmsg,wparam,lparam
```

The comma between the name of procedure and the first parameter is optional. The procedure instructions should follow in the next lines, ended with the `endp` macroinstruction. The stack frame is set up automatically on the entry to procedure, the EBP register is used as a base to access the parameters, so you should avoid using this register for other purposes. The names specified for the parameters are used to define EBP-based labels, which you can use to access the parameters as regular variables. For example the `mov eax,[hwnd]` instruction inside the procedure defined as in above sample, is equivalent to `mov eax,[ebp+8]`. The scope of those labels is limited to the procedure, so you may use the same names for other purposes outside the given procedure.

Since any parameters are pushed on the stack as double words when calling such procedures, the labels for parameters are defined to mark the double word data by default, however you can you specify the sizes for the parameters if you want, by following the name of parameter with colon and the size operator. The previous sample can be rewritten this way, which is again equivalent:

```
proc WindowProc,hwnd:DWORD,wmsg:DWORD,wparam:DWORD,lparam:DWORD
```

If you specify a size smaller than double word, the given label applies to the smaller portion of the whole double word stored on stack. If you specify a larger size, like far pointer of quad word, the two double word parameters are defined to hold this value, but are labelled as one variable.

The name of procedure can be also followed by either the `stdcall` or `c` keyword to define the calling convention it uses. When no such type is specified, the default is used, which is equivalent to STDCALL. Then also the `uses` keyword may follow, and after it the list of registers (separated only with spaces) that will be automatically stored on entry to procedure and restored on exit. In this case the comma after the list of registers and before the first parameter is required. So, the fully featured procedure statement might look like this:

```
proc WindowProc stdcall uses ebx esi edi,\
     hwnd:DWORD,wmsg:DWORD,wparam:DWORD,lparam:DWORD
```

To declare the local variable, you can use the `local` macroinstruction, followed by one or more declarations separated with commas, each one consisting of the name for variable followed by colon and the type of variable - either one of the standard types (must be upper case) or the name of data structure. For example:

```
local hDC:DWORD,rc:RECT
```

To declare a local array, you can follow the name of variable by the size of array enclosed in square brackets, like:

```
local str[256]:BYTE
```

The other way to define the local variables is to declare them inside the block started with "locals" macroinstruction and ended with "endl", in this case they can be defined just like regular data. This declaration is the equivalent of the earlier sample:

```
locals
  hDC dd ?
  rc RECT
endl
```

The local variables can be declared anywhere inside the procedure, with the only limitation that they have to be declared before they are used. The scope of labels for the variables defined as local is limited to inside the procedure, you can use the same names for other purposes outside the procedure. If you give some initialized values to the variables declared as local, the macroinstruction generates the instructions that will initialize these variables with the given values and puts these instructions at the same position in procedure, where the declaration is placed.

The `ret` placed anywhere inside the procedure, generates the complete code needed to correctly exit the procedure, restoring the stack frame and the registers used by procedure. If you need to generate the raw return instruction, use the `retn` mnemonic, or follow the `ret` with the number parameter, what also causes it to be interpreted as single instruction.

To recapitulate, the complete definition of procedure may look like this:

```
proc WindowProc uses ebx esi edi,hwnd,wmsg,wparam,lparam
  local hDC:DWORD,rc:RECT
  ; the instructions
  ret
endp
```

1.4 Procedures (64-bit)

In 64-bit Windows there is only one calling convention, and thus only two macroinstructions for calling procedures are provided. The `fastcall` calls directly the procedure specified by the first argument using the standard convention of 64-bit Windows system. The `invoke` macro does the same, but indirectly, through the pointer labelled by the first argument. Parameters are provided by the arguments that follow, and they can be of any size up to 64 bits. The macroinstructions use RAX register as a temporary storage when some parameter value cannot be copied directly into the stack using the `mov` instruction. If the parameter is preceded with `addr` word, it is treated as an address and is calculated with the `lea` instruction - so if the address is absolute, it will get calculated as RIP-relative, thus preventing generating a relocation in case of file with fixups.

Because in 64-bit Windows the floating-point parameters are passed in a different way, they have to be marked by preceding each one of them with `float` word. They can be either double word or quad word in size. Here is an example of calling some OpenGL procedures with either double-precision or single-precision parameters:

```
invoke glVertex3d,float 0.6,float -0.6,float 0.0
invoke glVertex2f,float dword 0.1,float dword 0.2
```

The stack space for parameters are allocated before each call and freed immediately after it. However, it is possible to allocate this space just once for all the calls inside some given block of code, for this purpose there are `frame` and `endf` macros provided. They should be used to enclose a block, inside which the RSP register is not altered between the procedure calls and they prevent each call from

allocating stack space for parameters, as it is reserved just once by the `frame` macro and then freed at the end by the `endf` macro.

