

Введение . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

Компилятор. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

Абстрактная машина. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Использование абстрактной машины. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Вызов «публичных» функций. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

Модули расширения. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

Написание «оберток». . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

Динамически загружаемые модули расширения. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

Ошибка проверки собственных функций. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

Настройка собственного диспетчера функций. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 0,37

Ссылка на функцию. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

Коды ошибок. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

Приложения . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 0,66

A: Сборка компилятора. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

Б: Построение абстрактной машины. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 0,71

C: Использование CMake. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86

D: абстрактный дизайн машины и ссылка. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

E: Поддержка отладки. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106

F: Примечания к генерации кода. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

G: Добавление сборщика мусора. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114

Н: Лицензия. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122

Индекс . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125



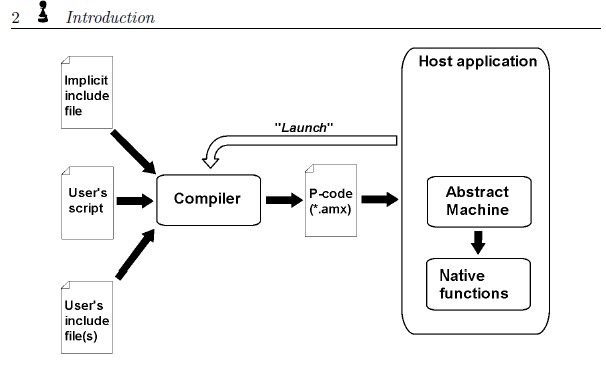
«pawn» — это простой, бестиповый, 32-битный язык расширений с синтаксисом, подобным C. Язык и функции описаны в сопроводительном буклете с подзаголовком «Язык». В этом «Руководстве по реализации» обсуждается, как внедрить язык сценариев pawn в хост-приложение. Инструментарий pawn состоит из двух основных частей: компилятор берет сценарий и преобразует его в P-код (или «байт-код»), который впоследствии выполняется на абстрактной машине (или «виртуальной машине»). Сам pawn написан в основном на языке программирования C (есть несколько файлов на ассемблере) и портирован на Microsoft Windows, Linux, PlayStation 2 и XBox. При встраивании pawn в хост-приложения, написанные не на C или C++, я предлагаю вам использовать библиотеки AMX DLL под Microsoft Windows.

Есть короткая глава о компиляторе. Большинство приложений выполняют компилятор как автономную утилиту с соответствующими параметрами. Даже когда вы связываете компилятор с основной программой, его API по-прежнему основан на параметрах, как если бы они были указаны в командной строке.

Абстрактная машина — это библиотека функций. Глава, посвященная этому, содержит несколько примеров для встраивания абстрактной машины в хост-приложение в дополнение к ссылке на все функции API. Приложения, наконец, дают инструкции по компиляции для различных платформ и справочную информацию — среди прочего, интерфейс отладчика и набор инструкций.

Язык и набор инструментов pawn были разработаны как язык расширения для приложений, в отличие от многих других языков сценариев, которые в первую очередь нацелены на командную оболочку операционной системы. Будучи языком расширения, инструменты и библиотеки набора инструментов pawn должны быть интегрированы с продуктом.

Две основные части набора инструментов pawn — это компилятор и абстрактная машина. Компилятор может быть либо связан с хост-приложением, либо может быть отдельным процессом, запускаемым из хост-приложения. Из соображений производительности абстрактная машина всегда встроена (подключена) внутрь основного приложения.



Компилятор Pawn принимает ряд текстовых файлов, содержащих код пользовательского скрипта и определения среды/хост-приложения. Один из включаемых файлов является неявным: компилятор pawn автоматически включит его в любой пользовательский скрипт, но если этот файл отсутствует, он автоматически завершится ошибкой. Имя по умолчанию для этого неявного включаемого файла (или «файла префикса») — «default.inc». Вы можете переопределить это имя с помощью параметра командной строки компилятора pawn.

Для хост-приложения рекомендуется создать неявный включаемый файл, содержащий:

⋄ все константы, специфичные для приложения;  
⋄ все встроенные функции, предоставляемые хост-приложением (или их базовое подмножество).  
родные функции);  
⋄ все перегруженные операторы (или их основное подмножество);  
⋄ все фондовые функции (или их основное подмножество);  
⋄ предварительные объявления всех публичных функций;  
⋄ объявления публичных переменных (если используются).

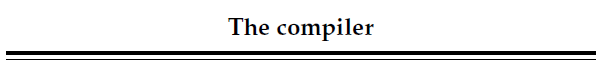
Вам нужно будет обратиться к ломбардному буклету «Язык» для написания деклараций, упомянутых в приведенном выше списке.

Причина наличия этих объявлений в неявно включенном файле заключается в том, что определения теперь всегда доступны. Это позволяет избежать ошибок, особенно в случае перегруженных операторов и публичных функций. Если определение перегруженного оператора отсутствует, во многих случаях компилятор пешки будет использовать оператор по умолчанию без предупреждения. Если пользователь допустит ошибку в объявлении публичной функции, хост-приложение не сможет ее вызвать, либо она пройдет

неправильные параметры. Предварительное объявление публичной функции перехватывает эту ошибку, потому что неправильная общедоступная функция не будет соответствовать предыдущему объявлению.

Помимо этого неявного включаемого файла, пользователь также может создавать собственные включаемые файлы и явно включать их. Кроме того, хост-приложение может предоставлять дополнительные «системные» включаемые файлы, которые не добавляются в проект автоматически и должны быть включены явно.

Следующие две главы посвящены пешечному компилятору и абстрактной машине соответственно. Наиболее распространенная конфигурация — это когда компилятор работает как отдельный процесс, порожденный хост-приложением.



Компилятор pawn в настоящее время является единственным транслятором (или синтаксическим анализатором), который реализует язык pawn. Компилятор pawn транслирует текстовый файл с исходным кодом в двоичный файл для абстрактной машины. Формат выходного файла находится в приложении D. Использование компилятора pawn описано в буклете pawn «Язык».

• **Deployment / installation**

В большинстве операционных систем компилятор представляет собой отдельную автономную исполняемую программу. Он может работать как есть, но он будет искать файл конфигурации в том же каталоге, где находится сам компилятор, и найдет (системные) включаемые файлы в определенном каталоге. Для розничных пакетов фактический компилятор находится в динамически загружаемой библиотеке. Эта библиотека называется «libpawnc.so» или «libpawnc.dll» (для Linux/Unix и «pawncc.exe» для Microsoft Windows соответственно).

Конкретно, чтобы настроить компилятор пешки в системе:

⋄ Скопируйте файл программы для компилятора (обычно «pawncc» для Linux/Unix и «pawncc.exe» для Microsoft Windows) в выбранную вами директорию, а также скопируйте библиотеку «libpawnc.so» или «libpawnc.dll» если он существует.

⋄ При желании скопируйте или создайте файл конфигурации с именем «pawn.cfg» в том же каталоге.

⋄ Добавьте подкаталог с именем «include» и скопируйте в него включаемые файлы — особенно добавьте в этот каталог файл с префиксом «default.inc», если применимо. и pawn.cfg, или он может быть на том же уровне, что и каталог, в котором находятся compile и pawn.cfg. Например, если в системе Windows pawncc.exe и libpawnc находятся в каталоге C:\Pawn\bin, компилятор будет искать включаемые файлы либо в каталоге C:\Pawn\bin\include, либо в C:\Pawn\include.

• **The configuration file**На платформах, которые его поддерживают (в настоящее время Microsoft DOS, Microsoft Windows и Linux), компилятор считывает параметры в «файле конфигурации» при запуске. Файл конфигурации должен иметь имя «pawn.cfg» и должен находиться в том же каталоге, что и исполняемая программа компилятора и/или динамически загружаемая библиотека компилятора.

В некотором смысле файл конфигурации является неявным файлом ответов (подробности о файлах ответов см. в буклете «Язык»). Параметры, указанные в командной строке, могут иметь приоритет над параметрами в файле конфигурации.

**Errors**

• **Compiler errors**

Сообщения об ошибках и предупреждения, создаваемые компилятором, описаны в сопроводительном буклете «Язык».

• **Run time errors**

Библиотека функций, формирующая абстрактную машину, возвращает коды ошибок. Эти коды ошибок охватывают как ошибки загрузки и инициализации двоичного файла, так и ошибки времени выполнения из-за ошибок программиста (проверка границ).



Абстрактная машина представляет собой библиотеку функций C. Существует несколько версий: одна написана на ANSI C и оптимизированные версии, использующие расширения GNU C или подпрограммы ассемблера.

• **Deployment / installation**

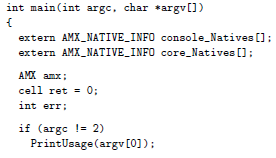
Абстрактная машина либо связана с основной программой, либо реализована в виде загружаемой библиотеки (DLL в Microsoft Windows или «общая библиотека» в Linux). Никаких особых соображений для перераспределения абстрактной машины не требуется.

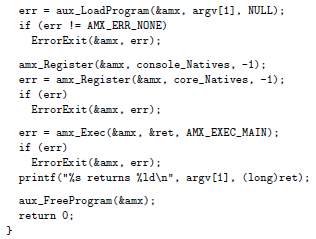
Если вы разрешаете динамическую загрузку модулей расширения, вам может потребоваться установить переменную среды в Linux/UNIX. Эти операционные системы ищут библиотеки по определенному пути, если для библиотеки не указан явный путь. В Linux/UNIX абстрактная машина строит определенный путь из комбинации переменной среды «AMXLIB» и имени библиотеки. Например, если для AMXLIB установлено значение «/opt/Pawn/bin», а модуль использует amxTime, абстрактная машина загрузит «/opt/Pawn/bin/amxTime.so». Имя переменной окружения настраивается, — см. стр. 74.

**Using the abstract machine**

Чтобы использовать абстрактную машину:  
1 инициализировать абстрактную машину и загрузить скомпилированный псевдокод;  
2 зарегистрировать все встроенные функции, которые предоставляет хост-программа, непосредственно с помощью amx\_Register или косвенно;  
3 запустить скомпилированный скрипт с помощью amx\_Exec;  
4 и очистить абстрактную машину и другие ресурсы.

Пример (в C) ниже иллюстрирует эти шаги:





Тип данных ячейки определен в AMX.H, обычно это 32-битное целое число.

Сначала программа проверяет наличие аргумента командной строки; если это так, программа предполагает, что это имя файла скомпилированного скрипта пешки. Функция PrintUsage обсуждается далее в этой главе.

Функция aux\_LoadProgram выделяет память для абстрактной машины, загружает скомпилированный псевдокод и инициализирует лот. Эта функция не является частью ядра пешки только из-за того, что она делает: выделение памяти и файловый ввод-вывод. Поэтому функция aux\_LoadProgram реализована в отдельном исходном файле и имеет префикс «aux\_», а не «amx\_» («aux» означает «вспомогательный»). Ниже мы рассмотрим реализацию aux\_LoadProgram.

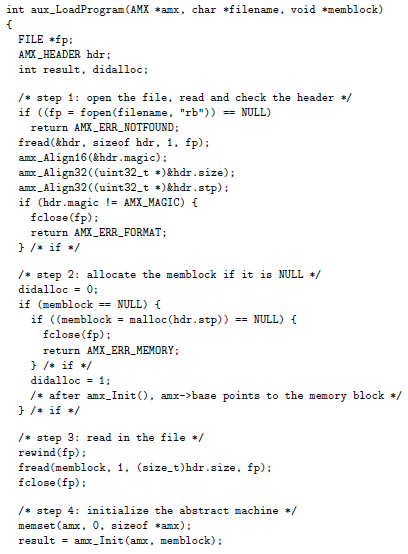
В программе есть объявления для двух наборов собственных функций: консольных функций из AMXCONS.C и основных функций из AMXCORE.C. Оба эти набора зарегистрированы в абстрактной машине. Функция amx\_Register возвращает код ошибки, если скомпилированный скрипт содержит неразрешенные вызовы нативных функций. Следовательно, необходимо проверить только результат последнего вызова amx\_Register.

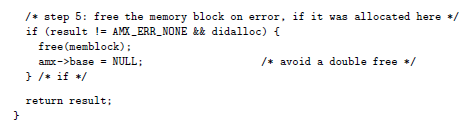
Вызов amx\_Exec запускает скомпилированный сценарий и возвращает как код ошибки, так и код результата программы. Ошибки, которые могут возникнуть во время выполнения amx\_Exec, включают деление на ноль, столкновение стека/кучи и другие распространенные ошибки времени выполнения, но встроенная функция или инструкция утверждения в исходном коде программы закладывания также может прервать сценарий закладки с кодом ошибки.

После завершения работы скрипта aux\_FreeProgram освобождает память и ресурсы, выделенные для него. Это тоже вспомогательная функция — пример реализации см. на стр. 10.

API абстрактной машины не имеет функций, которые считывают скомпилированный скрипт из файла в память; хост-программа должна их реализовать. Примером реализации, поставляемой с набором инструментов pawn, является aux\_LoadProgram. Это довольно большая функция, так как она:

1 открывает файл и проверяет/массирует заголовок;  
2 дополнительно выделяет блок памяти для хранения скомпилированного псевдокода (P-код);  
3 считывает полный файл P-кода;  
4 инициализирует абстрактную машину и подготавливает P-код к выполнению;  
5 очищает выделенные ресурсы на случай возникновения ошибки.





Step 1: pawn может работать как с архитектурой Little-Endian, так и с архитектурой Big-Endian, но

он использует единый формат файла для своего псевдокода. Многобайтовые поля в заголовке формата файла имеют формат Little Endian (или формат «Intel»). При работе на процессоре с обратным порядком байтов функция amx\_Init изменяет все поля в структуре AMX\_HEADER с прямого байта на обратный порядок байтов. Однако функция aux\_LoadProgram имеет дело с несколькими полями заголовков перед запуском amx\_Init, поэтому она должна выполнить правильное выравнивание явным образом на ЦП с обратным порядком байтов, используя функции amx\_Align16 и amx\_Align32. Вызов этих функций на машине с прямым порядком байтов не причинит вреда.

Заголовок скомпилированного скрипта содержит специальный номер. Мы тут же проверяем этот «волшебный файл», потому что, если мы найдем другое значение, все остальные поля в заголовке, скорее всего, тоже будут искажены.

Step 2: Размер бинарного образа скомпилированного скрипта не соответствует общей потребности в памяти — ему не хватает памяти для стека и кучи. Поле «stp» (верхняя часть стека) в заголовке формата файла указывает правильный размер памяти.

С приведенной выше реализацией aux\_LoadProgram вы можете загрузить скомпилированный скрипт либо в блок памяти, который вы выделили ранее, либо вы можете позволить aux\_LoadProgram выделить память для вас. Аргумент memblock должен либо указывать на блок памяти соответствующего размера, либо иметь значение NULL, и в этом случае функция выделяет блок.

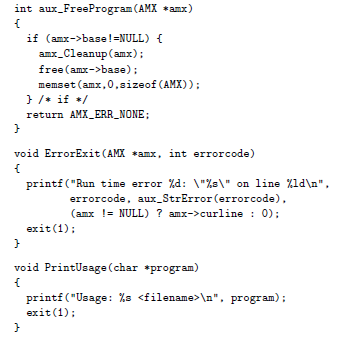
Step 3: Полный файл должен быть прочитан в блок памяти, включая заголовок, который мы читаем возле функции. После чтения файла в память его можно закрыть. Кроме того, значение hdr.size совпадает с длиной файла.

Step 4: Важно очистить структуру AMX перед вызовом amx\_Init, например, с помощью memset.

Step 5: amx\_Init выполняет несколько проверок заголовка и быстро выполняет

P-код для перемещения адресов переходов и переменных, а также для проверки недействительных инструкций. Если этот шаг проверки завершится неудачно, мы захотим освободить блок памяти, выделенный функцией, но только в том случае, если функция его выделила.

Наконец, для полноты картины функции aux\_FreeProgram, ErrorExit и PrintUsage приведены ниже:



• **Controlling program execution**

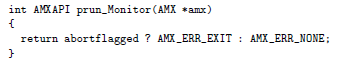
Приведенных фрагментов кода достаточно, чтобы сформировать интерпретатор программ-ломбардов. Недостатком, однако, является то, что программа-ломбард работает неконтролируемо, как только она запускается с помощью amx\_Exec. Например, если программа-пешеход входит в бесконечный цикл, единственный способ выйти из него — убить весь интерпретатор или, по крайней мере, поток, в котором работает интерпретатор. Особенно во время разработки удобно иметь возможность прерывания. пешка программа, которая работает наперекосяк.

Абстрактная машина имеет механизм для контроля за выполнением псевдокода, который называется «ловушка отладки». Абстрактная машина вызывает ловушку отладки, функцию, предоставляемую хост-приложением, при определенных событиях, таких как создание и уничтожение переменных и выполнение нового оператора. Очевидно, что хук отладки влияет на скорость выполнения абстрактной машины. Чтобы свести к минимуму потери производительности, хост-приложение может включать хук отладки «по мере необходимости» и отключать его, когда он не нужен.

Чтобы установить ловушку отладки, вызовите amx\_SetDebugHook. Функция ловушки отладки может проверять состояние абстрактной машины и просматривать символьную информацию (и исходные файлы) при ее вызове. Чтобы настроить ловушку отладки, вы должны добавить вызов amx\_SetDebugHook где-то между amx\_Init и amx\_Exec. В программе pawnrun, представленной на странице 6 (функция main), вы можете добавить следующую строку после вызова aux\_LoadProgram:



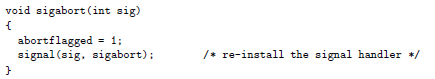
Функция amx\_Monitor становится функцией «ловушки отладки», прикрепленной к указанной абстрактной машине. Ниже приведена минимальная реализация этой функции:



Если ловушка отладки возвращает какой-либо код, отличный от AMX\_ERR\_NONE, выполнение останавливается, и amx\_Exec возвращает конкретный код ошибки. Код AMX\_ERR\_SLEEP представляет собой особый случай: он прерывает выполнение таким образом, что его можно перезапустить, передав специальный «индекс» AMX\_EXEC\_CONT в функцию amx\_Exec. Абстрактная машина вызывает ловушку отладки непосредственно перед выполнением нового оператора (в новой строке).

То, как именно хост-программа решает, продолжить работу или прервать работу абстрактной машины, зависит от реализации. В этом примере используется глобальная переменная abortflagged, которой присваивается ненулевое значение — с помощью какой-то волшебной процедуры — если абстрактная машина (машины) должна быть прервана.

Существует более или менее портативный способ достижения «магии», упомянутой в предыдущем абзаце. Если вы настроили сигнальную функцию для установки переменной с флагом прерывания в 1 в сигнале SIGINT, у вас есть одобренный «ANSI C» способ прерывания абстрактной машины. Фрагмент функции signal показан ниже:



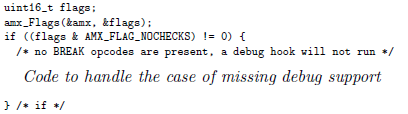
И где-то перед вызовом amx\_Exec вы добавляете строчку:



Функции отладки позволяют отслеживать использование стека, скорость выполнения профиля на уровне исходной строки и, ну. . . написать отладчик. Подробная информация о хуке отладки находится в приложении E данного руководства.

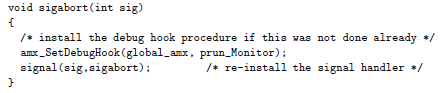
Одно предостережение заключается в том, что ловушка отладки зависит от наличия кодов операций прерывания. Когда программа-пешеход скомпилирована без отладочной информации, в P-коде отсутствуют коды операций прерывания, и хук отладки никогда не будет вызван. Это делает нашу функцию мониторинга неэффективной. Однако это нормально, потому что пользовательское или хост-приложение явно скомпилировано без отладочных проверок для повышения производительности во время выполнения — по умолчанию компилятор пешки имеет (минимальную) включенную отладочную информацию.

В вашем хост-приложении вы можете захотеть проверить наличие отладочной информации и, в случае ее отсутствия, предупредить пользователя о том, что некоторые функции могут быть недоступны. Чтобы проверить наличие отладочной информации, используйте следующий фрагмент:

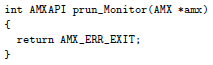


• **A smarter ‘‘break-out’’ hook**Описанный выше отладочный хук имеет большой недостаток: он замедляет работу скрипта, потому что абстрактная машина многократно вызывает функцию хука. В обычной ситуации (сигнал «abort» не подается) функция ловушки ничего не делает, кроме времени.

Улучшение состоит в том, чтобы нормально работать без ловушки отладки и настраивать ловушку отладки только после того, как уже был обнаружен разрыв сигнала прерывания. Для этого изменим сигнальную функцию на:

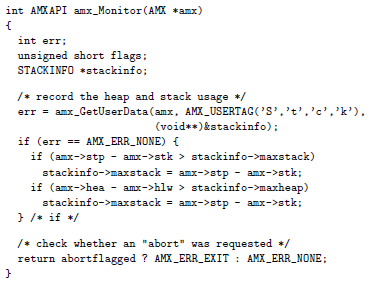


Если вы используете хук отладки только для проверки сигнала «break» или «abort», нет необходимости в глобальной переменной, которая помечает этот запрос: хук de debug будет вызываться только тогда, когда пользователь уже выдал команду break/. прерывание запроса, поэтому хук отладки всегда может вернуться с кодом ошибки, чтобы прервать сценарий.



В то время как глобальная переменная с флагом прерывания стала избыточной, я ввел новую глобальную переменную: global\_amx. Стандартная сигнальная функция из ANSI C не предусматривает передачу «пользовательского значения» через параметр, поэтому вам придется изобрести другой способ сделать абстрактную машину, которую вы хотите прервать, известной сигнальной функции. На практике ваше хост-приложение, скорее всего, будет иметь другую реализацию функции сигнала, например, процедуру обработки события в графическом интерфейсе.

• **Monitoring stack/heap usage**Полезная функция, которую может реализовать хук отладки, состоит в отслеживании того, сколько памяти использует скомпилированный сценарий во время выполнения, другими словами, проверка максимального использования стека и кучи. С этой целью в приведенном ниже примере расширена функция «монитора» отладки из предыдущих разделов и одновременно добавлено еще одно уточнение.



Эта расширенная версия amx\_Monitor по-прежнему проверяет переменную, отмеченную прерыванием (которая устанавливается по сигналу Ctrl-C или Ctrl-Break), но в то же время она также вычисляет текущее использование стека и кучи и записывает их в структуру. Используемое пространство стека — это разница между вершиной стека и текущей точкой стека; аналогично, использование кучи — это разница между текущим указателем кучи и дном кучи. Более интересно то, что функция хранит эти максимумы

вычисляемых значений в переменной stackinfo, которая представляет собой структуру со следующим определением:



Абстрактная машина позволяет хост-приложению устанавливать одно или несколько «пользовательских значений». В текущей реализации абстрактной машины может использоваться до четырех пользовательских значений. Чтобы указать, к каким пользовательским значениям вы хотите получить доступ, удобно использовать макрос AMX\_USERTAG с четырехбуквенной идентификационной строкой. В этом примере идентификационными символами являются «S», «t», «c», «k»

Функция монитора получает только указатель на структуру stackinfo и обновляет ее поля. В другом месте программы перед вызовом amx\_Exec присутствуют следующие строки для инициализации переменной и установки ее адреса в качестве пользовательского значения:



Поскольку вы, вероятно, захотите отслеживать использование стека с самого начала, перед вызовом amx\_Exec также необходимо настроить отладочную ловушку. Настройка ловушки отладки описана на странице 10.

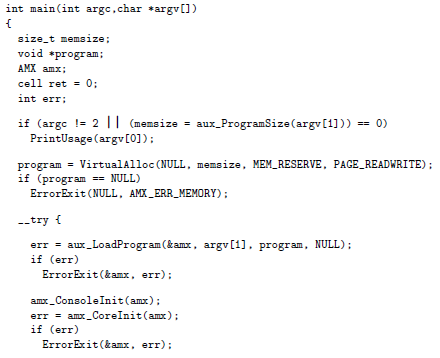
• **Preparing for memory-hungry scripts**Основные файлы времени выполнения, которые создают исполнитель абстрактной машины (AMX.C и AMXEXEC.ASM), специально разработаны для того, чтобы не использовать динамическую память или полагаться на конкретный распределитель памяти. предназначен для связывания с хост-приложениями, и на практике различные хост-приложения используют разные схемы распределения памяти — от инструментальных версий malloc до алгоритмов сборки мусора.

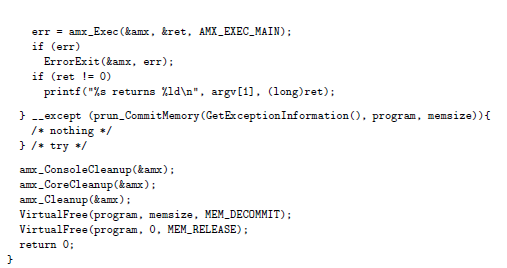


Однако недостатком этой схемы является то, что диапазон адресов, в котором выполняется скомпилированный сценарий, не может легко увеличиваться: сам исполнитель не может увеличить блок памяти, потому что он ничего не знает о распределителе памяти, который использует главная программа, и главная программа будет должны проникнуть внутрь исполнителя абстрактной машины после того, как он изменит размер блока памяти. Уже определяется, когда увеличивать блок. Следовательно, диапазон адресов, который может использовать сценарий, следует рассматривать как «фиксированный» или статический.

Проблема в том, что хост-приложение не может предвидеть, какие сценарии будут писать пользователи и сколько места потребуется для их сценариев. Пользователь может установить это значение самостоятельно с помощью #pragma dynamic, но это требует догадок и неудобно для пользователя. Когда хост-программа также запускает компилятор, он может установить размер кучи/стека на значение, достаточно большое для любого вообразимого сценария, но с риском того, что увеличение объема памяти хост-программы на этот размер повлияет на общую производительность всей системы (читай «вызывает чрезмерную подкачку»).

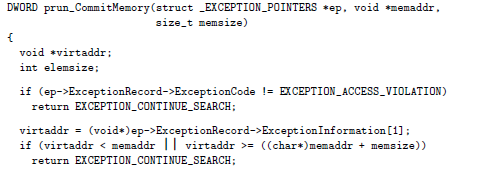
Современные операционные системы позволяют эффективно решить эту дилемму: выделить диапазон адресов памяти без резервирования памяти и впоследствии резервировать (или «фиксировать») память по мере необходимости. Фрагменты кода в этом разделе относятся к семейству Win32 MicrosoftWindows, но эта концепция применима ко многим операционным системам, предоставляющим виртуальную память.

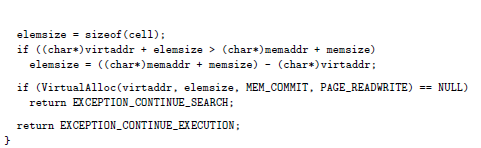




Приведенная выше основная функция является разновидностью функции на странице 6. Вместо использования malloc и free (косвенно через aux\_LoadProgram и aux\_FreeProgram) она вызывает функции Win32 VirtualAlloc и VirtualFree. Вызов VirtualAlloc резервирует диапазон адресов, но не «фиксирует» память, что означает, что в этот момент память не выделяется. Позже можно зафиксировать участки памяти внутри этого диапазона адресов с тем преимуществом, что теперь можно указать адрес памяти, который должен быть зафиксирован. В конце программы VirtualFree должен вызываться дважды, так как функция может освободить память только за один вызов, если она либо полностью зафиксирована, либо полностью освобождена. Первый вызов VirtualFree освобождает всю выделенную память.

