Использование памяти

В этой секции описывается, как HAL использует память и организовывает код, данные, стек и прочие секции логической памяти в физической памяти.

Секции памяти

По умолчанию, системы, основанные на HAL, компилируются, используя генерируемый скрипт компилятора, который создаётся в Nios II SBT. Скрипт компилятора контролирует размещение кода и данных в доступных секциях памяти. Автогенерируемый скрипт компилятора создаёт стандартные секции кода и данных (.text, .rodata, .rwdata и .bss), плюс секцию под каждое устройство физической памяти в системе. Например, если компонент памяти, называемый sdram, определён в файле **system.h** – это означает секцию памяти, называемую .sdram.

На рис. 6-3 показана организация обычной схемы памяти в HAL.

Устройства памяти, которые содержат адреса сброса и исключений процессора Nios II, - это отдельный случай. Инструментальная конструкция Nios II 32-байтной секции .entry, начинается по адресу сброса. Эта секция специально зарезервирована под использование обработчиком сброса. Схоже выглядит инструментальная конструкция секции .exceptions, начинающаяся по адресу исключений.

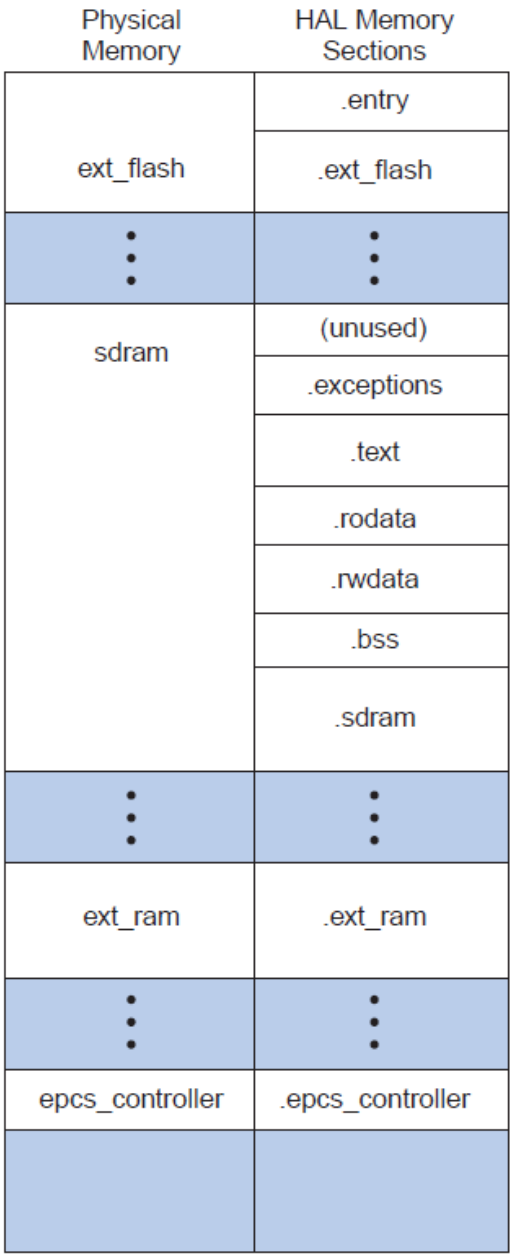
В устройстве памяти, содержащем адреса сброса и исключений, компилятор создаёт обычную (нерезервируемую) секцию памяти над секцией .entry или .exceptions. Если существует регион памяти выше секции .entry или .exceptions, то он не доступен для программы Nios II. На рис. 6-3 показан недоступный регион памяти, выше секции .exceptions.

Назначение кода и данных разделам памяти

В этой секции описывается, как контролировать размещение кода и данных программы в заданных секциях памяти. В основном, процесс разработки Nios II по умолчанию задаёт осмысленное разделение. Однако вы можете захотеть изменить разделение в отдельных случаях.

Например, для улучшения характеристик, общей практикой является размещение критичного к характеристикам кода и данных в RAM с быстрым временем доступа. Также принято отлаживать работу системы программой, размещённой в RAM, но при сбросе процессора загружать релиз программы из флеш памяти. В таких случаях вы должны вручную задать размещение кода по секциям памяти.