```
frame ; allocate stack space just once
  invoke TranslateMessage,msg
  invoke DispatchMessage,msg
endf
```

The `proc` macro for 64-bit Windows has the same syntax and features as 32-bit one (though `stdcall` and `c` options are of no use in its case). It should be noted however that in the calling convention used in 64-bit Windows first four parameters are passed in registers (RCX, RDX, R8 and R9), and therefore, even though there is a space reserved for them at the stack and it is labelled with name provided in the procedure definition, those four parameters will not initially reside there. They should be accessed by directly reading the registers. But if those registers are needed to be used for some other purpose, it is recommended to store the value of such parameter into the memory cell reserved for it. The beginning of such procedure may look like:

```
proc WindowProc hwnd,wmsg,wparam,lparam
  mov [hwnd],rcx
  mov [wmsg],edx
  mov [wparam],r8
  mov [lparam],r9
  ; now registers can be used for other purpose
  ; and parameters can still be accessed later
```

1.5 Customizing Procedures

It is possible to create a custom code for procedure framework when using `proc` macroinstruction. There are three symbolic variables, `prologue@proc`, `epilogue@proc` and `close@proc`, which define the names of macroinstructions that `proc` calls upon entry to the procedure, return from procedure (created with `ret` macro) and at the end of procedure (made with `endp` macro). Those variables can be re-defined to point to some other macroinstructions, so that all the code generated with `proc` macro can be customized.

Each of those three macroinstructions takes five parameters. The first one provides a label of procedure entry point, which is the name of procedure as well. The second one is a bitfield containing some flags, notably the bit 4 is set when the caller is supposed to restore the stack, and cleared otherwise. The third one is a value that specifies the number of bytes that parameters to the procedure take on the stack. The fourth one is a value that specified the number of bytes that should be reserved for the local variables. Finally, the fifth and last parameter is the list of comma-separated registers, which procedure declared to be used and which should therefore be saved by prologue and restored by epilogue.

The prologue macro apart from generating code that would set up the stack frame and the pointer to local variables has to define two symbolic variables, `parmbase@proc` and `localbase@proc`. The first one should provide the base address for where the parameters reside, and the second one should provide the address for where the local variables reside - usually relative to EBP/RBP register, but it is possible to use other bases if it can be ensured that those pointers will be valid at any point inside the procedure where parameters or local variables are accessed. It is also up to the prologue macro to make any alignments necessary for valid procedure implementation; the size of local variables provided as fourth parameter may itself be not aligned at all.

The default behavior of `proc` is defined by `prologuedef` and `epiloguedef` macros (in default case there is no need for closing macro, so the `close@proc` has an empty value). If it is needed to return to the defaults after some customizations were used, it should be done with the following three lines:

```
prologue@proc equ prologuedef
epilogue@proc equ epiloguedef
close@proc equ
```

As an example of modified prologue, below is the macroinstruction that implements stack-probing prologue for 32-bit Windows. Such method of allocation should be used every time the area of local variables may get larger than 4096 bytes.