Когда программа пытается получить доступ к незафиксированной памяти, возникает исключение «нарушение прав доступа». Функция main перехватывает исключения и обрабатывает их в функции ниже. Обратите внимание, что функция тщательно проверяет, получает ли она исключение, которое может обработать. pawn обычно обращается к элементам в ячейках, так что это размер по умолчанию для фиксации (переменная elemsize во фрагменте кода ниже), но этот размер корректируется, если он превышает диапазон выделяемой памяти.





Благодаря этим изменениям хост-программа (или пользователь) теперь может указать размер стека и кучи в несколько мегабайт при компиляции файла сценария и быть уверенным, что когда-либо будет выделена только та память, которая действительно используется программой. Microsoft Windows выделяет блоки памяти в виде «страниц» размером 4 КБ. То есть, хотя приведенный выше код фиксирует только одну ячейку (4 байта), фиксируется диапазон из 1024 ячеек.

Хост-программа может выбрать периодическое освобождение всей памяти для работающего сценария, чтобы уменьшить объем памяти, занимаемой сценарием (это не реализовано в приведенном выше фрагменте кода).

Еще одно изменение в main по сравнению с первой реализацией на странице 6 заключается в том, что он вызывает функции amx\_ConsoleInit и amx\_CoreInit, а не amx\_Register напрямую. Как объясняется в разделе о написании модулей расширения (модуль расширения — это собственная библиотека функций), предлагается, чтобы модуль расширения предоставлял функции инициализации и очистки; функция инициализации регистрирует собственные функции.

**Calling ‘‘public’’ functions**Реализации, представленные до сих пор, будут вызывать функцию main только в скомпилированном скрипте pawn. Многие реализации требуют нескольких точек входа и должны иметь возможность передавать входные параметры в эту точку входа. Нам нужно два шага, чтобы включить это:⋄ Сценарий должен предоставлять одну или несколько общедоступных функций.  
⋄ Хост-программа должна быть адаптирована для обнаружения общедоступной функции и передачи ее индекса (и параметров) в amx\_Exec.

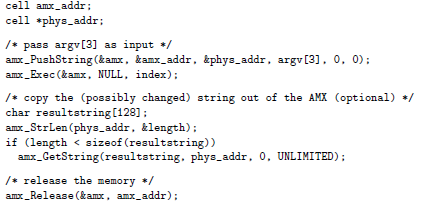
Чтобы начать с последнего шага, хост-программа адаптируется так, чтобы она находила конкретную общедоступную функцию по имени. Функция amx\_Exec принимает в качестве параметра индекс публичной функции; в предыдущих примерах использовалась специальная константа AMX\_EXEC\_MAIN для запуска с «основной» точки входа. Если известно имя публичной функции, amx\_FindPublic возвращает ее индекс. Для этого включите приведенный ниже фрагмент перед вызовом amx\_Exec (предполагается, что имя общедоступной функции находится в переменной argv[2]):



Общедоступной функции могут потребоваться входные аргументы. Если это так, они должны быть «вставлены» в стек amx перед вызовом amx\_Exec. Для числового параметра, который передается по значению, последовательность будет следующей:



Числовые («передаваемые по значению») параметры автоматически удаляются из стека при возврате функции amx\_Exec. Когда параметр является ссылочным параметром или массивом (или строкой), память должна быть явно освобождена после возврата amx\_Exec — это дает хост-приложению возможность проверять значения, сохраненные сценарием в этих параметрах. Например, чтобы передать строку из argv[3] в основной программе в общедоступную функцию на абстрактной машине, используйте следующий фрагмент кода:



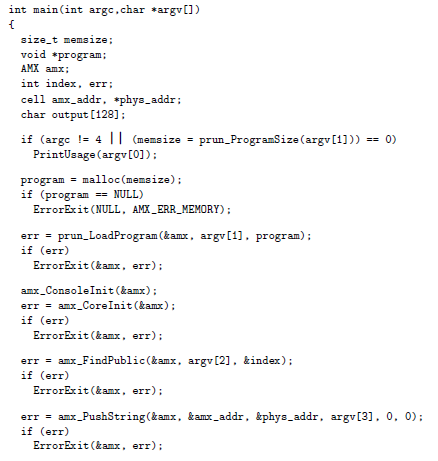
Приведенный выше фрагмент передает строку как «распакованную» строку, а это означает, что в скрипте каждая ячейка содержит один символ. API пешки (и сам язык пешки) поддерживает «расширенные символы» для приложений Unicode. В приведенном выше примере предполагается среда, не поддерживающая Unicode — в среде Unicode предпоследний параметр для amx\_SetString и amx\_GetString должен быть ненулевым.

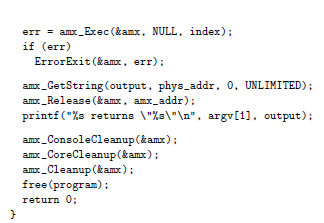
Помимо сохранения входного аргумента в стеке, функция amx\_PushString возвращает два адреса; здесь хранится в amx\_addr и phys\_addr. Переменная amx\_addr содержит адрес памяти относительно абстрактной машины — это адрес, который необходимо передать в amx\_Release, чтобы освободить выделенную память.

Переменная phys\_addr — это указатель непосредственно на стек amx, который хост-программа использует для проверки или копирования данных из абстрактной машины после возврата amx\_Exec. В этом примере хост-программа вызывает amx\_GetString для сохранения строки, измененной сценарием, в локальную переменную.

Если публичная функция имеет переменный список аргументов, все параметры в этом списке должны передаваться по ссылке. То есть вы должны следовать описанной выше процедуре для любого аргумента, попадающего в список переменных аргументов публичной функции. В качестве ссылочных аргументов передайте массив размером в одну ячейку.

Ниже приведена полная основная функция среды выполнения, которая позволяет выполнять любую общедоступную функцию и передавать строку. Эта программа, опять же, является модификацией примера программы на странице 6. Она включает вызовы amx\_FindPublic и amx\_PushString, упомянутые выше, а также показывает, как передать один дополнительный параметр через amx\_Exec.

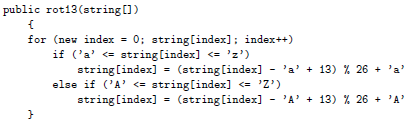




Когда программа возвращается из amx\_Exec, хост-программа может проверить возвращенные значения и освободить выделенное пространство. Представленная здесь программа использует amx\_GetString для извлечения строки, которую (возможно) изменила публичная функция. Функция amx\_Release освобождает память, выделенную amx\_PushString.

Когда вы передаете несколько аргументов строки или массива в общедоступную функцию, один вызов amx\_Release может освободить всю выделенную память, см. описание функции на стр. 58.

Чтобы продемонстрировать эту программу, мы также должны написать сценарий, содержащий общедоступную функцию и принимающий строковый параметр. Ниже приведен вариант примера сценария «ROT13» из буклета пешки «Язык». Существенной модификацией является ключевое слово public с префиксом имени функции «rot13» и удаление основной функции, которая теперь стала избыточной.



С этими изменениями и предположим, что мы построили программу C в исполняемый файл с именем «pawnrun», мы можем выполнить скрипт с помощью:  
pawnrun rot13.amx rot13 hello-world

По сути, та же процедура, что описана выше, применяется к передаче нестроковых массивов в публичную функцию:

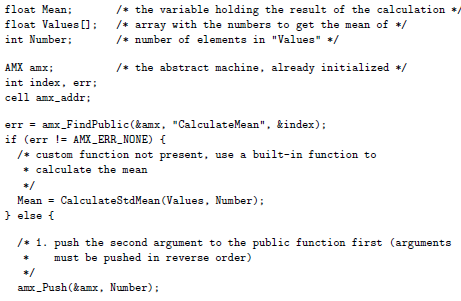
1 передать массив абстрактной машине с помощью amx\_PushArray;  
2 вызвать публичную функцию;  
3 опционально скопировать массив обратно из абстрактной машины — используя указатель «физический адрес», который вернул amx\_PushArray;  
4 освободить блок памяти в абстрактной машине с помощью amx\_Release, передав ему указатель «amx address», который также вернул amx\_PushArray.

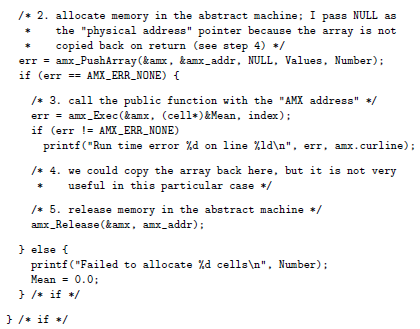
Реализация «pawnrun», вызывающая сценарий ROT13 (стр. 19), использует функции amx\_SetString и amx\_GetString для копирования строк в абстрактную машину и из нее. Причины использования этих функций связаны с разницей в размещении строк в памяти C/C++ по сравнению с pawn. При передаче массивов целых чисел (размером с ячейку) или значений с плавающей запятой вы можете просто использовать стандартные функции C memmove и memcpy.

Например, представьте себе хост-приложение, которое выполняет некоторую статистическую обработку списков чисел с плавающей запятой и позволяет пользователям приложения «настраивать» операцию, предоставляя альтернативную реализацию ключевых процедур в скрипте ломбарда. В частности, хост-приложение позволяет пользователю переопределить вычисление «среднего» с помощью сценария, содержащего общедоступную функцию CalculateMean со следующей сигнатурой:



Вот что делает хост-приложение (здесь я показываю только фрагмент кода, а не полную реализацию функции C/C++; контекст этого фрагмента см. на стр. 19):





Этот пример может показаться довольно абстрактным: «Какую альтернативную функцию среднего значения может изобрести пользователь, которая не была бы абсурдной или мошеннической?» — до тех пор, пока вы не погрузитесь в предмет и не обнаружите полный и сложный мир, стоящий за простой концепцией, такой как "Значение". Наиболее известным и наиболее часто используемым видом среднего, ставшим синонимом среднего, является «среднее арифметическое»: сумма всех элементов, деленная на число элементов. Хорошо известно, что среднее арифметическое чувствительно к выбросам, т.е. исходящий от шумного

данных, и в таких случаях «медиана» часто предлагается как стабильная альтернатива (арифметическому) среднему.

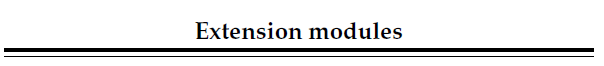
Медиана и среднее являются двумя крайними точками (арифметического) «усеченного среднего». Усеченное среднее отбрасывает самую низкую и самую высокую несколько выборок и вычисляет среднее арифметическое по оставшейся части. Количество отброшенных выборок является параметром функции усеченного среднего: если вы отбрасываете нулевые выборки, вы получаете стандартное среднее значение, а если вы отбрасываете все выборки, кроме одной, оставшаяся выборка является медианой.

Пример реализации усеченного среднего ниже отбрасывает только верхнюю и нижнюю выборки. Эта конкретная конфигурация усеченного среднего стала известна как «олимпийское среднее», ссылаясь на аналогичную процедуру, которая в прошлом использовалась для определения среднего результата спортсменов.





На этом завершается обработка аргументов массива и строки для общедоступной функции хост-приложением; то, что осталось, это ссылочные аргументы. Однако это не требует подробного обсуждения, поскольку хост-приложение может обрабатывать аргумент ссылки как аргумент массива размером в одну (1) ячейку.



Модуль расширения предоставляет программу-ломбард со специфическими для приложения («собственными») функциями. Нативная функция — это функция, реализованная в хост-приложении (в отличие от реализации в скрипте пешки), и обычно она реализуется на другом языке программирования.Создание модуля расширения — это трехэтапный процесс:

1 написание нативных функций (на C);

2 сделать функции известными абстрактной машине;

3 написание включаемого файла, который объявляет собственные функции для программ-ломбардов.

• **1.Writing the native functions**

Каждая нативная функция должна иметь следующий прототип:



Идентификатор «func» является заполнителем для имени по вашему выбору. Тип AMX — это структура, содержащая всю информацию о текущем состоянии абстрактной машины (регистры, стек и т. д.); он определен во включаемом файле AMX.H. Символ AMX\_NATIVE\_CALL содержит соглашение о вызовах функции. Файл AMX.H определяет его как пустой макрос (поэтому используется соглашение о вызовах по умолчанию), но в некоторых операционных системах или средах требуется другое соглашение о вызовах. Вы можете изменить соглашение о вызовах, отредактировав AMX.H или определив макрос AMX\_NATIVE\_CALL перед включением AMX.H. Общие соглашения о вызовах: \_cdecl, \_far \_pascal и \_stdcall.

Аргумент params указывает на массив, содержащий список параметров функции. Значение params[0] — это количество байтов, переданных функции (разделите на размер ячейки, чтобы получить количество параметров, переданных функции); params[1] — первый аргумент и так далее.

Для аргументов, которые передаются по ссылке, функция amx\_GetAddr преобразует адрес «абстрактной машины» из массива «params» в физический адрес. Указатель, возвращаемый amx\_GetAddr, позволяет напрямую обращаться к переменным внутри абстрактной машины. Функция amx\_GetAddr также проверяет, является ли входной адрес допустимым адресом.э



Когда нативная функция принимает переменное количество аргументов, все аргументы в «списке переменных аргументов» передаются нативной функции по ссылке. Четные (буквальные) константы, которые передаются в функцию, сначала сохраняются во временной ячейке стека, а затем адрес этой ячейки передается в функцию — таким образом, константа передается «по ссылке».

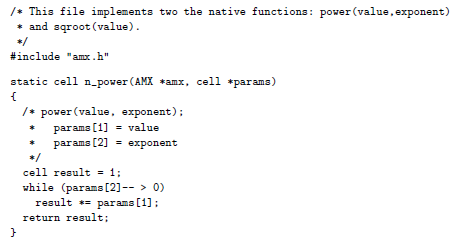
Строки, как и другие массивы, всегда передаются по ссылке. Однако ни упакованные строки, ни неупакованные строки не являются универсально совместимыми со строками C (на компьютерах с обратным порядком байтов упакованные строки совместимы со строками C). Таким образом, API абстрактной машины предоставляет две функции для преобразования строк C в строки пешек и обратно: amx\_GetString и amx\_SetString.

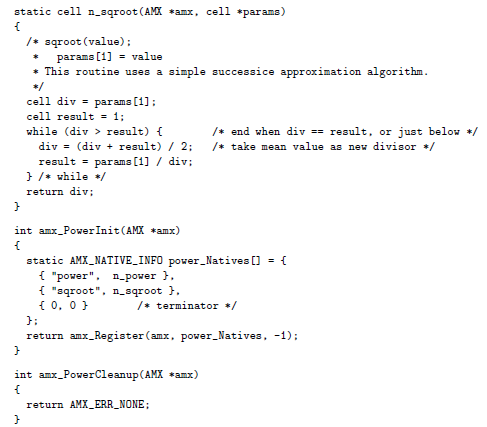
Собственная функция может прервать программу, вызвав amx\_RaiseError с ненулевым кодом. Ненулевой код — это то, что возвращает amx\_Exec.

• **2. Linking the functions to the abstract machine**

Приложение использует amx\_Register, чтобы сделать любые собственные функции известными абстрактной машине. Функция amx\_Register ожидает список структур AMX\_NATIVE\_INFO. Каждая структура содержит указатель на имя собственной функции и указатель на функцию.

Ниже приведен полный пример файла, реализующего две простые встроенные функции: возведение значения в степень и вычисление квадратного корня из значения. Список структур AMX\_NATIVE\_INFO находится в нижней части примера — он заключен в «функцию инициализации» под названием amx\_PowerInit.





В вашем приложении вы должны добавить вызов amx\_InitPower со структурой «amx» в качестве параметра, как показано ниже:



Первый пример «хост-приложений» для абстрактной машины пешки называется amx\_Register напрямую, ссылаясь на внешние массивы core\_Natives и console\_ Natives (являющиеся собственными таблицами функций). Во многих ситуациях выбранная здесь стратегия (вызов функции, предоставляемой модулем расширения для обработки регистрации собственной функции) предпочтительнее:

⋄ Предоставление функции «внешней» области видимости безопаснее, чем использование переменной; в отличие от функций переменные могут быть (случайно) изменены. Обратите внимание, кстати, что только функции amx\_PowerInit и amx\_PowerCleanup имеют внешнюю область видимости в приведенном выше примере.

⋄ Для модуля расширения может потребоваться дополнительный код запуска. Выполнение этого в той же подпрограмме, которая также регистрирует собственные функции, обеспечивает выполнение всех шагов инициализации в правильном порядке.

⋄ Для модуля расширения также может потребоваться код очистки. Когда все модули расширения предоставляют функции «инициализации» и «очистки», правила добавления модуля расширения в хост-приложение становятся универсальными. Это особенно важно, если существует соглашение об именах для этих функций инициализации и очистки.

По этой причине, несмотря на то, что модуль расширения «питание» не требует никакой очистки, была добавлена пустая функция очистки amx\_PowerCleanup.

• **3. writing an include file for the native functions**

На первом этапе реализуются собственные функции, а на втором этапе функции становятся известными абстрактной машине. Теперь третий шаг — сделать родные функции известными компилятору pawn. С этой целью пишется включаемый файл, содержащий прототипы собственных функций и все константы, которые могут быть полезны по отношению к собственным функциям.



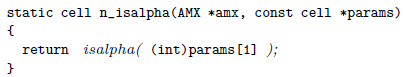
Строка библиотеки #pragma полезна при создании динамически загружаемого модуля расширения, как описано на стр. 35; это не требуется для модуля расширения, который статически связан с хост-приложением.

**Writing ‘‘wrappers’’**В предыдущих разделах описывалась реализация нескольких функций, которые были специально созданы как «собственные функции» для абстрактной машины пешки. Однако общепринятой практикой является то, что вместо написания новых функций для pawn вы делаете набор существующих функций C/C++ доступными для pawn. Чтобы «приклеить» существующие функции к закладке, вам нужно встроить каждую функцию в крошечную новую функцию с требуемой сигнатурой «родной функции». Такие новые функции называются функциями-оболочками.

Функции-оболочки также иллюстрируют проблемы с передачей параметров через границы пешек C/C++, а также то, что они предоставляют шаблоны для написания любых собственных функций.

• **Pass-by-value, the simplest case**Набор инструментов pawn был разработан, чтобы сделать интерфейс к родным функциям быстрым и простым. Для начала на примере сделаю обертку для функции isalpha из стандартной библиотеки C. Прототип isalpha:  


Обертывание isalpha в нативную функцию приводит к коду:

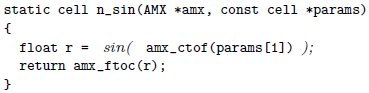


В дополнение к написанию вышеуказанной функции-оболочки, вы также должны добавить ее в таблицу для amx\_Register и добавить ее во включаемый файл для компилятора pawn.

• **Floating point**

Функции-оболочки, такие как isalpha, представляют собой простейший случай: функции, которые принимают параметры с «целочисленным» типом и возвращают «void» или целочисленный тип. Когда какой-либо из параметров или тип возвращаемого значения существующей функции является типом с плавающей запятой, эти параметры должны быть преобразованы в/из типа «ячейки», который использует pawn, но это преобразование должно происходить с помощью специального макроса. Например, рассмотрим функцию sin с прототипом:



Его функция-оболочка:  


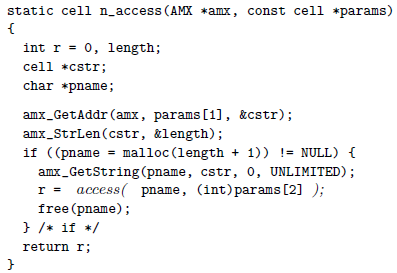
Символы amx\_ctof и amx\_ftoc представляют собой макросы, преобразующие тип «ячейка» в «плавающий» и наоборот, но в отличие от стандартных приведений типов C/C++ они не изменяют битовое представление приводимого значения. Таким образом, приведение нормального типа изменяет значение, а необходимо приведение, которое оставляет значение нетронутым — что и делают amx\_ctof и amx\_ftoc.



• **Strings**

Функции-оболочки, которые принимают строковые параметры, более сложны, потому что структура памяти строки в абстрактной машине пешки, вероятно, отличается от таковой в C/C++.† Это означает, что строки должны быть преобразованы между собственной функцией (оболочкой) и пешкой абстрактная машина. Стандартный доступ к функции C имеет прототип:



Его функция-оболочка:  


Когда абстрактная машина пешки передает массив нативной функции, она передает базовый адрес массива. Однако этот адрес относится к разделу данных абстрактной машины; это не указатель, который родная функция (в C/C++) может использовать как есть. Функция amx\_GetAddr преобразует «адрес абстрактной машины» (в params[1] в приведенном выше примере) в физический указатель для хост-приложения (т. е. cstr).

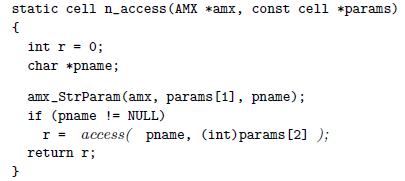
Следующим шагом является преобразование строки для формата, хранящегося в абстрактной машине, в формат, понятный C/C++. Функция amx\_GetString делает это, но перед ее использованием вы должны сначала проверить длину строки — отсюда и amx\_StrLen. Последний параметр amx\_GetString также позволяет вам ограничить количество символов, хранящихся в месте назначения; если вы знаете, что ваш буфер достаточно велик, вы можете передать константу UNLIMITED для размера.

Кстати, функция amx\_GetString распознает как упакованные, так и неупакованные строки.



Если вам нужно записать строку обратно в раздел данных абстрактной машины, вы можете использовать сопутствующую функцию amx\_SetString.

При создании оберток вручную макрос amx\_StrParam может быть удобен, поскольку он реализует «скаффолдинг-код». Оболочка для доступа к функции станет:



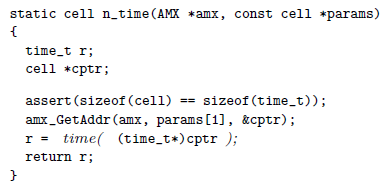
Функция-оболочка использует функцию C alloca для выделения памяти вместо malloc. Преимущество alloca в том, что память не нужно освобождать явно. Однако функция alloca не соответствует стандарту ANSI C и может быть недоступна на вашей платформе.

• **Pass-by-reference**

Функции C/C++, которые возвращают значения через указатели, нуждаются в такой же обертке, как и строки: pawn не понимает указатели, но поддерживает вызов по ссылке. Примером функции для этой оболочки является время функции C/C++ с прототипом:



Я делаю смелое предположение, что time\_t представлено как 32-битное целое число (которое также является ячейкой). Функция-оболочка становится:



В приведенной выше функции-оболочке время функции записывается непосредственно в ячейку памяти в разделе данных абстрактной машины. Это разрешено только в том случае, если значение, которое записывает функция, имеет тот же размер, что и ячейка (32 бита). На всякий случай приведенная выше оболочка проверяет это с помощью оператора assert. Если размер, возвращаемый функцией C/C++, отличается от размера ячейки, функция-оболочка должна преобразовать его в ячейку перед записью через указатель, полученный с помощью amx\_GetAddr.

• **Arrays**

Для интерфейса абстрактной машины с хост-приложением «ссылочный параметр» (см. предыдущий раздел) идентичен массиву с одним элементом. Таким образом, написание оболочек для функций, принимающих массив, похоже на написание функции, которая обрабатывает аргумент-ссылку. Основное отличие одномерных массивов состоит в том, что указатель, возвращаемый функцией amx\_GetAddr, теперь указывает на первую ячейку из потенциально многих ячеек.

Однако многомерные массивы должны обрабатываться по-разному, поскольку расположение памяти в C/C++ и pawn различается. По сравнению с C/C++, двумерные массивы в pawn имеют префикс одномерного массива, который содержит смещения в памяти до начала каждой «строки» в двумерном массиве. Этот дополнительный список позволяет каждой строке иметь различную длину столбца. В C/C++ каждый столбец в двумерном массиве должен иметь одинаковый размер.

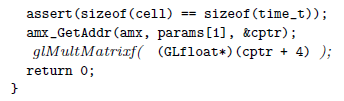
Если вы пишете функцию-оболочку для существующей функции C, а не пишете/адаптируете нативную функцию специально для использования функций pawn, вас не будут волновать массивы столбцов с переменной длиной — C/C++ их не поддерживает, поэтому ваш нативный функция не позволит их. Все, что нужно сделать, это пропустить список «смещений столбцов» с префиксом после получения адреса от amx\_GetAddr.

Например, я использую функцию OpenGL glMultMatrixf, которая умножает заданную матрицу 4 × 4 на текущую матрицу. Прототип функции:



Функция-обертка просто должна получить адрес своего параметра массива и добавить к ним четыре ячейки.





Для этого примера я выбрал функцию умножения матриц OpenGL, которая принимает матрицу значений с плавающей запятой «типа с плавающей запятой», потому что и ячейка, и типы с плавающей запятой имеют четыре байта (в обычной реализации пешки). Если вам нужно обернуть функцию, которая принимает массив значений «двойного типа», этот массив должен быть преобразован из значений с плавающей запятой в значения типа double — и, возможно, обратно в число с плавающей запятой после вызова обернутой функции.

• **Wrapping class methods (C**++ **interface)**

Интерфейс между абстрактной машиной и C/C++ основан на простых функциях. При попытке использовать метод класса в качестве нативной функции возникает сложность: (нестатическая) функция метода класса должна вызываться с неявным параметром this, о котором абстрактная машина не знает. Следовательно, абстрактная машина не может передать этот параметр напрямую, и для вызова метода класса требуется некоторый дополнительный промежуточный код.

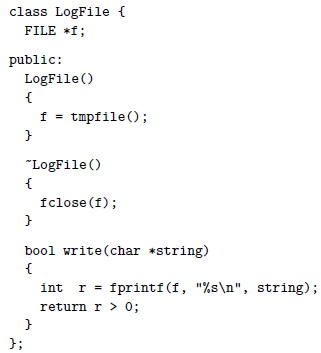
Причины, по которым вы хотите использовать методы класса как нативные функции, а не как обычные

Функции C/C++:  
1. улучшенная инкапсуляция,  
2. или возможность привязать другой экземпляр класса к каждой абстрактной машине  
(когда одновременно существует несколько абстрактных машин).

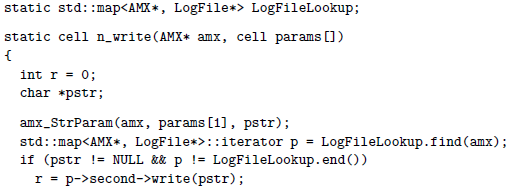
В первом случае решением является объявление методов класса и переменных-членов как «статических». Статические методы не получают этот параметр, но, в свою очередь, они не могут получить доступ к нестатическим переменным-членам. Таким образом, переменные-члены также должны быть статическими.

В этом разделе рассматривается второй случай: привязка абстрактной машины к экземпляру класса, который создается динамически. Для этой привязки интерфейсу нужны «пересылающие» функции, которые вызывают соответствующий (не статический) метод класса, и механизм поиска для сопоставления требуемого this с абстрактной машиной. Функции пересылки могут быть статическими методами в том же классе. Однако в приведенном ниже примере простые функции используются для переноса класса C++ без изменения класса.

