

Особенности реализации командного процессора cmd.exe операционных систем WinNT

Или УРОК ВАТ-АНИКИ 2-ой

Автор: Андрей Дибров

Источник: RSDN Magazine #4-2007

Опубликовано: 15.03.2008

Версия текста: 1.6

Предисловие Тестирование Спецсимволы Команда set

Создание переменной
Удаление переменной
Сброс ERRORLEVEL
Условный блок
Условный блок и ERRORLEVEL
Всё дело в скобках!

Командный процессор cmd.exe

Set и блок
Exit /b %ERRORLEVEL% и блок

Команда goto

Условный блок

Команда call

Команда call и блок
Двойное раскрытие переменной
Двойное раскрытие переменной и блок
Двойное раскрытие переменной и оператор "!"

Команды setlocal/endlocal

Оператор "!" или оператор "%"?
Восстанавливаем окружение после работы скрипта

Вызов с параметрами

Вызов внешнего приложения
Проблема обратного слеша (\)
Call-вызов скрипта и переменные %0-%9
Команда if и переменные %0-%9
Команда echo с точкой

И ещё немного о переменных

Спецсимволы в переменных
Комментарии в скриптах
Установка значения переменных из текстовых файлов
Установка значения переменных из стандартного вывода

Несколько слов о компиляторах

Есть ли замена cmd.exe?

Заключение

Приложение

Ключи реестра
Внешние утилиты и скрипты

Ссылки по теме

Использованные источники



Я спросил у сакуры,
Где та гейша, которая разбила мне сердце.
Сакура не ответила.
И это хорошо.
В нашем роду и так полно психов,
Которые говорят с деревьями и травой.
(<http://www.bash.org.ru/>)

ПРЕДИСЛОВИЕ

Желание организовать собранный материал в статью появилось после некоторых исследований работы bat-скриптов и накопления опыта по их составлению в проекте под Microsoft Visual Studio 2005, интенсивно использующего промежуточные bat-скрипты для взаимодействия непосредственно с некоторыми сторонними консольными утилитами, участвующими в сборке.

В каждом разделе статьи будет рассматриваться та или иная проблема, будут оговорены её последствия или приведено наилучшее, по мнению автора, решение. Проведённые изыскания можно рассматривать отдельно, но крайне желательно это делать в том порядке, в котором они встречаются в статье, т.к. многие проблемы связаны друг с другом и что-то в дальнейшем может быть непонятно из-за пропущенных деталей.

Статья предусматривает, что читатель уже знаком с написанием bat-скриптов NT (далее просто скриптов), знает о применении команд и умеет пользоваться справкой по встроенным командам и прилагаемым утилитам командного процессора. В статье не рассматриваются сторонние утилиты, а также их проблемы и особенности работы.

ПРИМЕЧАНИЕ

Некоторые проблемы, найденные в скриптах, могут быть связаны с их использованием (запуском) из проекта для Microsoft Visual Studio 2005 (C++), а также, возможно, и других сред, использующих обращения к bat-скриптам.

В конце статьи приведены тексты bat-скриптов и ссылки на некоторые консольные утилиты, использовавшиеся при тестировании примеров.

ТЕСТИРОВАНИЕ

Всё что здесь проверялось, а оно проверялось мной полностью только на одной системе, проверялось обычным образом, т.е. выводы о работе строились в основном на результатах выполнения тест-скриптов, их текст и приводятся в качестве аргументов к каждому пункту проблемы.

ПРЕДУПРЕЖДЕНИЕ

Здесь и далее все bat-скрипты тестировались в основном только под Windows XP Service Pack 2 (Build 2600.xpsp_sp2_gdr.070227-2254 : Service Pack 2). Возможно, это справедливо и для системы без сервис паков или с предыдущими сервис паками, но это особым образом не проверялось.

Статья не претендует на полный охват всех особенностей командного процессора, могут быть неточности в связи с уже установленными так называемыми фиксами от Microsoft или недосмотром самого автора.

Всё сказанное в статье справедливо для так называемого расширенного режима cmd.exe (установлена переменная CMDEXVERSION).

Чтобы детально ознакомиться с возможностями той или иной bat-команды, достаточно в командной строке окна консоли набрать **help <имя команды>** или **<имя команды> /?**. К примеру, **help set** или **echo /?**. Чтобы ознакомиться со списком поддерживаемых команд, нужно набрать просто **help**. Кроме того, вы можете ознакомиться со встроенным текстом справки Windows, набрав в окне консоли или в какой-либо командной оболочке "%windir%\help\ntcmds.chm", либо открыть Help Windows и найти тот же текст справки по ключевому тексту "Command-line reference".

СПЕЦСИМВОЛЫ

Прежде чем начать, следует рассказать о некоторых специальных символах, таких как:

| & () < > ^ " % ! ; , .

В подавляющем большинстве контекстов часть из них имеет приоритет над всеми остальными символами и командами. Поэтому надо не забывать их вовремя экранировать. К сожалению, в разных контекстах экранировать их следует по-разному. К примеру, внутри скрипта символы

| & () < > ^ "

можно экранировать с помощью символа (если не используются ограничивающие кавычки)

^

а внутри командной строки консоли *cmd.exe* - только с помощью кавычек.

Символы «;» и «,» представляют собой альтернативные разделители аргументов для командной строки скриптов и приложений, для их экранирования надо пользоваться только кавычками. Во всех остальных случаях использование и экранирование будет оговорено в соответствующем контексте.

КОМАНДА SET

СОВЕТ

Для правильного выполнения тестовых скриптов иногда стоит перезапускать консоль *cmd.exe* из-за того, что при создании переменных командой *set* значения созданных переменных остаются в памяти, и результат повторного выполнения скрипта может отличаться от первого выполнения.

Создание переменной

Рассмотрим небольшой пример:

Пример 1.1. Скрипт "1_1.bat"

```
@echo off
set A = B
set A2 = %A%
echo %A%
echo %A2%
```

Вывод скрипта "1_1.bat":

```
C:\rsdn>1_1.bat
ECHO is off.
ECHO is off.
C:\rsdn>_
```

Что здесь может быть неправильно, кроме того, что скрипт выводит не то, что нужно?

Что здесь произошло, можно понять после небольшого исправления.

Пример 1.2. Скрипт "1_2.bat"

```
@echo off
set A = B
set A2 = %A %
echo %A %
echo %A2 %
```

Вывод скрипта "1_2.bat":

```
C:\rsdn>1_2.bat
B
B
C:\rsdn>_
```

При использовании команды *set* слева и справа от равенства нельзя использовать пробелы, иначе командный процессор создаст переменную, название которой заканчивается пробелами, а значение начинается с пробелов и заканчивается последним символом (даже если это пробел). Это одна из причин, по которой при установке переменной не требуется использовать кавычки, т.к. здесь они не являются специальными ограничивающими символами (за исключением использования кавычек вокруг всего выражения *A=B*).

Следующий пример показывает бесполезность применения кавычек вокруг значения переменной для указания её границ.

Пример 1.3. Скрипт "1_3.bat"

```
@echo off
set A = B
rem Создаем переменную A2, содержащую два пробела в конце своего значения
set A2 = "%A %"
echo -%A %-
echo -%A2 %-
```

Вывод скрипта "1_3.bat":

```
C:\rsdn>1_3.bat
- B -
```

```
- " B" -  
C:\rsdn>_
```

Удаление переменной

Операция удаления переменной может вернуть ненулевой код возврата, если удаляется переменная, которая не существует. Чтобы каждый раз не писать проверку на существование переменной, можно воспользоваться скриптом *unset.bat*, который проверяет существование переменной перед её удалением. Скрипт приведён в приложении.

То же самое может быть справедливо и для установки переменной. К примеру, оператор *for* использует встроенные аргументы *%i*, *%j* и т.д (см. справку) для управления циклом. Их значения в общем случае могут отсутствовать. Поэтому, если попытаться для установки значения некоторой переменной скрипта использовать один из этих аргументов, можно получить ненулевой код возврата, так как присвоение отсутствующей переменной пустого (не заданного) значения будет интерпретировано cmd.exe как попытка удалить значение несуществующей переменной. В примере 1.3.1 демонстрируется эта ситуация.

Пример 1.3.1.

```
@echo off  
rem Некоторый цикл  
for /f "tokens=1,* delims=;" %%i in (myfile.txt) do (  
    rem Устанавливаем некоторую переменную.  
    rem Если %%j не существует, set установит ненулевой код возврата!  
    set MYVAR=%%j  
)
```

Происходит это из-за того, что значений аргументов *%j* и др. может не существовать в момент присвоения внутри *for*-блока. Вместо того, чтобы проверять их на существование, можно в данном случае – всегда устанавливать значение переменной, к примеру, в 0 перед присвоением ей значения *for*-аргумента.

Пример 1.3.2.

```
@echo off  
rem Некоторый цикл  
for /f "tokens=1,* delims=;" %%i in (myfile.txt) do (  
    rem Устанавливаем некоторую переменную.  
    set MYVAR=0  
    rem Если %%j не существует, set просто удалит созданную переменную.  
    set MYVAR=%%j  
)
```

Сброс ERRORLEVEL

Иногда при выполнении какого-то скрипта бывает необходимо сбросить код возврата. Допустим, запускается скрипт, который, в свою очередь, запускает последовательно два других. Так как первый скрипт может вернуть ненулевой код возврата, то перед выполнением второго нужно сбросить значение этого кода. Здесь важно заметить, что инструкция *set ERRORLEVEL=0* не сбрасывает переменную, а приводит только к созданию новой, которая перекрывает системную переменную *ERRORLEVEL*.

