Другие темы раздела

FASM Уроки Iczelion'a на FASM https://www.cyberforum.ru/ fasm/thread1240590.html
Уроки Iczelion'a на FASM Урок первый. MessageBox на FASM format PE GUI include 'win32ax.inc' ; import data in the same section invoke MessageBox,NULL,msgBoxText,msgBoxCaption,MB_OK ...

Побайтовый вывод файла FASM
Пытаюсь ввести в консоль файл в шестнадцатеричном виде, но происходит ошибка при выполнении. format PE console 4.0 include 'win32a.inc' xor ebx, ebx ; invoke CreateFile,\ ...

FASM Вывод адреса на консоль
Пытаюсь на консоль вывести адрес fin: invoke printf, не робит - как правильно надо? format PE console 4.0 entry start include 'win32a.inc' section '.data' data readable fin ...

FASM ГСЧ на макросах Всем привет. Понадобилось заюзать ГСЧ посредством макросов, чтобы каждый раз на стадии компиляции, использовалось уникальное значение. Учитывая семантику препроцессора (там чёрт ногу сломит),... https://www.cyberforum.ru/ fasm/thread1213146.html

FASM Создание окна на fasm https://www.cyberforum.ru/ fasm/thread1209394.html
Всем привет. Только что начал изучать ассемблер fasm. Возник первый вопрос: как создать окно? Прошу не просто дать мне код, а ещё объяснить что значит. Заранее благодарен

FASM Вызываем функции из clib (библиотека Си) в DOS
Вобщем, сбылась мечта идиота. Теперь, нежели писать свой ввод/вывод(особенно всегда напрягал ввод/вывод вещественных чисел на экран), можно воспользоваться стандартными ф-циями из библиотеки языка...

Организовать вычисления по формуле FASM
привет, всем активным участникам этого чудесного форума!!! помогите, пожалуйста, написать программу на Fasm Assembler. задание: Создать программу на языке Ассемблер, что позволяет организовать...

FASM Как сделать выход по ESC org 100h old dw 0 jmp start number dw 0 c dw 0 start: xor ax,ax mov es,ax cli https://www.cyberforum.ru/ fasm/thread1161834.html

FASM Получение CLSID image/png https://www.cyberforum.ru/ fasm/ thread1160365.html
Всем ку! int GetEncoderClsid(const WCHAR* format, CLSID* pClsid) { UINT num = 0; // number of image encoders UINT size = 0; // size of the image encoder array in bytes...

FASM Вывод трех строк в один MessageBox Здравствуйте, помогите, пожалуйста, с такой проблемой: не могу вывести 3 строки (Год+Месяц+День) в один MessageBox Вот такой код: format PE GUI 4.0 entry start include 'win32ax.inc' include... https://www.cyberforum.ru/ fasm/thread1142589.html

Mikl
Ушел с форума
13987 / 7000 / **813**
Регистрация: 11.11.2010
Сообщений: 12,592

01.09.2014, 06:04 **[TC]** **0**

Мануал по flat assembler

01.09.2014, 06:04. **Просмотров** 99783. **Ответов** 50
Метки (Все метки)

Ответ

3.1.4 Процедуры (64-разрядные)

In 64-bit Windows there is only one calling convention, and thus only two macroin- structions for calling procedures are provided. The fastcall calls directly the proce- dure 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 dierent 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:

Assembler Выделить код

```
1  invoke glVertex3d,float 0.6,float -0.6,float 0.0
2  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.

Assembler Выделить код

```
1  frame ; allocate stack space just once
2  invoke TranslateMessage,msg
3  invoke DispatchMessage,msg
4  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:

Assembler Выделить код

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

3.1.5 Customizing procedures

It is possible to create a custom code for procedure framework when using proc macroin- struction. 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 proce- dure (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 aswell. The second one is a bitfield containing some

ags, 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 an 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 также 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:

Assembler Выделить код

```
1  prologue@proc equ prologuedef
2  epilogue@proc equ epiloguedef
3  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.

Assembler Выделить код