Обертка предназначена для воображаемого класса, который позволяет писать в «лог-файлы». С помощью этой процедуры каждая абстрактная машина получит свой собственный файл журнала. Чтобы показать оболочку, класс сделан довольно упрощенно:



Когда новая абстрактная машина инициализирует свои собственные функции «файла журнала», она должна создать новый экземпляр класса и связать этот экземпляр (указатель this) с абстрактной машиной. Позже у функции упаковки/пересылки должен быть способ найти эту привязку или способ найти экземпляр класса LogFile, прикрепленный к абстрактной машине. Самый простой способ реализовать эту привязку — сохранить указатель на экземпляр класса в «пользовательских данных» абстрактной машины. Однако, поскольку количество пользовательских значений для абстрактной машины ограничено, это решение не является универсальным: если каждому модулю расширения (строковым функциям, консольным функциям, функциям даты/времени и т. д.) требуется пользовательское значение, вы запустите быстро. Альтернативным простым методом, сохраняющим привязку локальной для модуля расширения, является использование класса контейнера карты из стандартной библиотеки шаблонов (STL). STL теперь является частью стандартной библиотеки C++, поэтому, скорее всего, она будет доступна в вашей системе.





Функция-оболочка n\_write содержит обычный код для получения строкового параметра из абстрактной машины (см. стр. 30), а также ищет экземпляр класса LogFile для абстрактной машины, используя контейнер карты LogFileLookup. Функция amx\_LogFileInit создает новый экземпляр и добавляет его на карту в дополнение к регистрации собственных функций. Функция «очистить» для модуля расширения делает обратное: она удаляет экземпляр класса и удаляет его с карты. Обратите внимание, что функции amx\_LogFileInit и amx\_LogFileCleanup должны быть объявлены как «extern «C»» (но функция-оболочка n\_write не обязательно должна быть такой).

Контейнер карты из стандартной библиотеки шаблонов представляет собой реализацию общего назначения с удовлетворительной производительностью для очень маленьких и очень больших карт. Из описания свойств карты следует, что она использует структуру данных автобалансирующегося двоичного дерева. Если вы не знаете (или не контролируете), сколько абстрактных машин может работать одновременно, карта STL может быть хорошим выбором. С другой стороны, если вы можете оценить типичное количество и/или максимальное количество одновременно работающих абстрактных машин, вы обычно можете повысить производительность поиска, используя структуру данных, адаптированную к задаче и среде. .

В частности, хеш-таблица может дать почти постоянное время поиска — это означает, что поиск экземпляра класса выполняется одинаково быстро, когда имеется много параллельных абстрактных машин, и когда их всего несколько. Однако производительность хэш-таблицы быстро ухудшается, когда таблица заполняется, а очень большие хэш-таблицы имеют плохую общую производительность, поскольку они не помещаются в кэш процессора или диска.

**Dynamically loadable extension modules**

До этого момента описание разработки модулей расширения предполагало статическую компоновку модулей. Это означает, что объектный код для модулей встроен в ту же исполняемую программу/общую библиотеку, что и остальная часть основного приложения. Статическое связывание также означает, что если вы хотите добавить дополнительные собственные функции или исправить ошибку в одной из существующих собственных функций, вам потребуется доступ к исходному коду основного приложения.

Альтернативой является сборка модуля расширения в виде DLL (для Microsoft Windows) или в разделяемой библиотеке (для UNIX/Linux). При правильной настройке amx\_Init автоматически загрузит динамически загружаемый модуль расширения и зарегистрирует его функции. По завершении amx\_Cleanup очищает модуль расширения и выгружает его из операционной системы.

Помимо освобождения от написания нескольких строк (вам не нужно вызывать функции amx\_ModuleNameInit и amx\_ModuleNameCleanup), основным преимуществом динамической загрузки является то, что она позволяет легко расширять подсистему сценариев хост-приложения с помощью «подключаемых» модулей расширения. . Все, что конечный пользователь должен сделать для расширения среды сценариев, — это создать или загрузить новый модуль расширения в виде DLL/разделяемой библиотеки и скопировать его со связанным включаемым файлом (для пешечного компилятора) в соответствующие (системные) каталоги.

Чтобы построить модули расширения для динамической загрузки, придерживайтесь следующих правил:

⋄ Добавьте строку #pragma library... во включаемый файл компилятора pawn. Компилятор pawn использует эту #pragma для записи того, на какие модули расширения действительно ссылаются в скрипте. Компилятор пешки достаточно умен, чтобы не включать модуль расширения, если сценарий не вызывает ни одну из функций в этом модуле расширения.

⋄ Имя DLL или общей библиотеки должно совпадать с именем, указанным в строке библиотеки #pragma, но с префиксом из букв «amx» и расширением «.dll» или «.so», в зависимости от того, что подходит. .

⋄ Модуль расширения должен как минимум предоставлять внешнюю/экспортируемую функцию amx\_FilenameInit, где Filename — это, опять же, имя, указанное в строке библиотеки #pragma. Если библиотеке требуется код очистки, она также должна предоставлять функцию amx\_FilenameCleanup.

Например, при создании примерного модуля расширения «Питание» со страницы 25 в виде Windows DLL:

• имя файла должно быть «amxPower.dll»;  
• функции инициализации и очистки должны называться amx\_PowerInit и amx\_PowerCleanup соответственно (при этом можно не указывать ничего не делающую процедуру, такую как amx\_PowerCleanup);  
• и во включаемом файле вверху есть строка «#pragma library Power» — см. также стр. 27.

⋄ Обратите внимание, что в именах функций учитывается регистр (а в Linux — и в именах файлов). Пожалуйста, обратитесь к документации вашего компилятора для получения подробной информации о создании DLL или разделяемой библиотеки; также см. B для получения подробной информации о создании динамически загружаемого модуля расширения, особенно в разделе на странице 83.

⋄ Для развертывания в UNIX/Linux см. также стр. 6, где указана переменная среды, которую вам, возможно, потребуется установить.

Гибкость динамически загружаемых модулей расширения также является основной причиной, по которой вы можете захотеть отключить эту функцию: в интересах безопасности. Если вы тщательно и выборочно реализуете все нативные функции для вашего хост-приложения, вы хорошо понимаете, к каким частям хост-приложения и операционной системы имеют доступ конечные пользователи. С «подключаемыми» модулями расширения вся система эффективно открыта, как и в случае любой подключаемой системы.

Чтобы отключить поддержку динамически загружаемых модулей расширения, скомпилируйте абстрактную машину с определенным макросом AMX\_NODYNALOAD, см. приложение B.

**Error checking in native functions**

Сравнивая функции-оболочки для pawn с функциями для других языков сценариев, вы можете заметить, что функции-оболочки для pawn относительно малы и просты. Примечательно, что в функциях-оболочках отсутствует проверка типов и параметров, которую требуют другие языки сценариев. Функция-оболочка для isalpha, например, не проверяет количество параметров, которые передает скрипт pawn. Функция-оболочка могла бы проверять это количество аргументов, потому что pawn передает количество байтов, которое нативная функция получает в params[0 ], но в большинстве случаев эта дополнительная проверка избыточна.

Количество параметров, которые передаются нативной функции, и их теги следует проверять во время компиляции, а не во время выполнения. Следовательно, pawn требует определения нативных функций (в синтаксисе pawn) в дополнение к реализации — это был третий шаг в списке в начале главы «Модули расширения» (стр. 24).

Важно, чтобы объявления собственных функций (во включаемом файле) были точными и как можно более конкретными. Например, собственное объявление функции для функции glMultMatrixf будет таким:  


Приведенное выше объявление объявляет «m» как массив 4 × 4, содержащий значения, которые должны иметь тег «Float». Компилятор пешки теперь будет выдавать ошибку, если скрипт передает функции параметр, который не является массивом 4 × 4 или не содержит значений с плавающей запятой.

Проверки параметров, которые вы можете захотеть сделать во время выполнения из соображений безопасности, — это допустимость получаемых вами адресов. Для каждого ссылочного параметра или массива ваша собственная функция вызывает amx\_GetAddr для преобразования адреса, относящегося к абстрактной машине, в указатель, используемый C/C++. Поскольку pawn не позволяет разработчику сценария свободно манипулировать указателями, адреса, которые получает нативная функция, при нормальных обстоятельствах всегда действительны, но модифицированная версия компилятора pawn (или, возможно, ошибки в компиляторе и/или абстрактной машине) могут использоваться для передачи недопустимых адресов в собственную функцию.

Если для вашего продукта важна безопасность, вам следует проверить возвращаемое значение amx\_GetAddr; функция возвращает AMX\_ERR\_MEMACCESS, если указатель ввода недействителен. При использовании макроса amx\_StrParam указатель на выделенную память устанавливается в NULL, если адрес входного указателя недействителен.

**Customizing the native function dispatcher**Вышеупомянутые три шага для связывания собственных функций с абстрактной машиной подразумевают, что вы используете диспетчер собственных функций по умолчанию. Диспетчер по умолчанию является гибким и имеет небольшие накладные расходы, но для определенных целей вы можете создать собственный собственный диспетчер функций.

Сначала немного предыстории. Абстрактная машина очень похожа на ЦП, реализованный в программном обеспечении: у нее есть аккумулятор и несколько других «регистров», включая «указатель инструкции», указывающий на инструкцию, которая выполняется следующей. Когда функция в пешечной программе вызывает какую-то другую функцию, абстрактная машина видит инструкцию «вызов», которая настраивает инструкцию так, чтобы следующей выполняемой инструкцией была первая инструкция вызываемой функции. Пока все хорошо. Однако нативную функцию нельзя вызвать с помощью той же процедуры, так как нативная функция скомпилирована для реального ЦП, а абстрактная машина может обрабатывать только свой собственный набор инструкций. Собственная функция вызывается не с помощью инструкции «call», а с помощью инструкции «sysreq.c». Вместо корректировки указателя инструкций абстрактной машины «sysreq.c» запускает встроенный диспетчер функций. Для реальных ЦП эквивалентом «sysreq.c» будет прерывание, вызванное программным обеспечением.

Задача диспетчера собственных функций — найти правильную собственную функцию, вызвать функцию и вернуть результат функции. Прототип встроенного диспетчера функций:



где «index» — уникальный идентификатор нативной функции, «params» указывает на массив с параметрами, которые диспетчер должен передать нативной функции, а «result» — это место, где диспетчер должен хранить возвращаемое значение нативной функции. Предполагая, что у диспетчера нативных функций есть способ найти подходящую нативную функцию из индекса, диспетчер может вызвать нативную функцию с помощью:

  
Диспетчер собственных функций по умолчанию работает в сочетании с amx\_Register, который ищет функцию в «таблице собственных функций» в заголовке скомпилированного файла программы и сохраняет физический адрес функции непосредственно в этой таблице. После этого диспетчер по умолчанию может просто использовать параметр index в качестве индекса в собственной таблице функций и получить физический адрес функции. Несколько реализаций встроенного диспетчера функций по умолчанию идут еще дальше: после поиска адреса встроенной функции диспетчер изменяет код операции sysreq.c на sysreq.d и сохраняет адрес функции в качестве параметра sysreq.d. В результате каждый следующий вызов нативной функции будет переходить к нативной функции напрямую, без повторного прохождения диспетчера нативной функции.

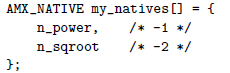


Это гибкая схема, так как она позволяет проверять скомпилированную программу и загружать только те пакеты с нативными функциями, которые программа действительно использует. Это также схема, которая налагает минимальные накладные расходы на вызов собственных функций.

С другой стороны, бывают ситуации, когда набор нативных функций, доступных программе-заложнику, фиксирован и известен заранее — например, для абстрактных машин, встроенных в (небольшие) аппаратные устройства. Для таких ситуаций у вас есть возможность жестко закодировать сопоставление индексов «sysreq» с собственными функциями.

Первый шаг, который нужно сделать, это настроить объявления нативных функций в файлах заголовков. Взяв пример функционального модуля «power», новые объявления станут такими:

  
Отличие от объявлений на стр. 27 состоит в том, что функция мощности теперь специально установлена на «sysreq» -1, а sqroot — на «sysreq» -2. Использование отрицательных чисел является обязательным: компилятор пешки резервирует положительные числа для своей схемы автоматической нумерации по умолчанию (обе схемы можно смешивать). Когда для нативной функции указан явный индекс «sysreq», компилятор pawn опускает его из таблицы нативных функций. То есть эта схема создает более компактные бинарные файлы.

Диспетчеру собственных функций по умолчанию требуется помощь, чтобы сопоставить отрицательные индексы с указателями функций; вам нужно создать таблицу с адресами нативных функций, а затем скомпилировать абстрактную машину с именем этой таблицы в макросе. В таблице первая запись относится к индексу -1, вторая - к индексу -2 и т. д.:  


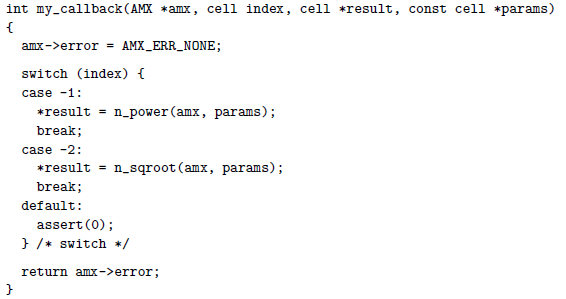
Второй шаг — скомпилировать абстрактную машину со следующим макросом:

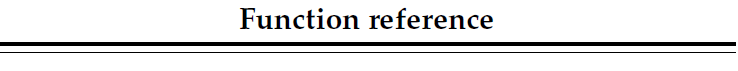


Обычно вы можете определить макрос в командной строке компилятора, например, с таким синтаксисом, как «-DAMX\_NATIVETABLE=my\_natives».

Если вам нужны специальные функции в обратном вызове, альтернативой является замена встроенного диспетчера функций пользовательской версией. Это состоит из двух шагов: создание нового собственного диспетчера функций и его установка. Последнее — это просто вызов:  

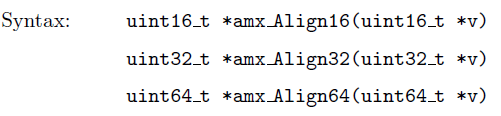

Ниже приведен пример собственного диспетчера функций — кстати, эта версия эквивалентна тому, что вы можете сделать с помощью макроса AMX\_NATIVETABLE, описанного выше.





За одним исключением все функции возвращают код ошибки в случае сбоя функции (исключением является amx\_NativeInfo). Нулевой код возврата означает «нет ошибки».

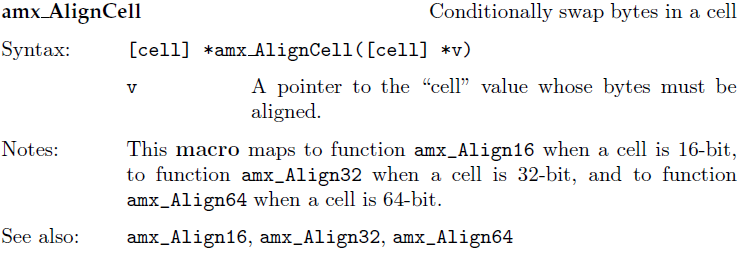
**amx Align16/32/64** Conditionally swap bytes in a 16-bit, 32-bit or 64-bit word

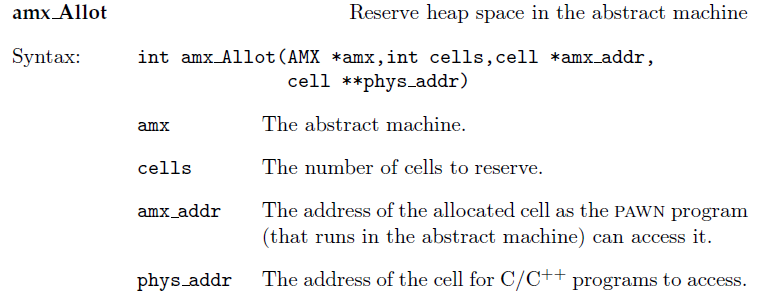


V Указатель на 16-битное значение, 32-битное значение или 64-битное значение, байты которого должны быть выровнены.  
Notes: Многобайтовые поля в заголовке в скомпилированном файле имеют формат Little Endian. При работе в архитектуре с обратным порядком байтов эти две функции меняют местами байты в 16-битном/32-битном/64-битном слове с прямым порядком байтов. Значение v остается неизменным, если код выполняется на ЦП с прямым порядком байтов, поэтому не будет никакого вреда в постоянном вызове этой функции.

amx\_Align64 доступен не во всех конфигурациях. Если пешка Abstract Machine была построена для 16-битной архитектуры, она, скорее всего, отсутствует.

See also: amx\_AlignCell





Notes:

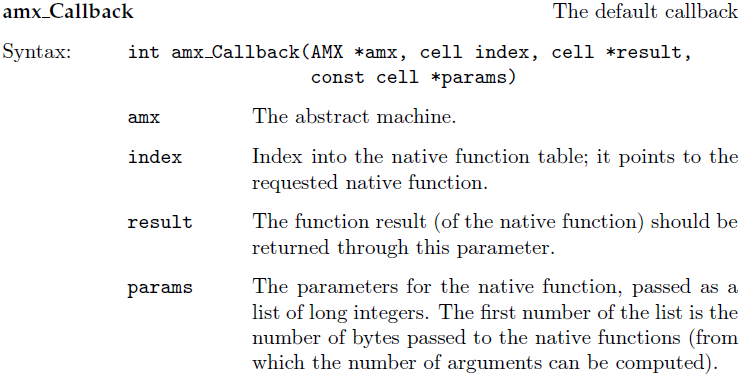
В более ранних версиях pawn массивы и строки должны были передаваться сценарию после явного выделения для них памяти в стеке amx. В текущем выпуске эта функциональность была в значительной степени заменена функциями amx\_PushArray и amx\_PushString.

Функция пешки может обращаться к памяти только внутри своей абстрактной машины. Если параметр должен быть передан «по ссылке» функции пешки, необходимо передать адрес этого параметра в amx\_Exec. Кроме того, сам этот адрес также должен находиться в диапазоне адресов абстрактной машины. Дополнительная сложность заключается в том, что абстрактная машина использует адреса, относящиеся к разделу данных абстрактной машины, а главная программа использует адрес относительно среды, которую ей дает операционная система.

amx\_Allot выделяет ячейки памяти внутри абстрактной машины и возвращает два адреса. Параметр amx\_addr — это адрес переменной относительно «раздела данных» абстрактной машины; это значение, которое вы должны передать в amx\_Exec (через amx\_Push). Параметр phys\_addr содержит адрес относительно адресного пространства хост-программы. Таким образом, программа C/C++ может использовать этот адрес и записывать в выделенную память.

осле возврата amx\_Exec вы можете проверить блок памяти (функция пешки, вызванная amx\_Exec, могла записать в него) и, наконец, освободить его, вызвав amx\_Release.



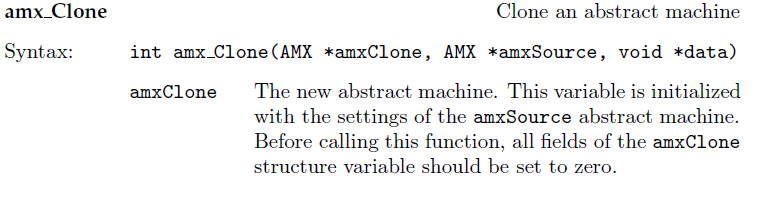


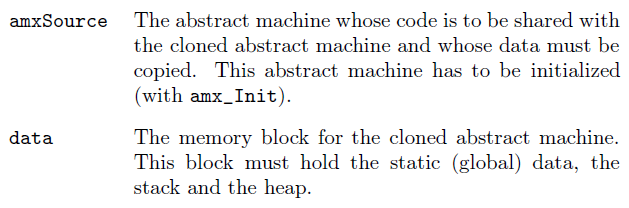
Returns: Обратный вызов должен возвращать код ошибки или ноль, если ошибки нет. Когда обратный вызов возвращает ненулевой код, amx\_Exec прерывает выполнение.

Notes: Абстрактная машина имеет функцию обратного вызова по умолчанию, которая работает в сочетании с amx\_Register. Вы можете переопределить операцию по умолчанию, установив другую функцию обратного вызова с помощью функции amx\_SetCallback.

Если вы переопределяете функцию обратного вызова по умолчанию, вам также может потребоваться предоставить альтернативную функцию для amx\_Registers.





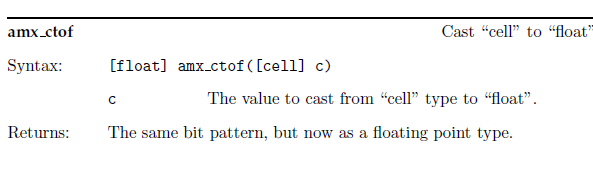


Notes: Используйте amx\_MemInfo для запроса размера статических данных и стека/кучи исходной абстрактной машины. Блок памяти, выделяемый для параметра данных, должен иметь размер, равный сумме глобальных данных и размера стека/кучи.

Клонированная абстрактная машина имеет отдельный раздел данных и отдельный раздел данных. стек скорости, но он разделяет исполняемый код с исходной абстрактной машиной. Исходная абстрактная машина не должна удаляться, пока какие-либо клонированные абстрактные машины могут оставаться активными.

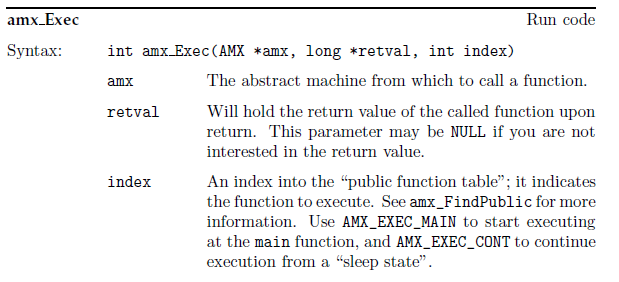
Состояние раздела данных (глобальные и статические переменные) копируется из исходной абстрактной машины в клон во время вызова amx\_Clone. Если исходная абстрактная машина изменила какие-либо глобальные/статические переменные перед клонированием, клон будет иметь эти значения в качестве своего начального состояния. На практике может оказаться целесообразным вообще не «запускать» исходную абстрактную машину, а использовать ее только для клонирования и запускать клоны.





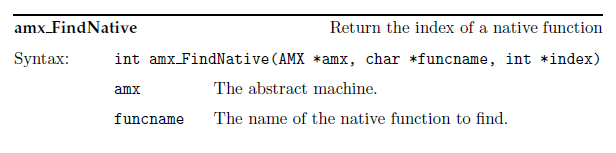
Notes: Этот макрос преобразует тип «ячейка» в тип «плавающий» без изменения битового шаблона. Приведение нормального типа в C/C++ изменяет представление выражения в памяти таким образом, чтобы его числовое значение в формате IEEE 754 было ближе всего к исходному целочисленному значению. Парсер пешки и абстрактная машина хранят значения с плавающей запятой в ячейке — при извлечении значения с плавающей запятой из ячейки битовый шаблон не должен изменяться.

See also: amx\_ftoc



Notes: Эта функция запускает скрипт, начиная с указанной функции. Он вызывает функцию обратного вызова для любого вызова собственной функции, который делает код в amx. amx\_Exec предполагает, что все нативные функции правильно инициализированы с помощью amx\_Register.

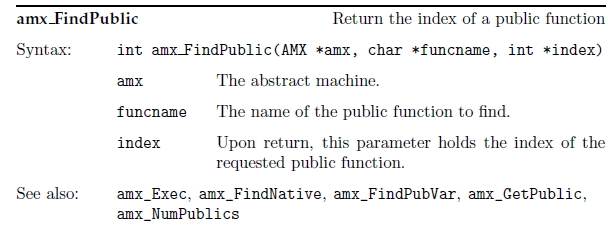


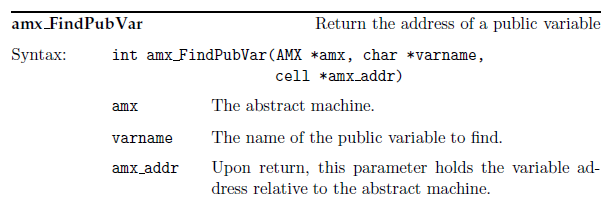




Notes: Возвращаемый индекс совпадает с тем, который абстрактная машина передала бы в amx\_Callback.

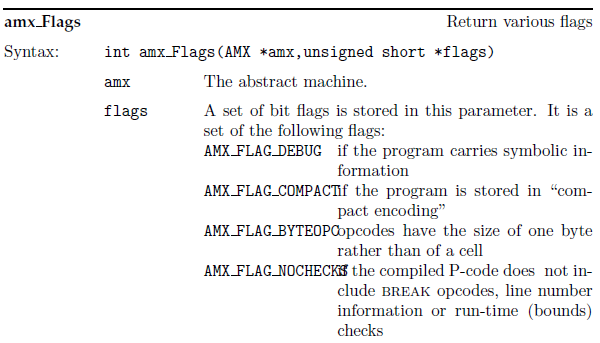




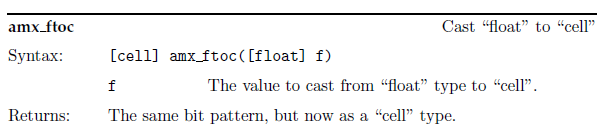


Notes: Возвращаемый адрес — это адрес относительно «раздела данных» в абстрактной машине. Используйте amx\_GetAddr для получения указателя на его «физический» адрес.



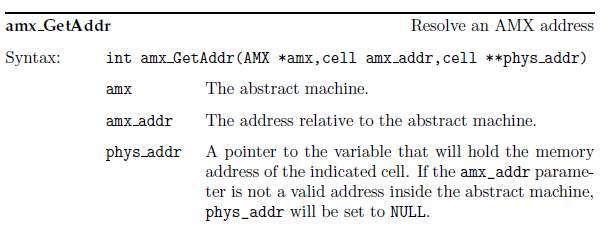


Notes: Обычно эта функция используется для проверки того, содержит ли скомпилированная программа символическую (отладочную) информацию. Запуск отладчика может быть бесполезен без символьной информации для отладки программы; если программа даже не содержит кодов операций break, установка обратного вызова отладчика может быть вообще пропущена.

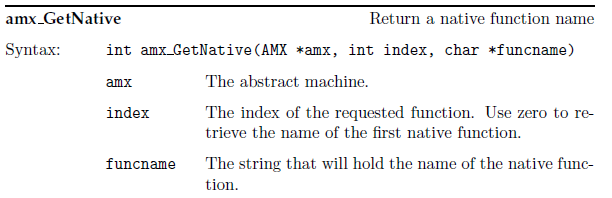


Notes: Этот макрос преобразует тип «плавающий» в тип «ячейка» без изменения битового шаблона. Приведение нормального типа в C/C++ изменяет представление выражения в памяти таким образом, что его числовое значение в целочисленном формате является целым (усеченным) значением исходного рационального значения. Парсер пешки и абстрактная машина сохраняют значения с плавающей запятой в ячейке — при сохранении значения с плавающей запятой в ячейке битовый шаблон не должен изменяться.