Пример 1.4. Скрипт "1_4.bat"

```
@echo off  
rem Инструкция, заведомо возвращающая ненулевой код возврата  
cd "z:\z\z\z\z\z"  
rem Проверяем это  
echo ERRORLEVEL=%ERRORLEVEL%  
rem Неправильная попытка сбросить код возврата  
set ERRORLEVEL=0  
rem Повторяем процедуру  
cd "z:\z\z\z\z\z"  
rem И проверяем, что код возврата не изменился  
echo ERRORLEVEL=%ERRORLEVEL%  
rem Удаляем перекрывающую переменную (фиктивный ERRORLEVEL)  
set ERRORLEVEL=  
rem И проверяем  
echo ERRORLEVEL=%ERRORLEVEL%
```

Вывод скрипта "1_4.bat":

```
C:\rsdn>1_4.bat  
The system cannot find the drive specified.  
ERRORLEVEL=1  
The system cannot find the drive specified.  
ERRORLEVEL=0  
ERRORLEVEL=1  
  
C:\rsdn>_
```

Чтобы правильно сбросить код возврата, можно воспользоваться внешней утилитой, которая вернёт заведомо нулевой код возврата, либо встроенной командой, которая предусматривает изменение кода возврата (к примеру, *cd*).

Пример 1.5. Скрипт "1_5.bat"

```
@echo off
```

```
rem Инструкция, заведомо возвращающая ненулевой код возврата
cd "z:\z\z\z\z\z"
rem Проверяем это
echo ERRORLEVEL=%ERRORLEVEL%
rem Инструкция, заведомо возвращающая нулевой код возврата
cd .
rem И проверяем
echo ERRORLEVEL=%ERRORLEVEL%
```

Вывод скрипта "1_5.bat":

```
C:\rsdn>1_5.bat
The system cannot find the drive specified.
ERRORLEVEL=1
ERRORLEVEL=0

C:\rsdn>_
```

ПРИМЕЧАНИЕ

ERRORLEVEL в 32-битной операционной системе Windows NT является 32-битным целым числом со знаком, т.е. представлена диапазоном значений: от -2147483648 до 2147483647. Таким образом, приложения могут возвращать отрицательный код возврата. Другие приложения могут не считать отрицательные значения за ошибку, к таким приложениям относится, к примеру, Microsoft Visual Studio 2005. Поэтому следует проверять код возврата на знак, если использующее скрипт внешнее приложение или скрипт также проверяет код возврата на знак.

Можно также воспользоваться скриптом, который сможет установить переданный ему код возврата (как все скрипты, внутри других скриптов его следует вызывать с применением оператора *call*).

Пример 1.6. Скрипт "errlvl.bat"

```
@echo off
if "%~1" == "" goto :EOF
exit /b %~1
```

Результат работы скрипта "errlvl.bat":

```
C:\rsdn>errlvl.bat 123

C:\rsdn>echo %ERRORLEVEL%
123

C:\rsdn>_
```

Условный блок

Из документации следует, что команды условного выполнения имеют в основном две формы. Первая форма – это условие-команда (then-переход), работающее по принципу "одно условие – одна команда". И вторая форма – условие-блок, работающее по принципу "одно условие – несколько (блок) команд". В документации это разделение особым образом не оговаривается, но оно есть. И более того, эти две формы могут работать несколько «по-разному». Это становится ясно по двум примерам.

Пример 2.1. Скрипт "2_1.bat"

```
@echo off
rem Условие всегда истинно
if "" == "" (
    rem Устанавливаем значение переменной
    set MYVALUE1=10
    rem Проверяем значение переменной
    echo MYVALUE1=%MYVALUE1%
)
rem Ещё раз проверяем значение переменной
echo MYVALUE1=%MYVALUE1%
```

Вывод скрипта "2_1.bat":

```
C:\rsdn>2_1.bat
MYVALUE1=
MYVALUE1=10

C:\rsdn>_
```

Пример 2.2. Скрипт "2_2.bat"

```
@echo off
rem Условие всегда истинно
if not "" == "" goto EXIT
    rem Устанавливаем значение переменной
    set MYVALUE2=10
    rem Проверяем значение переменной
    echo MYVALUE2=%MYVALUE2%
:EXIT
rem Ещё раз проверяем значение переменной
echo MYVALUE2=%MYVALUE2%
```

Вывод скрипта "2_2.bat":

```
C:\rsdn>2_2.bat
MYVALUE2=10
MYVALUE2=10

C:\rsdn>_
```

В первой примере, в отличие от второго, переменная получает своё значение только при выходе из блока условий.

Стоит заметить, что если блоки вложены друг в друга, то переменная получит значение только после выхода из самого внешнего блока условий выполняемого скрипта.

Условный блок и ERRORLEVEL

С предыдущей проблемой связана проблема задания значения встроенной переменной *ERRORLEVEL* внутри условия-блока. Фактически задание значения переменной *ERRORLEVEL* происходит за счёт выполнения командным процессором некоторой подпрограммы, в данном случае это будет ни что иное, как псевдокоманда "*set ERRORLEVEL=<код возврата>*" сразу же после выхода из вызванного приложения или скрипта. Это понятно по признакам работы обычной команды *set*. Таким образом, все проблемы, связанные с командой *set*, «автоматически наследуются» и подпрограммой командного процессора, устанавливающей значение переменной *ERRORLEVEL*.

Для проверки используется консольная утилита *errlvl.exe*, назначение которой – возвращать код ошибки, переданный ей в качестве аргумента. Эта утилита делает примерно то же самое, что и скрипт *errlvl.bat* (если вы захотите потом заменить её на этот скрипт, не забудьте добавить к его вызову оператор *call*, иначе результат будет отличаться от ожидаемого!).

Пример 3.1. Скрипт "3_1.bat"

```
@echo off
rem Сбрасываем предыдущий код возврата
cd .
rem Условие всегда истинно
if "" == "" (
    rem Устанавливаем значение переменной
    errlvl 10
    rem Проверяем значение переменной
    echo ERRORLEVEL=%ERRORLEVEL%
)
rem Ещё раз проверяем значение переменной
echo ERRORLEVEL=%ERRORLEVEL%
```

Вывод скрипта "3_1.bat":

```
C:\rsdn>3_1.bat
ERRORLEVEL=0
ERRORLEVEL=10

C:\rsdn>_
```

Пример 3.2. Скрипт "3_2.bat"

```
@echo off
rem Сбрасываем предыдущий код возврата
cd .
rem Условие всегда истинно
if not "" == "" goto EXIT
rem Устанавливаем значение переменной
errlvl 10
rem Проверяем значение переменной
echo ERRORLEVEL=%ERRORLEVEL%
:EXIT
rem Ещё раз проверяем значение переменной
echo ERRORLEVEL=%ERRORLEVEL%
```

Вывод скрипта "3_2.bat":

```
C:\rsdn>3_2.bat
ERRORLEVEL=10
ERRORLEVEL=10

C:\rsdn>_
```

Всё дело в скобках!

На самом деле проблема кроется не в самом операторе *if*, а в операторе "*скобки*" или, другими словами, в блоке кода, образуемом этими скобками (далее для краткости будет использоваться термин «блок»). Можно переписать вышеприведённые примеры без оператора *if*.

Пример 3.3. Скрипт "3_3.bat"

```
@echo off
```

```

rem Оператор "скобки"
(
    rem Устанавливаем значения переменных
    set MYVALUE1=10
    errlvl 10
    rem Проверяем значения переменных
    echo MYVALUE1=%MYVALUE1%
    echo ERRORLEVEL=%ERRORLEVEL%
)
rem Ещё раз проверяем значения переменных
echo MYVALUE1=%MYVALUE1%
echo ERRORLEVEL=%ERRORLEVEL%

```

Вывод скрипта "3_3.bat":

```

C:\rsdn>3_3.bat
MYVALUE1=
ERRORLEVEL=0
MYVALUE1=10
ERRORLEVEL=10

C:\rsdn>_

```

Таким образом, следует с осторожностью использовать встроенные команды, включающие использование скобок. Необдуманное применение такого оператора может привести к некорректному выполнению скрипта.

КОМАНДНЫЙ ПРОЦЕССОР CMD.EXE

Set и блок

Рассмотрим вызов скрипта через вызов cmd.exe с параметром /c (этот параметр приказывает выполнить строку, переданную в качестве аргумента, и завершить выполнение). Набор параметров *cmd.exe* и их назначение можно узнать из справки (источники указаны в начале статьи).