Figure 6–3. Sample HAL Link Map



Простые опции размещения

Код обработчика сброса всегда расположен в начале раздела .reset. Главный код обработки исключений всегда является первым кодом в секции, содержащей адрес исключения. По умолчанию оставшийся код и данные делятся на несколько следующих секций:

- .text – весь оставшийся код
- .rodata – данные только для чтения
- .rwdata – данные для чтения и записи
- .bss – данные, инициализированные нулём

Вы можете контролировать размещение .text, .rodata, .rwdata и других разделов памяти, манипулируя настройками BSP. За подробной информацией о контроле над настройками BSP обратитесь к секции "Настройки HAL BSP" на стр. 6-2.

Редактор Nios II BSP является самым удобным способом манипулировать схемой памяти компилятора. Редактор Nios II BSP графически отображает секции памяти и назначения регионов, позволяя вам видеть перекрывающиеся или неиспользуемые секции памяти. Редактор BSP доступен в Nios II SBT на Eclipse или в командной строке Nios II SBT.

За подробной информацией обратитесь к главе "Начало работы с командной строкой" в настольной книге программиста Nios II.

Расширенные опции размещения

В своём коде программы вы можете задать специальную область памяти под каждый фрагмент кода. В Си и C++ вы можете использовать атрибуты секций. Эти атрибуты должны быть размещены в качестве прототипа функции; вы не можете разместить их в декларации функции.

Код в примере 6-16 размещает переменную *foo* в памяти с именем ext_ram, а функция bar() находится в памяти с именем sdram.

Example 6–16. Manually Assigning C Code to a Specific Memory Section

```
/* data should be initialized when using the section attribute */
int foo __attribute__ ((section (".ext_ram.rwdata"))) = 0;

void bar (void) __attribute__ ((section (".sdram.txt")));

void bar (void)
{
  foo++;
}
```

В ассемблере вы можете делать тоже самое, используя директиву .section. Например, весь код после следующей строки будет размещён в памяти устройства, называемой ext_ram:

.section .ext_ram.txt

Имена секций ext_ram и sdram являются примером. Вам нужно использовать имена секций, соответствующих вашей аппаратной части. Когда вы создаёте имена секций, используйте следующие расширения:

- .txt для кода: например, .sdram.txt
- .rodata для данных только для чтения: например, .cfi_flash.rodata
- .rwdata для данных под чтение и запись: например, .ext_ram.rwdata

За подробной информацией об использовании этих средств, обратитесь к документации на GNU компилятор и ассемблер. Эта документация инсталлирована вместе с Nios II EDS. Чтобы найти её, откройте начальную страницу документации на Nios II EDS, прокрутите вниз до **Разработка программ** и кликните на **Использование коллекции компилятора GNU (GCC).**

Полнофункциональный способ манипулировать схемой памяти – это использовать редактор Nios II BSP. С помощью редактора Nios II BSP, вы можете задать размещение секций компилятора в заданных областях физической памяти, и затем в графическом виде посмотреть неиспользуемые и перекрывающиеся регионы. Запустить редактор BSP можно из командной строки Nios II. За подробной информацией об использовании редактора BSP, обратитесь к всплывающим подсказкам редактора.

Размещение кучи и стека

По умолчанию, куча (heap) и стек размещаются в одном разделе памяти в секции .rwdata. Стек растёт вниз от конца секции (навстречу младшему адресу). Куча растёт вверх от последнего используемого адреса памяти в секции .rwdata. Вы можете контролировать размещение кучи и стека, манипулируя настройками BSP.

По умолчанию, HAL не выполняет проверку стека и кучи. Это ускоряет вызов функций и распределение памяти, но при этом функции malloc() (в Си) и new (в C++) не смогут определить опустошение кучи. Вы можете разрешить проверку стека во время работы программы, манипулируя настройками BSP. С включенной проверкой стека, функции malloc() и new() смогут детектировать опустошение кучи.