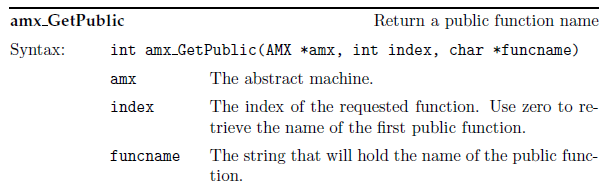


Notes: Эта функция возвращает адрес памяти адреса в абстрактной машине. Обычно эту функцию используют в модуле расширения, потому что она позволяет получить доступ к переменным внутри абстрактной машины.



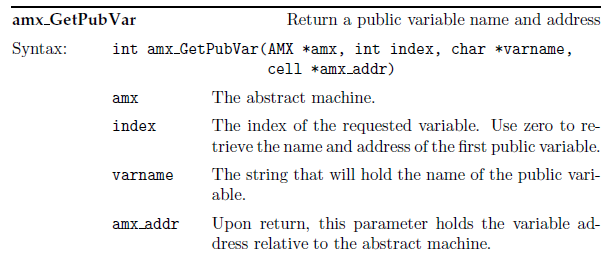
Notes: Строка должна быть достаточно большой, чтобы содержать самое длинное имя функции плюс завершающий нулевой байт. Используйте amx\_NameLength, чтобы запросить эту длину.





Notes: Строка должна быть достаточно большой, чтобы содержать самое длинное имя функции плюс завершающий нулевой байт. Используйте amx\_NameLength, чтобы запросить эту длину.

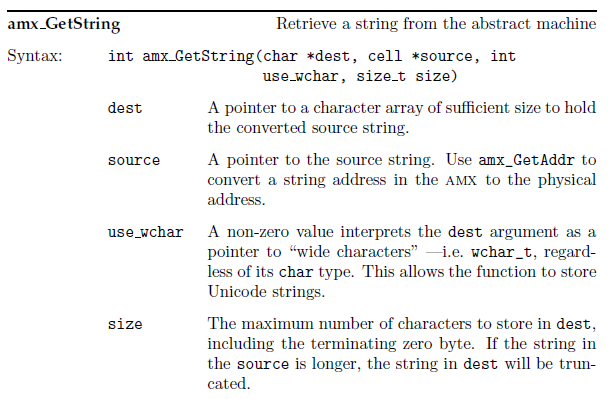




Notes: Строка должна быть достаточно большой, чтобы содержать самое длинное имя переменной плюс завершающий нулевой байт. Используйте amx\_NameLength, чтобы запросить эту длину.

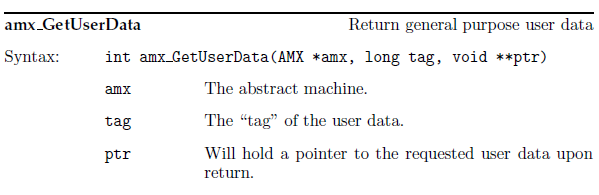
Возвращаемый адрес — это адрес относительно «раздела данных» в абстрактной машине. Используйте amx\_GetAddr для получения указателя на его «физический» адрес.





Notes: Эта функция преобразует как упакованные, так и неупакованные строки из формата «pawn» в формат «C». При извлечении неупакованной строки с параметром use\_wchar, установленным на ноль, функция может обрезать символы из расширенных символов в 8-битные ASCII/ANSI.

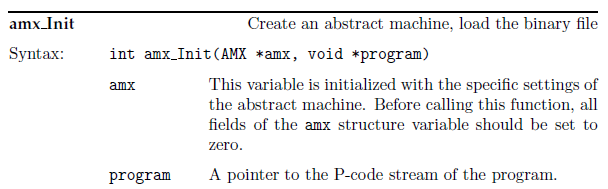




Notes: amx хранит несколько полей «пользовательских данных». Каждое поле должно иметь уникальный тег. Тег может быть любым значением (при условии, что он уникален), но обычно он формируется четырехбуквенной мнемоникой через макрос AMX\_USERTAG.

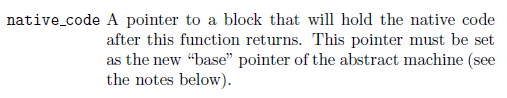
amx никоим образом не использует «данные пользователя». Хранилище можно использовать для любых целей.





Notes: amx\_Init инициализирует абстрактную машину настройками из бинарного файла. Перед вызовом этой функции вы должны установить все нули в структурную переменную amx.

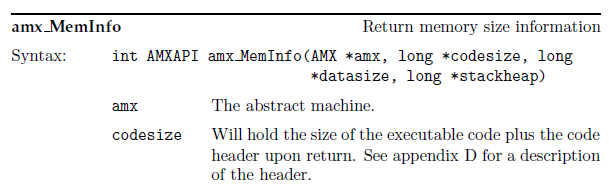
  

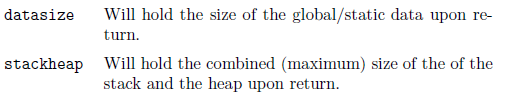



Notes: Функция amx\_Init заполняет два поля в структуре AMX, необходимые для JIT-компиляции: code\_size и reloc\_size. Оба поля представляют собой размеры буферов, которые должны быть выделены для amx\_InitJIT. Абстрактная машина будет скомпилирована в блок native\_code, который должен иметь размер code\_size (или больше), а JIT-компилятору при компиляции нужен вспомогательный блок, которым является reloc\_table размером reloc\_size.

Хост-приложение отвечает за выделение и освобождение необходимых блоков.

Функция amx\_Init дает консервативную минимальную оценку требуемого размера кода для собственных инструкций — это означает, что это значение всегда (или должно быть) слишком велико. Функция amx\_InitJIT настраивает поле code\_size на точное значение. После возврата функции amx\_InitJIT скомпилированный код необходимо присоединить к структуре amx, и перед этим вы можете уменьшить блок памяти до точного размера. Чтобы присоединить собственный код к абстрактной машине, присвойте указатель native\_code полю «base» структуры amx.

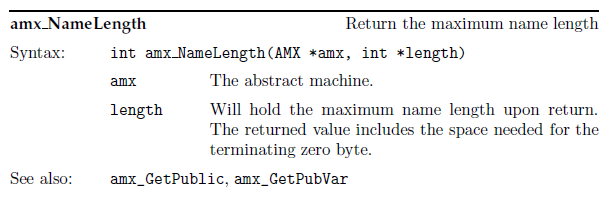
Кроме того, в некоторых архитектурах блок памяти для native\_code должен иметь соответствующие привилегии для выполнения машинных инструкций. Подробности см. на стр. 78.  
  


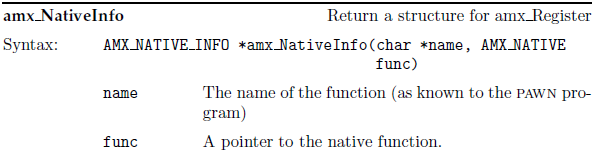


Notes: Все размеры указаны в байтах.

Стек и куча совместно используют область памяти; стек растет по направлению к куче, а куча растет по направлению к стеку.

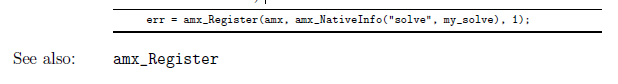


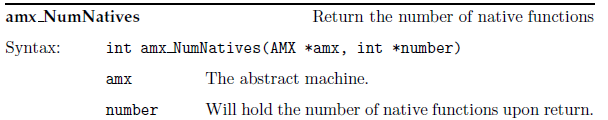




Возвращает: указатель на статическую запись (эта запись перезаписывается при каждом вызове; она не является потокобезопасной).

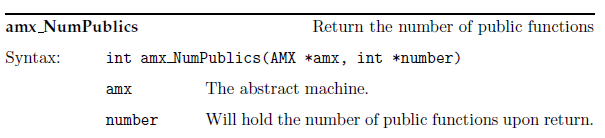
Примечания: Эта функция создает список с одной записью для amx\_Register. Чтобы зарегистрировать одну функцию, используйте фрагмент кода (где my\_solve — нативная функция):





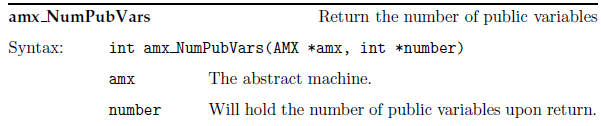
Notes: Функция возвращает количество записей в таблице «собственных функций» файла. Эта таблица содержит только собственные функции, на которые ссылается сценарий (т. е. функции, которые он вызывает). Чтобы получить имена функций, используйте \ amx\_GetNative.





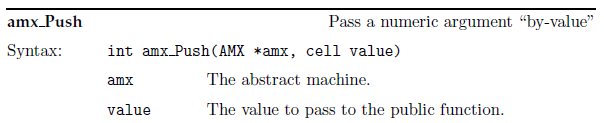
Notes: Функция возвращает количество записей в таблице «общедоступных функций» файла. Чтобы получить имена функций, используйте amx\_GetPublic.





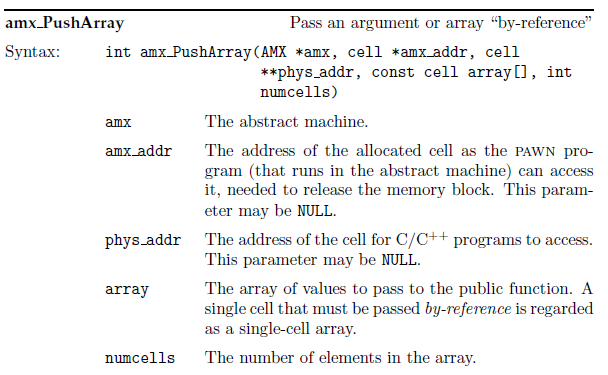
Notes: Функция возвращает количество записей в таблице «публичных переменных» файла. Чтобы получить имена переменных, используйте amx\_GetPubVar.





Notes: Любые параметры общедоступной функции должны быть переданы функции перед вызовом amx\_Exec. Если публичная функция имеет несколько аргументов, аргументы должны быть помещены в обратном порядке.

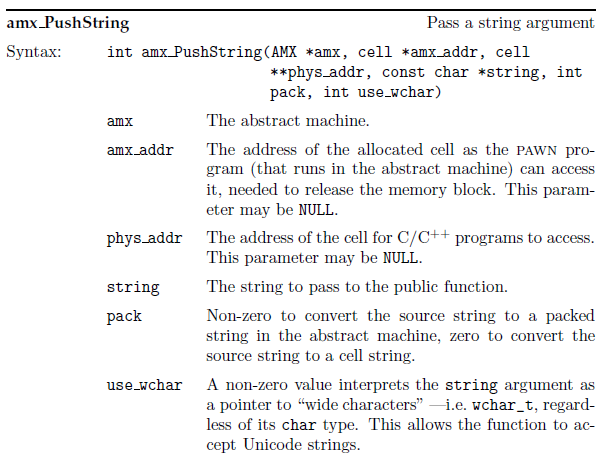




Notes: Любые параметры общедоступной функции должны быть переданы функции перед вызовом amx\_Exec. Если публичная функция имеет несколько аргументов, аргументы должны быть помещены в обратном порядке.

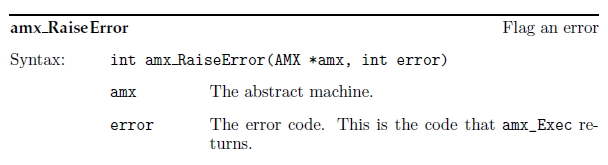
Функция выделяет память для массива внутри «кучи» абстрактной машины. Эта память должна быть освобождена с помощью amx\_Release. Подробную информацию о параметрах amx\_addr и phys\_addr см. в функции amx\_Allot.



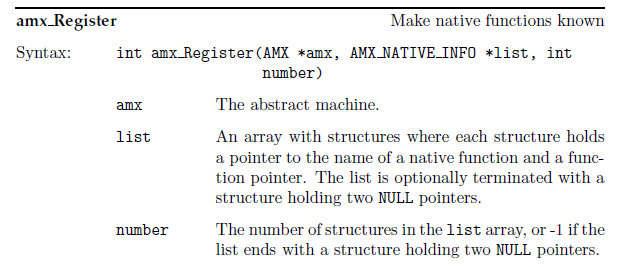


Notes: Любые параметры общедоступной функции должны быть переданы функции перед вызовом amx\_Exec. Если публичная функция имеет несколько аргументов, аргументы должны быть помещены в обратном порядке. Функция выделяет память для массива внутри «кучи» абстрактной машины. Эта память должна быть освобождена с помощью amx\_Release. Подробную информацию о параметрах amx\_addr и phys\_addr см. в функции amx\_Allot. Когда вы передаете строку Unicode и запрашиваете упакованный формат в абстрактной машине (т. е. и pack, и use\_wchar истинны), символы усекаются до 8 бит.





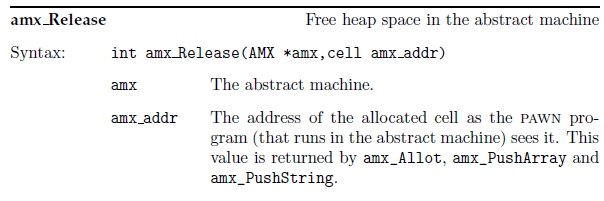
Notes: Эта функция должна вызываться из собственной функции. Это позволяет процедуре обратного вызова по умолчанию возвращать код ошибки.



Notes: В случае успеха эта функция возвращает 0 (AMX\_ERR\_NONE). Если эта функция возвращает код ошибки AMX\_ERR\_NOTFOUND, одна или несколько встроенных функций, используемых программой-ломбардом, не найдены в предоставленном списке. Вы можете снова вызвать amx\_Register, чтобы зарегистрировать дополнительные списки функций.

Чтобы проверить, зарегистрированы ли все собственные функции, используемые в скомпилированном скрипте, вызовите amx\_Register со списком параметров, равным NULL. Этот вызов не будет регистрировать никаких новых собственных функций, но по-прежнему будет возвращать AMX\_ERR\_NOTFOUND, если какая-либо собственная функция не зарегистрирована.

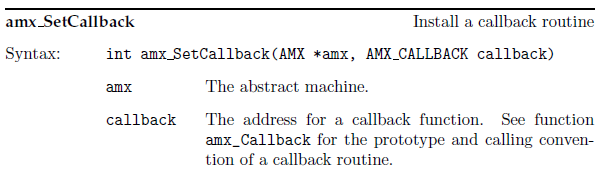




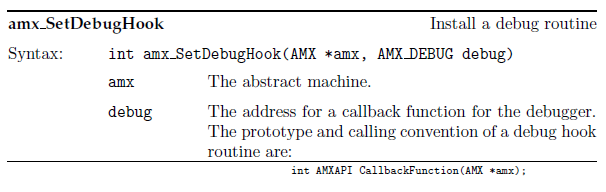
Notes: amx\_Allot выделяет память в куче в порядке возрастания (куча растет вверх). amx\_Release освобождает всю память выше значения входного параметра amx\_addr. То есть один вызов amx\_Release может освободить несколько вызовов amx\_Allot, если вы передадите значение amx\_addr первого выделения.

amx\_PushArray и amx\_PushString внутренне используют amx\_Allot, поэтому та же процедура применима и к этим функциям.

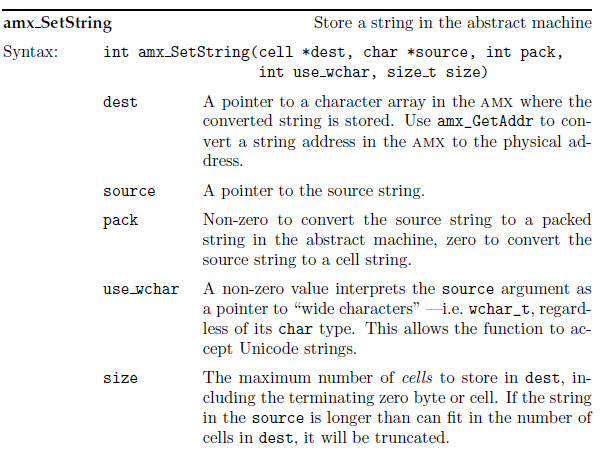




Notes: Если вы меняете функцию обратного вызова, вы не должны использовать функции amx\_Register или amx\_RaiseError. Эти функции работают в сочетании с функцией обратного вызова по умолчанию. Чтобы установить обратный вызов по умолчанию, установите параметр обратного вызова в функцию amx\_Callback. Вы можете установить обратный вызов до или после вызова amx\_Init.

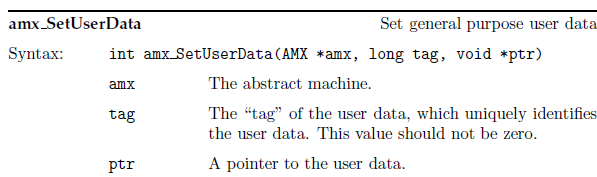


Notes: Чтобы отключить ловушку отладки, установите для параметра отладки значение NULL.



Notes: Когда вы передаете строку Unicode и запрашиваете упакованный формат в абстрактной машине (т. е. и pack, и use\_wchar истинны), символы усекаются до 8 бит.



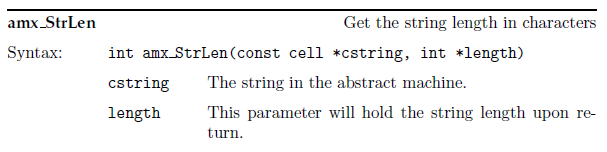


Notes: amx хранит несколько полей «пользовательских данных». Каждое поле должно иметь уникальный тег. Тег может быть любым значением (при условии, что оно уникально), кроме нуля, но обычно он формируется четырьмя символами через макрос AMX\_USERTAG.



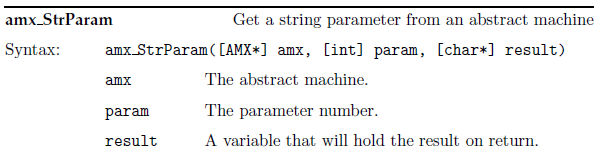
amx никоим образом не использует «данные пользователя». Хранилище можно использовать для любых целей.





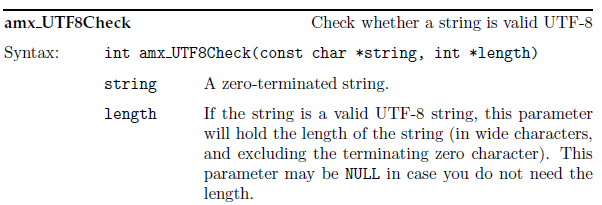
Notes: Эта функция определяет длину строки в символах, не включая завершающий нуль символ (или ячейку). Упакованная строка занимает меньше ячеек, чем ее количество символов. Если параметр cstring имеет значение NULL, параметр длины устанавливается равным нулю (0), и функция возвращается с кодом ошибки. Для конвертации распакованных строк в UTF-8 может быть более удобной функция amx\_UTF8Len.





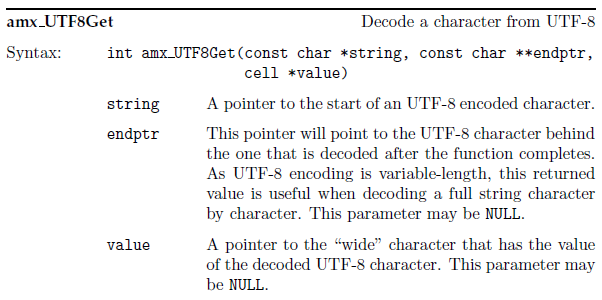
Notes: Этот макрос выделяет блок памяти (с alloca) и копирует строковый параметр (в собственную функцию) в этот блок. См. стр. 30 для примера использования этого макроса.





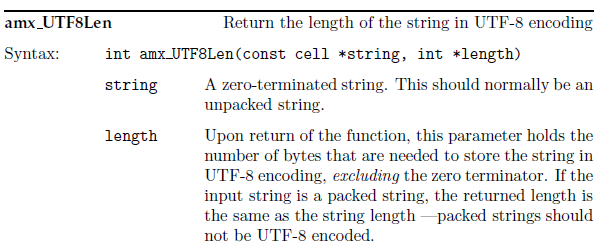
Notes: Функция проходит через строку с завершающим нулем и проверяет правильность кодировки UTF-8. Функция возвращает код ошибки, это AMX\_ERR\_NONE, если строка является допустимой UTF-8 (или действительной ASCII, если на то пошло). Если строка является допустимой UTF-8, вы можете использовать значение параметра length, чтобы выделить достаточно памяти для хранения строки, которую вы можете преобразовать с помощью amx\_UTF8Get.



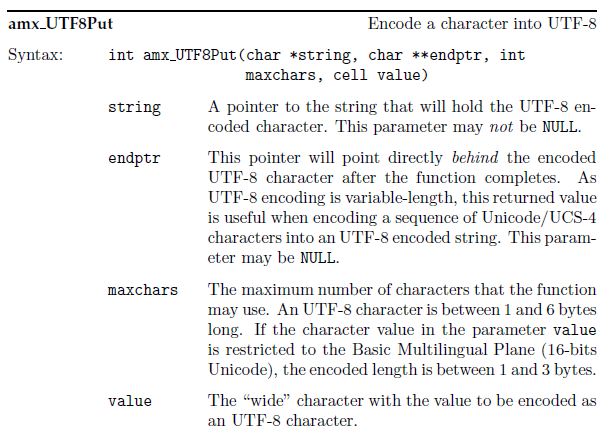


Notes: Функция возвращает код ошибки. При ошибке endptr указывает на начало символа (то же значение, что и входное значение для строкового параметра), а значение устанавливается равным нулю.





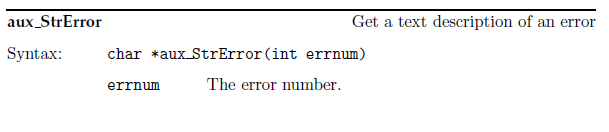
Notes: Если параметр cstring имеет значение NULL, параметр длины устанавливается равным нулю (0), и функция возвращается с кодом ошибки. 



Notes: Функция возвращает код ошибки, если параметр maxchars меньше необходимого количества байтов для кодировки UTF-8; в этом случае в строковом параметре ничего не сохраняется.

Функция не завершает строку нулем. Значения символов, недопустимые в Unicode/UCS-4, не могут быть закодированы в UTF-8 с помощью этой процедуры.

See also: amx\_UTF8Check, amx\_UTF8Get



Notes: Эта функция возвращает указатель на статическую строку с описанием ошибки с номером errnum. Некоторые коды «ошибок», такие как AMX\_ERR\_SLEEP, на самом деле не обозначают ошибочную ситуацию. Для этих кодов ошибок и недопустимых значений errnum функция возвращает описание, заключенное в круглые скобки.

**Error codes**

AMX ERR NONE No error. (0)

AMX ERR EXIT (1)

Программа прервала выполнение. Обычно это не ошибка.   
AMX ERR ASSERT (2)

Не удалось выполнить утверждение во время выполнения.   
AMX ERR STACKERR (3)

Переполнение стека или кучи; стек сталкивается с кучей.  
AMX ERR BOUNDS (4)

Индекс массива выходит за допустимые пределы.  
AMX ERR MEMACCESS (5)

Доступ к памяти, которая не выделена для программы.  
AMX ERR INVINSTR (6)

Неверная инструкция  
AMX ERR STACKLOW (7)

опустошение стека; из стека выталкивается больше элементов, чем помещается в него.  
AMX ERR HEAPLOW (8)

Недополнение кучи; из кучи удалено больше элементов, чем было вставлено в нее.  
AMX ERR CALLBACK (9)

Функция обратного вызова отсутствует, а программа вызывает нативную функцию.  
AMX ERR NATIVE (10)

Нативная функция запросила аборт абстрактной машины.  
AMX ERR DIVIDE (11)

Division by zero.  
AMX ERR SLEEP (12)

Скрипт или нативная функция заставили «заснуть». Хост-приложение может реализовать простую схему совместной многозадачности с инструкцией «засыпать».  
AMX ERR INVSTATE (13)

Функция была вызвана в состоянии, для которого функция не определена (и нет резервной функции).  
AMX ERR MEMORY (16)

Ошибка нехватки памяти общего назначения.  
AMX ERR FORMAT (17)

недопустимый формат образа памяти для абстрактной машины.  
AMX ERR VERSION (18)

Эта программа требует более новой версии абстрактной машины.  
AMX ERR NOTFOUND (19)

Запрошенные собственные функции не найдены.  
AMX ERR INDEX (20)

Недопустимый индекс (недопустимый параметр функции).

AMX ERR DEBUG (21)

Отладчик не может запуститься (это код ошибки, который может вернуть ловушка отладки).  
AMX ERR INIT (22)

Абстрактная машина не была инициализирована или была предпринята попытка ее двойной инициализации.  
AMX ERR USERDATA (23)

Невозможно установить поле данных пользователя (таблица заполнена) или невозможно получить данные пользователя (не найдено).

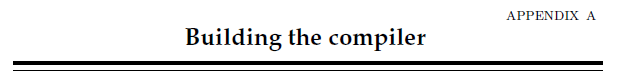
AMX ERR INIT JIT (24)  
 Компилятору Just-In-Time не удалось инициализироваться.

AMX ERR PARAMS (25)

Ошибка параметра общего назначения: один из параметров функции абстрактной машины был неправильным (например, вне допустимого диапазона).

AMX ERR DOMAIN (26)

«Ошибка домена»: результат выражения не помещается в переменную, которая должна его содержать. Эта ошибка может возникать в библиотеках поддержки фиксированной и плавающей запятой.



Исходные коды компилятора C в основном соответствуют ANSI C, поскольку набор инструментов предназначен для широкой переносимости. В целях распространения файла многоплатформенного проекта с исходным кодом я остановился на «CMake»: многоплатформенном инструменте для создания make-файлов; см. также приложение C. Файл проекта CMake для компилятора находится в подкаталоге «compiler», где установлен исходный код пешки. Существует еще один файл проекта CMake, который создает абстрактную машину в подкаталоге «amx».’

Если вы не можете использовать CMake, вы можете легко создать make-файл или проект для исходников. Для начала введите в терминале или в поле DOS команду, например:



Знаки «(...)» указывают на продолжение строки, кстати, их не следует набирать. Команда cl означает утилиту компиляции и компоновки для вашего компилятора. Это действительно «cl» для Microsoft Visual C/C++, «bcc32» для Borland C++ и «gcc» для GNU GCC. Приведенная выше команда создает компилятор pawn как один исполняемый файл без общей библиотеки/DLL. Теперь вы можете настроить сборку, добавив параметры компилятора. в командной строке.

Исходный код содержит участки кода, которые компилируются условно. См. руководство по компилятору, как указать параметры в командной строке или в «проекте», чтобы установить эти параметры. Исходный код компилятора также содержит утверждения, которые помогают мне выявлять ошибки при сопровождении кода. Чтобы собрать компилятор без утверждений, скомпилируйте компилятор с набором определений NDEBUG.

Файл проекта CMake создает компилятор как разделяемую библиотеку плюс крошечную программу-драйвер в консольном режиме, которая передает параметры командной строки в библиотеку. Вы также можете собрать компилятор как статическую библиотеку или как автономный исполняемый файл (как было объяснено в предыдущих абзацах). См. стр. 69 для создания статической библиотеки.