Пример 4.1. Скрипт "4_1.bat"

```

@echo off
rem Сбрасываем предыдущий код возврата
cd .
rem Оператор "скобки"
(
    rem Вызов скрипта
    cmd.exe /c 4_2.bat
    rem Проверка переменных
    echo 4_1. MYVALUE1=%MYVALUE1%
    echo 4_1. ERRORLEVEL=%ERRORLEVEL%
)
rem Ещё раз проверяем значения переменных
echo 4_1. MYVALUE1=%MYVALUE1%
echo 4_1. ERRORLEVEL=%ERRORLEVEL%

```

Пример 4.2. Скрипт "4_2.bat"

```

@echo off
rem Установка переменной
set MYVALUE1=10
rem Проверка переменной
echo 4_2. MYVALUE1=%MYVALUE1%
rem Изменения кода ошибки
errlvl 9
rem Проверка кода ошибки
echo 4_2. ERRORLEVEL=%ERRORLEVEL%

rem Выходим с не нулевым кодом ошибки
exit /b 10

```

Вывод скрипта "4_1.bat":

```

C:\rsdn>4_1.bat
4_2. MYVALUE1=10
4_2. ERRORLEVEL=9
4_1. MYVALUE1=
4_1. ERRORLEVEL=0
4_1. MYVALUE1=
4_1. ERRORLEVEL=0

C:\rsdn>_

```

В отличие от встроенных команд, внешние приложения не в состоянии в момент своего завершения «сохранять» значения переменных среды (в этом, собственно, и заключается одно из отличий скриптов командной оболочки от скриптов наподобие *Windows Scripting Host*, требующих для своего выполнения создания нового «процесса»), которые они изменили или создали в процессе своей работы, (за исключением встроенных переменных, таких как *ERRORLEVEL*, которые изменяются самим командным процессором). В этом отношении сам командный процессор *cmd.exe* является таким «внешним» приложением, т.к. переданный в качестве аргумента скрипт запускается в отдельном процессе *cmd.exe*, поэтому не удивительно, что значение переменной *MYVALUE1* после выхода из «вложенного» *cmd.exe* не сохранилось. Но, тем не менее, как видно из результата работы скрипта, состояние *ERRORLEVEL* не изменилось и после выхода из условного блока. А из предыдущего пункта было понятно, что это должно было произойти. Это ещё одна из проблем, свойственная скорее самому командному процессору – *cmd.exe*. То, что *cmd* возвращает код возврата вызванного приложения, можно продемонстрировать следующим

примером.

Пример 4.3. Скрипт "4_3.bat"

```
@echo off
rem Сбрасываем предыдущий код возврата
cd .
rem Оператор "скобки"
(
    rem Вызов скрипта
    cmd.exe /c errlvl 10
    rem Проверка переменной
    echo ERRORLEVEL=%ERRORLEVEL%
)
rem Ещё раз проверяем значения переменных
echo ERRORLEVEL=%ERRORLEVEL%
```

Вывод скрипта "4_3.bat":

```
C:\rsdn>4_3.bat
ERRORLEVEL=0
ERRORLEVEL=10

C:\rsdn>_
```

Можно устранить этот «эффект» игнорирования кода возврата из скрипта внутри блока простым добавлением *call* перед названием скрипта.

Пример 5.1. Скрипт "5_1.bat"

```
@echo off
rem Сбрасываем предыдущий код возврата
cd .
rem Оператор "скобки"
(
    rem Вызов скрипта
    cmd.exe /c call 4_2.bat
    rem Проверка переменных
    echo 5_1. MYVALUE1=%MYVALUE1%
    echo 5_1. ERRORLEVEL=%ERRORLEVEL%
)
rem Ещё раз проверяем значения переменных
echo 5_1. MYVALUE1=%MYVALUE1%
echo 5_1. ERRORLEVEL=%ERRORLEVEL%
```

Вывод скрипта "5_1.bat":

```
C:\rsdn>5_1.bat
4_2. MYVALUE1=10
4_2. ERRORLEVEL=9
5_1. MYVALUE1=
5_1. ERRORLEVEL=0
5_1. MYVALUE1=
5_1. ERRORLEVEL=10

C:\rsdn>_
```

Или просто не пользоваться "*cmd.exe /c*".

Пример 5.2. Скрипт "5_2.bat"

```
@echo off
rem Сбрасываем предыдущий код возврата
cd .
rem Оператор "скобки"
(
    rem Вызов скрипта
    call 4_2.bat
    rem Проверка переменных
    echo 5_2. MYVALUE1=%MYVALUE1%
    echo 5_2. ERRORLEVEL=%ERRORLEVEL%
)
rem Ещё раз проверяем значения переменных
echo 5_2. MYVALUE1=%MYVALUE1%
echo 5_2. ERRORLEVEL=%ERRORLEVEL%
```

Вывод скрипта "5_2.bat":

```
C:\rsdn>5_2.bat
4_2. MYVALUE1=10
4_2. ERRORLEVEL=9
5_2. MYVALUE1=
5_2. ERRORLEVEL=0
5_2. MYVALUE1=10
5_2. ERRORLEVEL=10

C:\rsdn>_
```

К тому же, во втором случае после выхода из самого внешнего блока мы также добьёмся «сохранения» переменных, созданных (или изменённых) во «внешнем» скрипте или во внутреннем блоке.

Таким образом, мы решили две проблемы, а именно, сохранили значение переменной *MYVALUE1* после выхода из блока и изменили значение *ERRORLEVEL* также после выхода из блока. Чтобы решить последнюю

проблему с изменением переменных только после выхода из самого внешнего блока, достаточно будет удалить все блоки, а если блоки использовались совместно с оператором `if`, то заменить их соответствующими конструкциями `goto` + метка (что, несомненно, ухудшит читаемость кода, но избавит от неявного поведения – прим.ред.).

Пример 5.3. Скрипт “5_3.bat”

```
@echo off
rem Сбрасываем предыдущий код возврата
cd .

    rem Вызов скрипта
    call 4_2.bat
    rem Проверка переменных
    echo 5_3. MYVALUE1=%MYVALUE1%
    echo 5_3. ERRORLEVEL=%ERRORLEVEL%

rem Ещё раз проверяем значения переменных
echo 5_3. MYVALUE1=%MYVALUE1%
echo 5_3. ERRORLEVEL=%ERRORLEVEL%
```

Вывод скрипта “5_3.bat”:

```
C:\rsdn>5_3.bat
4_2. MYVALUE1=10
4_2. ERRORLEVEL=9
5_3. MYVALUE1=10
5_3. ERRORLEVEL=10
5_3. MYVALUE1=10
5_3. ERRORLEVEL=10
C:\rsdn>_
```

Exit /b %ERRORLEVEL% и блок

Можно сначала подумать, что следующий пример, вопреки использованию `call`, всё равно не сохраняет код возврата.

Пример 6.1. Скрипт “6_1.bat”

```
@echo off
rem Сбрасываем предыдущий код возврата
cd .
if "%OUTPUT_REDIRECTED%" == "" (
    set OUTPUT_REDIRECTED=1
    rem Перенаправляем вывод ошибок
    cmd.exe /c call %0 %* 2> ".\MyErrLog.log"
    set OUTPUT_REDIRECTED=
    rem Проверяем ERRORLEVEL
    echo ERRORLEVEL=%ERRORLEVEL%
    rem Выходим из скрипта с текущим кодом возврата
    exit /b %ERRORLEVEL%
)

rem Проверяем ERRORLEVEL
echo ERRORLEVEL=%ERRORLEVEL%

rem Выходим с кодом ошибки 10
exit /b 10
```

Вывод скрипта “6_1.bat”:

```
C:\rsdn>6_1.bat
ERRORLEVEL=0
ERRORLEVEL=0

C:\rsdn>echo %ERRORLEVEL%
0
C:\rsdn>_
```

Но это происходит из-за того, что `ERRORLEVEL` на протяжении всего выполнения равен 0, и принимает новое значение только после выхода из самого внешнего блока. Причем это значение не равно 10, так как операция `exit /b %ERRORLEVEL%` сбрасывает его в 0. Достаточно заменить эту операцию эквивалентной, но приводящей к правильному результату – `goto :EOF`.

Пример 6.2. Скрипт “6_2.bat”

```
@echo off
rem Сбрасываем предыдущий код возврата
cd .
if "%OUTPUT_REDIRECTED%" == "" (
    set OUTPUT_REDIRECTED=1
    rem Перенаправляем вывод ошибок
    cmd.exe /c call %0 %* 2> ".\MyErrLog.log"
    set OUTPUT_REDIRECTED=
    rem Проверяем ERRORLEVEL
    echo ERRORLEVEL=%ERRORLEVEL%
    rem Выходим из скрипта с текущим кодом возврата
    goto :EOF
)

rem Проверяем ERRORLEVEL
echo ERRORLEVEL=%ERRORLEVEL%

rem Выходим с кодом ошибки 10
exit /b 10
```

Вывод скрипта "6_2.bat":

```
C:\rsdn>6_2.bat
ERRORLEVEL=0
ERRORLEVEL=0

C:\rsdn>echo %ERRORLEVEL%
10

C:\rsdn>_
```

КОМАНДА GOTO

Условный блок

При попытке использовать эту команду для перехода в условный блок, выполнение не будет прервано в конце условного блока, и если далее будут идти команды *else*, то они будут игнорироваться.

Пример 7.1. Скрипт "7_1.bat"

```
@echo off
if "%~1" == "0" (
:DEFAULT
  echo 0
) else if "%~1" == "1" (
  echo 1
) else if "%~1" == "2" (
  echo 2
  if "%~1" == "3" (
    echo 3
  ) else (
    echo 4
  )
) else (
  goto DEFAULT
)
```

При вызове данного скрипта без параметров, его код будет выводить числа 0, 1, 2, 4 в цикле. Для решения проблемы следует добавить *goto* в конец первого условного блока.