```
macro sp_prologue procname,flag,parmbytes,localbytes,reglist
{ local loc
  loc = (localbytes+3) and (not 3)
  parmbase@proc equ ebp+8
  localbase@proc equ ebp-loc
  if parmbytes | localbytes
   push ebp
   mov ebp,esp
   if localbytes
    repeat localbytes shr 12
     mov byte [esp-%*4096],0
    end repeat
    sub esp,loc
   end if
  end if
  irps reg, reglist \{ push reg \} }

prologue@proc equ sp_prologue
```

It can be easily modified to use any other stack probing method of the programmer's preference.

The 64-bit headers provide an additional set of prologue/epilogue macros, which allow to define procedure that uses RSP to access parameters and local variables (so RBP register is free to use for any other by procedure) and also allocates the common space for all the procedure calls made inside, so that `fastcall` or `invoke` macros called do not need to allocate any stack space themselves. It is an effect similar to the one obtained by putting the code inside the procedure into `frame` block, but in this case the allocation of stack space for procedure calls is merged with the allocation of space for local variables. The code inside such procedure must not alter RSP register in any way. To switch to this behavior of 64-bit `proc`, use the following instructions:

```
prologue@proc equ static_rsp_prologue
epilogue@proc equ static_rsp_epilogue
close@proc equ static_rsp_close
```

1.6 Exports

The `export` macroinstruction constructs the export data for the PE file (it should be either placed in the section marked as export, or within the `data export` block. The first argument should be quoted string defining the name of library file, and the rest should be any number of pairs of arguments, first in each pair being the name of procedure defined somewhere inside the source, and the second being the quoted string containing the name under which this procedure should be exported by the library. This sample:

```
export 'MYLIB.DLL',\
```

```
        MyStart,'Start',\
        MyStop,'Stop'
```

defines the table exporting two functions, which are defined under the names `MyStart` and `MyStop` in the sources, but will be exported by library under the shorter names. The macroinstruction can take care of the alphabetical sorting of the table, which is required by the PE format.

1.7 Component Object Model

The `interface` macro allows to declare the interface of the COM object type, the first parameter is the name of interface, and then the consecutive names of the methods should follow, like in this example:

```
interface ITaskBarList,\
          QueryInterface,\
          AddRef,\
          Release,\
          HrInit,\
          AddTab,\
          DeleteTab,\
          ActivateTab,\
          SetActiveAlt
```

The `comcall` macro may be then used to call the method of the given object. The first parameter to this macro should be the handle to object, the second one should be name of COM interface implemented by this object, and then the name of method and parameters to this method. For example:

```
comcall ebx,ITaskBarList,ActivateTab,[hwnd]
```

uses the contents of EBX register as a handle to COM object with `ITaskBarList` interface, and calls the `ActivateTab` method of this object with the `[hwnd]` parameter.

You can also use the name of COM interface in the same way as the name of data structure, to define the variable that will hold the handle to object of given type:

```
ShellTaskBar ITaskBarList
```

The above line defines the variable, in which the handle to the COM object can be stored. After storing the handle to an object, its methods can be called with the `cominvk`. This macro needs only the name of the variable with assigned interface and the name of method as first two parameters, and then parameters for the method. So, the `ActivateTab` method of object whose handle is stored in the `ShellTaskBar` variable as defined above can be called this way:

```
cominvk ShellTaskBar,ActivateTab,[hwnd]
```

which does the same as:

```
comcall [ShellTaskBar],ITaskBarList,ActivateTab,[hwnd]
```

1.8 Resources

There are two ways to create resources, one is to include the external resource file created with some other program, and the other one is to create resource section manually. The latter method, though doesn't need any additional program to be involved, is more laborious, but the standard headers provide the assistance - the set of elementary macroinstructions that serve as bricks to compose the resource section.

The `directory` macroinstruction must be placed directly in the beginning of manually built resource data and it defines what types of resources it contains. It should be followed by the pairs of values, the first one in each pair being the identifier of the type of resource, and the second one the label of subdirectory of the resources of given type. It may look like this:

```
directory RT_MENU,menus,\
          RT_ICON,icons,\
          RT_GROUP_ICON,group_icons
```

The subdirectories can be placed anywhere in the resource area after the main directory, and they have to be defined with the `resource` macroinstruction, which requires first parameter to be the label of the subdirectory (corresponding to the entry in main directory) followed by the trios of parameters - in each such entry the first parameter defines the identifier of resource (this value is freely chosen by the programmer and is then used to access the given resource from the program), the second specifies the language and the third one is the label of resource. Standard equates should be used to create language identifiers. For example, the subdirectory of menus may be defined this way:

```
resource menus,\
         1,LANG_ENGLISH+SUBLANG_DEFAULT,main_menu,\
         2,LANG_ENGLISH+SUBLANG_DEFAULT,other_menu
```

If the resource is of kind for which the language doesn't matter, the language identifier `LANG_NEUTRAL` should be used. To define the resources of various types there are specialized macroinstructions, which should be placed inside the resource area.

The bitmaps are the resources with `RT_BITMAP` type identifier. To define the bitmap resource, use the `bitmap` macroinstruction with the first parameter being the label of resource (corresponding to the entry in the subdirectory of bitmaps) and the second being the quoted string containing the path to the bitmap file, like:

```
bitmap program_logo,'logo.bmp'
```

There are two resource types related to icons, the `RT_GROUP_ICON` is the type for the resource, which has to be linked to one or more resources of `RT_ICON` type, each one containing single image. This allows to declare images of different sizes and color depths under the common resource identifier. This identifier, given to the resource of `RT_GROUP_ICON` type may be then passed to the `LoadIcon` function, and it will choose the image of suitable dimensions from the group. To define the icon, use the `icon` macroinstruction, with first parameter being the label of `RT_GROUP_ICON` resource, followed by the pairs of parameters declaring the images. First parameter in each pair should be the label of `RT_ICON` resource, and the second one the quoted string containing the path to the icon file. In the simplest variant, when group of icon contains just one image, it will look like:

```
icon main_icon,icon_data,'main.ico'
```

where the `main_icon` is the label for entry in resource subdirectory for `RT_GROUP_ICON` type, and the `icon_data` is the label for entry of `RT_ICON` type.

The cursors are defined in a way similar to icons, with the `RT_GROUP_CURSOR` and `RT_CURSOR` types and the `cursor` macro, which takes parameters analogous to those taken by `icon` macro. So the definition of cursor may look like this:

```
cursor my_cursor,cursor_data,'my.cur'
```

The menus have the `RT_MENU` type of resource and are defined with the `menu` macroinstruction followed by few others defining the items inside the menu. The `menu` itself takes only one parameter - the label of resource. The `menuitem` defines the item in the menu, it takes up to five parameters, but only two are required - the first one is the quoted string containing the text for the item, and the second one is the identifier value (which is the value that will be returned when user selects the given item from the menu). The `menuseparator` defines a separator in the menu and doesn't require any parameters.

The optional third parameter of `menuitem` specifies the menu resource flags. There are two such flags available - `MFR_END` is the flag for the last item in the given menu, and the `MFR_POPUP` marks that the given item is the submenu, and the following items will be items composing that submenu until the item with `MFR_END` flag is found. The `MFR_END` flag can be also given as the parameter to the `menuseparator` and is the only parameter this macroinstruction can take. For the menu definition to be complete, every submenu must be closed by the item with `MFR_END` flag, and the whole menu must also be closed this way. Here is an example of complete definition of the menu:

```
menu main_menu
     menuitem '&File',100,MFR_POPUP
              menuitem '&New',101
              menuseparator
              menuitem 'E&xit',109,MFR_END
     menuitem '&Help',900,MFR_POPUP + MFR_END
              menuitem '&About...',901,MFR_END
```

The optional fourth parameter of `menuitem` specifies the state flags for the given item, these flags are the same as the ones used by API functions, like `MFS_CHECKED` or `MFS_DISABLED`. Similarly, the fifth parameter can specify the type flags. For example, this will define item checked with a radio-button mark:

```
menuitem 'Selection',102, ,MFS_CHECKED,MFT_RADIOCHECK
```

The dialog boxes have the `RT_DIALOG` type of resource and are defined with the `dialog` macroinstruction followed by any number of items defined with `dialogitem` ended with the `enddialog`.

The `dialog` can take up to eleven parameters, first seven being required. First parameter, as usual, specifies the label of resource, second is the quoted string containing the title of the dialog box, the next four parameters specify the horizontal and vertical coordinates, the width and the height of the dialog box window respectively. The seventh parameter specifies the style flags for the dialog box window, the optional eighth one specifies the extended style flags. The ninth parameter can specify the menu for window - it should be the identifier of menu resource, the same as one specified in the subdirectory of resources with `RT_MENU` type. Finally, the tenth and eleventh parameter can be used to define the font for the dialog box - first of them should be the quoted string containing the name of font, and the latter one the number defining the size of font. When these optional parameters are not specified, the default MS Sans Serif of size 8 is used.

This example shows the `dialog` macroinstruction with all the parameters except for the menu (which is left with blank value), the optional ones are in the second line:

```
dialog about,'About',50,50,200,100,WS_CAPTION+WS_SYSMENU,\
             WS_EX_TOPMOST, ,'Times New Roman',10
```

The `dialogitem` has eight required parameters and one optional. First parameter should be the quoted string containing the class name for the item. Second parameter can be either the quoted string

containing text for the item, or resource identifier in case when the contents of item has to be defined by some additional resource (like the item of `STATIC` class with the `SS_BITMAP` style). The third parameter is the identifier for the item, used to identify the item by the API functions. Next four parameters specify the horizontal, vertical coordinates, the width and height of the item respectively. The eighth parameter specifies the style for the item, and the optional ninth specifies the extended style flags. An example dialog item definition:

```
dialogitem 'BUTTON','OK',IDOK,10,10,45,15,WS_VISIBLE+WS_TABSTOP
```

And an example of static item containing bitmap, assuming that there exists a bitmap resource of identifier 7:

```
dialogitem 'STATIC',7,0,10,50,50,20,WS_VISIBLE+SS_BITMAP
```

The definition of dialog resource can contain any number of items or none at all, and it should be always ended with `enddialog` macroinstruction.

The resources of type `RT_ACCELERATOR` is created with `accelerator` macroinstruction. After first parameter traditionally being the label of resource, there should follow the trios of parameters - the accelerator flags followed by the virtual key code or ASCII character and the identifier value (which is like the identifier of the menu item). A simple accelerator definition may look like this:

```
accelerator main_keys,\
            FVIRTKEY+FNOINVERT,VK_F1,901,\
            FVIRTKEY+FNOINVERT,VK_F10,109
```

The version information is the resource of type `RT_VERSION` and is created with the `versioninfo` macroinstruction. After the label of the resource, the second parameter specifies the operating system of PE file (usually `VOS__WINDOWS32`), third parameter the type of file (the most common are `VFT_APP` for program and `VFT_DLL` for library), fourth the subtype (usually `VFT2_UNKNOWN`), fifth the language identifier, sixth the code page and then the quoted string parameters, being the pairs of property name and corresponding value. The simplest version information can be defined like:

```
versioninfo vinfo,VOS__WINDOWS32,VFT_APP,VFT2_UNKNOWN,\
                  LANG_ENGLISH+SUBLANG_DEFAULT,0,\
            'FileDescription','Description of program',\
            'LegalCopyright','Copyright et cetera',\
            'FileVersion','1.0',\
            'ProductVersion','1.0'
```

Other kinds of resources may be defined with `resdata` macroinstruction, which takes only one parameter - the label of resource, and can be followed by any instructions defining the data, ended with `endres` macroinstruction, like:

```
resdata manifest
  file 'manifest.xml'
endres
```

1.9 Text Encoding

The resource macroinstructions use the `du` directive to define any Unicode strings inside resources - since this directive simply zero extends the characters to the 16-bit values, for the strings containing some non-ASCII characters, the `du` may need to be redefined. For some of the encodings the macroinstructions redefining the `du` to generate the Unicode texts properly are provided in the `ENCODING`

subdirectory. For example, if the source text is encoded with Windows 1250 code page, such line should be put somewhere in the beginning of the source:

```
include 'encoding\win1250.inc'
```

2. Extended Headers

The files `win32ax.inc`, `win32wx.inc`, `win64ax.inc` and `win64wx.inc` provide all the functionality of base headers and include a few more features involving more complex macroinstructions. Also if no PE format is declared before including the extended headers, the headers declare it automatically. The files `win32axp.inc`, `win32wxp.inc`, `win64axp.inc` and `win64wxp.inc` are the variants of extended headers which additionally perform checking the count of parameters to procedure calls.

2.1 Procedure Parameters

With the extended headers the macroinstructions for calling procedures allow more types of parameters than just the double word values as with basic headers. First of all, when the quoted string is passes as a parameter to procedure, it is used to define string data placed among the code, and passes to procedure the double word pointer to this string. This allows to easily define the strings that don't have to be re-used, just in the line calling the procedure that requires pointers to those strings, like:

```
invoke MessageBox,HWND_DESKTOP,"Message","Caption",MB_OK
```

If the parameter is the group containing some values separated with commas, it is treated in the same way as simple quoted string parameter.

If the parameter is preceded by the `addr` word, it means that this value is an address and this address should be passed to procedure, even if it cannot be done directly - like in the case of local variables, which have addresses relative to EBP/RBP register. In 32-bit case the EDX register is used temporarily to calculate the value of address and pass it to the procedure. For example:

```
invoke RegisterClass,addr wc
```

in case when the `wc` is the local variable with address EBP-100h, will generate this sequence of instructions:

```
lea edx,[ebp-100h]
push edx
call [RegisterClass]
```

However, when the given address is not relative to any register, it is stored directly.

In 64-bit case the `addr` prefix is allowed even when only standard headers are used, as it can be useful even in case of the regular addresses, because it enforces RIP-relative address calculation.

With 32-bit headers, if the parameter is preceded by the word `double`, it is treated as 64-bit value and passed to the procedure as two 32-bit parameters. For example:

```
invoke glColor3d,double 1.0,double 0.1,double 0.1
```

will pass the three 64-bit parameters as six double words to procedure. If the parameter following `double` is the memory operand, it should not have size operator, the `double` already works as the size override.

Finally, the calls to procedures can be nested, that is call to one procedure may be used as the parameter to another. In such case the value returned in EAX/RAX by the nested procedure is passed as the parameter to the procedure which it is nested in. A sample of such nesting:

```
invoke MessageBox,<invoke GetTopWindow,[hwnd]>,\
                  "Message","Caption",MB_OK
```

There are no limits for the depth of nesting the procedure calls.

2.2 Structuring the Source

The extended headers enable some macroinstructions that help with easy structuring the program. The `.data` and `.code` are just the shortcuts to the declarations of sections for data and for the code. The `.end` macroinstruction should be put at the end of program, with one parameter specifying the entry point of program, and it also automatically generates the import section using all the standard import tables. In 64-bit Windows the `.end` automatically aligns the stack on 16 bytes boundary.

The `.if` macroinstruction generates a piece of code that checks for some simple condition at the execution time, and depending on the result continues execution of following block or skips it. The block should be ended with `.endif`, but earlier also `.elseif` macroinstruction might be used once or more and begin the code that will be executed under some additional condition, when the previous conditions were not met, and the `.else` as the last before `.endif` to begin the block that will be executed when all the conditions were false.

The condition can be specified by using comparison operator - one of the `=`, `<`, `>`, `<=`, `>=`, and `<>` - between the two values, first of which must be either register or memory operand. The values are compared as unsigned ones, unless the comparison expression is preceded by the word `signed`. If you provide only single value as a condition, it will be tested to be zero, and the condition will be true only if it's not. For example:

```
.if eax
  ret
.endif
```

generates the instructions, which skip over the `ret` when the EAX is zero.

There are also some special symbols recognized as conditions: the `ZERO?` is true when the ZF flag is set, in the same way the `CARRY?`, `SIGN?`, `OVERFLOW?` and `PARITY?` correspond to the state of CF, SF, OF and PF flags.

The simple conditions like above can be composed into complex conditional expressions using the `&`, `|` operators for conjunction and alternative, the `~` operator for negation, and parenthesis. For example:

```
.if eax<=100 & ( ecx | edx )
  inc ebx
.endif
```

will generate the compare and jump instructions that will cause the given block to get executed only when EAX is below or equal 100 and at the same time at least one of the ECX and EDX is not zero.

The `.while` macroinstruction generates the instructions that will repeat executing the given block (ended with `.endw` macroinstruction) as long as the condition is true. The condition should follow the `.while` and can be specified in the same way as for the `.if`.

The pair of `.repeat` and `.until` macroinstructions define the block that will be repeatedly executed until the given condition will be met - this time the condition should follow the `.until` macroinstruction, placed at the end of block, like:

```
.repeat
  add ecx,2
.until ecx>100
```

Copyright © 1999-2018, Tomasz Grysztar.
Powered by rwasa.

Table of Contents