При компиляции исходников под Linux вам может понадобиться сначала преобразовать окончания строк CR/LF в окончания строк LF — существует два архива исходного кода для pawn: ZIP-файл имеет окончания строк в стиле DOS/Windows (CR/LF) и файл TAR-GZIP имеет окончания строк в стиле Unix (LF). Некоторые инструменты (например, некоторые версии компилятора GCC) чувствительны к тому, как заканчиваются строки. Утилита «dos2unix» — самый удобный способ перевода исходных файлов.

Обратите внимание, что компилятор также использует включаемый файл из подкаталога «amx», поэтому, вероятно, лучше запускать dos2unix для всех исходных файлов во всех подкаталогах.

• **Compile-time options**

Компилятор представляет собой отдельную программу. Если вы хотите связать его с приложением, можете скомпилировать исходники с определением макроса NO\_MAIN. Это удалит из программы «основную» функцию и набор функций ввода/вывода. Подробности смотрите в разделе «Встраивание компилятора в приложение» (ниже).

Если вам нужен компилятор пешки, который выводит 16-битный P-код, добавьте определение «PAWN\_CELL\_SIZE=16» в параметры компилятора. Обратите внимание, что это не связано с тем, является ли сам компилятор 16-битной или 32-битной исполняемой программой. Заголовочный файл использует точные типы для компилятора, соответствующего стандарту C99, но для более старых (то есть «большинства») компиляторов смело предполагается, что «короткий int» — 16-битный, а «длинный int» — 32-битный. Если это не так для вашего компилятора, вы должны изменить определение типа ячейки в SC.H, но вы также должны проверить места, где появляются разделы, условно скомпилированные на значении PAWN\_CELL\_SIZE. Н.Б. Инструменты пешки не тестируются регулярно с 16-битными ячейками.

За базовой генерацией кода следует простой оптимизатор глазка. Если вы наткнетесь на ошибку генерации кода, первое, что вы, возможно, захотите узнать, — это ошибка, связанная с генерацией кода или оптимизатором. Для этого используйте параметр -d3 компилятора пешки (это заменяет макрос NO\_OPTIMIZE в предыдущих выпусках для «условной компиляции» оптимизатора глазка).

Для экономии места для данных (что важно для 16-битной версии компилятора, где данные и стек должны умещаться в одном сегменте по 64 КБ), две таблицы строк сжаты; эти таблицы находятся в SC5.SCP и SC7.SCP. Если вы изменяете эти строки (или добавляете к ним), строки должны быть повторно сжаты с помощью утилиты SCPACK. Перед этим вам нужно собрать сам SCPACK — это простая программа ANSI C, не зависящая от других файлов.

Компилятор pawn включает в себя препроцессор, выполняющий замену текста (с параметрами или без них). Возможности сопоставления текста у препроцессора pawn еще более гибкие, чем у препроцессора C/C++, и, как следствие, он также не менее «опасен» в запутывании кода. Вы можете решить не включать препроцессор (и ключевое слово #define), установив параметр времени компиляции NO\_DEFINE.

Компилятор pawn читает исходные файлы в наборе символов ascii и в наборе символов UTF-8. Поддержку UTF-8 можно отключить, определив макрос PAWN\_NO\_UTF8. Декодер UTF-8 в Pawn поддерживает полный 31-битный набор символов UCS-4.

Несколько функций компилятора pawn являются второстепенными гаджетами. В случаях, когда имеет значение размер компилятора, их можно удалить, скомпилировав с помощью макроса PAWN\_LIGHT. Если этот макрос определен, компилятор пропустит:

⋄ отчет об использовании (перекрестная ссылка); то есть опция «-r»,

⋄ Оценка использования стека/кучи с параметрами «-d2» и «-d3»,

⋄ возможность парсить файлы ответов; параметр командной строки «@filename» игнорируется,

⋄ поддержка файла pawn.cfg, параметры которого считываются неявно.

⋄ генерация макроинструкций и упакованных опкодов (т.е. опция «-O2» отключена).

• **Summary of definitions**AMX COMPACTMARGIN Размер буфера, необходимый для формата файла с «компактным кодированием». Подробную информацию о компактном кодировании см. на стр. 104. Значение по умолчанию — 64 (ячейки). Когда это значение установлено равным нулю, поддержка компактного кодирования полностью удаляется из абстрактной машины. Если требуется поддержка компактного кодирования, рекомендуется установить это значение не менее 30.

AMX\_NO\_MACRO\_INSTR  
Удаляет возможность генерировать макрокоманды из компилятора пешки. Инструкции макросов несовместимы с JIT.

AMX NO OPCODE PAKING  
AMX NO OPCODE PAKING Удаляет возможность генерировать упакованные коды операций из компилятора pawn. Упакованные коды операций несовместимы с JIT.  
LINUX Скомпилируйте для Linux (или, возможно, для других версий Unix).  
NDEBUG Компилировать без утверждений.  
NO MAIN Удалите из программы функции main() и I/O.  
NO DEFINE Удалите текстовый препроцессор из компилятора pawn (т.е. директиву #define).  
PAWN CELL SIZE Размер ячейки в битах: 16, 32 или 64.  
PAWN LIGHT Убрать поддержку вывода перекрестных ссылок и файлов ответов. Некоторые оптимизации также отключены. Этот код подразумевает AMX\_NO\_MACRO\_INSTR и AMX\_NO\_OPCODE\_PACKING.  
PAWN NO CODEPAGE  
Удалите поддержку кодовых страниц из компилятора пешки.  
PAWN NO UTF8  
Удалите возможность чтения UTF-8.  
  
• **Embedding the compiler into an application**Если вы хотите связать компилятор pawn с приложением, вам придется удалить из него функцию «main» (см. параметр NO\_MAIN выше). Но это только первый шаг. Кроме того, вам следует:

⋄ Обратите внимание на загрязнение глобального пространства имен многими-многими функциями и глобальными переменными компилятора pawn.

⋄ Отменить функции, которые компилятор пешки вызывает для ввода/вывода.

Архив содержит файл libpawnc.c, который иллюстрирует выполнение этих шагов. По сути, вы реализуете все функции файлового ввода-вывода, которые требуются компилятору пешки. Этим функциям не нужно читать из файла или записывать в файл, вы можете компилировать из памяти в память при условии, что вы реализуете функции, которые это делают.

Затем из вашего приложения вызовите pc\_compile, передав все аргументы. Прототип функции:  


Как видите, эта функция выглядит как стандартная функция main; при вызове pc\_compile вы должны заполнить массив аргументов, включая argv[0] (поскольку компилятор создает путь для включаемых файлов из пути/имени файла в argv[0]).  
Другими функциями, которые вы можете вызывать из приложения (до вызова pc\_compile), являются pc\_addconstant и pc\_addtag. Функция pc\_compile удаляет все символы перед возвратом, включая все константы и тэги, которые вы добавили с помощью pc\_addconstant и pc\_addtag.

Файл libpawnc.c также может служить основой для DLL или общей библиотеки. Как есть, его можно использовать как DLL/общую библиотеку для консольных приложений — функция pc\_error выводит сообщения об ошибках на консоль. Кроме того, вы можете добавить опцию «-e» в список аргументов pc\_compile, чтобы перенаправить весь вывод в файл и использовать библиотеку libpawnc без изменений в приложениях с графическим интерфейсом.

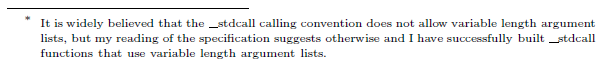
Компиляция libpawnc в «общую библиотеку» (Linux, UNIX) мало чем отличается от компиляции в DLL, но вы, вероятно, захотите уменьшить количество символов в результирующей библиотеке после этапа компоновки. Исходные коды компилятора экспортируют множество функций, предназначенных только для внутреннего использования. Команда «strip» позволяет удалить все символы из библиотеки (общие или статические), за исключением нескольких выбранных (pc\_compile и т. д.). Для DLL этот шаг не нужен, поскольку DLL экспортирует только те символы, которые были явно экспортированы через файл .DEF.

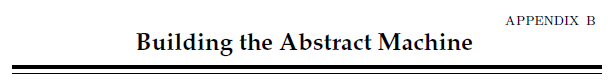
Обратите внимание, что экспортируемые функции в libpawnc предполагают соглашение о вызовах по умолчанию для компилятора. Во многих компиляторах это \_\_cdecl. Для DLL обычно используется \_\_stdcall. Вы можете изменить соглашение о вызовах компилятора по умолчанию с помощью опции (командной строки). Однако некоторые функции компилятора pawn используют списки аргументов переменной длины, и ваш компилятор может не обеспечивать поддержку списков аргументов переменной длины в соглашении о вызовах \_\_stdcall.

DLL-версия libpawnc может быть загружена из rundll/rundll32. Командная строка для 32-битной версии:



Среди рекомендуемых опций: «-D» (установить активный каталог для выходных файлов и файлов ошибок), «-i» (установить включаемый путь) и «-e» (отправить ошибку сообщения в файл).





Файлы проекта для создания примера среды выполнения консоли «pawnrun» доступны для Microsoft Visual C/C++ (в подкаталоге «msvc») и для CMake. Подробности см. в приложениях A и C. Однако в большинстве случаев вы захотите встроить абстрактную машину в приложение, а не использовать отдельную среду выполнения. Таким образом, предоставленный проект и make-файл имеют ограниченное применение.

Библиотека для «Абстрактной машины» (amx) полностью реализована в одном файле C: AMX.C. Этот файл содержит исходный код всех функций, но без встроенной функции. Ключевая подпрограмма в библиотеке, amx\_Exec, называется основной функцией amx и существует в различных версиях:  
⋄ ANSI C: самое медленное, но самое портативное ядро;  
⋄ GNU GCC оптимизирован: все еще реализован на C, но с использованием специального GNU GCC. расширения, которые делают его значительно быстрее, чем версия ANSI C;  
⋄ Ассемблер Intel Pentium: это единая конструкция, но двойная реализация для поддержки широкого спектра ассемблеров;  
⋄ Компиляторы Just-In-Time: самое быстрое ядро (но наименее переносимое).

Наряду с базовой библиотекой amx, набор инструментов поставляется с различными модулями расширения (собственными библиотеками функций), которые добавляют консольный ввод/вывод, арифметику с фиксированной и плавающей запятой, а также вспомогательные процедуры для поддержки языка. Эти модули расширения технически не являются частью «Абстрактной машины».  
Исходники C содержат разделы кода, которые компилируются условно. См. руководство по компилятору, как указать параметры в командной строке или в «проекте», чтобы установить эти параметры

Исходный код amx содержит утверждения, которые помогают мне выявлять ошибки при поддержке кода. В розничной версии amx вы захотите скомпилировать без утверждений, потому что этот код замедляет его работу. Для этого скомпилируйте исходные файлы с набором определений NDEBUG.

Сборка «по умолчанию» для инструментов предназначена для DOS/Windows. Чтобы скомпилировать для Linux, добавьте определение макроса LINUX в командную строку компилятора.  
Основные подпрограммы библиотеки amx не используют динамическое выделение памяти, файловый ввод-вывод или консольный ввод-вывод и не зависят от них, но собственные функции могут это делать. Например, функции «свойства» в модуле расширения AMXCORE.C используют malloc/free; вы можете удалить эти функции набора/получения свойств, скомпилировав файл AMXCORE.C с определением AMX\_NOPROPLIST.

Функции консольного ввода-вывода в AMXCONS.C (еще один модуль расширения) в значительной степени используют стандарт C. Для некоторых расширенных функций файл явно поддерживает терминальные коды ANSI и VT100 (ANSI.SYS в DOS, xterm и большинство оболочек в Linux), а также консольные программы Win32. Файл AMXCONS.C предоставляет функции-ловушки, которые ваше хост-приложение может реализовать для выполнения вывода на консоль. По умолчанию AMXCONS.C использует функции консоли Win32 при компиляции для Microsoft Windows и коды терминала ANSI/VT100 при компиляции для Linux или Unix. Если в системе Windows вы предпочитаете использовать терминальные коды ANSI/VT100, компилируйте с помощью макроса VT100; если вы хотите использовать свои собственные функции «консольного ввода-вывода», вместо этого определите AMX\_TERMINAL — примеры см. в разделе «Добавление терминала к абстрактной машине» на стр. 81.

В зависимости от возможностей хост-приложения и операционной системы может потребоваться включить поддержку Unicode или «расширенных символов» для подсистемы сценариев. Компилятор pawn отличается гибкостью в обработке кодовых страниц и переводе расширенных символов ascii и UTF-8 в расширенные символы (т. е. Unicode). Для хост-приложения, по сути, есть два подхода:  
Поддержка Unicode или UCS-4 и интерпретация распакованных строк как строк, содержащих «расширенные» символы. Компилятор пешки не генерирует суррогатные пары Unicode. Если требуются символы за пределами BMP («базовая многоязычная плоскость»), а хост-приложение (или операционная система) не поддерживает полную кодировку UCS-4, хост-приложение должно разделить 32-битную символьную ячейку, предоставленную компилятором пешки, на суррогатная пара. 1 Поддерживать кодировку UTF-8 и анализировать строки в хост-приложении или, если операционная система изначально поддерживает UTF-8, передавать строки на более высокий уровень без дальнейшей обработки.

Основные модули абстрактной машины не зависят от того, использует ли хост-приложение Unicode или UTF-8; основные модули абстрактной машины по умолчанию имеют встроенную поддержку широких символов. Несколько вспомогательных модулей, например AMXCONS.C (поддержка консольного ввода-вывода), необходимо скомпилировать с макросами UNICODE или \_UNICODE, определенными для включения поддержки Unicode. Оба макроса имеют одинаковый эффект. Если вы хотите удалить поддержку Unicode/расширенных символов, добавьте определение AMX\_ANSIONLY в параметры компилятора. Этот параметр также отключает поддержку UTF-8.

Соглашения о вызовах всегда являются важным вопросом при переносе программного обеспечения. Pawn amx определяет используемое соглашение о вызовах с помощью трех макросов. Эти макросы пусты по умолчанию, чтобы максимально приблизиться к ANSI C. Путем (пере)определения одного (или обоих) этих макросов вы можете настроить соглашения о вызовах:

AMX NATIVE CALL Соглашение о вызовах собственных функций. Вы можете установить для этого параметра значение \_\_stdcall при компиляции для Win32.

AMXAPI Соглашение о вызовах, используемое для всех функций интерфейса абстрактной машины (например, amx\_Init), включая собственный диспетчер функций и обратный вызов отладчика. Вам нужно изменить это, например, если вы помещаете amx в Windows DLL.

AMXEXPORT Когда вы создаете динамически загружаемые модули расширения, функции инициализации и очистки должны быть «видимы» извне. Для общей библиотеки Unix/Linux любая нестатическая функция доступна автоматически, но для Microsoft Windows функция должна быть явно экспортирована. Кроме того, рекомендуется, чтобы экспортируемые функции использовали соглашение о вызовах \_\_stdcall. Подробности см. на стр. 83.

Если вы намерены использовать ассемблерное ядро amx, необходимо учитывать еще два соглашения о вызовах.

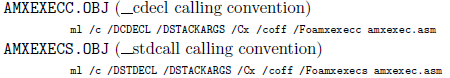
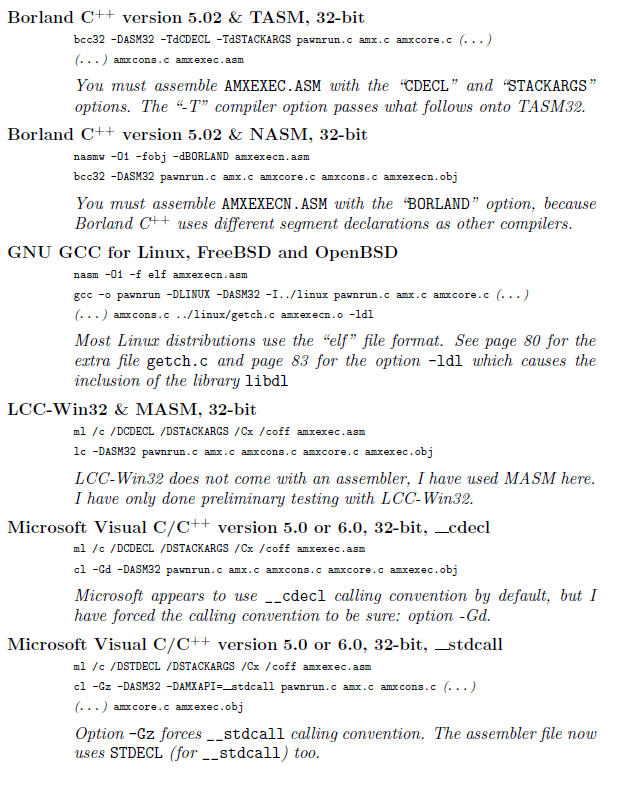
Как вы могли заметить, проблема «соглашения о вызовах» является отличительной сложностью Microsoft Windows. В Unix-подобных операционных системах вы обычно можете игнорировать проблему соглашений о вызовах.  
Метод многопоточности по умолчанию для реализаций GNU C/C++, Intel C/C++ и ассемблера — «прямая многопоточность». Вы можете переключить это на «потоковую передачу токена» с помощью параметра AMX\_TOKENTHREADING. Потоки токенов имеют больше накладных расходов на декодирование (один доступ к памяти на инструкцию P-кода), но прямая потоковая обработка требует исправления кодов операций в потоке P-кода, а потоковая передача токенов — нет. Поэтому многопоточность токена является требованием, когда скомпилированный скрипт запускается из ПЗУ или когда действуют «упакованные коды операций». Версия ANSI C поддерживает только «переключение потоков».  
• **Summary of definitions**AMX ANSIONLY Удалите поддержку Unicode и UTF-8 из абстрактной машины.  
AMX COMPACTMARGIN Размер буфера, необходимый для формата файла с «компактным кодированием». Подробную информацию о компактном кодировании см. на стр. 104. Значение по умолчанию — 64 (ячейки). Когда это значение установлено равным нулю, поддержка компактного кодирования полностью удаляется из абстрактной машины. Если требуется поддержка компактного кодирования, рекомендуется установить это значение не менее 30.  
AMX LIBPATH Имя переменной среды, используемой для поиска динамически загружаемых модулей расширения (см. стр. 35). Эта переменная среды применима только к Linux и UNIX. Его значение по умолчанию — «AMXLIB» — см. стр. 6.  
AMX NATIVE CALL соглашение о вызовах собственных функций (применяется к AMX.C и модулям расширения);  
AMX NATIVETABLE добавить собственную таблицу с «фиксированными» функциями, см. стр. 39.  
AMX NODYNALOAD отключить поддержку динамически загружаемых модулей расширения, см. обсуждение на стр. 35 (AMX.C);  
AMX NOPROPLIST удалить функции получения/установки свойств из AMXCORE.C;  
AMX NORANDOM удалить генератор псевдослучайных чисел из AMXCORE. C предположительно для замены его лучшим генератором псевдослучайных чисел;  
AMX TERMINAL для AMXCONS.C не используйте функции консоли (консоль Win32, ANSI/VT100 или обычная консоль);  
AMX TOKENTHREADING использовать «токен-потоки» (действительно только для реализаций GNU C/C++, Intel C/C++ и ассемблера);  
AMXAPI соглашение о вызовах интерфейсных функций; это переопределяет любые макросы CDECL или STDECL (AMX.C);  
AMXEXPORT соглашение о вызовах функций инициализации и очистки модулей расширения;  
ASM32 скомпилировать версию на ассемблере (AMX.C);  
CDECL устанавливает для AMXAPI значение \_\_cdecl для совместимости с ядром ассемблера (AMX.C);  
FIXEDPOINT для AMXCONS.C добавьте поддержку фиксированной точки, см. также параметр FLOATPOINT  
FLOATPOINT для AMXCONS.C добавьте поддержку с плавающей запятой, см. отдельный раздел ниже  
JIT добавить поддержку компилятора Just-In-Time (AMX.C);  
LINUX скомпилировать для Linux (или, возможно, для других версий Unix);  
NDEBUG компилировать без утверждений (все файлы);

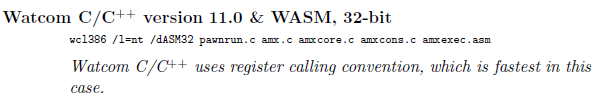
STDECL устанавливает для AMXAPI значение \_\_stdcall для совместимости с ядром ассемблера (AMX.C);  
UNICODE, UNICODE Включите Unicode в модуле ввода-вывода консоли и, возможно, в других вспомогательных библиотеках.  
VT100 для AMXCONS.C используйте терминальные коды ANSI/VT100 (неявно для Linux)  
Все примеры компиляции (перечисленные ниже) имеют столько параметров командной строки, сколько необходимо. Обратитесь к документации компилятора, чтобы добавить отладочную информацию или включить оптимизацию. Программа, которую компилирует каждый из примеров, — это SRUN, простой интерпретатор P-кода, который разрабатывается, начиная со страницы 6.  
Кроме того, файлы «проекта» и «рабочей области» для Microsoft Visual C/C++ (для компилятора и исходных файлов библиотеки Abstract Machine) можно найти в подкаталоге «msvc» того места, где установлен pawn.

• **ANSI C (see the GNU C section for Linux)**

Borland C++ version 3.1, 16-bit   
  
16-разрядный компилятор в пакете Borland C++ 5.0 имеет несколько ошибок генератора кода, поэтому либо используйте более раннюю версию компилятора Borland, либо компилируйте в 32-разрядной версии.  
LCC-Win32, 32-bit  
  
Microsoft Visual C/C++ version 5.0 or 6.0, 32-bit  
  
При работе с уровнем предупреждения 4, параметром «-W4», Visual C/C++ выдает несколько предупреждений о неиспользуемых аргументах функции.  
Watcom C/C++ version 11.0, 32-bit  
  
Приведенный выше список далеко не исчерпывающий. Абстрактная машина пешки переносима на многие компиляторы и многие операционные системы/архитектуры.

• **Assembler core for the AbstractMachine**Ассемблерная реализация абстрактной машины Марка Питера в настоящее время работает со всеми 32-битными компиляторами C для Microsoft Windows. Это (примерно) в пять раз быстрее, чем версия ANSI C. Как вы можете видеть в командной строке, файлы C требуют определения макроса ASM32.  
В реализации ассемблера есть две проблемы с «соглашением о вызовах» (в дополнение к упомянутым на стр. 72):  
⋄ Соглашение, с которым вызывается сам amx\_exec\_asm. Соглашение о вызовах по умолчанию — это соглашение о вызовах регистра Watcom. Для других компиляторов измените это на \_\_cdecl, установив макрос STACKARGS  
⋄ Соглашение о вызове «ловушек» (собственный диспетчер функций и обратный вызов отладчика). Опять же, по умолчанию используется соглашение о вызовах регистров Watcom. Используйте макросы CDECL или STDECL для \_\_cdecl и \_\_stdcall соответственно. (Поскольку STDCALL является зарезервированным словом на ассемблере, мне пришлось выбрать другое имя для макроса, следовательно, STDECL.) В AMX.C соглашение о вызовах для функций ловушки устанавливается с помощью макроса AMXAPI. Возможно, вам придется настроить макрос AMXAPI так, чтобы он не конфликтовал с соглашением о вызовах для функций ловушек, которые предполагает ядро ассемблера.

В архив включены два предварительно собранных объектных файла для тех из вас, у кого нет ассемблера (обратите внимание, что masm от Microsoft теперь находится в свободном доступе на веб-сайте Microsoft, и теперь также поддерживается бесплатный «Netwide assembler»). Два ассемблерных файла отличаются только используемым соглашением о вызовах. Ниже приведены имена файлов и команды, которые я использовал для их сборки:  
  
Оба предварительно скомпилированных ассемблерных файла были собраны из файла AMXEXEC.ASM (но с разными параметрами). Этот файл ассемблера совместим с Microsoft masm, Borland tasm и Watcom wasm. Ассемблер Netwide (nasm) имеет синтаксис, похожий на синтаксис masm/tasm/wasm, но несовместимый с ним. Файл «AMXEXECN.ASM» (обратите внимание на «N» после «AMXEXEC») представляет собой ту же реализацию ядра ассемблера для amx, но с использованием синтаксиса «Netwide assembler». Ассемблер Netwide — это бесплатный ассемблер, работающий на различных платформах.  
Версия кода amx на ассемблере Netwide не поддерживает соглашение Watcom о «регистрации вызовов» — она всегда использует \_\_cdecl для самой функции amx\_exec\_asm. Соглашение о вызовах для «ловушек» по умолчанию — \_\_cdecl, но его можно изменить на \_\_stdcall, установив макрос STDECL в командной строке nasm.  
У меня были проблемы с инкрементным компоновщиком при смешивании ассемблера с C/C++, как для компиляторов Borland, так и для компиляторов Microsoft. Когда программа дает сбой по загадочным причинам или когда отладчик показывает ассемблерный код или адреса переменных, которые явно не совпадают с соответствующим исходным кодом, сначала выполните полную сборку (и особенно полную «ссылку»).  


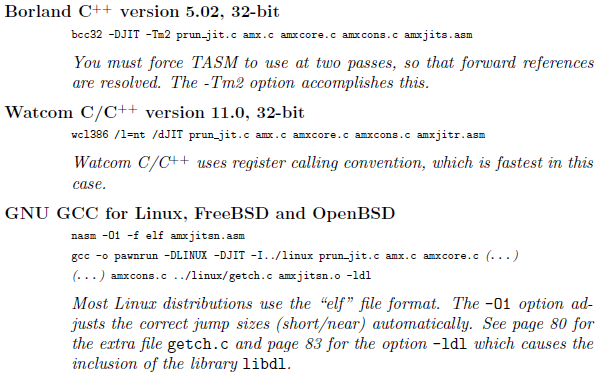


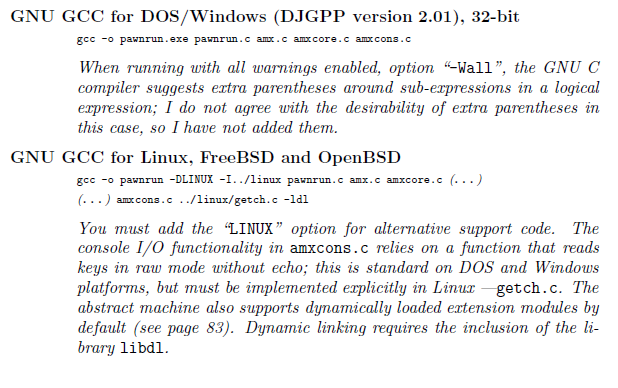
• **Just-In-Time compiler**Третий вариант — добавить компилятор Just-In-Time, а также вспомогательные процедуры. JIT компилирует P-код AMX в собственный машинный код во время выполнения. Полученный код более чем в два раза быстрее, чем ассемблерная версия абстрактной машины (которая уже была довольно быстрой). Чтобы добавить поддержку JIT, вы должны определить макрос «JIT» с помощью переключателя командной строки.

В дополнение к компиляции с определенным макросом JIT хост-приложение должно:  
⋄ установить AMX\_FLAG\_JITC в поле «flags» структуры AMX перед вызовом функции amx\_Init;  
⋄ вызвать функцию amx\_InitJIT после amx\_Init.

Функция amx\_InitJIT, в свою очередь, нуждается в двух дополнительных блоках памяти: один для собственных машинных инструкций, которые генерирует компилятор, а другой — для любых перемещений. После возврата amx\_InitJIT буфер таблицы перемещений может быть освобожден. Блок памяти, содержащий исходные инструкции P-кода пешки, больше не нужен и также может быть освобожден.

