Валерий Яковлев

Написание пользовательской DLL доступа к универсальному OPC-серверу Fastwel

Во всем мне хочется дойти
До самой сути.

Б.Л. Пастернак

У многих системных интеграторов при необходимости реализации системы управления верхнего уровня, где часто используются такие современные программные средства, как SCADA-системы, возникает потребность программного сопряжения старого оборудования (для общности будем говорить о контроллере) с собственно пакетом SCADA. Современные SCADA-системы в качестве программного элемента сопряжения с оборудованием используют OPC-серверы. Чаще всего программный интерфейс старого оборудования реализует «самобытный» протокол связи, на текущий момент позабытый, и в силу этого не существует готового OPC-сервера, позволяющего достаточно просто реализовать вопрос стыковки. При этом есть два пути решения возникшей проблемы. Первый — переписать программу контроллера с целью обеспечения стандартного протокола обмена, и второй — написать собственный OPC-сервер. К сожалению, довольно часто первый путь затруднителен, так как связан с существенными программными и материально-техническими издержками, либо просто невозможен. Остаётся безальтернативный путь — писать собственный OPC-сервер. Это довольно сложная работа, требующая наличия программиста высокой квалификации, причём неизбежны большие временные затраты на разработку и отладку. В этой ситуации фирмой Fastwel предложен оригинальный программный продукт — универсальный OPC-сервер [1], позволяющий существенным образом снизить требования к квалификации программиста и сократить временные затраты, связанные с написанием собственного OPC-сервера, так как часть работы (наиболее «наукоёмкая») уже проделана. После приобретения этого программного продукта разработчику предлагается написать библиотеку DLL (Dynamic Link Library) пользователя, осуществляющую только обмен с устройством, всю остальную работу по обеспечению обмена данными между OPC и SCADA осуществляет OPC-сервер. Поставляется и программная заготовка на языке C++, на основе которой пользователь может создавать эту DLL. В данной статье рассматривается программная заготовка для написания пользовательской DLL на ассемблере. Программирование на ассемблере под Windows не только не сложнее написания программ на ассемблере под DOS, а даже проще! Необходимость навыков работы с этим языком актуальна до сих пор, и популярность его ничуть не уменьшается, а иногда применение его остаётся единственным эффективным средством решения проблемы.

Выбор инструментов программирования

На текущий момент хорошо известны следующие средства разработки приложений на ассемблере под Windows: MASM (Microsoft), TASM (Borland), NASM, FASM. Основными пакетами (в пакет входят, как минимум, транслятор, компоновщик и компилятор ресурсов) являются MASM v.6.1X и TASM v.5.0.

Для решения поставленной задачи выбран пакет MASM32 v.8.2, являющийся достаточным для программирования приложений для ОС Windows и содержащий, кроме транслятора MASM v.6.14, компоновщика и редактора ресурсов, компактный редактор, набор специализированных утилит и большое количество примеров. Пакет распространяется бесплатно (http://www.masm32.cjb.net). Программировать на ассемблере под Windows и пользоваться командной строкой и bat-файлами, на мой взгляд, слишком консервативно. Я опробовал две интегрированные среды разработки (IDE) для ассемблера – WinAsm Studio v.4.0.1.266 (http://www.winasm.net) и RadASM v.2.1.0.6 (http://radasm.visualassembler.com/download/radasm.html). Версии IDE указаны на момент написания статьи. Оба проекта достаточно динамично развиваются и распространяются бесплатно. Я остановился на RadASM. Установка этой IDE и работа с ней достаточно просты, и я не буду останавливаться на её описании. Автор RadASM программирует на ассемблере (сама оболочка также написана на этом языке), и потому всё продуманно, функционально и быстро. Кроме того, на сайте есть законченные примеры проектов, облегчающих первые шаги в увлекательный мир ассемблера для Windows.

Введение в программирование на ассемблере в ОС Windows

Безусловно, программирование под Windows имеет некоторые отличия по отношению к «старому стилю» програм-

мирования на ассемблере, и они связаны с изменившимся «лицом» операционной системы (ОС), осуществляющей встроенную поддержку многого из того, что ранее для DOS реализовывалось на уровне приложений и требовало существенных усилий программиста [2]. Так как статья не посвящена глубокому анализу организации Windows, рассмотрим те моменты, которые существенны для решения поставленной задачи.

Первым приятным моментом является то, что операционная система Windows, работая в защищённом режиме, обеспечивает для каждого запущенного на выполнение приложения отдельное виртуальное адресное пространство размером в 4 Гбайт. Теперь, программируя на ассемблере, не приходиться беспокоиться о существовавшей в DOS проблеме ограничения адресного пространства сегментов в 64 кбайт. Таким образом, в 32-разрядный Windows используется только одна модель памяти – «плоская», обозначаемая ассемблерной директивой MODEL FLAT.

