Разработка встроенных приложений

• Разработка встроенного программного обеспечения с использованием инструментов сборки IAR.
• Процесс сборки - обзор.
• Выполнение приложения - обзор
• Создание приложений - обзор
• Базовая конфигурация проекта.

Разработка встроенного программного обеспечения с использованием инструментов сборки IAR

Обычно встроенное программное обеспечение, написанное для выделенного микроконтроллера, представляет собой бесконечный цикл, ожидающий возникновения каких-либо внешних событий. Программное обеспечение находится в ПЗУ и запускается при сбросе. При написании такого программного обеспечения вы должны учитывать несколько аппаратных и программных факторов. Чтобы помочь вам, включены параметры компилятора, расширенные ключевые слова, директивы pragma и т. д.

КАРТА ПАМЯТИ

Встроенные системы обычно содержат различные типы памяти, такие как встроенная RAM, внешняя DRAM или SRAM, ROM, EEPROM или флэш-память.

Как разработчик встроенного программного обеспечения, вы должны понимать особенности различных типов памяти. Например, оперативная память на кристалле часто работает быстрее, чем другие типы памяти, и переменные, к которым часто обращаются, в критичных по времени приложениях выиграют от их размещения здесь. И наоборот, некоторые данные конфигурации могут быть редко доступны, но должны сохранять свое значение после выключения питания, поэтому их следует сохранять в EEPROM или флэш-памяти.

Для эффективного использования памяти компилятор предоставляет несколько механизмов для управления размещением функций и объектов данных в памяти. Для получения дополнительной информации см. Управление размещением данных и функций в памяти, стр. 240.

Компоновщик помещает разделы кода и данных в память в соответствии с директивами, указанными в файле конфигурации компоновщика, см. Размещение кода и данных - файл конфигурации компоновщика, стр. 99

СВЯЗЬ С ПЕРИФЕРИЙНЫМИ УСТРОЙСТВАМИ

Если к микроконтроллеру подключены внешние устройства, может потребоваться инициализация и управление сигнальным интерфейсом, например, с помощью контактов выбора микросхемы, а также обнаружение и обработка сигналов внешних прерываний. Обычно это необходимо инициализировать и контролировать во время выполнения. Обычный способ сделать это - использовать регистры специальных функций (SFR). Обычно они доступны по выделенным адресам и содержат биты, управляющие конфигурацией микросхемы.

Стандартные периферийные устройства определены в файлах заголовков ввода-вывода для конкретных устройств с расширением имени файла h. См. Под-

держка устройства, стр. 53. Для примера см. Доступ к регистрам специальных функций, стр. 253.

УПРАВЛЕНИЕ СОБЫТИЯМИ

Во встроенных системах использование прерываний - это метод немедленной обработки внешних событий, например, обнаружения нажатия кнопки. Как правило, когда в коде происходит прерывание, ядро немедленно прекращает выполнение кода, которое оно запускает, и вместо этого начинает выполнять процедуру обработки прерывания.

Компилятор предоставляет различные примитивы для управления аппаратными и программными прерываниями, что означает, что вы можете писать свои процедуры прерывания на C, см. Функции прерывания для устройств Cortex-M, стр. 80 и Функции прерывания для Arm7 / 9/11, Cortex-A и Устройства Cortex-R, стр. 81. См. Также Функции исключений для 64-битного режима, стр. 86.

ЗАПУСК СИСТЕМЫ

Во всех встроенных системах код запуска системы выполняется для инициализации системы - как аппаратного, так и программного обеспечения - перед вызовом основной функции приложения. ЦП требует этого, начиная выполнение с фиксированного адреса памяти.

Как разработчик встроенного программного обеспечения, вы должны убедиться, что код запуска расположен по выделенным адресам памяти или может быть доступен с помощью указателя из таблицы векторов. Это означает, что код запуска и начальная таблица векторов должны быть помещены в энергонезависимую память, такую как ROM, EPROM или флэш-память.

Кроме того, приложению C / C ++ необходимо инициализировать все глобальные переменные. Эта инициализация выполняется компоновщиком вместе с кодом запуска системы. Для получения дополнительной информации см. Выполнение приложения - обзор, стр. 64.

ОПЕРАЦИОННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Во многих случаях встроенное приложение является единственным программным обеспечением, работающим в системе. Однако использование ОСРВ имеет некоторые преимущества.