```
1   macro sp_prologue procname,flag,parmbytes,localbytes,reglist
2   - local loc
3   loc = (localbytes+3) and (not 3)
4   parmbase@proc equ ebp+8
5   localbase@proc equ ebp-loc
6   if parmbytes | localbytes
7   push ebp
8   mov ebp,esp
9   if localbytes
10  repeat localbytes shr 12
11  mov byte [esp-%*4096],0
12  end repeat
13  sub esp,loc
14  end if
15  end if
16  irps reg, reglist \{ push reg \} }
17  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 также 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 eect 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:

Assembler Выделить код

Assembler Выделить код

```
1  prologue@proc equ static_rsp_prologue
2  epilogue@proc equ static_rsp_epilogue
3  close@proc equ static_rsp_close
```

3.1.6 Экспорт

Макроинструкция **export** создает экспорт данных для PE файла (она должно быть помещена в раздел, отмеченный как export, или в пределах блока экспорта данных). Первый параметр должен быть символьная строка, определяющая название библиотечного файла, а остальное должно быть парами параметров, сначала в каждой паре название процедуры, определенной где-нибудь в исходнике, и потом символьная строка, содержащей название, под которым эта процедура должна экспортироваться библиотекой. Этот пример

Assembler Выделить код

```
1  export 'MYLIB.DLL',\
2      MyStart,'Start',\
3      MyStop,'Stop'
```

определяет таблицу, экспортирующую две функции, которые определены под названиями MyStart и MyStop в исходниках, но экспортируются библиотекой под более короткими названиями. Макрокоманда заботится об алфавитной сортировке таблицы, которая требуется форматом PE.

3.1.7 Component Object Model

Макрос **interface** позволяет объявлять интерфейс для объекта типа COM, первый параметр — название интерфейса, и затем последовательность из имен методов, как в этом примере:

Assembler Выделить код

```
1  interface ITaskBarList,\
2      QueryInterface,\
3      AddRef,\
4      Release,\
5      HrInit,\
6      AddTab,\
7      DeleteTab,\
8      ActivateTab,\
9      SetActiveAlt
```

Тогда может использоваться макрос comcall, чтобы вызвать методы данного объекта. Первым параметром для этого макроса должен быть хендл объекта, второй должен быть названием интерфейса COM, осуществленного этим объектом, и затем название метода и параметры для этого метода. Например:

Assembler Выделить код

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

здесь регистр EBX содержит хендл объекта COM с интерфейсом **ITaskBarList**, и вызывает метод этого объекта ActivateTab с параметром [hwnd].

Вы можете также использовать название интерфейса COM тем же самым способом как название структур, объявляя переменную, которая будет содержать хендл объекта данного типа например:

Assembler Выделить код

```
1  ShellTaskBar ITaskBarList
```

Вышеупомянутая строка объявляет переменную размером в двойное слово, в которой может быть сохранен хендл COM объекта. После сохранения там хендла объекта, его методы можно вызывать с cominvk. Эта макроинструкция нуждается только в названии обьявленной переменной с назначенным интерфейсом и названием метода в качестве первых двух параметров, и затем параметры для метода. Так что метод ActivateTab объекта, хендл которого сохранен в переменной ShellTaskBar как определено выше, можно вызвать вот так:

Assembler Выделить код

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

который делает то же самое, как и этот:

Assembler Выделить код

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

Вернуться к обсуждению:
Мануал по flat assembler

Следующий ответ

0

IT_Exp Эксперт **87844** / 49110 / **22898** Регистрация: 17.06.2006 Сообщений: 92,604	01.09.2014, 06:04

Ошибка в flat assembler
начал изучать ассемблер сталкнулся с такой проблемой: перепечатываю пример из книги: org...

flat assembler массив
У меня есть задание "Упорядочить по убыванию элементы каждого столбцу матрицы" Числа произвольные...

Массив в Flat Assembler
Всем добрый день! Подскажите, почему не работает массив в Flat Assembler? org 100h jmp start...

Как использовать Flat Assembler в Free Pascal?
Я недавно хотел разработать так ради прикола мини ОС с использованием в связке Free Pasal и Flat...

0