Особое внимание следует уделить блоку, который будет содержать инструкции собственного машинного кода: для блока должно быть установлено разрешение на выполнение машинного кода из блока памяти. В процессорах Intel любой блок памяти, который имеет «доступ для чтения», неявно имеет «доступ для выполнения». Чтобы заблокировать обработку переполнений буфера, которые позволяют выполнять произвольный код, AMD ввела бит «не выполнять» (NX) в дескрипторе страницы памяти, и Intel приняла этот дизайн, хотя и назвал его «выполнение запрещено» ( ХД). В операционной системе, в которой бит NX/XD установлен по умолчанию, вы должны убедиться, что в блоке памяти, в котором JIT-компилятор генерирует инструкции, бит NX/XD очищен.  
Самому JIT-компилятору требуется только доступ для чтения и записи к блоку памяти для собственных машинных инструкций (это значение по умолчанию для блока памяти, который вы выделяете). Выполнение JIT-компилируемого кода через amx\_Exec требует полного доступа к блоку памяти: чтение, запись и выполнение. Блоку требуется доступ для записи, потому что код операции SYSREQ.C заменяется на SYSREQ.D после первого поиска (это оптимизация, ищите адрес нативной функции только один раз). В Microsoft Windows функция VirtualAlloc может выделить блок памяти с полным доступом; в качестве альтернативы VirtualProtect может изменить права доступа к существующему блоку памяти. В версиях Linux, поддерживающих биты NX/XD, вы можете использовать vmalloc\_exec для получения блока с полным доступом или настроить права доступа к уже выделенному блоку с помощью функции mprotect. Если ваша версия Linux не предоставляет vmalloc\_exec, она, вероятно, не будет поддерживать бит NX/XD. Для процессоров или операционных систем, не поддерживающих бит NX/XD, выполнение кода неявно разрешено. Вы можете использовать стандартный malloc вместо VirtualAlloc и vmalloc\_exec.  
Во время компиляции JIT-компилятору требуется доступ для записи к своему собственному сегменту кода: JIT-компилятор вставляет параметры P-кода в свой собственный сегмент кода во время компиляции. Чтобы сделать эти исправления возможными, amx\_InitJIT временно разрешает «доступ для записи» в собственном сегменте кода для операционных систем, которым это требуется.  
amx\_Init дает консервативную оценку размера блока памяти, необходимого для компиляции собственного машинного кода. Консервативная оценка означает, что блок памяти гарантированно будет достаточно большим и, вероятно, будет намного больше, чем действительно необходимо. Когда amx\_InitJIT возвращается, он вычисляет реальный требуемый объем памяти. Поэтому для экономии памяти вы можете уменьшить или перераспределить блок памяти после возврата amx\_InitJIT.  
Инструментарий поставляется с исходным кодом prun\_jit.c, который является модификацией программы «pawnrun» (пример программы для встраивания абстрактной машины, см. стр. 6) для JIT-компилятора. В этом примере программы излагаются шаги, описанные выше.  
На самом деле существует три версии JIT, все они предназначены для процессорной архитектуры 80x86 (Intel Pentium, AMD Athlon):  
AMXJITR.ASM использует соглашения о вызовах на основе регистров и требует Watcom C/C++;  
AMXJITS.ASM использует соглашения о вызовах \_\_cdecl или \_\_stdcall (оба основаны на стеке) и должны работать с другими компиляторами Win32.  
AMXJITSN.ASM такой же, как AMXJITS.ASM, но реализованный в «nasm», что делает JIT-компилятор доступным для Linux и Unix-подобных операционных систем.

За исключением соглашений о вызовах и синтаксиса ассемблера, три версии JIT идентичны.  
Исходные файлы AMXJITR.ASM, AMXJITS.ASM и AMXJITSN.ASM содержат несколько определений, с помощью которых вы можете обменять производительность на другие параметры. См. исходные файлы для описания этих определений.  
  
JIT не поддерживает ловушку отладчика. То есть JIT компилирует P-код в собственный машинный код для процессора, но полностью пропускает код операции break. Таким образом, при запуске собственного машинного кода функция ловушки отладчика никогда не будет вызываться. Если вы хотите иметь решение, в котором код сценария может работать с максимальной скоростью и в то же время иметь возможность отлаживать сценарий, один из вариантов — включить в абстрактную машину как ядро ассемблера (см. стр. 75), так и JIT. . В этом случае в командной строке компилятора будут указаны макросы JIT и ASM32, и в проект будут добавлены как файлы ассемблера для JIT, так и ядро ассемблера.  
С комбинированным ядром JIT и ассемблера вы можете отлаживать сценарий, запуская ядро ассемблера, и запускать розничный код на JIT. Хост-приложение может решить, на каком «ядре» запускать сценарий, прежде чем вызывать amx\_Init.

• **Direct threaded interpreterwith GNU C extensions**Файл AMX.C содержит специальный код для компилятора GNU C (GCC), который делает абстрактную машину примерно в два раза быстрее, чем версия ANSI C. Однако ядро ассемблера и JIT еще быстрее.   
• **Adding a terminal to the abstract machine**Простой текстовый терминал часто удобен для пользователей продукта, поскольку он позволяет им распечатывать текстовые строки и вводить данные простым и понятным способом. Строки, напечатанные на консоли, также могут служить для пользователя средством отладки или отслеживания.  
Примеры консольных функций находятся в файле AMXCONS.C, они позволяют печатать форматированный текст и читать ввод с клавиатуры. Реализация консольного интерфейса по умолчанию записывает в стандартную консоль вывода для приложения «текстового режима»: это «ящик DOS» для Microsoft Windows и активный терминал для Linux/Unix. В Linux/Unix функции поддерживают терминал VT100, а в Microsoft Windows аналогичная функциональность эмулируется. Есть запасной вариант, использующий только функции стандартного C — это, конечно, накладывает ряд ограничений, но работает везде.

Для лучшего встраивания в приложение вы можете написать собственный терминал. В качестве примера того, как написать код поддержки, набор инструментов pawn поставляется с двумя альтернативными реализациями терминала:

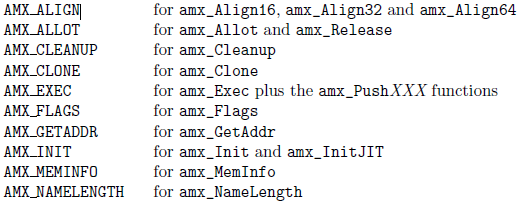
termwin Терминал для графических («оконных») приложений Microsoft Windows. Он может быть скомпилирован для использования консольного ввода-вывода ASCII/ANSI или Unicode. Хотя количество столбцов и строк фиксировано, окно терминала можно изменять в размерах и прокручивать, а также терминал позволяет масштабировать шрифт. Эта реализация поддерживает несколько параллельных терминалов.

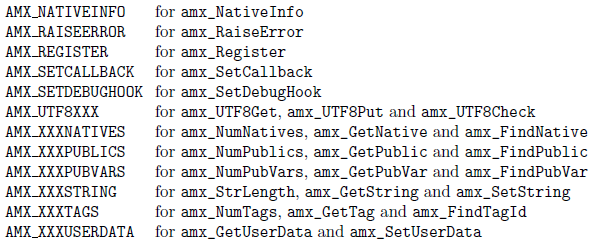
term ga Терминал, реализованный в кроссплатформенной библиотеке «GraphApp»; он работает на Microsoft Windows, Linux, FreeBSD и Macintosh. Этот терминал изначально поддерживает UTF-8, а также может быть скомпилирован с поддержкой Unicode («широкосимвольный»).  
Для компиляции со специальным терминалом необходимо отключить стандартную реализацию терминальных функций ввода-вывода в AMXCONS.C и добавить в проект исходный файл с нужным терминалом. Например, в Watcom C/C++ командная строка для использования termwin будет выглядеть так:  
  
Если вы компилируете терминал Microsoft Windows для Unicode, вам необходимо добавить определение макроса «UNICODE» в командной строке. Если вы хотите, чтобы терминал Unicode работал также и в Microsoft Windows 9x, вам необходимо связать библиотеку «unicows» («Microsoft Layer for Unicode» в Windows 95/98/ME). Подробную информацию о Unicode и unicows см. на сайте Microsoft.  
Использование терминала «GraphApp» требует лишь немного больше работы: GraphApp требует переопределения точки входа в программу (функция main). Самый простой способ запустить его — включить файл «grahpapp.h» в SRUN.C. Конечно, библиотеки GraphApp также должны быть скомпилированы.  
• **Support for floating point in the AbstractMachine**Определения определяемых пользователем операторов для подпрограмм с плавающей запятой находятся в файле «FLOAT.INC». Вы можете использовать арифметику с плавающей запятой в своих программах пешек, включив этот файл. Включаемый файл содержит определения собственных функций, которые выполняют основные операции с плавающей запятой, и определяемые пользователем операторы для их сопоставления с распространенными операторами сложения/вычитания/умножения/деления. См. буклет pawn «Язык» для получения дополнительной информации об определяемых пользователем операторах.

Абстрактная машина также должна поддерживать операции с плавающей запятой. Это требует двух или трех дополнений к компиляции абстрактной машины:  
1. необходимо определить макрос «FLOATPOINT» при компиляции исходных файлов;  
2. необходимо добавить файл FLOAT.C в список файлов  
3. в зависимости от компилятора/компоновщика C вам может потребоваться добавить параметр компилятора или файл библиотеки для компоновщика.

Эти два/три шага применимы ко всем «командным строкам компилятора», приведенным выше. Например, первая командная строка (ANSI C, с использованием 16-разрядного компилятора Borland C++) выглядит так:  
  
Компилятор Borland C++ не требует дополнительных опций для компиляции программ с плавающей запятой. Однако компилятор GNU GCC должен быть проинструктирован добавить «математическую» библиотеку на этапе компоновки с параметром -lm. Командная строка для GCC для Linux становится следующей:  


Кстати, поддержка фиксированной точки добавляется примерно таким же образом: вы добавляете макрос FIXEDPOINT в командную строку компилятора и включаете файл FIXED.C в список файлов. В вашей программе залога вы должны включить файл FIXED.INC для определений и определяемых пользователем операторов.

• **Compiling ‘‘dynamically loadable’’ modules**В приведенном выше разделе о добавлении плавающей запятой к абстрактной машине это было сделано путем статической компиляции/связывания поддержки во время выполнения. Альтернативой является компиляция абстрактной машины только с минимальным набором модулей расширения и собственных функций, а также создание дополнительных библиотек в виде динамически загружаемых модулей (или «плагинов»).  
Чтобы создать динамически загружаемый модуль расширения, файл C/C++, реализующий модуль, должен быть собран как DLL (MicrosoftWindows) или совместно используемая библиотека (Unix/Linux).  
В Microsoft Windows функции amx\_FilenameInit и amx\_FilenameCleanup должны быть помечены как «экспортированные», и они также должны иметь соглашение о вызовах «\_\_stdcall». Для этой цели AMX.H определяет макрос AMXEXPORT: рекомендуется, чтобы определения amx\_FilenameInit и amx\_FilenameCleanup были помечены этим макросом и чтобы вы установили для него соответствующее (зависящее от компилятора) соглашение о вызовах в командной строке компилятора.  
Кроме того, имена экспортируемых функций не должны быть «искажены». В проекте C++ файлы должны быть объявлены как extern "C", чтобы избежать искажения имен. Компиляторы для Microsoft Windows также обычно искажают функции C (например, amx\_PowerInit становится amx\_PowerInit@4), и это должно быть явно отключено с помощью файла компоновщика «.DEF» или параметра компилятора. Watcom C/C++ использует файл «.LBC» вместо файла .DEF.  
Усложнение в MicrosoftWindows, наряду с изменением имени, — это соглашение о вызовах. Для библиотек динамической компоновки обычно экспортируемые функции используют соглашение о вызовах «\_\_stdcall». Технически нативные функции не должны использовать то же соглашение о вызовах, что и экспортированные функции (amx\_FilenameInit и amx\_FilenameCleanup), но из соображений схожести и функциональной совместимости я советую вам также установить соглашение о вызовах нативных функций и «ловушки». функции для \_\_stdcall. Это, в свою очередь, означает, что абстрактный машинный код также должен быть построен с использованием соглашения о вызовах \_\_stdcall для собственных функций и функций-ловушек. Подробности см. на стр. 72 (установите для всех трех макросов AMX\_NATIVE\_CALL, AMXAPI и AMXEXPORT значение \_\_stdcall).  
Пример командной строки для создания модуля расширения «арифметики с плавающей запятой» в виде DLL для Microsoft Windows с использованием Borland C++ 5.0:  
  
Обратите внимание, что хост-программа теперь также должна использовать соглашение о вызовах \_\_stdcall для собственных функций и функций ловушек. Модуль расширения консольного ввода-вывода (AMXCONS.C) также содержит некоторую поддержку значений с фиксированной и плавающей запятой, которые должны быть включены отдельно — см. предыдущий раздел.  
Библиотека собственных функций, созданная как DLL/общая библиотека, должна быть связана с несколькими функциями в файле AMX.C, в частности с amx\_Register. Однако включать в модуль все функции из AMX.C — пустая трата места: маловероятно, например, что модуль будет вызывать amx\_Init или amx\_Exec. Чтобы удалить ненужные функции из AMX.C, определите макросы в командной строке компилятора, чтобы указать набор функций, которые вам нужны:  






CMake — это кроссплатформенная система make с открытым исходным кодом, которая генерирует «makefile» или файлы проектов для различных компиляторов и платформ. Он изначально работает в Linux, различных вариантах Unix и Windows (без Cygwin) и поддерживает различные компиляторы. Дополнительную информацию о CMake, а также бесплатно загружаемую копию можно найти на http://www.cmake.org/.  
Инструментарий pawn поставляется с двумя файлами проекта CMake. Первый строит компилятор как разделяемую библиотеку и «внешний интерфейс» консольного режима. Второй строит простую исполняемую программу, включающую абстрактную машину, простой консольный отладчик и различные модули расширения в виде общих библиотек. Файлы проекта CMake находятся в подкаталогах «compiler» и «amx», где установлен исходный код инструментария pawn. При распаковке исходного кода пешки из архива .ZIP или .TGZ файлы проекта CMake находятся в этих подкаталогах ниже каталога, в который вы распаковали архив.  
• **Microsoft Windows**1. Запустите CMakeSetup.

2. Выберите в качестве каталога исходного кода подкаталог «compiler» или «amx» в дереве каталогов для инструментария, в зависимости от того, что вы хотите собрать. Например, если вы установили инструментарий в папку C:\Pawn, исходным каталогом для компилятора будет C:\Pawn\source\compiler. Вы также можете собрать компилятор и абстрактную машину за один прогон, выбрав исходный каталог верхнего уровня в качестве исходного каталога CMake.

3. Выберите в качестве места назначения подкаталог «bin» или любой другой каталог по вашему выбору. Makefile (или файлы проекта) будут сгенерированы в целевом каталоге.

4. Также выберите используемый компилятор. В Microsoft Windows CMake поддерживает компиляторы Microsoft и Borland, а также GNU GCC.

5. Нажмите кнопку «Настроить». После первоначальной настройки элементы могут отображаться красным цветом. Этим CMake указывает, что эти элементы могут нуждаться в корректировке, но в случае пешки это требуется редко. Нажмите «Настроить» еще раз для окончательной конфигурации.  
6. Нажмите кнопку «ОК». Это закрывает программу CMakeSetup после создания ряда файлов в целевом подкаталоге.

7. Соберите программу обычным способом. Для Microsoft Visual C/C++ CMake создал проект Visual Studio и файлы «Workspace»; для других компиляторов CMake создает make-файл.

• **Linux / Unix**

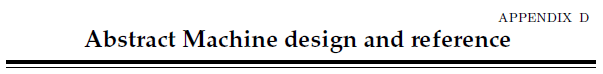
1. Перейдите в каталог, в котором вы хотите сгенерировать make-файл (и собрать исполняемый файл). Как правило, это будет подкаталог «bin» в дереве каталогов для инструментария. Например, если вы установили инструментарий в /opt/Pawn, подкаталог bin для компилятора будет /opt/Pawn/bin. Если вы установили pawn как «root», вам также нужно быть root, чтобы перекомпилировать pawn.

2. Запустите «ccmake ../source/compiler» или «ccmake ../source/amx», в зависимости от того, что вы хотите собрать (компилятор или абстрактную машину). Вы можете собрать оба одновременно с помощью «ccmake ../source».

3. Нажмите клавишу «c» для «настройки». После первоначальной настройки у вас могут быть элементы в списке со знаком «\*» перед их значением. Этим CMake указывает, что эти элементы могут нуждаться в корректировке, но в случае пешки это требуется редко. Введите «c» еще раз для окончательной конфигурации.

4. Нажмите кнопку «g» для «создать и выйти». Затем соберите программу, набрав «make».

5. При желании вы также можете выполнить «make install», чтобы скопировать исполняемые файлы по пути, выбранному с помощью ccmake.



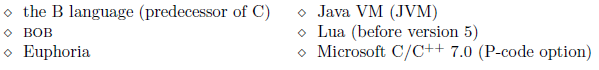
Первый вопрос: зачем вообще абстрактная машина? При компиляции на родном машинном языке процессора по вашему выбору производительность будет намного выше.  
Есть только одна реальная причина использовать абстрактную машину: кросс-платформенная совместимость скомпилированного бинарного кода. На момент разработки этой пешки для меня были важны как 16-битные, так и 32-битные платформы на процессорах серии 80x86. К тому времени, когда я смогу забыть о 16-разрядных операционных системах, альтернативные микропроцессоры (такие как PowerPC и DEC Alpha) могут стать необходимыми.

Другими причинами (хотя и не существенными) являются:

⋄ Гораздо проще поддерживать работу программы на абстрактной машине внутри ее «песочницы». Например, неограниченная рекурсия в абстрактной машине приводит к сбою самой абстрактной машины, но не более того. Если вы запускаете собственный машинный код, рекурсивная процедура может повредить системный стек и привести к сбою приложения. Хотя современные операционные системы поддерживают многопоточность с отдельным стеком для каждого потока, действие по умолчанию при переполнении любого стека по-прежнему заключается в завершении работы всего приложения.  
⋄ Легче разработать язык, в котором объект данных (массив) может содержать P-код, который затем выполняется. Современные операционные системы разделяют разделы кода и данных: вы не можете писать в раздел кода и не можете выполнять данные; то есть не без серьезных усилий.

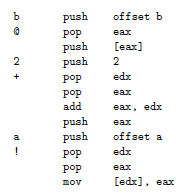
Текущий пеший язык не имеет возможности выполнять P-код из массива, но абстрактная машина не слишком тесно связана с языком. То есть будущие версии языка пешек могут предоставить средства для выполнения потока кода из переменной, не требуя от меня перепроектирования абстрактной машины.

Моей первой попыткой спроектировать абстрактную машину было рассмотрение текущих реализаций. Похоже, что реализация абстрактных машин в виде стековых машин является своего рода традицией, даже несмотря на то, что дизайн микропроцессоров переместился в сторону реализации на основе регистров. Все абстрактные машины, с которыми я сталкивался, основаны на стеке. Это включает:

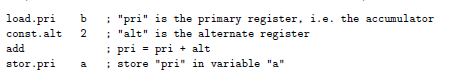
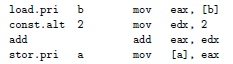




Стековые машины, безусловно, компактны, гибки и просты в реализации, но их также сложнее оптимизировать по скорости. Чтобы понять почему, давайте проанализируем конкретный пример.

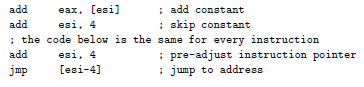
  
Native code  
В 32-битном ассемблере это будет:  
  
Stack based abstract machine  
Форт — это архетип стековой машины, поэтому я буду использовать его в качестве примера. Та же процедура в Форте будет выглядеть так:  
  
где каждая буква — это инструкция («@» означает «выборка», а «!» — «сохранение»; обратите внимание, что стековые машины выполняют код в «обратной шлифовальной нотации»). Итак, это шесть инструкций в P-коде, но код расширяется до:  
  
Два наблюдения: 1. стековая машина сильно использует память (плохо для производительности) и 2. расширенный код довольно велик по сравнению с собственным кодом (12 инструкций против 3).  
Расширенный код — это то, что может сделать из него компилятор «точно в срок» (JIT) (хотя можно ожидать, что оптимизирующий JIT несколько уменьшит избыточные «проталкивания» и «выталкивания»). При запуске кода на абстрактной машине абстрактная машина также должна расширять код, но, кроме того, у нее есть накладные расходы на выборку и декодирование инструкций. Эти накладные расходы составляют как минимум две собственные инструкции на инструкцию P-кода (подробнее об этом позже). Для шести инструкций P-кода к 12 собственным инструкциям расширенного кода следует добавить еще 12 собственных инструкций. И все же пример сильно упрощен, потому что код работает в системном стеке и использует системное адресное пространство.  
Другими словами, абстрактная машина, основанная на стеке, выполняет собственный фрагмент кода из 3 инструкций в 6 инструкциях P-кода, которые, как оказывается, требуют 24 собственных инструкций и даже больше, если вы хотите запустить абстрактную машину в своем собственном стеке и в своем собственном стеке. собственное (защищенное) пространство данных.

Register-based abstract machine  
Микропроцессоры использовали регистры с момента их теоретического создания фон Нейманом. Распространение этой архитектуры на абстрактную машину вполне естественно. Есть два преимущества: инструкции абстрактной машины лучше сопоставляются с собственными инструкциями (вы фактически можете использовать регистры процессора для реализации регистров абстрактной машины) и количество виртуальных инструкций, необходимых для выполнения простого выражения, может быть уменьшено.

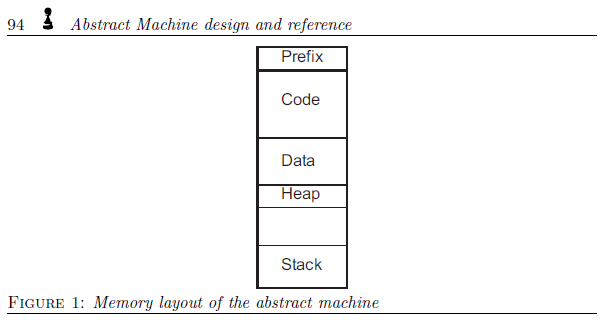
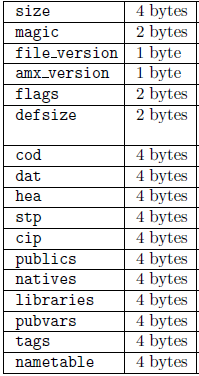
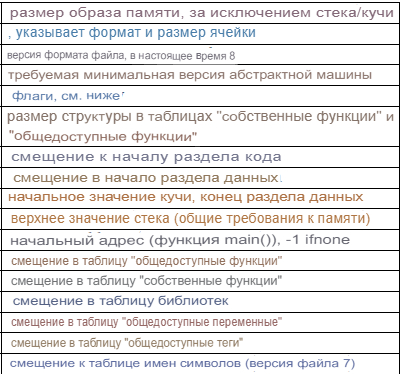
В качестве примера, вот код пешки «amx», двухрегистровой абстрактной машины (amx означает «Abstract Machine eXecutor»):  
  
  
В расширенном коде это будет:  


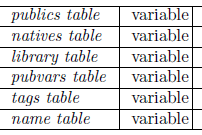
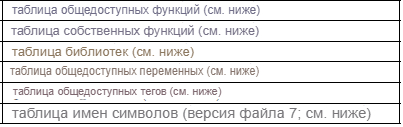
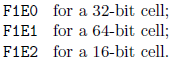
Четыре инструкции байт-кода прекрасно соответствуют собственным инструкциям. Здесь снова нам придется добавить накладные расходы на выборку и декодирование инструкций P-кода (2 собственные инструкции на инструкцию P-кода). По сравнению с абстрактной машиной на основе стека абстрактная машина на основе регистра работает в два раза быстрее; в 12 собственных инструкциях по сравнению с 24 собственными инструкциями для абстрактной машины на основе стека.  
Более того: по моему опыту, абстрактные машины на основе стека легче оптимизировать по размеру, а абстрактные машины на основе регистров легче оптимизировать по скорости. Таким образом, абстрактная машина на основе регистров действительно может быть в два раза быстрее абстрактной машины на основе стека.  
Немного подробнее об оптимизации: я намеренно решил добавить «2» к переменной. Увеличение или уменьшение значения на единицу или два — настолько распространенный случай, что в Forth есть специальный оператор для них: слово «2+» добавляет 2 к значению. Предполагая, что хорошая (на основе стека) абстрактная машина также имеет специальные коды операций для общих операций, использование этого слова «2+» вместо общих слов «2» и «+» удаляет одну инструкцию P-кода и 3 собственные инструкции. Это уменьшит количество встроенных инструкций до 21. Однако тот же прием оптимизации применим к абстрактной машине на основе регистров. Абстрактная машина пешки имеет код операции «add.c», который добавляет постоянное значение в первичный регистр. Оптимизированная последовательность будет следующей:  
  
что приводит к 3 собственным инструкциям плюс 6 инструкций служебных данных для выборки и декодирования инструкций P-кода. Абстрактная машина на основе регистров (которой требуется 9 собственных инструкций) по-прежнему примерно в два раза быстрее, чем абстрактная машина на основе стека (с 21 собственной инструкцией).  
• **Threading**В «токеновом» интерпретаторе каждый код операции является индексом в таблице, которая содержит «адрес перехода» для каждой инструкции. В «прямопоточном» интерпретаторе код операции — это сам адрес перехода. Прямая многопоточность часто требует, чтобы все коды операций были перемещены на адреса перехода при компиляции или при загрузке предварительно скомпилированного файла. Формат файла абстрактной машины пешки разработан таким образом, что возможны как потоки маркеров, так и прямые потоки.  
Потоковая абстрактная машина обычно пишется на ассемблере, потому что большинство языков высокого уровня не могут хранить адреса меток в массиве. Однако компилятор GNU C (GCC) расширяет язык C с помощью унарного оператора «&&», который возвращает адрес метки. Этот адрес можно сохранить в переменной типа «void \*» и использовать позже в инструкции goto. По сути, следующий фрагмент кода делает то же самое, «идти домой»:  
  
Версия абстрактной машины ANSI C использует большой оператор switch для выбора правильных инструкций для каждого кода операции. Из-за прямой многопоточности версия абстрактной машины GNU C работает примерно в два раза быстрее, чем версия ANSI C. К счастью, GNU C работает на многих платформах. Это означает, что быстрая версия GNU C по-прежнему достаточно переносима.  
• **Optimizing in assembler**В следующем обсуждении предполагается процессор Intel 80386 или совместимый. Эта же методика применима и к 16-битным процессорам, и к процессорам других марок, но имена (и количество) регистров будут другими.  
Выгодно использовать регистры процессора для реализации регистров абстрактной машины. Детали абстрактной машины для пешечной системы следуют далее в этом приложении. Дальнейшие предположения:  
⋄ PRI — псевдоним регистра процессора EAX, а ALT — EDX.