Например, время выполнения высокоприоритетных задач не зависит от других частей программы, которые выполняются в задачах с более низким приоритетом. Обычно это делает программу более детерминированной и может снизить энергопотребление за счет эффективного использования ЦП и перевода ЦП в состояние с низким энергопотреблением в режиме ожидания.

Использование ОСРВ может облегчить чтение и обслуживание вашей программы, а во многих случаях также сделать ее меньше. Код приложения можно четко разделить на независимые друг от друга задачи. Это упрощает командную работу, поскольку работу по разработке можно легко разделить на отдельные задачи, которыми занимается один разработчик или группа разработчиков.

Наконец, использование ОСРВ снижает зависимость от оборудования и создает чистый интерфейс для приложения, что упрощает перенос программы на другое целевое оборудование.

См. Также Управление многопоточной средой, стр. 165.

ВЗАИМОДЕЙСТВИЕ С ДРУГИМИ ИНСТРУМЕНТАМИ РАЗРАБОТКИ

Компилятор и компоновщик IAR обеспечивают поддержку AEABI, двоичного интерфейса встроенного приложения для Arm. Для получения дополнительной информации об этой спецификации интерфейса посетите веб-сайт www.arm.com.

Преимуществом этого интерфейса является возможность взаимодействия между поставщиками, поддерживающими его: приложение может быть построено из библиотек объектных файлов, созданных разными поставщиками и связанных с компоновщиком от любого поставщика, если они соответствуют стандарту AEABI.

AEABI указывает полную совместимость для объектного кода C и C ++, а также для библиотеки C. AEABI не включает спецификации для библиотеки C ++.

Дополнительные сведения о поддержке AEABI в инструментах сборки IAR см. В разделе Соответствие AEABI, стр. 228.

Инструменты сборки IAR для Arm с номерами версий от 8.xx и выше не полностью совместимы с более ранними версиями продукта. Для получения дополнительной информации см. Руководство по миграции IAR Embedded Workbench® для ARM.

Для получения дополнительной информации см. Оптимизация компоновщика, стр. 127.

Процесс сборки (build) - обзор

В этом разделе дается обзор процесса сборки - как различные инструменты сборки (компилятор, ассемблер и компоновщик) сочетаются друг с другом, переходя от исходного кода к исполняемому образу.

Чтобы ознакомиться с этим процессом на практике, вам следует изучить учебные материалы, доступные в Информационном центре IAR.