Вторым приятным моментом, существенно изменяющим характер программирования в ОС Windows, является то, что при программировании приложения активно используется огромное количество функций интерфейса API (Application Programming Interface), предоставляемых в ***рамках самой операционной системы Windows*** и позволяющих значительно минимизировать труд программиста по реализации каких-либо прикладных задач, будь то создание графического изображения окна или работа с файлом, сводя эту работу к простым вызовам в своей программе. Под DOS приходилось применять системные вызовы (системные прерывания INT 21H) или прибегать к прерываниям BIOS (особенно часто это касалось работы по созданию интерфейса пользователя). Большое богатство функций сосредоточено в системных библиотеках динамической компоновки DLL, основными из которых являются User32.dll, Gdi32.dll, Advapi32.dll и Kernel32.dll. Эти библиотеки предоставляют разработчику программ документированный интерфейс между программами и вызываемыми функциями подсистем. User32.dll реализует функции, связанные с поддержкой пользовательского интерфейса, Gdi32.dll обеспечивает графический интерфейс, а Kernel32.dll отвечает за поддержку работы с памятью и взаимодействие с процессами. В отличие от статически компонуемых библиотек (раннее связывание), привычных нам при программировании в DOS, библиотеки динамической компоновки загружаются в память и обращение к ним идёт динамически, по мере необходимости. При этом распознавание необходимости DLL для загружаемой программы происходит либо автоматически при запуске программы, либо через вызов в этой программе соответствующей функции. Второй способ иногда удобней, так как программа может скорректировать своё выполнение по результатам загрузки DLL и продолжить выполнение, скажем, с ограничением своей функциональности, или самостоятельно выгрузить ранее загруженную и ставшую ненужной DLL. В первом случае Windows сообщит об ошибке и выгрузит приложение. Разные способы загрузки вносят принципиальные изменения и в код программы. Для первого случая в коде программы необходимо наличие строки с директивой ***includelib*** и название так называемой библиотеки импорта (например User32.lib). Эта библиотека позволяет редактору связей получить информацию о требуемой при динамической компоновке DLL, которая вносится в исполняемый файл (библиотеки импорта по имени совпадают с именем представляемых ими DLL и имеют расширение .lib). Кроме того, в исполняемом файле необходимо вызывать функцию ***GetProcAddress*** для каждой используемой функции, что несколько увеличивает код. При самостоятельной загрузке библиотеки импорта в тексте программы не используются, при этом не исключается необходимость чёткого знания количества и формата передаваемых параметров в используемых функциях. Существует два механизма определения функции при её связывании: по номеру и по имени этой функции. Чаще используется второй вариант, имеющий, как минимум, преимущество читабельности исходной программы, так как название отражает выполняемую функцию. Являясь разделяемым ресурсом в ОС Windows, DLL существенным образом экономят системные ресурсы, так как в оперативной памяти находится только одна копия DLL при обращении к ней нескольких программ. Кроме того, DLL существенно уменьшают объём самих создаваемых программ, тем самым экономя системный ресурс в части занимаемого программой места на жёстком диске. DLL, кроме исполняемого кода, могут содержать и другие данные, например, графические изображения, шрифты и.т.д.

Мы подошли к третьему, очень важному положительному моменту, являющемуся одним из ключевых для ОС Windows, – многозадачности. Как мы помним, в DOS, чтобы написать программу, обеспечивающую параллельное (точнее, псевдопараллельное) выполнение программного кода, приходилось очень потрудиться, так как сама DOS была спроектирована как однозадачная ОС. В Windows эти проблемы в основном решаются на системном уровне, хотя определённые тонкие моменты при программировании остаются (вопросы синхронизации при межзадачном обмене и использовании общесистемных ресурсов, например COM-порта). Вопроса синхронизации доступа для разделяемого участка программы при обращении к общим данным мы ещё коснёмся.

При активном использовании вызовов функций необходимо помнить о порядке передачи параметров через стек (и не только через стек!) в вызываемые процедуры и функции. С этими моментами программисты должны были сталкиваться и раньше, при написании процедур и функций на ассемблере в программах, написанных на языках высокого уровня. Кроме того, при разных подходах является актуальным вопрос о том кто «чистит» стек: вызывающая программа или вызываемая подпрограмма. Для распространенных языков программирования C и Pascal существуют следующие соглашения по этим двум вопросам. По соглашению для C параметры в вызываемую функцию передаются справа налево и вызывающая программа должна очистить стек. Например вызов функции Test

```
Test (param_1, param_2)
```

в ассемблерном виде выглядит как

```
push [param_2]
push [param_1]
call Test
add sp,8
```