Пример 7.2. Скрипт "7_2.bat"

```
@echo off
if "%~1" == "0" (
:DEFAULT
  echo 0
  goto BREAK10
) else if "%~1" == "1" (
  echo 1
) else if "%~1" == "2" (
  echo 2
  if "%~1" == "3" (
    echo 3
  ) else (
    echo 4
  )
) else (
  goto DEFAULT
)
:BREAK10
```

Вывод скрипта "7_2.bat":

```
C:\rsdn>7_2.bat
0

C:\rsdn>_
```

КОМАНДА CALL

Команда call и блок

Внутри справки (внешней справки – *ntcmds.chm*, см. начало статьи) по командному процессору на счёт команды *call* можно прочитать интересное заявление, что якобы она не останавливает выполнение скрипта родителя, т.е. продолжает выполнение запускаемого скрипта параллельно (выделено курсивом):

Call - Calls one batch program from another *without stopping the parent batch program*.

В общем случае это не так, и не имеет ничего общего с действительным поведением по умолчанию. Скрипт-родитель ждёт завершения вызванного скрипта.

ПРИМЕЧАНИЕ

Следует сделать уточнение, если вызывается не скрипт, а внешняя утилита, то сценарий

выполнения зависит от этой утилиты. Если она была собрана (создана), как консольная, то скрипт ждёт завершения её выполнения. Если как GUI, то не ждёт.

Но суть не в этом, вообще команда *call* обладает некоторым «магическим» действием. То, что раньше не работало, с её использованием начинает работать.

Пример 8.1. Скрипт "8_1.bat"

```
@echo off
(
  (
    7_2.bat
    echo 1
    goto BREAK
  )
  echo 2
  goto BREAK
)
echo 3
:BREAK
echo 4
```

Вывод скрипта "8_1.bat"

```
C:\rsdn>8_1.bat
0
1
2
C:\rsdn>_
```

Добавим *call*.

Пример 8.2. Скрипт "8_2.bat"

```
@echo off
(
  (
    call 7_2.bat
    echo 1
    goto BREAK
  )
  echo 2
  goto BREAK
)
echo 3
:BREAK
echo 4
```

Вывод скрипта "8_2.bat":

```
C:\rsdn>8_2.bat
0
1
4
C:\rsdn>_
```

Вообще, такое поведение свойственно блоку, поэтому в данном случае использование *call*-вызова лишь устраняет «дефекты работы» блока.

Двойное раскрытие переменной

Данный пункт не относится к проблемным местам в написании скриптов, а скорее является так называемым хинтом.

Если в нашем распоряжении имеется переменная, которая уже имеет какое-то ранее присвоенное значение и имя этой переменной присутствует в значении другой переменной, чтобы раскрыть значение второй переменной так, чтобы значение первой переменной подставилось вместо ее "%" - имени в значение второй переменной, можно воспользоваться синтаксисом переустановки переменной через вызов *call*.

Пример 9.1. Скрипт "9_1.bat"

```
@echo off
rem Первая переменная
set MYINSTALL_DIR=C:\subfolder
rem Вторая переменная
set МУРАТН=%MYINSTALL_DIR%\subfolder
rem Двойное раскрытие переменной
call set МУРАТН=%МУРАТН%
rem Проверка первой переменной
echo МУРАТН=%МУРАТН%
```

Вывод скрипта "9_1.bat":

```
C:\rsdn>9_1.bat
МУРАТН=C:\subfolder\subfolder
```

```
C:\rsdn>_
```

В этом случае всё, что записано справа от вызова *call* до первого неэкранированного спецсимвола раскрывается два раза. Первый раз – перед выполнением любой команды внутри скрипта, то есть перед интерпретацией строчки текста в скрипте (при этом раскрывается вся строчка целиком), второй раз – перед самим вызовом *call* (раскрывается часть строчки от *call*-команды до первого неэкранированного спецсимвола, например, перенаправления ввода-вывода). Для проверки достаточно создать скрипт с именем переменной и вызвать его рекурсивно изнутри.

Пример 9.2. Скрипт "%9_2%.bat"

```
@echo off
echo Hello, we are in
rem Рекурсивный вызов, %~0 раскроется два раза
call %~0
```

Вывод скрипта "%9_2%.bat":

```
C:\rsdn>%9_2%
Hello, we are in
'_2' is not recognized as an internal or external command,
operable program or batch file.
C:\rsdn>_
```

ПРИМЕЧАНИЕ

Кстати, в скриптах в качестве имён переменных можно использовать числа и некоторые спецсимволы, а неинициализированные переменные, введённые непосредственно в командную строку консоли cmd.exe, не удаляются, как это происходит в скриптах, т.к. это могут быть обращения к именам файлов. Инициализированные переменные раскрываются всегда.

Поскольку в нашем примере переменная %9_2% не была инициализирована, то, в отличие от скрипта, где бы она была замещена «пустым местом» (или, попросту говоря, удалена), здесь она не была замещена. Что касается текста ошибки, то _2 – это всё, что осталось после раскрытия %9_2% внутри скрипта, где %9 – это встроенная неинициализированная переменная, а % – просто проигнорированный спецсимвол.

Устранить игнорирование двойного раскрытия несложно, достаточно заменить %~0 на %%~0.

Пример 9.3. Скрипт "%9_3%.bat"

```
@echo off
echo Hello, we are in
rem Рекурсивный вызов, %%~0 раскроется два раза
call %%~0
```

Вывод скрипта "%9_3%.bat":

```
C:\rsdn>%9_3%
Hello, we are in
Hello, we are in
...
***** B A T C H   R E C U R S I O N   exceeds STACK limits *****
Recursion Count=1240, Stack Usage=90 percent
*****           B A T C H   PROCESSING IS   A B O R T E D           *****
C:\rsdn>_
```

Получили рекурсивный вызов, как и требовалось. Единственное, что можно было бы здесь рассмотреть ещё, – это раскрытие инициализированных переменных непосредственно в окне консоли.

Вывод скрипта "%9_3%.bat":

```
C:\rsdn>set 9_3=10
C:\rsdn>%9_3%
'10' is not recognized as an internal or external command,
operable program or batch file.
C:\rsdn>_
```

Здесь скрипт даже не начал выполняться. Способ решения такой же, как и внутри встроенной команды *echo*, «экранировать выполнение» спецсимвола %.

Вывод скрипта "%9_3%.bat":

```
C:\rsdn>set 9_3=10
C:\rsdn>^%9_3^%
Hello, we are in
Hello, we are in
...
***** B A T C H   R E C U R S I O N   exceeds STACK limits *****
Recursion Count=1240, Stack Usage=90 percent
```

```
*****      B A T C H      P R O C E S S I N G   I S   A B O R T E D      *****  
C:\rsdn>_
```

ПРИМЕЧАНИЕ

Способы экранирования символа % непосредственно внутри скрипта и при вводе прямым текстом в окне консоли несколько отличаются. В скрипте для экранирования надо использовать двойной знак % - %, а в окне консоли - ^%.

Двойное раскрытие переменной и блок

Двойное раскрытие переменной обладает ещё одной особенностью. С его помощью можно обратиться к значению переменной, не ожидая выхода из самого внешнего блока.

Пример 10.1. Скрипт "10_1.bat"

```
@echo off  
rem Устанавливаем код возврата  
errlvl 10  
rem Сбрасываем переменную  
set MYVALUE=0  
set MYVALUE=  
rem Оператор " скобки"  
(  
    rem Устанавливаем значения переменных  
    set MYVALUE=123  
    rem Устанавливаем код возврата  
    errlvl 20  
    rem Проверяем значения переменных  
    call echo 1. MYVALUE=%MYVALUE%  
    call echo 1. ERRORLEVEL=%ERRORLEVEL%  
)  
rem Ещё раз проверяем значения переменных  
echo 2. MYVALUE=%MYVALUE%  
echo 2. ERRORLEVEL=%ERRORLEVEL%
```

Вывод скрипта "10_1.bat":

```
C:\rsdn>10_1  
1. MYVALUE=123  
1. ERRORLEVEL=0  
2. MYVALUE=123  
2. ERRORLEVEL=0  
C:\rsdn>_
```

ПРИМЕЧАНИЕ

ERRORLEVEL равен нулю, так как его сбрасывает вызов операций с помощью команды call.

Его можно также использовать для вызова скриптов и блоков кода внутри блока, и при этом все переменные на момент вызова будут раскрыты, в том числе и те, которые были установлены до вызова call внутри блока.

Пример 10.2. Скрипт "10_2.bat"

```
@echo off  
rem Устанавливаем код возврата  
errlvl 10  
rem Устанавливаем переменную и проверяем код возврата  
set MYVALUE1=1  
echo 1. ERRORLEVEL=%ERRORLEVEL%  
rem Оператор " скобки"  
(  
    rem Устанавливаем код возврата  
    errlvl 20  
    rem Переустанавливаем переменную и проверяем код возврата  
    call set MYVALUE1=%MYVALUE1%+2  
    call echo 2. ERRORLEVEL=%ERRORLEVEL%  
    rem Оператор " скобки"  
    (  
        call :TEST_ROUTINE  
    )  
)  
goto :EOF  
:  
:TEST_ROUTINE  
rem Проверяем переменную и код возврата  
echo 3. MYVALUE1=%MYVALUE1%  
echo 3. ERRORLEVEL=%ERRORLEVEL%  
goto :EOF
```

Вывод скрипта "10_2.bat":

```
C:\rsdn>10_2.bat  
1. ERRORLEVEL=10  
2. ERRORLEVEL=0  
3. MYVALUE1=1+2  
3. ERRORLEVEL=0
```

```
C:\rsdn>_
```

Первый недостаток – оператор *call* сбрасывает код возврата, даже если выполняет команду не меняющую его, в данном случае это команды *echo* и *set* (с установкой не пустого значения). Поэтому, это приводит к тому, что значение переменной *ERRORLEVEL* сбрасывается после оператора *call*. Для обхода этой проблемы можно воспользоваться отдельным скриптом раскрытия переменных, который восстанавливает код возврата, который был перед его вызовом.