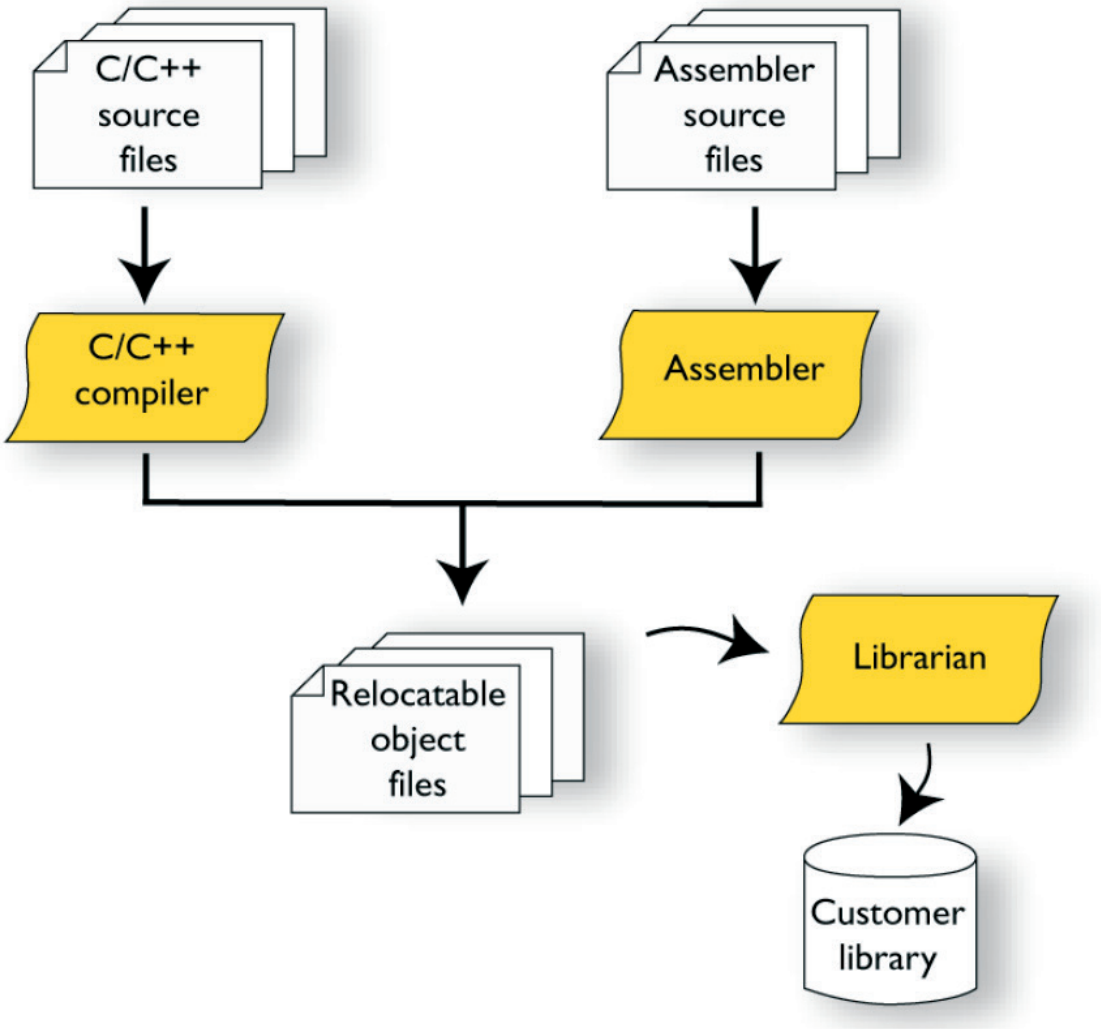
ПРОЦЕСС ТРАНСЛЯЦИИ

В среде IDE есть два инструмента, которые переводят исходные файлы приложения в промежуточные объектные файлы: компилятор IAR C / C ++ и ассемблер IAR. Оба производят перемещаемые объектные файлы в стандартном промышленном формате ELF, включая формат DWARF для отладочной информации.

Примечание. Компилятор также можно использовать для перевода исходного кода C в исходный код ассемблера. При необходимости вы можете изменить исходный код ассемблера, который затем может быть преобразован в объектный код. Для получения дополнительной информации об ассемблере IAR см. Руководство пользователя ассемблера IAR для Arm.

После перевода вы можете упаковать любое количество модулей в архив или, другими словами, в библиотеку. Важная причина, по которой вы должны использовать библиотеки, заключается в том, что каждый модуль в библиотеке условно связан в приложении или, другими словами, включается в приложение только в

том случае, если модуль прямо или косвенно используется модулем, предоставленным как объектный файл. При желании вы можете создать библиотеку, а затем использовать утилиту IAR ***iarchive***.



ПРОЦЕСС КОМПОНОВКИ

Перемещаемые модули в объектных файлах и библиотеках, созданные компилятором и ассемблером IAR, не могут быть выполнены как есть. Чтобы стать исполняемым приложением, они должны быть связаны.

Примечание. Модули, созданные набором инструментов от другого поставщика, также могут быть включены в сборку. Имейте в виду, что для этого также может потребоваться библиотека служебных программ компилятора от того же поставщика.

Компоновщик IAR ILINK (***ilinkarm.exe***) используется для создания окончательного приложения. Обычно компоновщик требует в качестве входных данных следующую информацию:

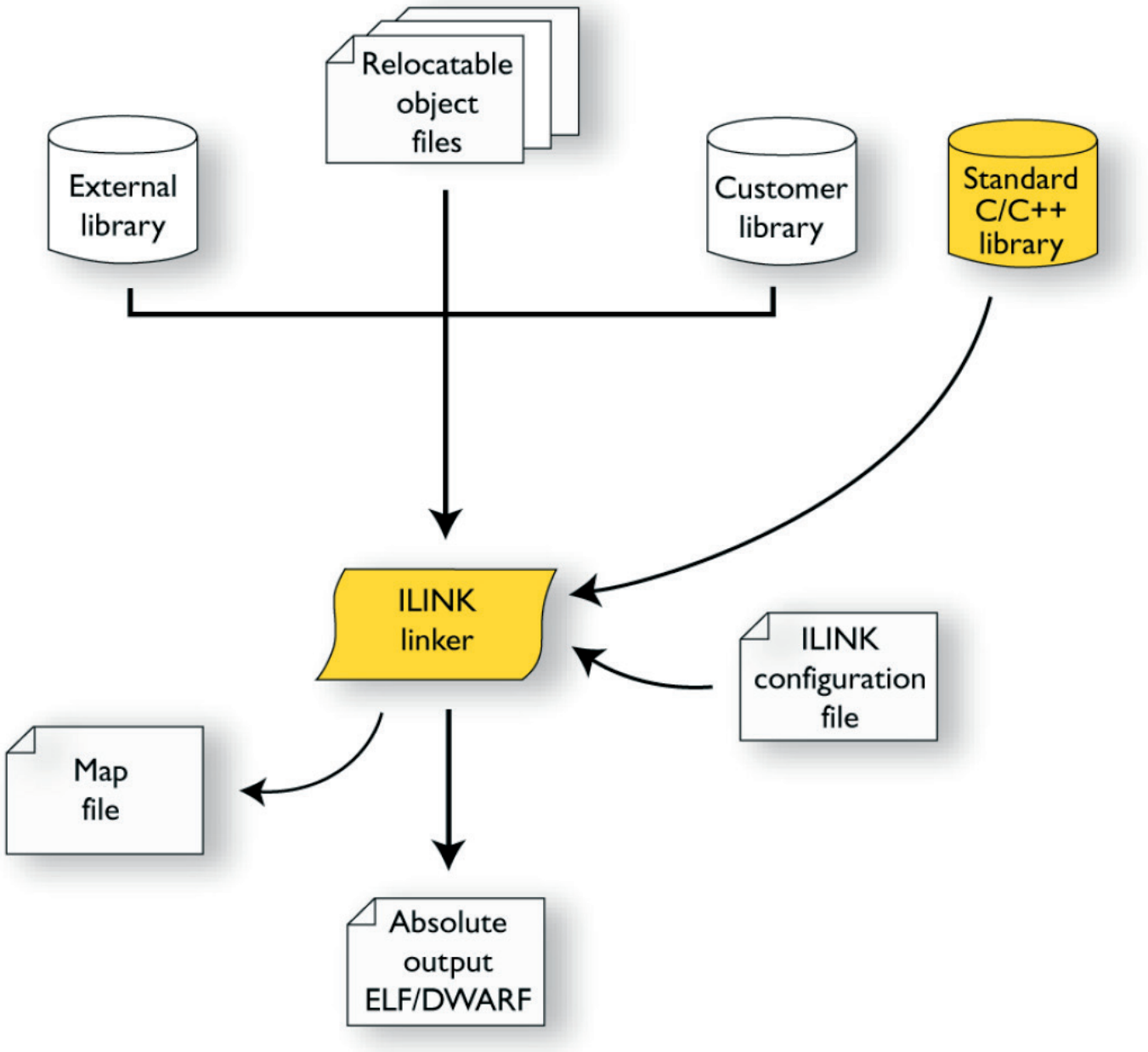
- Несколько объектных файлов и, возможно, некоторые библиотеки.
- Метка начала программы (устанавливается по умолчанию).
- Файл конфигурации компоновщика, описывающий размещение кода и данных в памяти целевой системы.

Примечание. Стандартная библиотека C / C ++ содержит процедуры поддержки для компилятора и реализацию функций стандартной библиотеки C / C ++.

При связывании компоновщик может выдавать сообщения об ошибках и сообщения журнала на ***stdout*** и ***stderr***. Сообщения журнала полезны для понима-

ния того, почему приложение было связано таким образом, например, почему был включен модуль или удален раздел.

Для получения дополнительной информации о процедуре, выполняемой компоновщиком, см. Подробные сведения о процессе связывания, стр. 97.