По соглашению для Pascal всё с точностью до наоборот, то есть передача параметров в функцию осуществляется слева направо и вызываемая функция ответственна за очистку стека. Для платформ Win32 используется комбинированный вариант этих соглашений STDCALL. Согласно этому варианту соглашения данные в вызываемую функ-

цию передаются справа налево и вызываемая функция чистит стек. Говоря о вызовах функций, нельзя не коснуться темы существования в MASM32 очень удобной директивы INVOKE, позволяющей минимизировать текстовые затраты программиста на ассемблере при оформлении вызова функции в тексте программы. Как известно, стандартный способ вызова функции на ассемблере — через оператор Call, то есть вызов функции `Test2 (param_1,param_2)` с двумя параметрами будет выглядеть следующим образом:

```
Push param_2
Push param_1
Call Test2@N
```

Здесь N – количество отправляемых в стек байтов.

Использование директивы INVOKE позволяет нам оформить вызов функции `Test2` более коротко:

```
INVOKE Test2 param_1, param_2
```

При этом передача параметров в стек и очистка стека будут автоматически оформлены компилятором. Параметры, передаваемые в функцию, при этом могут являться адресом (смещением — OFFSET или ADDR), непосредственным значением или значением в регистре. Естественно, для того чтобы транслятор и редактор связей правильно обработали эту директиву (проверили количество и тип параметров вызываемой функции), они должны иметь прототип функции. Прототипы функций имеют вид [имя функции] ключевое слово [PROTO] и далее через двоеточие в соответствии с числом параметров их тип, например DWORD, и хранятся в многочисленных inc-файлах (\masm32\INCLUDE*.inc). Прототип для `Test2` выглядит как `Test2 PROTO: DWORD: DWORD`.

Одними из фундаментальных понятий в Windows являются окно, сообщение, цикл обработки очереди сообщений. Окно в упрощённом виде можно рассматривать как программный каркас приложения в Windows, в котором определены все свойства этого приложения, включая видимый интерфейс пользователя и внутреннюю логику работы самого приложения. Каждое окно для идентификации операционной системой имеет свой дескриптор окна — четырёхбайтовое число, позволяющее операционной системе понимать, какая из программ её о чём-то просит. Для этого в большинстве вызываемых из приложения функций, связанных с использованием системных ресурсов, в качестве параметра присутствует этот дискриптор. Окно не обязательно должно иметь видимый пользователю интерфейс. Операционная система Windows генерирует и распределяет между запущенными приложениями сообщения о событиях во «внутреннем мире» ОС.

Событиями являются нажатия на клавишу клавиатуры, перемещение мыши, поступление информации в COM-порт и т.д. Приложение получает информацию о событиях в виде заполненных структур. Приложение не обязано реагировать на все события и может выбирать, какие из поступивших событий будет обрабатывать. Для каждого окна существует своя очередь предназначенных ему сообщений, обрабатываемая в рамках главной процедуры этого окна. Таким образом, каждое приложение имеет внешний интерфейс пользователя (в случае консольных приложений он отсутствует) с меню и кнопками управления, программу, получающую сообщения об имеющих место событиях, и собственно код главной процедуры окна, обрабатывающий эти события и обеспечивающий функциональность конкретного приложения. Подчеркну ещё раз (это важный момент), что приложению нет необходимости что-либо предпринимать для получения и формирования очереди сообщений — эта функция лежит на самой операционной системе. Программный код главной процедуры приложения только обрабатывает события, и всё. Это, безусловно, упрощает программирование в ОС Windows, так как в случае приложения под DOS оно должно само следить за происходящим в ОС, не полагаясь на её «помощь». DLL не является полнофункциональным приложением в привычном понимании, но тем не менее у неё есть так называемая процедура входа (на неё указывает метка за директивой END), которая получает и обрабатывает события, передаваемые ей операционной системой. Подчинённый характер DLL заключается в ограниченном числе получаемых сообщений и выгрузке из памяти при закрытии основного приложения, которое «вызвало к жизни» DLL, хотя само приложение при этом может продолжать свою работу. При вызове процедуры входа DLL в стеке для неё передаются три параметра, один из которых поясняет, по какому поводу её «побеспокоили». DLL передаются следующие параметры: первый — идентификатор DLL-библиотеки, второй – сообщение о причине вызова, третий — зарезервирован для дальнейших расширений. Причины вызова могут быть следующие:

`DLL_PROCESS_ATTACH` – сообщение о том, что библиотека загружена в адресное пространство вызвавшего её процесса;

`DLL_THREAD_ATTACH` – сообщение о создании вызвавшим библиотеку процессом нового потока;

`DLL_PROCESS_DETACH` — сообщение о том, что библиотека выгружается из адресного пространства вызвавшего ее процесса;

`DLL_THREAD_DETACH` – сообщение об уничтожении процессом ранее созданного им потока.