Пример 10.3. Скрипт “expandvar.bat”

```
@echo off
rem Скрипт раскрывает строку %2 и сохраняет результат в переменной,
rem имя которой передается через параметр %1.
if not "%~1" == "" (
    if not "%~2" == "" (
        rem Раскрываем переменную %2
        call set "%~1=%~2"
        rem Выходим с предыдущим кодом возврата
        exit /b %ERRORLEVEL%
    )
)
rem Выходим с текущим кодом возврата
```

Результат работы скрипта “expandvar.bat”:

```
C:\rsdn>set B=10
C:\rsdn>set A=%B%
C:\rsdn>expandvar EXPAND_VALUE %A%
C:\rsdn>echo EXPAND_VALUE=%EXPAND_VALUE%
EXPAND_VALUE=10
C:\rsdn>_
```

Второй недостаток – оператор *call* неприменим к некоторым встроенным командам, к примеру, к команде *if*. Поэтому не получится воспользоваться двойным раскрытием переменных внутри этой встроенной команды.

Двойное раскрытие переменной и оператор “!”

Существует стандартный способ двойного раскрытия переменной, который изначально выключен. Чтобы активировать его, независимо от установок по умолчанию, нужно воспользоваться встроенной командой *setlocal*.

Пример 11.1. Скрипт “11_1.bat”

```
@echo off
set MYVVALUE1=123
set MYVVALUE2=MYVVALUE1
rem Активируем двойное раскрытие переменных через оператор !
setlocal ENABLEDELAYEDEXPANSION
echo !%MYVVALUE2%!
rem Восстанавливаем настройки по-умолчанию
endlocal
echo !%MYVVALUE2%!
```

Вывод скрипта “11_1.bat”:

```
C:\rsdn>11_1
123
!MYVVALUE1!
C:\rsdn>_
```

Преимущество данного способа заключается в обходе проблемы с раскрытием переменных только после выхода из самого внешнего оператора скобки.

Пример 11.2. Скрипт “11_2.bat”

```
@echo off
rem Устанавливаем переменную
set MYVALUE=10
(
    set MYVALUE=20
    setlocal ENABLEDELAYEDEXPANSION
    echo 1. MYVALUE=!MYVALUE!
    endlocal
    echo 1. MYVALUE=%MYVALUE%
)
setlocal ENABLEDELAYEDEXPANSION
echo 2. MYVALUE=!MYVALUE!
endlocal
echo 2. MYVALUE=%MYVALUE%
```

Вывод скрипта “11_2.bat”:

```
C:\rsdn>11_2
```

```
1. MYVALUE=20
1. MYVALUE=10
2. MYVALUE=20
2. MYVALUE=20
```

```
C:\rsdn>_
```

К сожалению, и данный способ содержит недостаток, а именно, “двойной эффект”, и, кроме “включения” оператора “!”, сохраняет все переменные в стек (см. описание команд `setlocal/endlocal` далее).

КОМАНДЫ SETLOCAL/ENDLOCAL

Оператор “!” или оператор “%”?

Фактически, оператор “!” — ни что иное, как оператор “%”, но с меньшим приоритетом.

В следующем примере мы создаём переменную, имя которой состоит из одного символа — “!”. Так как приоритет символа “%” выше, чем “!”, то “%-переменные раскроются до “!”-переменных.

Пример 12.1. Скрипт “12_1.bat”

```
@echo off
rem Создаём переменные
set MYVVALUE1=123
set MYVVALUE2=MYVVALUE1
rem Создаём переменную “!”
set !=!
rem Активируем двойное раскрытие переменных через оператор !
setlocal ENABLEDELAYEDEXPANSION
echo 1. !%MYVVALUE2%!
echo 2. %!MYVVALUE2!%
echo 3. %!%MYVVALUE2%!%
rem Восстанавливаем настройки по умолчанию
endlocal
```

Вывод скрипта “12_1.bat”:

```
C:\rsdn>12_1
1. 123
2.
3. MYVVALUE1
C:\rsdn>_
```

В дополнение, оператор “!” не экранируем ни самим собой, ни другими конструкциями спецсимволов. В данном случае рекомендуется в качестве замещения экранированию использовать переменную со значением “!”.

Далее показывается, как можно выводить символ “!” при включённом раскрытии “!”-переменных.

Пример 12.2. Скрипт “12_2.bat”

```
@echo off
rem Создаём переменную “!”
set EXCLAMATION=!
rem Активируем двойное раскрытие переменных через оператор !
setlocal ENABLEDELAYEDEXPANSION
echo !EXCLAMATION!
printargs !EXCLAMATION!
echo " !EXCLAMATION!"
printargs " !EXCLAMATION!"
rem Восстанавливаем настройки по-умолчанию
endlocal
```

Вывод скрипта “12_2.bat”:

```
C:\rsdn>12_2
!
"! "
"! "
"! "
C:\rsdn>_
```

Лишние кавычки здесь — детали вызова `printargs.exe` с параметрами (см. “Вызов с параметрами”).

Восстанавливаем окружение после работы скрипта

Что, если требуется после работы скрипта восстановить окружение в том виде, который был до его вызова? Очевидно, это можно сделать вручную, но иногда достаточно воспользоваться для этого командами `setlocal/endlocal`. Кроме того, вам может понадобиться удалить переменные, созданные чужим скриптом (имена которых неизвестны).

Пример 13.1. Скрипт “13_1.bat”

```
@echo off
```

```
rem Этот сторонний скрипт что-то делает...
set WINDIR=E:\Windows
```

Пример 13.2. Скрипт "13_2.bat"

```
@echo off
rem Проверяем переменную
echo WINDIR=%WINDIR%
rem Сохраняем окружение
setlocal
rem Изменяем переменную
set WINDIR=D:\Windows
rem Сохраняем окружение
setlocal
rem Вызываем "не наш" скрипт
call "%~dp013_1.bat"
rem Проверяем переменную
echo WINDIR=%WINDIR%
rem Восстанавливаем окружение
endlocal
rem Проверяем переменную
echo WINDIR=%WINDIR%
rem Восстанавливаем окружение
endlocal
rem Проверяем переменную
echo WINDIR=%WINDIR%
```

Результаты работы скрипта "13_2.bat":

```
C:\rsdn>13_2
WINDIR=C:\Windows
WINDIR=E:\Windows
WINDIR=D:\Windows
WINDIR=C:\Windows

C:\rsdn>_
```

ПРИМЕЧАНИЕ

К сожалению, *setlocal/endlocal* (с параметрами и без) работают только внутри скрипта (*endlocal* даже не нужно вызывать явно, после выполнения скрипта, он будет вызван столько же раз, сколько был вызван *setlocal* в этом же скрипте) и не работают вне его (в окне консоли).

ВЫЗОВ С ПАРАМЕТРАМИ

Вызов внешнего приложения

В общем случае вызов какого-либо приложения с определёнными аргументами не представляет особого интереса, за исключением того, что если параметры берутся в кавычки, то сами кавычки не входят в значения аргументов. В данном примере консольная утилита *printargs.exe* выводит свои аргументы, взяв при этом каждый из них в кавычки.

Результат выполнения утилиты *printargs.exe* с аргументами:

```
C:\rsdn>printargs.exe "1 1" 2 2 " 3 " 4
"1 1" "2" "2" " 3 " "4"

C:\rsdn>_
```

Проблема обратного слеша (\)

Существует одна проблема, которая не позволяет использовать завершающий обратный слеш (т.е. символ "\") внутри параметров скриптов и приложений, вызываемых с использованием *cmd.exe*. В этом случае *cmd.exe* работает так, что завершающий обратный слеш приводит к «экранированию» кавычки, расположенной сразу за ним.

Результат выполнения утилиты *printargs.exe* с аргументами:

```
C:\rsdn>printargs.exe "1 1\" 2 2 " 3 " 4
"1 1" 2 2 " "3" " 4"

C:\rsdn>_
```

Чтобы избавиться от этого «эффекта», достаточно добавить ещё один обратный слеш.

Результат выполнения утилиты *printargs.exe* с аргументами:

```
C:\rsdn>printargs.exe "1 1\\" 2 2 " 3 " 4
"1 1\" "2" "2" " 3 " "4"

C:\rsdn>_
```


ПРЕДУПРЕЖДЕНИЕ

Будьте осторожны при использовании встроенных переменных внутри таких аргументов. К примеру, такие переменные как `%~dp0` и `%~p0` всегда содержат завершающий обратный слеш, и будут приводить к тому же «эффекту».