Чтобы задать ограничение размера кучи, установите символ предпроцессора ALT_MAX_HEAP_BYTES в максимальное значение размера кучи десятичным числом. Например, аргумент предпроцессора -DALT_MAX_HEAP_SIZE=1048576 устанавливает ограничение размера кучи до 0x100000. Вы можете задать эту опцию в командной строке в настройках BSP. За подробной информацией о контроле настроек BSP, обратитесь к секции "Настройки HAL BSP" на странице 6-2.

Проверка стека снижает характеристики программы. Если вы хотите не проверять стек, вы должны написать вашу программу так, чтобы она работала в рамках доступной памяти под стек и кучу.

Обратитесь к главе "Инструменты создания программы под Nios II" в настольной книге программиста Nios II за подробной информацией о выборе размещения стека и кучи, и настройки проверки стека.

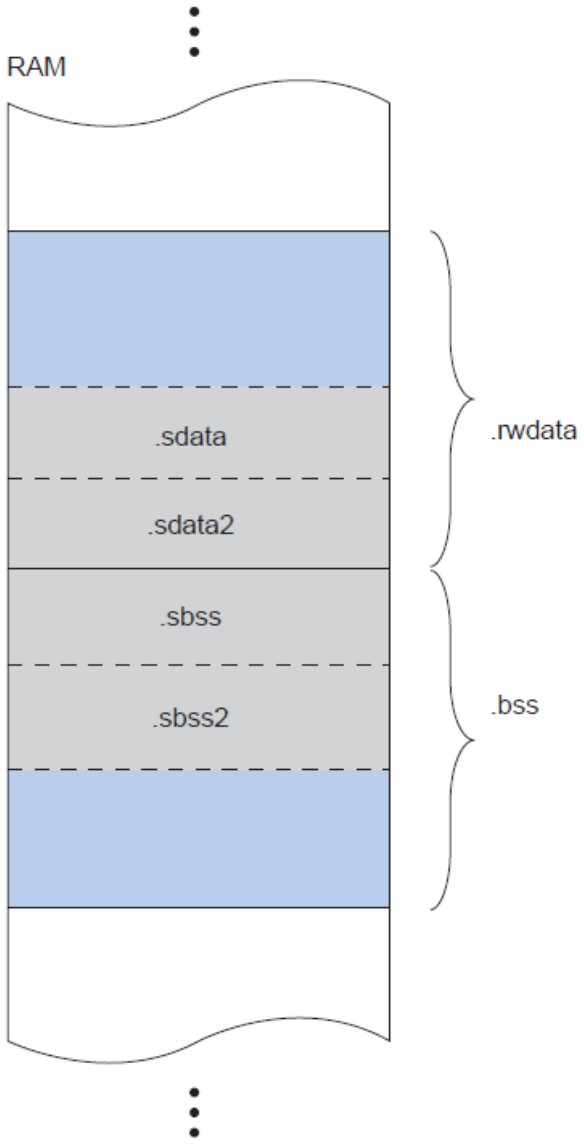
Регистр глобального указателя

Регистр глобального указателя разрешает быстрый доступ к структуре данных в программах под Nios II. Компилятор Nios II реализует глобальный указатель и определяет, какие структуры данных будут доступны через него. Вам не нужно ничего делать, пока вы не захотите изменить поведение компилятора по умолчанию.

Регистр глобального указателя может иметь доступ к одному непрерывному региону памяти в 64 Кб. Чтобы избежать переполнения этого региона, компилятор использует только глобальный указатель с малыми глобальными структурами данных. По умолчанию, порог – 8 Байт.

Малые глобальные структуры данных распределяются в малых глобальных секциях данных - .sdata,.sdata2, .sbss и .sbss2. Малые глобальные секции данных являются подсекциями секций .rwdata и .bss. Они распределяются вместе, как показано на рис. 6-4, чтобы разрешить доступ к ним глобального указателя.