Основной «каркас» простейшей DLL

```
;-------------------------------------------------
;             SimpleDLL.asm
;-------------------------------------------------
.386
.model flat, stdcall
include \masm32\include\windows.inc
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
.data
.code
DllEntry proc hInstSimpleDLL: HISTANCE, reason:DWORD,
future: DWORD
mov eax, TRUE
ret
DllEntry endp

FirstFunction proc param1:DWORD, param2:DWORD
Mov ebx,param1
Mov eax,param2
Add eax,ebx
ret
FirstFunction endp

End DllEntry
```

```
;-------------------------------------------------------
;              SimpleDLL.inc
;-------------------------------------------------------
FirstFunction proc param1:DWORD, param2:DWORD

;-------------------------------------------------------
;              SimpleDLL.def
;-------------------------------------------------------
LIBRARY SimpleDLL
EXPORTS FirstFunction
```

Удивительно, но перед нами текст совершенно работоспособной DLL (точнее, три текстовых файла .asm, .inc и .def). Согласитесь, что всё достаточно прозрачно и лаконично. Процедура DllEntry вызывается операционной системой в случаях, определённых перечнем приведенных ранее сообщений. При успешном завершении действий, связанных с конкретным сообщением (реально его можно и не отрабатывать, как в приведенном «каркасе»), процедура возвращает TRUE.

Если для нормальной работы DLL при её загрузке в память необходима инициализация внутренних переменных или просто выделение блока памяти, а это не может быть выполнено, то, вернув FALSE, Вы тем самым сообщаете операционной системе о «неудаче» и DLL будет выгружена. Наша DLL содержит всего лишь одну функцию FirstFunction, которая, получив два параметра, складывает их и возвращает результат сложения (надо сказать, что возвращаются значения в регистре eax). В файле определений (.def) после ключевого слова LIBRARY указывается имя библиотеки. Все функции DLL, которые мы хотели бы сделать доступными для внешних программ, перечисляются с ключевым словом EXPORTS. После компиляции проекта мы получаем два файла, саму библиотеку SimpleDLL.DLL и файл SimpleDLL.lib, последний является библиотекой импорта, о которой мы уже говорили. Напомню, что она необходима при создании программы пользователя, использующей созданную Вами библиотеку в режиме явного связывания. Собственно сам «каркас» приведённой DLL уже является подобным примером, использующим системные библиотеки.

Массивы и структуры пользовательской DLL

Для начала будем рассматривать простейший вариант, когда в адресном пространстве сервера будет видимым только одно устройство. Как Вы, наверно, помните, все теги, видимые клиентом (публикуемые OPC-сервером), делятся на три типа: аналоговые, битовые и целые. Теги представляют собой структуру данных. Структура – это конструкция в ассемблере (и не только в нём), позволяющая объединять данные различных типов. Таким образом, структура определяет составной тип данных, в котором содержится один или более элементов данных, называемых членами (полями) структуры. Синтаксис определения структуры в ассемблере довольно прост:

```
Имя STRUC
ENDS
```

Параметр «Имя» – это присваиваемое структуре имя. Все данные, находящиеся между ключевыми словами STRUC и ENDS, являются членами структуры.

Обращение к членам структуры выглядит следующим образом: «Имя.ИмяПоляСтруктуры», то есть первым идёт имя структуры и далее через разделитель «.» — имя поля структуры. Как уже говорилось, важно, что каждый член структуры (или поле структуры) может иметь разный тип. Учитывая это, при помощи определения различных структур можно создавать описания объектов с набором признаков, присущих этим объектам. В нашем случае каждый тег имеет как минимум два поля: имя тега и само значение. Поле имени тега содержит указатель на строковую переменную, которая содержит имя тега. В поле тега «значение» в зависимости от того, какой тип тега определяется структурой, будет переменная, содержащая в явном виде значение переменной соответствующего типа для ассемблера. Посмотрим, как всё это выглядит на ассемблере.

Структура, описывающая тег, содержащий аналоговую переменную:

```
anTag struc
ptrname dd NULL
value dd 0
anTag ends
```

Структура, описывающая тег, содержащий битовую переменную:

```
bitTag struc
ptrname dd NULL
value db 0
bitTag ends
```

Структура, описывающая тег, содержащий переменную целого типа:

```
intTag struc
ptrname dd NULL
value dd 0
intTag ends
```

При определении структуры переменные инициализировать не обязательно. Если Вы всё же указываете какие-либо значения, они принимаются значениями по умолчанию при объявлении в тексте программы переменной с типом этой структуры.