Также можно наблюдать отсутствие «эффекта» в случае со встроенной командой `echo`.

Результат выполнения встроенной команды `echo` с аргументами:

```
C:\rsdn>echo "1 1\" 2 2 " 3 " 4
"1 1\" 2 2 " 3 " 4
C:\rsdn>_
```

Call-вызов скрипта и переменные %0-%9

При вызове внешней утилиты с параметрами, например, `printargs.exe`, аргументы передаются в неё без ограничивающих кавычек. Другое дело – вызов скрипта с аргументами, здесь всё иначе. При заключении аргументов в кавычки, кавычки передаются вместе со значениями и так сохраняются во встроенные переменные `%0-%9`.

Пример 14.1. Скрипт "14_1.bat"

```
@echo off
rem Использование внешней утилиты
printargs.exe %1 + %2 + %3 + %4 + %5
rem Использование встроенной команды
echo %1 + %2 + %3 + %4 + %5
```

Вывод скрипта "14_1.bat":

```
C:\rsdn>14_1.bat "1 1" 2 2 " 3 " 4
"1 1" "+" "2" "+" "2" "+" " 3 " "+" "4"
"1 1" + 2 + 2 + " 3 " + 4
C:\rsdn>_
```

ПРИМЕЧАНИЕ

Символы «+» добавлены для более ясной картины происходящего.

Для устранения кавычек, если они имеются, нужно пользоваться "~-"-модификацией этих переменных (полное описание - `help call` в окне консоли).

Пример 14.2. Скрипт "14_2.bat"

```
@echo off
rem Использование внешней утилиты
printargs.exe %~1 + %~2 + %~3 + %~4 + %~5
rem Использование встроенной команды
echo %~1 + %~2 + %~3 + %~4 + %~5
```

Вывод скрипта "14_2.bat":

```
C:\rsdn>14_2.bat "1 1" 2 2 " 3 " 4
"1" "1" "+" "2" "+" "2" "+" "3" "+" "4"
1 1 + 2 + 2 + 3 + 4
C:\rsdn>_
```

ПРИМЕЧАНИЕ

При этом пробелы справа и слева в значении переменных никуда не пропадают.

Команда if и переменные %0-%9

Порой можно встретить интересные записи условий с использованием переменных-аргументов. Вот некоторые из них.

Пример 15.1. Пример записи условной команды

```
...
rem Использование внешней утилиты
if %1. == .SomeValue. goto PROCEED1
...
```

Или пример из скрипта `vcvarsall.bat` в Microsoft Visual Studio 2005:

```
@echo off
if "%1" == "" goto x86
if not "%2" == "" goto usage

if /i %1 == x86      goto x86
if /i %1 == amd64    goto amd64
```

К сожалению, такие формы записи приводят к невозможности передачи параметров с кавычками. Сложности начинаются, когда значения параметров могут содержать пробелы.

```
@echo off
if %1. == .My Value. echo Test1 passed
if %1 == My Value echo Test2 passed
if "%1" == "My Value" echo Test3 passed
```

Вывод скрипта `"15_3.bat"`:

```
C:\rsdn>15_1.bat "My Value"
Value"" was unexpected at this time.

C:\rsdn>_
```

Сообщение об ошибке относится к третьей строчке с `if`. Попробуем разобраться, что произошло, а точнее, просто подставить `"My Value"` (вместе с кавычками под `%1`).

```
@echo off
if ."My Value". == .My Value. echo Test1 passed
if "My Value" == My Value echo Test2 passed
if ""My Value"" == "My Value" echo Test3 passed
```

Вывод скрипта `"15_4.bat"`:

```
C:\rsdn>15_4.bat
Value"" was unexpected at this time.

C:\rsdn>_
```

Как видно, ничего не изменилось, но стало понятно, какую строчку препроцессор не пропускает. Этого можно избежать, если всегда внутри условий использовать кавычки и `"~"`-модификации встроенных переменных `%0`-`%9`.

```
@echo off
if "%~1" == "MyValue" echo Test1 passed
if "%~2" == "My Value" echo Test2 passed
```

Вывод скрипта `"15_5.bat"`:

```
C:\rsdn>15_5.bat MyValue "My Value"
Test1 passed
Test2 passed

C:\rsdn>_
```

Исключение составляет сравнение чисел вместо строк. В этом случае использование кавычек может привести к неверному результату.

```
@echo off
rem if 10 >= 9
if "10" GEQ "9" (
    echo 10 ">=" 9
) else (
    echo 10 "<" 9!
)
```

Вывод скрипта `"15_6.bat"`:

```
C:\rsdn>15_6.bat
10 "<" 9!

C:\rsdn>_
```

```
@echo off
rem if 10 >= 9
if 10 GEQ 9 (
    echo 10 ">=" 9
) else (
    echo 10 "<" 9!
)
```

Вывод скрипта "15_7.bat":

```
C:\rsdn>15_7.bat
10 ">=" 9

C:\rsdn>_
```

Команда echo с точкой

Есть одна неприятная особенность, связанная с командой echo. При попытке вывести текст, начинающийся с "/", echo вместо предлагаемого текста выводит текст своей справки.

Пример 16.1. Скрипт "16_1.bat"

```
@echo off
echo Usage %~n0 [option]
echo    /? Show help
echo    etc
```

Вывод скрипта "16_1.bat":

```
C:\rsdn>16_1.bat
Usage 16_1 [option]
Displays messages, or turns command-echoing on or off.

    ECHO [ON | OFF]
    ECHO [message]

Type ECHO without parameters to display the current echo setting.
etc

C:\rsdn>_
```

Решение – использовать «echo с точкой».

Пример 16.2. Скрипт "16_2.bat"

```
@echo off
echo.Usage %~n0 [option]
echo.    /? Show help
echo.    etc
```

Вывод скрипта "16_2.bat":

```
C:\rsdn>16_2.bat
Usage 16_2 [option]
    /? Show help
    etc

C:\rsdn>_
```

Кроме того, такое использование удобно в случае вывода аргументов скрипта, т.к. не требует их проверки на существование.

Пример 16.3. Скрипт "16_3.bat"

```
@echo off
echo.%~1
echo.%~2
```

Вывод скрипта "16_3.bat":

```
C:\rsdn>16_3.bat "" arg2
arg2

C:\rsdn>_
```

И ЕЩЁ НЕМНОГО О ПЕРЕМЕННЫХ

Спецсимволы в переменных

Иногда значения переменных могут содержать символы форматирования:

```
| & ( ) < > ^
```

Поэтому некоторые операции над такими значениями могут перестать работать.

Пример 17.1.1. Скрипт "17_1.bat"

```
@echo off
set /P MYVAR=< MyVar1.var
echo.%MYVAR%
printargs %MYVAR%
```

Пример 17.1.2. Файл "MyVar1.var"

```
(1 < 2) & (3 > 4) ^ 1
```

Вывод скрипта "17_1.bat":

```
C:\rsdn>17_1.bat
1 was unexpected at this time.

C:\rsdn>_
```

Общий способ обхода данной проблемы заключается в использовании ограничивающих кавычек вокруг выражения, содержащего специальные символы форматирования.

Пример 17.2. Скрипт "17_2.bat"

```
@echo off
set /P MYVAR=< MyVar1.var
echo "%MYVAR%"
printargs "%MYVAR%"
```

Вывод скрипта "17_2.bat":

```
C:\rsdn>17_2.bat
"(1 < 2) & (3 > 4) ^ 1"
"(1 < 2) & (3 > 4) ^ 1"

C:\rsdn>_
```

Но дело может не ограничиваться обычным выводом.

Пример 17.3.1. Скрипт "17_3.bat"

```
@echo off
set /P MYVAR=< MyVar1.var
set /P MYVAR2=< MyVar2.var
call set MYVAR2=%MYVAR%
echo "%MYVAR2%"
printargs "%MYVAR2%"
```

Пример 17.3.2. Файл "MyVar2.var"

```
(1 < 2) & (3 > 4) ^ %MYVAR%
```

Вывод скрипта "17_3.bat":

```
C:\rsdn>17_3.bat
1 was unexpected at this time.

C:\rsdn>_
```

Можно обернуть всё выражение команды `set` в кавычки.

Пример 17.4. Скрипт "17_4.bat"

```
@echo off
set /P MYVAR=< MyVar1.var
set /P MYVAR2=< MyVar2.var
call set "MYVAR2=%MYVAR2%"
echo "%MYVAR2%"
printargs "%MYVAR2%"
```

Вывод скрипта "17_4.bat":

```
C:\rsdn>17_4.bat
"(1 < 2) & (3 > 4) ^^ (1 < 2) & (3 > 4) ^ 1"
"(1 < 2) & (3 > 4) ^^ (1 < 2) & (3 > 4) ^ 1"
```

```
C:\rsdn>_
```

Или поступить несколько иначе:

Пример 17.5. Скрипт "17_5.bat"

```
@echo off
set /P MYVAR=< MyVar1.var
set /P MYVAR2=< MyVar2.var
call set MYVAR2= "%MYVAR2%"
echo "%MYVAR2:~1,-1%"
printargs "%MYVAR2:~1,-1%"
```

Вывод скрипта "17_5.bat":

```
C:\rsdn>17_5.bat
"(1 < 2) & (3 > 4) ^^ (1 < 2) & (3 > 4) ^ 1"
"(1 < 2) & (3 > 4) ^^ (1 < 2) & (3 > 4) ^ 1"

C:\rsdn>_
```

Проблемой в данном случае при двойном раскрытии переменной через оператор *call* будет «авто-экранирование» (дублирование) спецсимвола

^

Поэтому стоит избегать использования подобных символов в значениях переменных или пользоваться заместителями таких символов в виде других переменных, содержащих эти символы.