⋄ ESI — указатель кодовых инструкций (CIP)

⋄ EDI указывает на начало сегмента данных, ECX — это указатель стека (STK), EBX — указатель кадра (FRM), а EBP доступен как промежуточный регистр общего назначения; остальные регистры в amx (STP и HEA) являются локальными переменными.  
К каждому коду операции прикреплен набор машинных инструкций, а также трейлер, который переходит к следующей инструкции. Трейлер идентичен для каждого кода операции. В качестве примера ниже приведена реализация опкода ADD.C:  
  
Обратите внимание, что «трейлер», который связывается со следующей инструкцией через (прямую) поточность, состоит из двух инструкций; этот трейлер был источником предпосылки служебных данных с двумя инструкциями для выборки и декодирования инструкций в более раннем анализе.

При реализации абстрактной машины можно дополнительно оптимизировать последовательности вручную. В приведенном выше примере две инструкции «добавить esi, 4», конечно, можно сложить в одну инструкцию, добавляющую 8 к ESI.

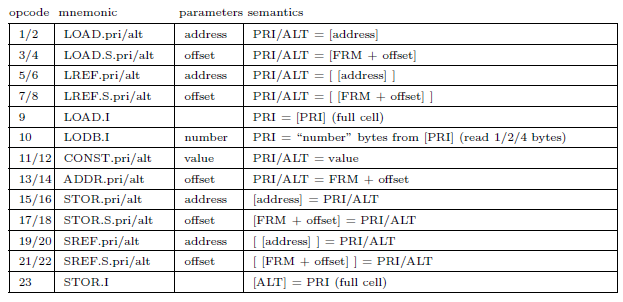
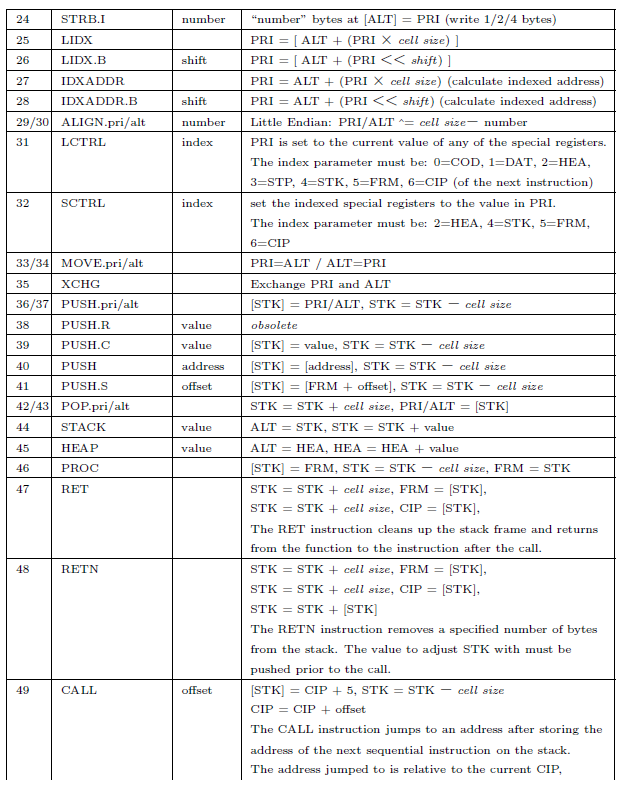
**Abstract Machine reference**Абстрактная машина состоит из набора регистров, предлагаемой (или навязанной) структуры памяти и набора инструкций. Каждый обсуждается в отдельном разделе.  
• **Register layout**Абстрактная машина имитирует двухрегистровый процессор. В дополнение к двум регистрам общего назначения у него есть несколько внутренних регистров. Ниже приведен список с именами и описанием всех регистров:  
PRI первичный регистр (ALU, общего назначения).  
ALT альтернативный регистр (общего назначения).  
FRM указатель кадра стека, операции чтения и записи памяти относительно стека относятся к адресу в этом регистре.  
CIP указатель инструкции кода.  
DAT смещение к началу данных.  
COD смещение к началу кода.  
STP вершина стека.  
STK индекс стека, указывает текущую позицию в стеке. Стек идет вниз от регистра STP к нулю.  
HEA указатель кучи. Динамически выделяемая память поступает из кучи, а регистр HEA указывает вершину кучи.  
Примечательно, что в наборе регистров отсутствует регистр «флагов». Абстрактная машина не хранит отдельного набора флагов; вместо этого все условные переходы выполняются в зависимости от содержимого регистра pri.  
• **Memory image**Куча и стек совместно используют блок памяти. Стек растет вниз от stp к нулю; куча растет вверх. Исключение возникает при конфликте регистров stk и hea. (Исключение означает, что абстрактная машина завершает работу с сообщением об ошибке. В настоящее время нет механизма перехвата исключений.)  
На рис. 1 показана предполагаемая схема образа памяти, которую стандартная абстрактная машина предполагает для автономной «работы» amx. Возможны альтернативные планировки. Например, когда вы «клонируете» задание amx, новое задание будет разделять префикс и разделы кода с исходным заданием, а разделы данных/кучи/стека будут находиться в другом блоке памяти. В частности, реализация может хранить кучу и стек в отдельном блоке памяти рядом с блоком памяти для кода, данных и префикса. Верхняя часть рисунка представляет самый низкий адрес в памяти.  
Бинарный файл (на диске) состоит из «префикса», разделов кода и данных. Разделы кучи и стека не хранятся в бинарном файле, абстрактная машина может построить их из информации в разделе «префикс». Префикс также содержит информацию о запуске и определения собственных и общедоступных функций.  
Символическая (отладочная) информация может следовать за разделами кода и данных в файле. Эта символическая информация обычно не считывается в память (по крайней мере, абстрактной машиной). Подробности см. в приложении E.  
  
Все многобайтовые значения в префиксе сохраняются с младшим байтом по младшему адресу (Little Endian или «младший байт первым»). Порядок байтов в сгенерированном коде и разделах данных — либо в Little Endian, либо в компактном кодировании — см. стр. 104 для получения подробной информации о компактном кодировании.  
 

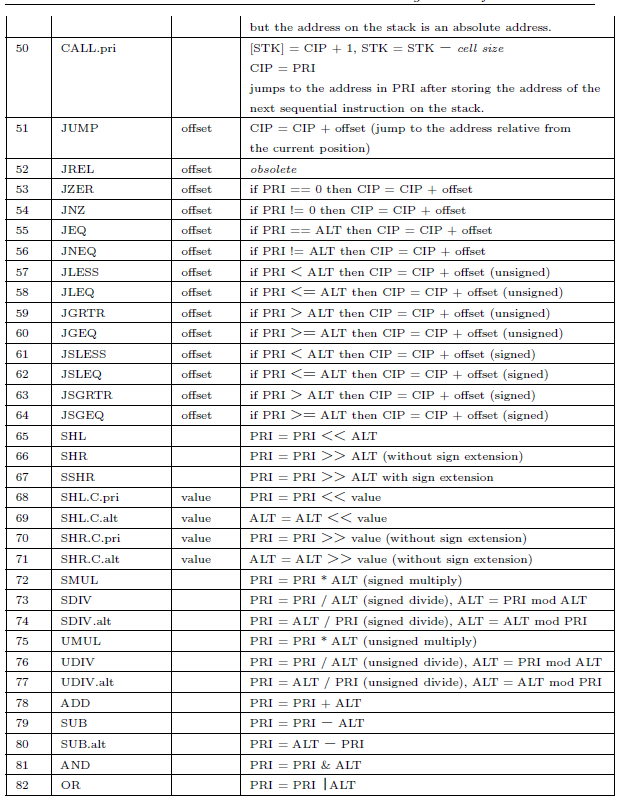
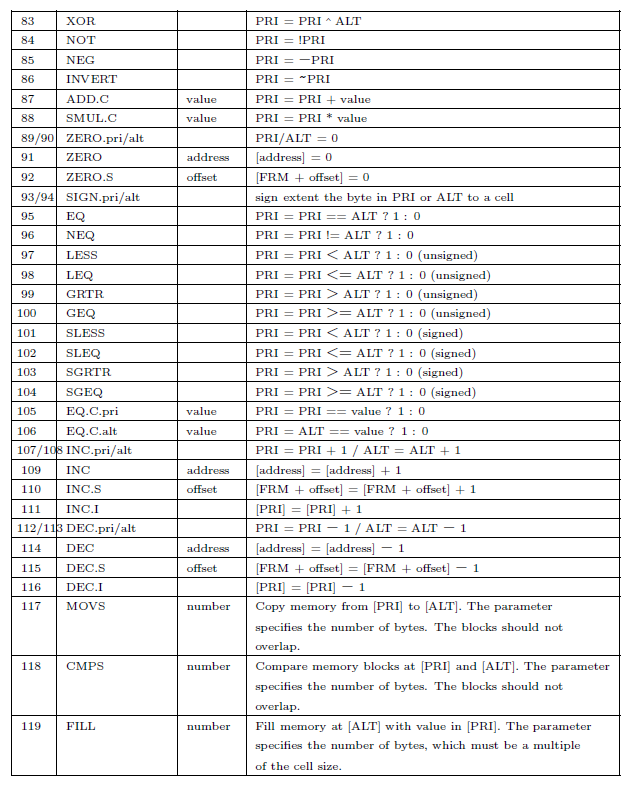
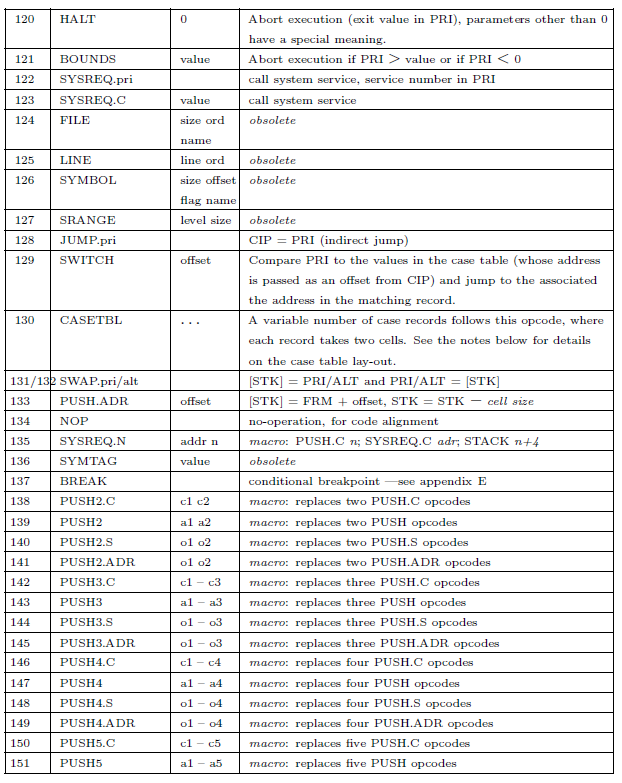
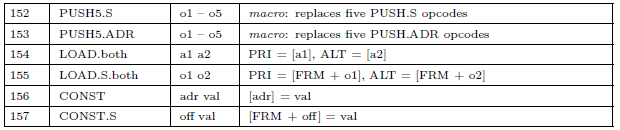
   
  
  
Магическое значение указывает размер ячейки в P-коде скомпилированной программы. Это значение (в шестнадцатеричном формате):  
  
Каждый бит в поле флагов содержит одну настройку. В настоящее время определены следующие биты (биты, которые не упоминаются, в настоящее время не определены):  
0 reserved - в настоящее время не используется  
1 (AMX\_FLAG\_DEBUG) если установлено, файл содержит символическую (отладочную) информацию  
2 (AMX\_FLAG\_COMPACT) если установлено, файл сжимается с помощью «компактной кодировки»\ — см. стр. 104  
3 reserved - в настоящее время не используется  
4 (AMX\_FLAG\_NOCHECKS) if set, the code has no debug support at all (no array

bounds-checking, no assertions, no line-tracing support)  
14 reserved — этот бит используется внутри  
15 reserved — этот бит используется внутри

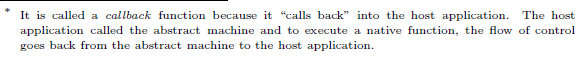
За фиксированной частью префикса следует ряд таблиц. Каждая таблица содержит ноль или более записей. Таблица имен имеет переменный размер записи; размер записей в других таблицах указан в поле defsize в префиксе. Чтобы найти количество записей в таблице, вычтите смещение до таблицы из смещения до следующей таблицы и разделите полученное значение на defsize. Например, количество записей в таблице туземцев:   
P-код следует сразу за префиксом, но обратите внимание, что префикс может быть дополнен для выравнивания разделов кода и данных (это параметр компилятора). Поле cod в заголовке — это смещение файла до начала P-кода.

В версиях от 0 до 6 файлов P-кода записи в таблице общедоступных функций имеют формат:  
  
  
Как видно, имя публичной функции присутствует в записи. Максимальная длина имени общедоступной функции ограничена размером записи (за вычетом количества байтов в ячейке для байтов, занимаемых полем адреса).  
Формат таблицы собственных функций очень похож (см. ниже, это снова формат для версий файлов 0–6). Порядок записей в таблице важен, потому что параметр инструкции SYSREQ.C является индексом в таблице собственных функций.  
Таблица библиотеки имеет тот же формат, что и таблица собственных функций. Поле «адрес» используется внутренне и должно быть равно нулю в файле. Поле «имя» содержит имя библиотеки.  
Таблица «общедоступных переменных», опять же, имеет такое же расположение записей, как и таблица общедоступных функций. Поле адреса общедоступной переменной содержит адрес переменной относительно раздела dat.

Таблица «tags» также использует тот же формат. Эта таблица содержит только те теги, имя или номер которых могут быть полезны для основного приложения или модулей расширения: теги, которые используются с инструкциями выхода или перехода в спящий режим или используются с оператором tagof. Поле адреса записи тега содержит идентификатор тега.  
Начиная с версии файла 7, скомпилированный файл включает «таблицу имен». Эта таблица содержит имена символов для символов, на которые ссылаются другие таблицы. Каждое имя находится в записи переменного размера в виде строки, заканчивающейся нулем. Преимущество этой схемы в том, что она позволяет использовать произвольно длинные имена символов, сохраняя при этом эти имена в компактном виде.  
Поскольку имена символов больше не нужно хранить в таблицах для общедоступных и собственных функций, общедоступных переменных, тегов и библиотек, записи для этих таблиц также изменились. Вместо поля имени записи содержат 4-байтовое смещение относительно начала «префикса» файла до начала имени символа в таблице имен. Размер записи в заголовке, «defsize», устанавливается равным размеру одной ячейки плюс смещение в 4 байта, т.е. 8 для реализации 32-битной ячейки и 12 для реализации 64-битной ячейки. Ниже приведено определение нативной/общедоступной функции/переменной в формате файла 7.  
  
• **Instruction reference**Каждая инструкция состоит из кода операции, за которым следует ноль или один параметр. Каждый код операции имеет размер один байт; параметр инструкции имеет размер ячейки (обычно четыре байта). Несколько «отладочных» инструкций (в конце списка) составляют исключение из этих правил: они имеют два или более параметра, и эти параметры не всегда имеют размер ячейки.  
Многие инструкции подразумевают регистры в качестве операндов. Это уменьшает количество операндов, необходимых для декодирования инструкции, и, следовательно, сокращает время, необходимое для декодирования инструкции. В некоторых случаях подразумеваемый регистр является частью имени кода операции. Например, PUSH.pri — это имя кода операции, в котором регистр pri хранится в стеке. Эта инструкция не имеет параметров: ее параметр (pri) подразумевается в имени кода операции.  
Ссылка на инструкцию упорядочена по коду операции. Описание двух опкодов иногда объединяется в одной строке таблицы, потому что опкоды различаются только исходным или целевым регистром. В этих случаях коды операций и варианты регистров разделяются знаком «/».  
Столбец «семантика» дает краткое описание того, что делает опкод. Он использует синтаксис языка C для операторов, который такой же, как и в языке пешек. Элемент в квадратных скобках указывает на доступ к памяти (относительно регистра dat, за исключением инструкций перехода и вызова). Итак, PRI=[адрес] означает, что значение, считанное из памяти по адресу DAT+адрес, хранится в pri.  
  


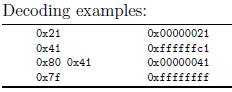
• **Branching**За некоторыми исключениями, инструкции перехода (вызов, переход и т. д.) используют относительные целевые адреса. Параметры этих кодов операций являются смещениями относительно адреса самого кода операции. Смещение может быть как положительным, так и отрицательным. Использование относительных адресов переходов делает двоичный код «независимым от позиции».

Исключениями являются: инструкции ret и retn, выполняющие переход к абсолютному адресу, который был сохранен в стеке предыдущей инструкцией вызова, и инструкции jump.pri и call.pri, выполняющие переход к вычисляемому адресу.  
• **Macro instructions**Для повышения скорости и уменьшения объема памяти скомпилированных программ абстрактная машина включает несколько макроинструкций. Эти макроинструкции представляют собой последовательность «простых» инструкций в одном коде операции. Это, конечно, уменьшает размер памяти, потому что теперь один код операции заменяет два или более простых кода операции; это также повышает производительность благодаря уменьшению накладных расходов декодера P-кода внутри абстрактной машины. Обычные коды операций не имеют параметров или имеют один параметр, тогда как макрокоды операций имеют два или более кодов операций. Несколько кодов операций отладки в приведенной выше таблице также имели более одного параметра, но теперь все эти коды операций устарели. Макроинструкции несовместимы с текущими JIT-компиляторами — поскольку JIT-компилятор устраняет накладные расходы на декодирование инструкций, макрокоманды не имеют преимуществ перед простыми кодами операций при использовании JIT-компилятора. Компилятор пешки может дополнительно отключить генерацию макроинструкций.  
• **Opcode packing**Последние версии компилятора pawn поддерживают «упакованные коды операций». Это инструкции, в которых код операции и его параметр упакованы в одну ячейку. Все упакованные коды операций имеют один параметр. Концепция упакованных кодов операций — это «оптимизация пространства», чтобы уменьшить размер, который работающий скрипт занимает в памяти. Упаковка кода операции требует потоковой передачи токена.  
• **Native call opcodes**Есть два кода операции, которых нет в таблице кодов операций. Они называются SYSREQ.D и SYSREQ.ND. Эти коды операций являются вариантами прямого вызова SYSREQ.C и SYSREQ.N соответственно. Компилятор pawn никогда их не генерирует, поэтому их нет в таблице. Эти коды операций генерируются самой абстрактной машиной.  
Когда сценарий вызывает собственную функцию, текущая версия компилятора pawn генерирует код операции SYSREQ.N, а более старые версии генерируют код операции SYSREQ.C. Оба этих кода операции вызывают переход от абстрактной машины к подпрограмме, которая обрабатывает диспетчеризацию собственных функций. Вы можете установить такую подпрограмму с помощью amx\_SetCallback, но есть также подпрограмма по умолчанию, называемая amx\_Callback. Функция обратного вызова/диспетчера должна искать собственную функцию в параметре исходного кода операции SYSREQ.\*, а затем вызывать эту собственную функцию с переданными параметрами функции. В этой цепочке есть двойной вызов: опкод SYSREQ.\* вызывает вызов callback-функции, которая затем вызывает запрошенную нативную функцию.  
Коды операций SYSREQ.D и SYSREQ.ND удаляют один вызов и тем самым улучшают производительность собственной ссылки вызова. После того, как функция обратного вызова нашла адрес собственной функции, она вставляет этот адрес прямо в поток кода скомпилированного скрипта и изменяет код операции SYSREQ.N на SYSREQ.ND или код операции SYSREQ.C на SYSREQ.D. для старых систем. При следующем вызове этой нативной функции появляется новый код операции, который вызывает адрес нативной функции напрямую, минуя обратный вызов.  
Этот «трюк» работает, только если вы используете обратный вызов по умолчанию или если вы реализуете аналогичную функциональность исправления в своем пользовательском обратном вызове. Также требуется, чтобы поток P-кода был доступен для записи. Если вы сохраните раздел кода скомпилированного сценария в (Flash) ПЗУ, функция обратного вызова не сможет исправить коды операций.  
  
• **Compact file format**

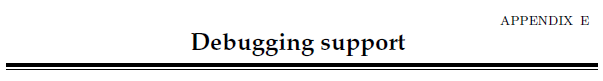
Компилятор pawn генерирует выходной P-код либо в виде простого дампа кодов операций, либо в кодировке переменной длины, аналогичной файлам MIDI «SMF». «Простая» кодировка использует Little Endian для всех кодов операций в качестве слов данных, а это означает, что процессор Big Endian должен поменять местами все ячейки, которые он считывает из файла P-кода, перед их выполнением. Альтернатива, «компактные двоичные файлы», не только имеет уменьшенный размер, формат файла также является универсальным для компьютеров Big Endian и Little Endian.

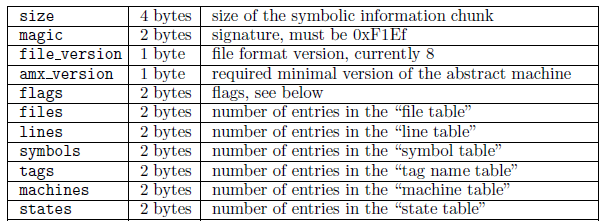
Заголовок модуля (см. стр. 94) и все таблицы (публичные функции, нативные функции, общедоступные переменные библиотек) не сжимаются — они всегда в формате Little Endian. Данные, которые следуют за этими таблицами, кодируются кодами переменной длины: каждая четырехбайтовая ячейка кодируется от одного до пяти байтов.

Старший бит каждого байта является битом «продолжения». Если он установлен, следуют другие байты с семью более значащими битами. Старшие 7 битов сохраняются первыми (по младшему смещению файла/адресу памяти). Когда последовательность байтов декодирована, бит 6 (следующий по значимости бит) первого байта повторяется, чтобы заполнить полные 32 бита.

  
• **Cross-platform support**В абстрактной машине есть некоторый уровень кроссплатформенной поддержки. Адресация памяти как с прямым порядком байтов, так и с прямым порядком байтов.

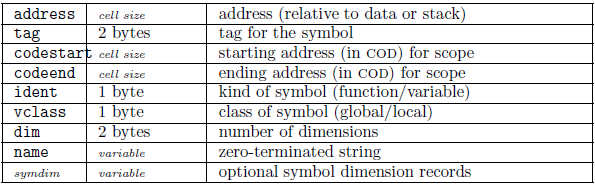
схемы широко используются сегодня. Big Endian — это «сетевой порядок байтов», поскольку он используется для различных сетевых протоколов, в частности для набора интернет-протоколов. Серии ЦП Intel 80x86 и Pentium используют адресацию с прямым порядком байтов.  
Абстрактная машина оптимизирована для манипулирования «ячейками», 32-битными величинами. Байты или 16-битные слова могут быть прочитаны или записаны только косвенно, сначала сгенерировав адрес, а затем используя инструкции LODB.I или STRB.I. Инструкция ALIGN.pri помогает сгенерировать адрес.  
Абстрактная машина предполагает, что когда в ячейке упаковано несколько символов, первый символ занимает старшие биты в ячейке, а последний символ — младшие биты ячейки. Вот как язык пешек хранит упакованные строки.  
На компьютере с обратным порядком байтов порядок символов является «естественным» в том смысле, что первый символ пакета находится по младшему адресу, а последний символ — по старшему. На компьютере с прямым порядком байтов порядок символов обратный. При доступе ко второму символу пакета вы должны читать/писать с более низкого адреса, чем при доступе к первому символу пакета.  
Компилятор пешки может легко сгенерировать необходимый дополнительный код для настройки адреса для каждого символа в наборе. Недостатком будет то, что модуль, написанный для компьютера с обратным порядком байтов, не будет работать на компьютере с прямым порядком байтов, и наоборот. Поэтому вместо этого компилятор пешки генерирует специальную инструкцию ALIGN, семантика которой зависит от того, работает ли абстрактная машина на компьютере с прямым порядком байтов или с прямым порядком байтов. В частности, инструкция ALIGN ничего не делает на компьютере с прямым порядком байтов и выполняет простую побитовую операцию «исключающее или» на компьютере с прямым порядком байтов.  
• **The ‘‘switch’’ instruction and case table lay-out**Инструкция переключения сравнивает значение pri со значением case в каждой записи в соответствующей таблице case и, если находит совпадение, переходит к адресу в совпадающей записи. Код операции switch имеет один параметр, который представляет собой адрес таблицы прецедентов относительно cip (указатель инструкции). По этому адресу должен появиться опкод casetbl.

Каждая запись в таблице вариантов, кроме первой, содержит значение варианта и адрес перехода в указанном порядке. Адрес перехода относится к адресу самой записи. Первая запись хранит количество последующих записей в таблице прецедентов в своей первой ячейке и адрес перехода «нет совпадения» во второй ячейке. Если ни одно из значений case последующих записей не совпадает с pri, инструкция переключения переходит к этому «несовпадающему» адресу. Обратите внимание еще раз, что первая запись исключается из поля «количество записей» в первой записи.  
Записи в таблице прецедентов сортируются по их значению. Абстрактная машина может использовать эту схему для поиска в таблице с помощью бинарного поиска.  
  
Поддержка отладки включает два компонента: коды операций break и символьный формат информации. Компилятор pawn вставляет код операции break перед любой последовательностью инструкций, которая запускает оператор в исходном коде. Следовательно, когда настроена ловушка отладки, функция ловушки вызывается до того, как абстрактная машина выполнит P-код для оператора.

Когда хук отладки служит полным (символьным) отладчиком, ему нужно просмотреть символьную информацию для исходных файлов. Компилятор прикрепляет символическую информацию к двоичному файлу P-кода. Символическая информация состоит из заголовка и нескольких таблиц переменного размера.  


После заголовка в следующем порядке:

⋄ файловая таблица  
⋄ линейная таблица  
⋄ таблица символов  
⋄ таблица имен тегов  
⋄ таблица имен машин  
⋄ таблица названий штатов

Когда в какой-либо таблице нет записей, сама таблица полностью отсутствует в файле. Большинство таблиц имеют записи переменной длины, а это означает, что вам нужно просмотреть символьную информацию, чтобы найти конкретную запись.  
• **The file table**Записи в файле имеют следующий формат:  
  
  
Поле адреса содержит адрес в сегменте кода, с которого начинается сгенерированный P-код для файла. Вся таблица сортируется по этому полю адреса. Имея адрес относительно псевдорегистра cod, вы можете найти файл, из которого был сгенерирован P-код по этому адресу. P-код из одного файла может попасть в несколько диапазонов адресов в P-коде — например, посредством включения файлов. В таком случае в таблице файлов есть несколько записей для одного и того же файла.  
• **The line table**Таблица строк содержит номера строк; он используется в сочетании с файловой таблицей:  
  
Эта таблица сопоставляет адреса в P-коде с номерами строк. Имена файлов (относительно которых номера строк) должны быть найдены в таблице файлов. Таблица строк сортируется по полю адреса. Адрес — это наименьший адрес, с которого начинается сгенерированный P-код для строки исходного кода. Обычно в этой строке появляется инструкция разрыва.  
• **The symbol table**Записи для символа (переменной, функции) более сложные:  
  
Адрес относится либо к сегменту кода (cod), либо к сегменту данных (dat), либо к фрейму текущей функции, чей адрес находится в псевдорегистре frm. Поля ident и vclass указывают, к какому псевдорегистру относится адрес, и позволяют определить местонахождение значения символа. Возможные значения поля ident:  
1 переменная

2 «ссылка», переменная, содержащая адрес другой переменной (другими словами, указатель).