Следующая структура TDataForOPCServer является основной, так как содержит поля, полностью описывающие конкретное устройство, с точки зрения наличия (или отсутствия) у устройства определённых типов данных (по сути, реальных входов-выходов устройства) и количества логических входов. Структура выглядит следующим образом:

```
TDataForOPCServer struc
dataIsValid db 1 ; признак качества данных
anTagsQnty dw 1 ; количество аналоговых тегов
TAnTag dd 0 ; указатель на массив аналоговых тегов
bitTagsQnty dw 1 ; количество битовых тегов
TBitTag dd 0 ; указатель на массив битовых тегов
intTagsQnty dw 1 ; количество тегов с значением целого
типа
TIntTag dd 0 ; указатель на массив тегов целого типа
structVersionNumber dw 0 ; поддержка версии 0 OPC-сер-
вера Fastwel
DeviceName db "Device1" ; имя устройства
DeviceNameL db 57 dup (" ") ;
Reserved db 128 dup(0) ; зарезервировано
TDataForOPCServer ends
```

Смысл полей достаточно ясен из текста комментария. Необходимо сделать акцент на том, что поле версии OPC-сервера Fastwel — это не просто порядковый номер текущей версии универсального сервера, а данные, влияющие на характер и количество полей самой структуры TDataForOPCServer, интерпретируемые сервером, то есть

определяющие функциональные возможности универсального OPC-сервера в целом.

Так, если в поле structVersionNumber стоит версия 1, то структура является описанием некоторого массива структур, что позволяет «видеть» несколько устройств.

Функции, реализуемые пользовательской DLL

Функции, вызываемые непосредственно OPC-сервером, которые необходимо реализовать в пользовательской DLL (и экспортировать для возможности их использования), следующие:

```
1. const TDataForOPCServer * PASCAL
GetDataForOPCServer( void );

2. unsigned char PASCAL SetAnTagValue(
        unsigned  number, // устройства (HIWORD) и
            // номер тега в массиве (LOWORD)
        float value      // новое значение параметра
        );

3.  unsigned char PASCAL SetBitTagValue(
        unsigned  number, // устройства (HIWORD) и
            // номер тега в массиве (LOWORD)
        unsigned char value // новое значение параметра
        );

4. unsigned char PASCAL SetIntTagValue(
        unsigned  number, // устройства (HIWORD) и
            // номер тега в массиве (LOWORD)
        int value     // новое значение параметра
        );
```

Описание функций взято из заголовочного файла dataserv.h, входящего в поставку универсального OPC-сервера фирмы Fastwel.

Как можно видеть, первая функция возвращает OPC-серверу указатель на структуру TDataForOPCServer, о которой мы говорили ранее, тем самым давая информацию о наличии и количестве соответствующих тегов, остальные функции позволяют OPC-серверу изменять значения соответствующих тегов. Посмотрим, как выглядит простейшая реализация этих функций на ассемблере:

```
1. GetDataForOPCServer proc
      mov eax,offset TestTDataForOPCServer
       ret
   GetDataForOPCServer endp

2.  SetAnTagValue proc number:DWORD, value:DWORD

      mov eax,value
      mov TestAnTag.value,eax
      mov eax,1
      ret
     SetAnTagValue endp

3.  SetBitTagValue proc number:DWORD, value:BYTE
      mov al,value
       mov TestbitTag.value,al
      mov eax,1
      ret
    SetBitTagValue endp
4. SetIntTagValue proc number:DWORD, value:DWORD
      mov eax,value
      mov TestintTag.value,eax
      mov eax,1
      ret
   SetIntTagValue endp
```

Как видим, всё достаточно просто. Приведённый текст не является объявлением функций, как это видим в прототипе на языке C++ 5. Я ввёл некоторую конкретику для сокращения дальнейшего обращения к тексту на ассемблере. Реально объявление для функции установки аналогового тега выглядит так:

```
SetAnTagValue proc number:DWORD, value:DWORD
    ret
 SetAnTagValue endp
```

Результат работы первой функции — это указатель, возвращаемый в регистре eax. Если в eax возвращается NULL, это ошибка. Остальные функции получают в качестве параметров номер устройства (старшее слово), в нашем примере оно одно, так как версия 0, номер соответствующего тега в массиве (младшее слово) параметра number и его новое значение — параметр value. При этом приведённые в качестве примера функции не изменяют выходные значения в реальном устройстве, а просто изменяют значение в массиве и сообщают OPC-серверу, что всё в порядке (значение TRUE или 1 в регистре eax).

В действительности в функциях, изменяющих значения тегов, Вам необходимо реализовать не формальное изменение значение тега в массиве данных, а реальное изменение значения в устройстве (это может быть значение в регистре платы, установленной в слот компьютера, или значение на выходе контроллера, обмен с которым осуществляется по коммуникационному порту).