Комментарии в скриптах

В основном существует два способа комментировать строки текста внутри скриптов, встроенная команда

```
rem
```

и оператор

```
::
```

Было замечено, что при определённых условиях оператор двойного двоеточия может вызывать ошибки выполнения скриптов, поэтому рекомендуется пользоваться только встроенной командой *rem*.

ПРИМЕЧАНИЕ

Ошибки могут проявляться в основном в достаточно редких и сложных случаях (вы даже можете не понять, что проблема в комментариях). Проблема решалась простой заменой какого-то одного-двух ":"-комментариев на "rem"-комментарии, поэтому рекомендуется не пользоваться ":"-комментариями вообще.

Установка значения переменных из текстовых файлов

При чтении значений переменных из текстовых файлов командой "*set /P*", попытка использовать формат текстового файла, отличный от формата текстовых файлов *Windows* (здесь и далее имеется ввиду формат перевода строк, который, в отличие от *Unix*-фомата, имеет два символа перевода строки вместо одного), может привести к ошибкам в точках раскрытия таких переменных.

Пример 18.1.1. Скрипт "18_1.bat"

```
@echo off
set /P MYVAR=< MyVar3.var
if not "%MYVAR%" == "" (
    echo "MYVAR=%MYVAR%"
)
```

Пример 18.1.2. Файл "MyVar3.bat"

```
12345
1234
```

Вывод скрипта "18_1.bat":

```
C:\rsdn>18_1.bat
```

```
The syntax of the command is incorrect.
C:\rsdn>_
```

В качестве решения помогает переустановка переменной сразу после её чтения из файла.

Пример 18.2. Скрипт "18_2.bat"

```
@echo off
set /P MYVAR=< MyVar3.var
set "MYVAR=%MYVAR%"
if not "%MYVAR%" == "" (
    echo "MYVAR=%MYVAR%"
)
```

Вывод скрипта "18_2.bat":

```
C:\rsdn>18_2.bat
"MYVAR=12345"
C:\rsdn>_
```

Установка значения переменных из стандартного вывода

В стандартных утилитах *Windows NT* нет *Unix*-подобной утилиты *printf*, с помощью которой можно было бы сохранить вывод скрипта или утилиты в переменную. Тем не менее, этого можно добиться с помощью команды *for*.

Пример 19.1. Скрипт "setvarfromstd.bat"

```
@echo off
rem Скрипт читает стандартный вывод в переменную STDOUT_VALUE.
rem Команда "for" пропускает пустые строки до первой непустой.
if "%~1" == "" exit /b 65
rem Сбрасываем код возврата.
cd .
set STDOUT_VALUE=0
set STDOUT_VALUE=
rem Выполнение '*' (%* выдает значение всех аргументов одной строкой)
rem не устанавливает значение ERRORLEVEL.
for /F "usebackq tokens=*" %i in ('%*') do (set STDOUT_VALUE=%i) && goto :EOF
```

Вывод скрипта "setvarfromstd.bat":

```
C:\rsdn> setvarfromstd.bat echo.10
C:\rsdn>echo "STDOUT_VALUE=%STDOUT_VALUE%"
"STDOUT_VALUE=10"
C:\rsdn>_
```

Следует заметить, что для выполнения аргумента *for* скрипт создает новую консоль *cmd.exe*, в которой выполняет команду. Для проверки скрипт можно запустить так:

```
call setvarfromstd.bat pause
```

и проверить появление нового процесса *cmd.exe* в списке процессов.

Несколько слов о компиляторах

Не существует способа абсолютно переносимо перевести текст bat-файла в exe-программу. И как это не смешно бы звучало, связано это скорее с особенностями самих скриптов (здесь даже речь не идёт о чтении и изменении скрипта самого себя как текстового файла, используя переменную *%0*), а также дизайна как такового, который и мешает самой переносимости.

В Интернете вы можете найти несколько компиляторов:

- Quick Batch File Compiler (<http://www.battoexe.com>)
- Batch File Compiler (<http://www.bdargo.com>)
- Cmd2Exe (<http://www.mailsend-online.com>)

Есть ли замена CMD.EXE?

Есть. Существует множество замен, начиная от "неродных", перенесенных из-под *Unix* и *Linux*, и заканчивая разработками, ориентированными только на *Windows*. Останавливаться подробно на «неродных» командных процессорах я бы здесь не хотел (самый известный из них — *Cygwin*, который не только предоставляет набор утилит, но и **NIX*-совместимый *API*). Приведу список нескольких командных процессоров для *Windows*, которые я смог найти в интернете.

- PowerShell от Microsoft
- 4NT от JP Software (В данный момент поставляется вместе с “Take Command”)
- BPL от Jim Lawless
- WinBatch от Wilson WindowWare
- XLNT от Advanced Systems Concepts

ПРИМЕЧАНИЕ

Некоторые командные процессоры, такие как 4NT, предлагают интеграцию на таком уровне, что скрипты, написанные под *cmd.exe*, свободно работают и под 4NT без каких-либо дополнительных модификаций, но это обманчиво. 4NT – это *сторонний* командный процессор, и он не в состоянии повторить *все* “багофичи” *cmd.exe*. Чтобы убедиться в этом, достаточно будет протестировать представленные здесь примеры.

На наш взгляд, в реальной работе имеет смысл использовать *PowerShell*, если вы полностью сосредоточены на работе с ОС *Microsoft*, или *Cygwin*, который существует на множестве платформ в неизменном виде и предоставляет знакомый unix-подобный интерфейс. Возможно, для некоторых задач можно вообще отказаться от командных процессоров с их скриптами, и предпочесть им утилиты управления процессом сборки типа *ANT* или *MSBuild* – прим. ред.

ЗАКЛЮЧЕНИЕ

Я постарался привести здесь всё, с чем мне приходилось столкнуться при написании bat-скриптов, но не исключено, что что-то могло быть упущено.

Итак, мы пришли к некоторому набору условий и правил, которые следует выполнять и помнить для написания корректных скриптов, а это:

1. Не забывать там, где это необходимо, экранировать приоритетные спецсимволы;
2. Не вставлять пробелы при присваивании переменных командой *set* и не использовать в ней кавычки в значениях переменных в качестве ограничивающих символов;
3. При возможном использовании спецсимволов в значениях переменных, пользоваться кавычками вокруг выражения *Variable=Value* выражения *set Variable=Value*.
4. При установке переменной убедиться, что правая часть выражения не может быть пустой.
5. Сбрасывать переменные с помощью вспомогательного скрипта *unset.bat*.
6. По возможности вместо условия-блока использовать условие-переход.
7. При использовании блока (оператор “скобки”) помнить, что фактическое присваивание переменных (включая установку кода возврата) выполняется после выхода из самого внешнего блока, располагающегося внутри выполняемого скрипта;
8. Всегда вызывать скрипты из других скриптов через оператор *call*.
9. Всегда завершать выполнение скрипта с текущим кодом возврата через *goto :EOF* вместо *exit /b %ERRORLEVEL%*.
10. При вызове *call* внутри скрипта помнить, что строка *call* (до первого спецсимвола перенаправления ввода/вывода) раскрывается 2 раза, т.е. все “%”-переменные в ней раскрываются 2 раза, а символ «^» - автоэкранируется;
11. Всегда пользоваться кавычками в выражениях условий, если сравниваются строки и не использовать, если числа;
12. Помнить, что вызов скриптов с аргументами приводит к передаче кавычек внутрь переменных %0-%9, поэтому там, где надо (к примеру, в выражениях условий), использовать вместо них переменные %~0-%~9;
13. Использовать команды *setlocal/endlocal* в случае необходимости удалять и восстанавливать промежуточные переменные, соответственно, создаваемые и изменяемые в процессе работы какого-либо скрипта.

ПРИЛОЖЕНИЕ

Ключи реестра

Приведу некоторые ключи реестра, отвечающие за взаимодействие с командным процессором – *cmd.exe*.

Запрет режима командной строки

```
HKEY_CURRENT_USER\Software\Policies\Microsoft\Windows\System\DisableCMD:REG_DWORD
```

Для запрета режима командной строки и обработки bat-файлов создайте в этом ключе параметр с именем «*DisableCMD*», который может принимать значения:

- «0» (или отсутствие записи в реестре) — система может использовать режим командной строки и обрабатывать bat-файлы;
- «1» — система не может использовать режим командной строки, но может обрабатывать bat-файлы;
- «2» — система не может использовать режим командной строки и обрабатывать bat-файлы.
- Указанные значения проверяются при запуске новой консоли, поэтому перезагружаться не требуется.