3 массив

4 ссылка на массив (указатель на массив)

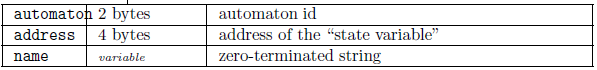
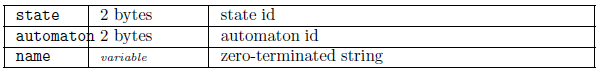
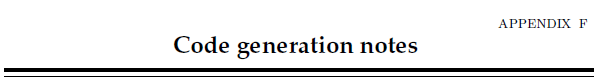
9 функция

10 ссылка на функцию (указатель на функцию) Значения поля vclass:

0 символ относится к глобальной переменной (относительно dat) или к функции (относительно cod)

1 символ относится к локальной переменной с относительным адресом стека (относительно frm; поле адреса может быть положительным или отрицательным)

2 символ указывает на «статическую» локальную переменную; адрес относится не к стеку, а к dat  
Адреса codestart и codeend относятся к псевдорегистру cod. Эти адреса дают диапазон адресов, в котором символ находится «в области действия». Для локальных переменных это диапазон адресов составного блока, в котором происходит объявление переменной. Глобальные переменные имеют конечный адрес, равный последнему допустимому адресу P-кода в файле. Для функций начальный и конечный адреса представляют собой диапазон адресов, который принимает P-код функции. Для функции поля address и codestart равны.  
Поле тега представляет собой числовой идентификатор тега. Имя тега можно найти в таблице тегов.  
Когда символ является массивом или ссылкой на массив, поле dim указывает количество измерений и количество записей «размер символа» (symdim), которые следуют за символом. Когда массив имеет два измерения, dim равен 2, и две записи symdim следуют за полем имени, заканчивающимся нулем. Каждая запись symdim имеет формат:  


Когда поле «размер» записи symdim равно нулю, размер массива (для этого измерения) не определен.  
• **The tag name table**Таблица имен тегов позволяет искать имя тега по идентификатору:  
  
  
Отладчик может использовать имя тега для выбора подходящего формата отображения. Например, когда отладчик определяет, что имя тега символа — «Float:», он может автоматически отображать значение символа как значение с плавающей запятой, а не (неверно) интерпретировать его как целое число.  
Таблица «тегов» в «префиксе» абстрактной машины (см. рис. 1 и стр. 94) также содержит отображение идентификаторов тегов в имена тегов. Таблица в области префикса содержит только «экспортированные» теги (которые могут понадобиться любой хост-программе). Таблица в отладочной информации содержит все теги, которые используются в программе.  
• **The automaton table**Для сценариев, определяющих несколько автоматов, таблица автоматов содержит имена всех автоматов. Компилятор пешки также генерирует «переменную состояния» для каждого автомата. Адрес этой переменной (относительно псевдорегистра dat) также есть в таблице автомата.  
  
• **The state table**Для каждого состояния, определенного в сценарии, есть запись в таблице состояний. Каждое состояние относится к автомату. Если автомат не определен явно, состояние использует идентификатор автомата 0 (ноль).  
  
• **Functions**Чтобы упростить просмотр отладочной информации, набор инструментов pawn поставляется с функциями поддержки, которые выполняют основные задачи поиска функций, переменных и номеров строк исходного кода по адресам. Эти функции находятся в файлах amxdbg.c и amxdbg.h. Эти функции предоставляются в качестве «примеров реализации» — вы можете (или должны) адаптировать их к своей среде.  
  
Генерация кода пешечного компилятора довольно проста (в том числе из-за простоты абстрактной машины). Стоит упомянуть несколько моментов:

⋄ У абстрактной машины есть инструкции, которые компилятор пешки в настоящее время не генерирует. Например, инструкция LREF.pri работает как оператор разыменования («\*») в C/C++. pawn не поддерживает указатели напрямую, но ссылки — это просто замаскированные указатели. Однако pawn поддерживает только ссылки в аргументах функции, а это означает, что «операции с указателями» в pawn всегда относятся к стеку. Другими словами, компилятор pawn не генерирует инструкцию LREF.pri, хотя генерирует инструкцию LREF.S.pri. Абстрактная машина довольно независима от пешечного языка, даже если они были разработаны друг для друга. Язык пешек может легко расширяться в будущем, возможно, с типом переменной «ссылка», тем самым давая инструкции LREF.pri причину существования. Однако абстрактная машина не может легко расти, потому что новые инструкции немедленно делают новую абстрактную машину несовместимой с предыдущими версиями. То есть программы, скомпилированные для новой абстрактной машины, не будут работать в более ранней версии.

⋄ Для нативной функции компилятор pawn генерирует инструкцию SYSREQ.C вместо обычного вызова функции. Параметр инструкции SYSREQ.C является индексом в таблице собственных функций. Функция в pawn очищает свои аргументы, которые были помещены в стек, потому что она возвращается с инструкцией RETN. Инструкция SYSREQ.C не удаляет элементы из стека, поэтому компилятор пешки делает это явно с инструкцией STACK после инструкции SYSREQ.C.

Аргументы нативной функции помещаются в стек так же, как и для обычной функции. В «пешеходной» реализации абстрактной машины (см. стр. 6) инструкции «системного запроса» связаны с установленной пользователем функцией обратного вызова. Таким образом, нативная функция в программе-заложнике выполняет вызов определяемой пользователем функции обратного вызова в абстрактной машине.

⋄ При вызове функции программа-заложник помещает аргументы функции в стек в обратном порядке (то есть справа налево). Он заканчивает список аргументов функции в стеке, помещая в стек количество байтов, которые были помещены в стек. Поскольку компилятор пешки передает функции только аргументы размером с ячейку, количество байтов равно количеству аргументов, умноженному на размер ячейки.

Заложенная функция заканчивается инструкцией RETN. Эта инструкция удаляет аргументы функции из стека.

⋄ Когда функция имеет «ссылочный» аргумент со значением по умолчанию, компилятор выделяет место для этого значения по умолчанию в куче. Для функции, которая имеет аргумент массива со значением по умолчанию, компилятор выделяет место для значения массива по умолчанию в куче. Однако, если аргумент массива (со значением по умолчанию) также является константой, компилятор пешки передает массив по умолчанию напрямую (здесь нет необходимости делать копию в куче, так как функция не будет пытаться изменить аргумент массива и , таким образом перезаписывая значение по умолчанию).

⋄ Аргументы функции, которая имеет «переменные аргументы» (обозначается оператором ..., см. брошюру «Язык»), всегда передаются по ссылке. Для констант и выражений, не являющихся lvalue, компилятор копирует значения в ячейку, выделенную из кучи, и передает адрес ячейки в функцию.

⋄ Для инструкции «switch» компилятор pawn генерирует код операции switch и таблицу case с кодом операции casetbl. Таблица прецедентов создается в сегменте трески; они считаются данными «только для чтения». «Несовпадающий» адрес в таблице прецедентов переходит к инструкции прецедента по умолчанию, если таковая имеется. Блоки кейсов в закладке не пропускаются. В конце каждой инструкции в списке case компилятор pawn генерирует переход к метке «выход» сразу после инструкции переключения. Компилятор pawn генерирует таблицу case между кодом для последнего case и меткой выхода. При этом каждый случай, включая случай по умолчанию, перемещается по таблице случаев.

⋄ Многомерные массивы реализованы в виде векторов, содержащих смещения к подмассивам. Например, двумерный массив с четырьмя «строками» и тремя «столбцами» состоит из одномерного массива с четырьмя элементами, где каждый элемент представляет собой смещение к трехэлементному одномерному массиву. Общий объем памяти массива составляет 4 +4×3 ячейки. Многомерные массивы в pawn аналогичны массивам указателей в C/C++. Как указано выше, «главное измерение» многомерных массивов содержит смещения к подмассивам. Это смещение указывается в байтах (не в ячейках) и относится к адресу ячейки, из которой было считано смещение. Возвращаясь к примеру двумерного массива с четырьмя строками и тремя столбцами (при условии, что размер ячейки равен четырем байтам), блок памяти, выделенный для массива, начинается с массива из четырех ячеек для «строк», за которым следует четыре массива с тремя элементами в каждом. Первый массив «column» начинается с четырех ячеек позади массива «rows», и поэтому первый элемент массива «rows» содержит значение 4×cellsize (16 для 32-битной ячейки). Второй массив столбцов начинается с трех ячеек позади первого массива столбцов, что на семь ячеек отстает от начала массива строк. Смещение второго столбца массива хранится во втором элементе массива строк, а смещение второго столбца относительно второй ячейки массива строк равно шести ячейкам. Таким образом, второе значение в массиве строк равно 6 × размер ячейки.  
Для конкретного примера предположим, что массив объявлен как:  
  
Последовательность значений в памяти для этого массива, где суффикс «c» в числе означает, что значение должно быть масштабировано для размера ячейки в байтах, выглядит следующим образом:  
  
Для трехмерного массива элементы вектора для основного измерения содержат смещения векторов для каждого дополнительного измерения.

⋄ Оператор деструктора принимает на вход массив с одним измерением, и этот массив содержит все элементы переменной, которые должны быть уничтожены:

• Для простых переменных переменная передается по ссылке, поэтому она выглядит как массив с одним элементом.

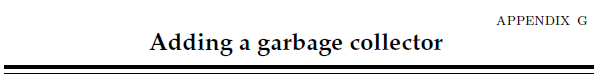
• Для массивов с одним измерением массив передается без изменений

• Для массивов с двумя или более измерениями оператор деструктора получает адрес за «таблицами косвенного обращения» для основных измерений. Как описано выше, многомерный массив начинается с векторов для основных измерений, каждое из которых содержит смещения к измерению, находящемуся под ним. Данные для самого массива упакованы за этими массивами смещения. Передавая адрес, где начинаются данные массива, оператор деструктора может получить доступ к элементам массива, как если бы это был массив с одним измерением.

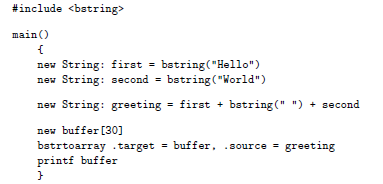
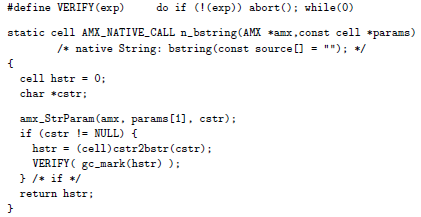
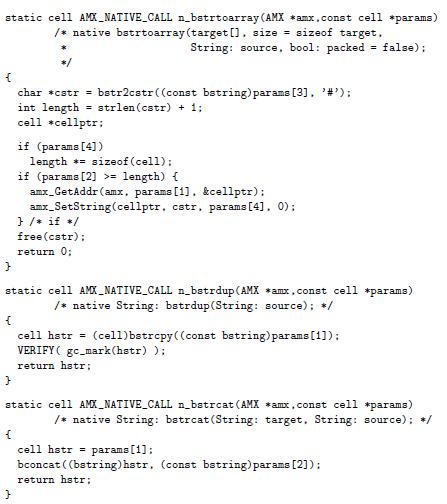
⋄ Начиная с версии 2.0 компилятор pawn помещает код операции HALT в начало кода (т. е. по адресу кода 0). Прежде чем перейти к точке входа (функции), абстрактная машина помещает в стек нулевой адрес возврата. Когда точка входа возвращается, она возвращается к нулевому адресу и видит инструкцию HALT.

⋄ Инструкция сна генерирует код операции HALT с кодом ошибки 12 («сон»). Когда абстрактная машина видит этот специальный код ошибки, она сохраняет состояние стека/кучи (а не сбрасывает его), чтобы иметь возможность перезапустить абстрактную машину.

⋄ Компилятор pawn добавляет в ассемблерный файл специальные комментарии (с формами «;$exp» и «;$par»), чтобы помочь оптимизатору-глазку принимать правильные решения. Эти комментарии отмечают конец «оператора выражения» или конец параметра функции. Код, сгенерированный компилятором, не переносит значение регистра из одного оператора/выражения в другое, и оптимизатор глазка использует эту информацию, чтобы избежать сохранения регистров, значения которых в любом случае не будут использоваться снова.

⋄ Для функций, которые имеют состояния, компилятор создает таблицу переходов по адресу функции; каждая запись переходит к конкретной реализации. Технически эта таблица представляет собой «таблицу вариантов», аналогичную той, которая используется для оператора «переключатель», и она действительно использует код операции переключения для перехода к правильной реализации. Случай «по умолчанию» указывает на резервную функцию или на точку выхода из ошибки, если резервная функция отсутствовала. Кроме того, эта схема более эффективна, чем закодированный вручную переключатель переменной состояния, потому что закодированный вручную переключатель должен находиться внутри собственной функции, используя дополнительный кадр функции и дополнительный стек для хранения другого возврата. адрес и форвард и параметры.   
pawn использует только статическое размещение для всех своих объектов. Преимущество статического распределения заключается в том, что требования к памяти для скрипта pawn легко определить (компилятор pawn делает это с опцией -d2), а объем памяти и производительность во время выполнения становятся полностью детерминированными. Тем не менее, для работы с объектами динамического размера в пешке очень удобен сборщик мусора. В этом приложении описывается, как можно добавить сборщик мусора в хост-приложение, использующее инструментарий pawn. Он реализован в виде отдельной библиотеки.  
• **How to use**Целью сборщика мусора является уведомление вашей программы об объектах, которые больше не используются и, следовательно, могут быть освобождены. Для этого сборщику мусора нужна структура данных для регистрации используемых объектов, а также способ уведомлять основное приложение (вашу программу) о избыточных объектах. Эти два элемента должны быть инициализированы до того, как вы начнете использовать сборщик мусора. Структура данных, которая записывает объекты, которые «используются», представляет собой хеш-таблицу. Его размер должен быть степенью двойки — на самом деле, параметр, который вы передаете функции gc\_settable, — это «мощность». То есть передача 10 в качестве аргумента gc\_settable создает хеш-таблицу, содержащую 210 или 1024 элемента. Существует нижняя граница размера 128 элементов, что означает, что параметр экспоненты должен быть не менее 7. Максимальный размер хэш-таблицы — это максимальное значение целого числа со знаком: 32 767 для 16-разрядных платформ и 2 147 483 648 для 32-разрядных платформ. битовые платформы (максимальный показатель степени равен 15 или 31 для 16-битных и 32-битных платформ соответственно). Второй параметр gc\_settable — это набор флагов. Единственный флаг, определенный на момент написания этой статьи, — GC\_AUTOGROW, который сообщает сборщику мусора, что он может автоматически увеличивать размер хеш-таблицы, когда она заполняется. Для каждого объекта, на который больше нет ссылок ни в одной из просканированных абстрактных машин, сборщик мусора вызывает функцию обратного вызова, чтобы освободить его. Но сначала вы, конечно, должны зарегистрировать эту функцию обратного вызова. Это вы делаете с помощью функции gc\_setcallback. По замыслу сигнатура для функции обратного вызова была сделана совместимой со стандартной бесплатной функцией C. Если ваша хост-программа выделяет свои объекты с помощью malloc, вы можете установить стандартную функцию free в качестве обратного вызова сборщика мусора. Если вам нужен дополнительный код очистки или если вы не выделяете объекты с помощью malloc, вы должны написать соответствующий обратный вызов.

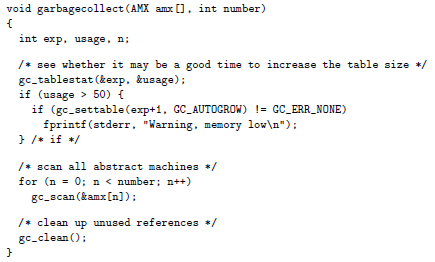
Как только хэш-таблица и обратный вызов установлены, ваша хост-программа (или ваша собственная библиотека функций) может выделять объекты и помечать их как «используемые» с помощью функции gc\_mark. Значение, которое вы передаете, должно быть ненулевым значением, которое однозначно идентифицирует объект, и оно должно быть типом данных «ячейка» — типом данных языка пешек. Если размер указателя такой же, как размер ячейки (или меньше), вы можете пометить указатель на объект (просто приведя его к ячейке). Другие механизмы заключаются в том, что вы выделяете объект из списка, который вы поддерживаете внутри, и «отмечаете» индекс объекта в этом списке. Важно, чтобы вы отмечали точно такое же значение, которое нативная функция возвращает скрипту пешки. Время от времени, по таймеру или в любой другой удобный момент, хост-программа должна вызывать gc\_scan один или несколько раз, после чего следует один вызов gc\_clean. Перед завершением gc\_clean вызывает функцию обратного вызова для каждого объекта, на который больше нет ссылок. Параметр функции обратного вызова — это то же значение, которое вы передали gc\_mark для функции. Функция gc\_scan обнаруживает «живые» объекты, функция gc\_clean очищает любой неживой объект. Хост-приложение может одновременно запускать несколько сценариев и, следовательно, может иметь несколько абстрактных машин, существующих в любое время. Сборщик мусора собирает ссылки на объекты для всех объектов, которые были выделены для всех абстрактных машин. При выполнении сборки мусора программа должна сканировать все абстрактные машины (используя gc\_scan) и завершать работу одним вызовом gc\_clean. Когда абстрактная машина исчезает, все объекты, выделенные для этой абстрактной машины (на которые не ссылаются другие абстрактные машины), очищаются при последующем запуске сборки мусора — просто потому, что gc\_scan не вызывается на абстрактной машине, которая исчезла. В конце программы вызовите gc\_settable с нулевым размером. Ранее я писал, что существует нижняя граница входного значения gc\_settable, равная 7, но нулевое значение — это особый случай. Кроме того, gc\_settable внутренне вызывает gc\_clean, когда экспонента таблицы равна нулю, чтобы удалить любой оставшийся объект в таблице.

• **Rescaling the garbage collector**Сборщик мусора построен на хеш-таблице, которая выделяется динамически. Хэш-таблица — это структура данных, которая позволяет быстро искать. Он делает это, вычисляя значение индекса из некоторого произвольного свойства объекта и сохраняя ссылку на объект по этому вычисляемому индексу в таблице. Для сборщика мусора индекс вычисляется из параметра «значение», которое вы передаете в функцию gc\_mark.  
Хеш-таблица не должна быть слишком маленькой, потому что она не может хранить больше объектов, чем помещается в таблицу, и она не должна быть слишком большой, так как это приведет к трате памяти и снижению производительности. Сборщик мусора делает размер таблицы регулируемым: вы можете начать работу с маленькой таблицы и увеличивать ее по мере необходимости. При желании вы также можете уменьшить размер хеш-таблицы. Увеличение или уменьшение хеш-таблицы сохраняет объекты, находящиеся в настоящее время в таблице.  
Проблема с хеш-таблицами в целом заключается в «коллизиях»: два разных объекта могут получить один и тот же индекс в хеш-таблице. Существуют различные стратегии выхода из этой ситуации; сборщик мусора использует самый простой: «прощупывание». Если происходит столкновение, новый объект сохраняется не с его вычисленным индексом, а с фиксированным смещением от вычисленного индекса. Чтобы избежать кластеров в таблице, смещение уменьшается примерно с четверти размера таблицы (за исключением таблиц, превышающих 64 КБ) до 1; чтобы избежать «слепых пятен» в таблице, смещение зондирования всегда является простым числом.  
Когда хеш-таблица заполнена, gc\_mark может сначала попытаться увеличить таблицу (в зависимости от того, был ли установлен GC\_AUTOGROW при вызове gc\_settable). Он возвращается с кодом ошибки, если увеличение таблицы не удается или если это не разрешено. Затем хост-программа может выполнить сборку мусора в надежде, что это освободит некоторые слоты в хеш-таблице; хост-программа также может попытаться увеличить саму хэш-таблицу. Поскольку хэш-таблица выделяется динамически, попытка изменить ее размер также может завершиться неудачно. Конечным результатом является то, что gc\_mark может выйти из строя, и ваша хост-программа не сможет восстановиться после этого.  
Однако можно избежать непоправимого сбоя gc\_mark: вместо того, чтобы ждать заполнения таблицы, хост-программа может решить увеличить таблицу задолго до того, как она заполнится. Если это не удастся, gc\_mark все равно будет успешным, и следующие несколько вызовов gc\_mark также будут успешными. Следовательно, хост-приложение имеет возможность освободить память или сообщить пользователю о «недостаточно памяти» — сообщение, которое более дружелюбно, чем сообщение «недостаточно памяти, не может продолжаться».  
Есть еще одна причина, по которой ранний рост хеш-таблицы является хорошей стратегией: производительность. Линейное зондирование — это простой метод борьбы с коллизиями, но он также приводит к сильному снижению производительности при заполнении хеш-таблицы. Вероятно, лучше всего, когда использование хеш-таблицы не превышает 50%. Функция gc\_tablestat возвращает текущую «загрузку» хеш-таблицы в процентах от ее размера.  
• **An example implementation**Чтобы использовать сборщик мусора в примере, мы должны сначала иметь собственную библиотеку функций, которая создает мусор. Для этого примера я выбрал «Better String library» Пола Хси, библиотеку, которая позволяет работать с динамически выделяемыми строками переменной длины в C/C++.  
Первым шагом является создание функций-оболочек для подмножества библиотеки. Для демонстрации сборщика мусора я выбрал минимальное подмножество, достаточное для запуска приведенной ниже примера программы — в реальных приложениях вы бы добавили значительно больше функций:  
  
Две основные встроенные функции, реализованные ниже, выполняют преобразование в массивы пешек или из них: n\_bstring и n\_bstrtoarray. Преобразование из массива в тип «bstring» (из библиотеки Better String) необходимо для обработки литеральных строк; преобразование обратно в массив пешек необходимо, потому что собственные функции в модуле расширения «консольный ввод-вывод» не поддерживают тип bstring. Опять же, на практике вы, вероятно, модифицировали бы printf и другие встроенные функции для работы с bstring, так что обратное преобразование в массивы пешек никогда не потребуется.  
  
  
Функции-оболочки, которые выделяют новые экземпляры bstring, отличаются от обычных функций-оболочек тем, что они вызывают gc\_mark. Обратите внимание, что функциям-оболочкам, которые не создают новых экземпляров bstring, не нужно помечать объект для сборщика мусора.  
Проверка ошибок в этом примере примитивна. Когда хэш-таблица сборщика мусора заполнена и не может расти, программа просто прерывается. Как обсуждалось в предыдущем разделе, рекомендуется увеличивать таблицу задолго до того, как она заполнится. Теперь мы должны изменить хост-приложение, чтобы настроить сборщик мусора. В моем случае это адапта  
Инициализация сборщика мусора — простой шаг, потому что освободитель памяти для «библиотеки Better String» совместим с функцией обратного вызова сборщика мусора. Все, что нужно сделать, это вставить следующие строки где-нибудь перед вызовом amx\_Exec:



Очистить сборщик мусора перед выходом тоже легко:  
  
Очистить сборщик мусора перед выходом тоже легко:  


Более сложная часть запускает сборщик мусора в нужное время. С одной стороны, вам захочется регулярно вызывать сборщик мусора, чтобы в таблице не было слишком много «мусора»; с другой стороны, слишком частый вызов сборщика мусора снижает общую производительность. На самом деле было бы лучше, если бы сборщик запускался в то время, когда загрузка ЦП низкая.

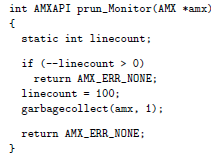
Даже если мы просто хотим вызывать сборщик мусора через регулярные промежутки времени, незначительная проблема заключается в том, что нет переносимого способа сделать это. В Linux и Unix вы можете использовать функции сигнала и тревоги, а в Microsoft Windows может быть полезна функция SetTimer. Многопоточность — еще один вариант, но имейте в виду, что вы должны реализовать доступ «взаимное исключение» самостоятельно (например, с семафорами или критическим разделом).  
Функция, выполняющая сборку мусора, может быть похожа на приведенную ниже. Функция ожидает, что абстрактные машины будут сканировать массив. Он увеличивает хеш-таблицу, когда ее использование превышает 50%.  
  
С целью предоставить полный пример, который компилируется и работает на всех платформах, которые в настоящее время поддерживает инструментарий pawn, я «прицепил» функцию «garbagecollect» (реализованную выше) к хуку отладки. То есть ведущее приложение устанавливает ловушку отладки, а функция ловушки отладки вызывает сбор мусора при различных событиях. Делать это в чем-либо, кроме демонстрационной программы, не рекомендуется по нескольким причинам:

⋄ Ловушка отладки может отслеживать только одну абстрактную машину, тогда как в реальных проектах у вас, скорее всего, будет несколько параллельных абстрактных машин.

⋄ Для вызова сборщика мусора через регулярные промежутки времени лучше всего отслеживать код операции DBG\_LINE. Однако этот отладочный код никогда не будет отправлен, если сценарий был скомпилирован без отладочной информации.

⋄ Обработчик отладки не учитывает нагрузку на систему, тогда как сборка мусора должна выполняться, особенно когда система не занята.

⋄ Ловушка отладки несет некоторые накладные расходы (хотя и небольшие). Что позади нас, ниже находится отладочный хук, который вызывает сборщик мусора. Он вызывает сборщик мусора после выполнения каждых 100 строк и после каждого возврата функции. Воздействие на код DBG\_RETURN позволяет обойти проблемы для скриптов пешки, которые скомпилированы без отладочной информации.



• **Other notes**Как обсуждалось ранее, функция gc\_clean вызывает функцию обратного вызова для освобождения любого объекта, который больше не используется на какой-либо абстрактной машине, которая была просканирована. Функция предполагает, что обратный вызов действительно освобождает объект: он не будет сообщать об этом снова.  
  
Каждый объект должен быть в хеш-таблице только один раз. Если вы вызываете gc\_mark со значением, которое уже есть в хеш-таблице, функция возвращает ошибку. Это не фатальная ошибка, но тем не менее лучше не добавлять один и тот же указатель/объект дважды в таблицу сборки мусора.  
Алгоритм проверки, используемый сборщиком мусора, отличается как от хорошо известных линейных, так и от квадратичных алгоритмов проверки, но его свойства (относящиеся к кластеризации или «слипанию») аналогичны свойствам квадратичной проверки.  
Дизайн хорошей хэш-функции/уравнения — еще одна повторяющаяся тема в исследованиях. Поскольку обсуждаемый здесь сборщик мусора является универсальным, нельзя ничего предполагать относительно входного ключа (параметр gc\_mark). Алгоритм генерации хэша, используемый в сборщике мусора, «складывает» значение ключа несколько раз, в зависимости от размера «ячейки» и размера хеш-таблицы. Сворачивание означает, что значение ключа делится пополам, и две половины объединяются операцией «исключающее ИЛИ». Конкретно, если показатель степени хэш-таблицы (первый параметр gc\_settable) меньше или равен 16, 32-битное значение ключа сначала разбивается на два 16-битных значения, а затем верхняя половина обрабатывается по принципу «исключающее ИЛИ». к первой половине, в результате чего получается одно 16-битное значение — новый ключ. Когда показатель степени таблицы меньше или равен 8, свертывание происходит дважды.  
  
Часто источником значения ключа является указатель. В типичных менеджерах памяти младшие биты фиксированы. Например, обычно память выделяется по адресу, кратному 8 байтам, чтобы обеспечить оптимальное выравнивание данных. Функция хэш-таблицы пытается выполнить копирование с этим конкретным аспектом, меняя местами все биты младшего значащего байта.