Figure 6–4. Small Global Data sections



Если общий размер малой глобальной структуры данных больше 64 Кб, то эти структуры данных переполняют регион глобального указателя. Компилятор выдаёт сообщение об ошибке: "Unable to reach <*variable name*> ... from the global pointer ... because the offset ... is out of the allowed range, -32678 to 32767. Невозможно взять (имя переменной) из глобального указателя, поскольку офсет выходит за разрешённый диапазон от -32678 до 32767."

Вы можете зафиксировать это с помощью опции компилятора –G. Эта опция устанавливает размер чувствительности. Например, -G 4 ограничивает глобальному указателю использовать под структуры данных не более 4 байт. Уменьшение порога глобального указателя уменьшает размер секций малых глобальных данных.

Опции –G имеют десятичный аргумент. Вы можете задать эту опцию компилятора в настройках проекта. За подробной информацией о контроле настроек проекта, обратитесь к секции "Настройки HAL BSP" на странице 6-2.

Вы должны установить для этой опции одинаковое значение для проекта BSP и для проекта приложения.

Режимы загрузки

Загрузочная память процессора – это память, содержащая вектор сброса. Это может быть внешним чипом флеш памяти или чипом последовательной конфигурации Altera EPCS, или внутри чиповой RAM. В зависимости от типа загрузочной памяти, системы, основанные на HAL, конструируются так, чтобы все секции программы и данных в начальном положении находились в них. HAL предлагает маленькую программу загрузчик, которая копирует эти секции во время загрузки на место их размещения при запуске программы. Вы можете задать размещение во время работы программы для данных и кода, манипулируя настройками BSP.

Если размещение секции .text рабочего цикла программы находится за пределами загрузочной памяти, флеш программатор Altera размещает загрузчик по адресу сброса. Этот загрузчик используется для загрузки всех секций кода и данных до вызова _start. Когда выполняется загрузка из чипа EPCS, эта функция загрузчика реализуется аппаратно.

Однако, если размещение секции .text рабочего цикла программы находится в загрузочной памяти, то системе не требуется отдельный загрузчик. Вместо этого в исполняемой программе HAL прямо вызывается точка входа _reset. Функция _reset инициализирует кэш инструкций и затем вызывает _start. Такая последовательность инициализации позволяет вам разрабатывать приложения, которые загружаются и исполняются прямо из флеш памяти.

Когда запущен этот режим, исполняемая программа HAL должна нести ответственность за загрузку любой секции, которую необходимо загрузить в RAM. Секции .rwdata, .rodata и .exceptions загружаются, соответственно, перед вызовом alt_main(). Такая загрузка выполняется функцией alt_load(). Для загрузки любых других секций, используйте функцию alt_load_section().

За дополнительной информацией о функции alt_load_section(), обратитесь к главе "Справка по HAL API" в настольной книге программиста Nios II.

Работа с исходными файлами HAL

Вы можете захотеть сравнить файлы в HAL, особенно заголовочные файлы. В этой секции описывается, как находить и использовать исходные файлы HAL.

Поиск HAL файлов

Вы задаёте размещение исходных файлов HAL, когда создаёте BSP. Исходные файлы HAL (и другие BSP файлы) копируются в директорию BSP.

За подробной информацией обратитесь к главе "Справка по инструментам создания программ Nios II" в настольной книге разработчика Nios II.

Подмена функции HAL

Исходные файлы HAL копируются в вашу директорию BSP при создании вашего BSP. Если вы регенерируете BSP, все исходные файлы HAL, отличающиеся от инсталлированных файлов, тоже копируются. Избегайте изменения BSP файлов. Чтобы подменить HAL код по умолчанию, используйте настройки BSP или собственные драйверы устройств, или пакеты программ.

За информацией о том, что получается при регенерации BSP, обратитесь к секции "Исправление вашего BSP" в главе "Инструменты создания программ Nios II" в настольной книге разработчика Nios II.

Избегайте изменений исходных файлов HAL. Если вы измените исходные файлы HAL, вы не сможете регенерировать BSP без потери ваших изменений. Сложно сохранить согласованность изменённого BSP с основной системой SOPC Builder.

За информацией обратитесь к секции "Программные проекты Nios II" в главе "Инструменты создания программ Nios II" в настольной книге разработчика Nios II.