Создание потока и зачем это нужно

Наверняка у читателя возникнет вопрос, каким образом осуществляется обновление данных в структуре TDataForOPCServer. Законный вопрос. Необходимо иметь некую подпрограмму, которая периодически считывала бы значения из нашего устройства и заносила эти значения в поле value соответствующего тега.

Роль подобных подпрограмм, выполняемых в режиме псевдопараллельности, играют ***потоки***, создаваемые основным приложением. Каждое приложение имеет хотя бы один первичный поток (можно считать, что это основная программа), который может создавать вторичные потоки, которые, в свою очередь, могут создавать свои вторичные потоки, и т.д. Между создающим и создаваемым потоком существуют «родительские» взаимоотношения, выражающиеся в том, что поток, создающий вторичный поток, может его и завершить, обратное неверно. Выделением кванта времени для исполнения какого-либо потока (и собственно его выполнением) занимается сама операционная система.

DLL ничем не хуже других приложений Windows, и поэтому в рамках пользовательской DLL нам необходимо создать поток (подпрограмму), который и будет заниматься вопросом обновления данных. Рассмотренный вопрос о потоках является системным, относящимся к организации ОС Windows, и, конечно же, его необходимо учитывать при программировании приложений не только на ассемблере.

В рамках API есть функции, при помощи которых и осуществляется процесс создания и управления потоками. Допустим, есть процедура, которая через определённый интервал времени, определяемый программной задержкой, считывает текущее значение переменной целого типа из структуры TDataForOPCServer, увеличивает его на единицу и записывает получившееся значение обратно. Надо отметить, что код создаваемых потоков имеет доступ ко всем переменным приложения.

```
DllThread proc
      DllThreadLoop:
      pushad
      mov cx,100
      Thread_delay:
      loop Thread_delay
      mov eax, TestintTag.value
      inc eax
      mov TestintTag.value, eax
      popad
      jmp DllThreadLoop
      ret
 DllThread endp
```

Теперь в рамках программы, которая будет выполняться при загрузке DLL в память, необходимо эту процедуру запустить:

```
invoke CreateThread,NULL,0,offset DllThread,
[Dll_Handle],0,offset HTHR
```

Как видим, это осуществляется вызовом соответствующей функции API (CreateThread) с передачей ей необходимых параметров:

1) указатель на структуру атрибутов безопасности доступа (актуально для NT). Обычно не используется — NULL;

2) размер стека потока. Если значение параметра равно 0, принимается размер стека родительского потока;

3) адрес исполняемой процедуры потока. В нашем случае это DllThread, которая выполняет порученную ей работу по обновлению данных;

4) параметр для функции потока;

5) флаг состояния потока. Если параметр нулевой, выполнение потока начинается немедленно. Если созданный поток ожидает запуска (функцией ResumeThread), используется флаг CREATE_SUSPEND;

6) адрес переменной, в которую записывается дескриптор созданного потока.

Как мы говорили, родительский поток, кроме того что может породить вторичный поток, может его и «убить», вызовом соответствующей функции. Это осуществляется в рамках окончания работы приложения OPC-сервера, которое сообщает DLL о том, что работа приложения завершается и ему необходимо освободить все ресурсы и завершить все созданные потоки.

```
invoke TerminateThread,[HTHR],0
```