Автовыполнение команд из разделов реестра «AutoRun»

Для текущего сеанса пользователя:

```
HKEY_CURRENT_USER\Software\Microsoft\Command Processor\AutoRun:REG_EXPAND_SZ
```

Для всех пользователей:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Command Processor\AutoRun:REG_SZ
```

Если в командной строке для *cmd* не указан параметр */d*, программа *cmd.exe* выполняет поиск указанных подразделов реестра (сначала для *HKEY_LOCAL_MACHINE*, потом для *HKEY_CURRENT_USER*). Если присутствуют один или оба подраздела реестра, они выполняются самыми первыми. Перезапустите консоль, чтобы изменения вступили в силу.

Запрет расширенного режима командного процессора

Для текущего сеанса пользователя:

```
HKEY_CURRENT_USER\Software\Microsoft\Command Processor\EnableExtensions:REG_DWORD
```

Для всех пользователей:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Command Processor\EnableExtensions:REG_DWORD
```

Для запрета расширенного режима командного процессора — *cmd.exe* создайте в этом ключе параметр с именем «*EnableExtensions*» и значением «0». Для включения расширенного режима используйте значение «1». Перезапустите консоль, чтобы изменения вступили в силу.

Расширенный режим командного процессора можно также включить или отключить для консольного процесса с помощью параметра */e:{on|off}* или несколько короче: */x* – включение, */y* – выключение.

Расширенный режим вносит изменения и дополнения в следующие команды:

- DEL или ERASE
- COLOR
- CD или CHDIR
- MD или MKDIR
- PROMPT
- PUSHHD
- POPD
- SET
- SETLOCAL
- ENDLOCAL
- IF
- FOR
- CALL
- SHIFT
- GOTO
- START (также вносит изменения в семантику вызова скриптов и утилит вне команды START)
- ASSOC
- FTYPE

Отложенное раскрытие переменных среды

Для текущего сеанса пользователя:

```
HKEY_CURRENT_USER\Software\Microsoft\Command Processor\DelayedExpansion:REG_DWORD
```

Для всех пользователей:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Command Processor\DelayedExpansion:REG_DWORD
```

Данный параметр эквивалентен применению команды

```
setlocal ENABLEDELAYEDEXPANSION
```

на “глобальном” уровне, соответственно, для текущего сеанса пользователя или для всех пользователей.

Отложенное расширение переменных среды по умолчанию не включено. Для включения используйте параметр «*DelayedExpansion*» со значением «1». Удалите созданный параметр или измените его значение на «0», чтобы вернуть настройки к первоначальному виду. Перезапустите консоль, чтобы изменения вступили в силу.

Также отложенное раскрытие переменных среды можно включить или отключить для консольного процесса *cmd.exe* с помощью параметра */v:{on/off}*.

Автозавершение имён файлов и папок

Для текущего сеанса пользователя:

```
HKEY_CURRENT_USER\Software\Microsoft\Command Processor\CompletionChar:REG_DWORD  
HKEY_CURRENT_USER\Software\Microsoft\Command Processor\PathCompletionChar:REG_DWORD
```

Для всех пользователей:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Command Processor\CompletionChar:REG_DWORD  
HKEY_LOCAL_MACHINE\Software\Microsoft\Command Processor\PathCompletionChar:REG_DWORD
```

Позволяет автоматически завершать имена файлов и папок в окне консоли *cmd.exe*. Для использования следует ввести часть пути и нажать назначенную клавишу, соответственно, для автозавершения имени файла или папки.

По умолчанию автозавершение имен файлов и папок не используется. Можно включить или отключить завершение имен файлов и папок для всех процессов *cmd.exe*, задав значения для *CompletionChar* и *PathCompletionChar*. Чтобы задать значение, введите шестнадцатеричное число управляющего символа для определенной функции (например, *0x09* — это *TAB*, а *0x08* — это *BACKSPACE*). Перезапустите консоль, чтобы изменения вступили в силу.

Также автозавершение имён файлов и папок можно включить или отключить для консольного процесса *cmd.exe* с помощью параметра */f:{on/off}*.

Внешние утилиты и скрипты

В этом разделе описаны утилиты и скрипты, прилагаемые к статье. Некоторые представленные здесь скрипты могут вызывать другие, поэтому следует рассматривать их вместе.

ПРИМЕЧАНИЕ

Из-за общего ограничения (дизайна) NT-скриптов, переменные не должны содержать значения, в которых встречаются кавычки, поэтому не стоит пытаться передавать в скрипты переменные или значения, содержащие кавычки, или указывать файлы для обработки, которые также содержат кавычки.

errlvl.exe/errlvl.bat

Устанавливает значение переменной *ERRORLEVEL*, переданное утилите/скрипту в качестве аргумента. С их помощью можно также сбросить значение, передав им ноль в качестве параметра. Сбросить код возврата можно также, вызвав команду, устанавливающую код возврата, например «*cd .*».

printargs.exe

Выводит на консоль свои входные аргументы с добавлением ограничивающих кавычек.

unset.bat

Безопасно удаляет переменную.

printfile.bat

Печатает текстовый файл на экран.

Того же эффекта можно добиться и с помощью команды *type*:

```
rem Вывод файла на экран  
type mytext.txt
```

ПРИМЕЧАНИЕ

В отличие от команды *more*, *type* не ждет подтверждения следующей страницы для вывода.

appendvar.bat

Добавляет к значению переменной, имя которой передано первым аргументом, строку, переданную вторым аргументом. Значение и строка могут отделяться символом-разделителем, передаваемым посредством третьего аргумента.

Применяется, в основном, для добавления значений к переменной *PATH*, а также к другим переменным с другими разделителями значений.

expandvar.bat

Раскрывает строку, переданную вторым аргументом, и сохраняет значение в переменную, имя которой передано первым аргументом. Скрипт восстанавливает код возврата, сбрасываемый оператором *call* внутри скрипта.

setvarfromstd.bat

Получает в качестве параметров команду и ее аргументы, выполняет команду, передав ей эти аргументы, и помещает результат в переменную *STDOUT_VALUE*.

joinvars.bat

Объединяет все строки из входного текстового файла в одну строку (используя как разделитель точку с запятой), и присваивает её переменной.

Пример использования:

Пример 22.1.2. Файл "MyVars1.vars"

```
1/1
2/2
3/3 4/4
5/5
```

Результат работы скрипта "joinvars.bat":

```
C:\rsdn>joinvars MyValue MyVars1.vars -t
C:\rsdn>echo -%MyValue%-
-1/1;2/2;3/3 4/4;5/5;-
C:\rsdn>_
```

Пример 22.1.2. Файл "MyVars1.vars"

```
1/1
2/2
3/3 4/4
5/5
```

Результат работы скрипта "joinvars.bat":

```
C:\rsdn>joinvars MyValue MyVars1.vars -t
C:\rsdn>echo -%MyValue%-
-1/1;2/2;3/3 4/4;5/5;-
C:\rsdn>_
```

splitvars.bat

Разбивает все строки из входного файла/литерала, объединённых через точку с запятой, на несколько строчек и последовательно выводит их.

Пример использования:

Пример 23.1.2. Файл "MyVars2.vars"

```
1 1;2/2;3,3;4%%4 ; 5 5 ; 6
%B%;7 7 ;8 8
```

Результат работы скрипта "splitvars.bat":

```
C:\rsdn>set B=100
C:\rsdn>splitvars MyVars2.vars -f -t -e
1 1
2/2
3,3
4%%4
5 5
6
100
7 7
```

```
8 8
C:\rsdn>_
```

iffexist.bat

Проверяет существование файла в каталогах из списка который может быть задан текстовым файлом или строкой где отдельные значения разделяются точкой с запятой (как в переменной *PATH*) или переменной (например, переменной *PATH*). Устанавливает переменную *FOUND_PATH* и возвращает 0 в случае существования файла по одному из путей в списке и 1 – в случае его отсутствия.

Пример использования:

Результат работы скрипта "iffexist.bat":

```
C:\rsdn>iffexist cmd.exe "%PATH%" -s -t
C:\rsdn>echo "FOUND_PATH=%FOUND_PATH%"
"FOUND_PATH=C:\Windows\system32\cmd.exe"
C:\rsdn>_
```

ПРЕДУПРЕЖДЕНИЕ

Так как скрипт *iffexist.bat* возвращает созданную им переменную, то, соответственно, его нельзя использовать в блоке, иначе переменная, как и код возврата, будут установлены только после выхода из самого внешнего блока.

Результат работы скрипта "iffexist.bat":

```
C:\rsdn>iffexist cmd.exe PATH -a
C:\rsdn>echo "FOUND_PATH=%FOUND_PATH%"
"FOUND_PATH=C:\Windows\system32\cmd.exe"
C:\rsdn>_
```

Ссылки по теме

1. <http://forum.ru-board.com/topic.cgi?forum=5&topic=25393&start=60> - "Командная строка, батники\сценарии (bat, cmd)"
2. http://coop.chuvashia.ru/kartuzov/os/Articles/05/Bat_NT.htm - "Командные файлы NT"
3. http://en.wikipedia.org/wiki/Scripting_language#Types_of_scripting_languages - "Wiki: Список скриптовых языков"
4. <http://web.archive.org/web/20041104042922/http://www.cmdtools.com/> - "Список командных инструментов и утилит"

ИСПОЛЬЗОВАННЫЕ ИСТОЧНИКИ

1. Windows Help: Command-line reference ("%windir%/help/ntcmds.chm")
2. win-da.by.ru – Реестр «Windows»
3. computery.ru – Эффективная консоль
4. Google.ru

Эта статья опубликована в журнале *RSDN Magazine #4-2007*. Информацию о журнале можно найти [здесь](#)