После рассмотрения всех этих вопросов можно взглянуть на то, что у нас получилось:

```
; Project DLL for v.0 Universal OPC Server Fastwel
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
include DATASERV.Inc

.data
Dll_Handle dd ?
HTHR  dd ?
; переменные устройства 0
TestAnTagName  db "TestAnTag",0
TestbitTagName db "TestbitTag",0
TestintTagName db "TestintTag",0

TestTDataForOPCServer TDataForOPCServer <>


TestAnTag anTag <TestAnTagName,2.3>
TestbitTag bitTag <?>
TestintTag intTag <?>

.code
DllThread proc
 DllThreadLoop:
      pushad
      mov cx,100
      Thread_delay:
      loop Thread_delay
      mov eax, TestintTag.value
      inc eax
      mov TestintTag.value, eax
      popad
      jmp DllThreadLoop
      ret

DllThread endp
DllEntry proc hInstDLL:HINSTANCE, reason:DWORD,
reserved1:DWORD

    mov eax,reason
    cmp eax,DLL_PROCESS_DETACH
    je @dll1 ; закрытие библиотеки
    cmp eax,DLL_PROCESS_ATTACH
    je @dll2 ; открытие библиотеки
    cmp eax,DLL_THREAD_DETACH
    je @dll3
    cmp eax,DLL_THREAD_ATTACH
    je @dll4
    ; неизвестная команда !
    mov eax,TRUE
    jmp dll_exit
;----------------------------------------
    ; закрытие библиотеки процессом
    @dll1:
    invoke TerminateThread,[HTHR],0
    mov eax,TRUE
    jmp dll_exit
;-------------------------------------------------
    ; открытие библиотеки процессом
    @dll2:
    mov eax,dword ptr [ebp+8] ; Взяли дескриптор
    mov Dll_Handle,eax

  ; инициализируем структуру битовой переменной
  ; устройства 0
```

```
        mov TestbitTag.ptrname, offset TestbitTagName
        mov TestbitTag.value,1
       ; инициализируем структуру переменной целого типа
устройства 0
        mov TestintTag.ptrname,offset TestintTagName
        mov TestintTag.value,650

       ; инициализируем структуру TDataForOPCServer
       mov TestTDataForOPCServer.dataIsValid,1
       mov TestTDataForOPCServer.anTagsQnty,1
       mov TestTDataForOPCServer.TAnTag,offset TestAnTag
       mov TestTDataForOPCServer.bitTagsQnty,1
       mov TestTDataForOPCServer.TBitTag,offset TestbitTag
       mov TestTDataForOPCServer.intTagsQnty,1
       mov TestTDataForOPCServer.TIntTag,offset TestintTag

       ; создаём поток опроса значений
    invoke CreateThread,NULL,0,offset DllThread,
[Dll_Handle], 0,offset HTHR
       ;-------------------------------------
       mov eax,TRUE
       jmp dll_exit
  ;-----------------------------------------------------
       ; закрылся один из потоков
       @dll3:
       mov eax,TRUE
       jmp dll_exit
  ;-----------------------------------------------------
       ; открылся один из потоков
       @dll4:

         mov eax,TRUE
         dll_exit:
         ret

  DllEntry endp
  GetDataForOPCServer proc
        mov eax,offset TestTDataForOPCServer

        ret

  GetDataForOPCServer endp
  SetAnTagValue proc number:DWORD, value:DWORD

        mov eax,value
        mov TestAnTag.value,eax
        mov eax,1
        ret

  SetAnTagValue endp
  SetBitTagValue proc number:DWORD, value:BYTE

        mov al,value
        mov TestbitTag.value,al
        mov eax,1
        ret

  SetBitTagValue endp
  SetIntTagValue proc number:DWORD, value:DWORD

        mov eax,value
        mov TestintTag.value,eax
        mov eax,1
        ret

  SetIntTagValue endp
  End DllEntry
```

Создание критических секций, и зачем они нужны

В рамках идеологии объектного программирования доступ к переменным (чтение или запись) некоторого объекта должен осуществляться единообразным способом, или ***методом***. То есть если у нас определена подпрограмма SetIntTagValue изменения значения переменной в массиве TDataForOPCServer, то и в программе, отвечающей за считывание и обновление значений переменных, необходимо использовать ту же самую процедуру (метод), а не писать значение напрямую, как это осуществляется у нас в примере кода потока. Выглядеть это должно так:

```
   DllThread proc
         DllThreadLoop:
         pushad
         mov cx,100
         Thread_delay:
         loop Thread_delay
         mov eax, TestintTag.value
         inc eax
          mov ebx,0
         invoke SetIntTagValue, ebx, eax
         popad
         jmp DllThreadLoop
         ret
     DllThread endp
```

Использование одной и той же подпрограммы для изменения состояния одной и той же переменной (её атрибутов), при том что доступ к ней имеют программы DLL и сервера (асинхронный доступ), может приводить к неверным конечным результатам. Чтобы избежать подобной ситуации, существует программный инструмент синхронизации доступа – критическая секция. Для использования этого инструмента необходимо создать структуру `DataservCriticalSection RTL_CRITICAL_SECTION <?>` и инициализировать её соответствующей функцией

```
  invoke InitializeCriticalSection,addr
DataservCriticalSection
```

Теперь при реализации функции изменения значения переменной при входе и выходе вызываются соответствующие функции, функция `invoke EnterCriticalSection` – войти в критическую секцию и `invoke LeaveCriticalSection` — покинуть критическую секцию. То есть в общем виде это выглядит так:

```
  SetIntTagValue proc number:DWORD, value:DWORD
               invoke EnterCriticalSection,addr
DataservCriticalSection

    ; здесь код процедуры изменения значения переменной

  invoke LeaveCriticalSection,addr
DataservCriticalSection
        mov eax,1
        ret

  SetIntTagValue endp
```

Подобная реализация гарантирует, что программа, ограниченная функцией входа и выхода в критическую секцию, будет выполняться только одним потоком приложения.

При завершении работы программы DLL, когда все ресурсы необходимо освободить, вызывается функция для удаления ранее созданного объекта — критическая секция `invoke DeleteCriticalSection, addr DataservCriticalSection`. В приведённом примере DLL пользователя не используется критическая секция, но сказать о ней необходимо, так как это один из стандартных способов решения вопроса синхронизации работы потоков при программировании (наряду с семафорами, событийной синхронизацией и мьютексами).

Заключение, или Что впереди

В статье рассматривался вариант реализации самой простой версии интерфейса (версия 0) пользовательской DLL доступа к данным. Разработчики фирмы Fastwel постоянно совершенствуют свой программный продукт, и в зависимости от версии Universal OPC-сервер появлялись новые версии интерфейса DLL, который может выбрать для реализации разработчик DLL. Новые версии интерфейса предлагают тот или иной дополнительный сервис при написании кода DLL и публикации дополнительных параметров тегов. В отличие от версии 0, в версии 1 появляется возможность задания нескольких устройств в адресном пространстве OPC-сервера. Версия 2 отличается от версии 1 признаком качества данных в каждом теге. Версия 3 отличается от версии 2 дополнительным полем времени в каждом теге, а также наличием функций синхронизации при обновлении атрибутов тега со стороны DLL.

Интересным моментом при использовании этих функций является то, что параметр tstamp, имеющий тип FILETIME, передаётся в функцию не через стек, а через регистр сопроцессора. В целом реализация пользовательской DLL на ассемблере с учётом возможностей последующих версий интерфейса (1, 2, и 3) не несёт в себе каких-либо принципиальных моментов по отношению к рассмотренным нами вопросам. Хочется надеяться, что базовые моменты описаны достаточно полно и послужат отправной точкой для реализации продвинутых вариантов пользовательской DLL. Главное, чтобы программист понял, что использование ассемблера при реализации программ для Windows ничуть не сложнее того, к чему он привык в DOS. Серьёзные изменения в программировании под Windows в основном связаны с появлением большого количества структур системных объектов, API и с необходимостью учёта многозадачности Windows (вопрос написания драйверов в этой ОС — это отдельная тема). Но тут, как говорится, учи матчасть, без её знания и раньше жилось непросто. ●

Литература

1. Fastwel Universal OPC-сервер версия 1.0. Руководство пользователя. — М.: Fastwel, 1999.
2. Пирогов В.Ю. Ассемблер для Windows. — М.: ИД Молгачева С.В., 2002.

Автор — сотрудник фирмы ПРОСОФТ
Телефон: (812) 325-3790
Факс: (812) 325-3791
E-mail: info@spb.prosoft.ru

НОВОСТИ НОВОСТИ НОВОСТИ НОВОСТИ НОВОСТИ НОВОСТИ

ПРОСОФТ — лучший дистрибьютор Advantech IAG в 2004 году

В начале июня 2005 года в американском городе Цинциннати, штат Огайо, прошла трёхдневная конференция Advantech eAutomation World Partner Conference, в которой приняли участие 160 представителей и дистрибьюторов из Америки, Европы и Азии.

Это масштабное мероприятие корпорация Advantech, а точнее её подразделение, занимающееся промышленной автоматизацией, Industrial Automation Group, проводит раз в два года. Его основная цель — ознакомить партнеров с перспективами развития бизнеса Advantech на ближайшие пять лет как в области технологий производства промышленной электроники, так и в сфере маркетинга и работы с клиентами.

Открывая конференцию, президент Advantech Industrial Automation Group г-н Минг Чин Ву (Ming-Chin Wu) отметил важное изменение в концепции развития компании: оставаясь лидером рынка промышленной автоматизации, компания должна ориентироваться не только на техническое совершенство продукции, но и на оказание полного комплекса услуг клиентам. Не

Награда Star Award, присужденная фирме ПРОСОФТ

каждое региональное представительство или дистрибьютор Advantech способно удовлетворить все потребности клиента, начиная с отлаженной логистики, технической поддержки и обучения и заканчивая участием в совместных разработках аппаратуры и систем промышленной автоматики.

Россия на конференции была представлена компанией ПРОСОФТ, имеющей наивысший статус среди независимых дистрибьюторов Advantech — Premier Channel Partner. Этого статуса компания была удостоена на предыдущей всемирной конференции в Тайбэе два года назад, но и на этот раз представителю ПРОСОФТ не пришлось уезжать с пустыми руками. Новая награда Star Award была вручена компании ПРОСОФТ за выдающиеся достижения в 2004 году.

По объёму продаж оборудования Advantech для промышленной автоматизации компания ПРОСОФТ достигла рекордных показателей, опередив многих дистрибьюторов даже из экономически развитых стран Западной Европы и Северной Америки. Наибольшим спросом у заказчиков ПРОСОФТ в России и других государствах бывшего СССР пользовались устройства промышленной коммуникации, популярные модули серий ADAM-4000 и ADAM-5000, промышленные мониторы FPM для человеко-машинного интерфейса (HMI) и мощные процессорные платы для индустриального применения серии PCA. ●