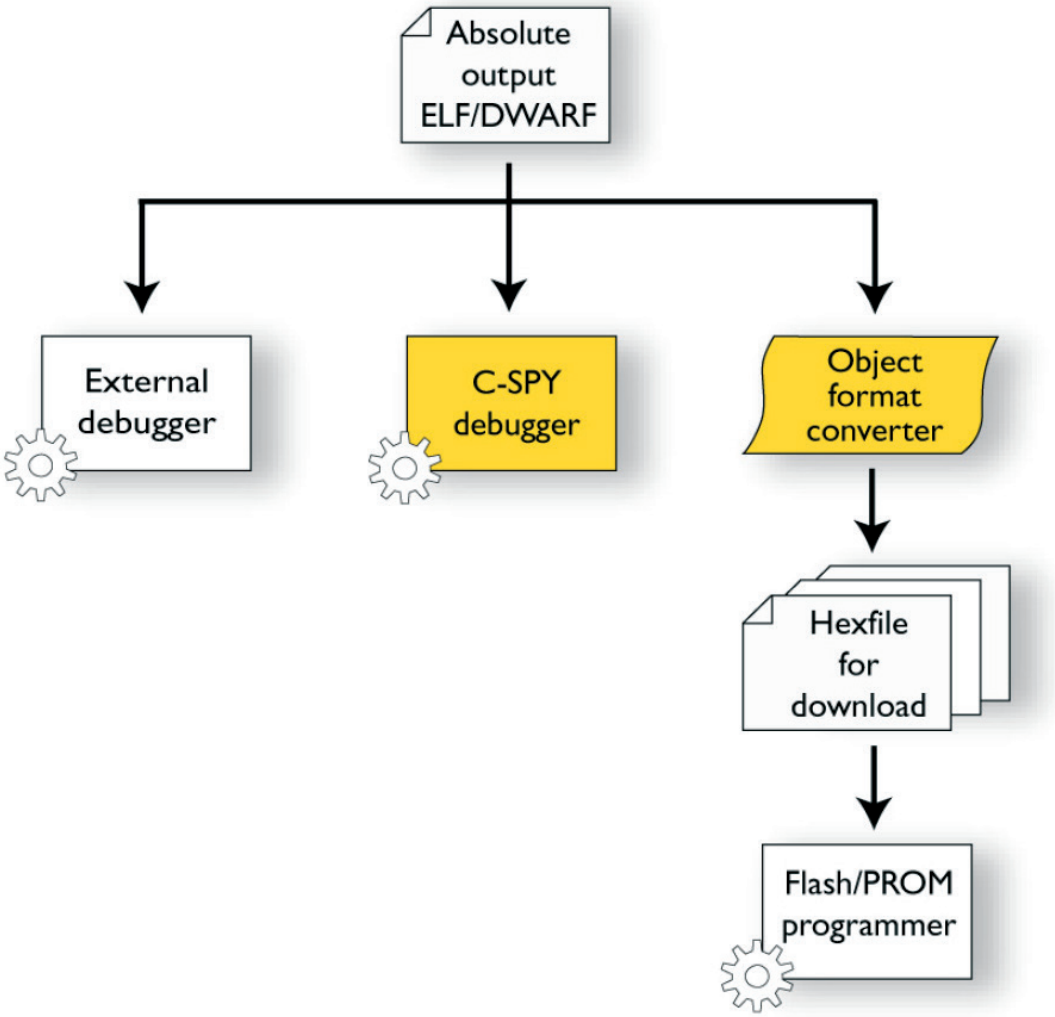
ПОСЛЕ Компоновки

Компоновщик IAR ILINK создает абсолютный объектный файл в формате ELF, содержащий исполняемый образ. После связывания созданный абсолютный исполняемый образ можно использовать для:

• Загрузка в отладчик IAR C-SPY или любой другой совместимый внешний отладчик, который читает ELF и DWARF.

• Программирование на flash / PROM с помощью программатора flash / PROM. Прежде чем это станет возможным, фактические байты изображения должны быть преобразованы в стандартный 32-разрядный формат S-записи Motorola или формат Intel Hex-32. Для этого используйте ***ielftool***, см. Инструмент IAR ELF - ielftool, стр. 549.

На этой иллюстрации показаны возможные варианты использования файла ELF / DWARF с абсолютным выводом:



Выполнение приложения (execution) - обзор

В этом разделе дается обзор выполнения встроенного приложения, разделенного на три этапа:

- Фаза инициализации
- Этап выполнения
- Завершающая фаза.

ФАЗА ИНИЦИАЛИЗАЦИИ

Инициализация выполняется при запуске приложения (сброс ЦП), но до входа в основную функцию (***main***). Для простоты этап инициализации можно разделить на:

- Аппаратная инициализация, которая, как минимум, обычно инициализирует указатель стека. Инициализация оборудования обычно выполняется в коде запуска системы ***cstartup.s*** и, если требуется, с помощью дополнительной низкоуровневой процедуры, которую вы предоставляете. Это может включать в себя сброс / перезапуск остального оборудования, настройку ЦП и т. д. В рамках подготовки к программной инициализации системы C / C ++.
- Инициализация системы программного обеспечения C / C ++

Как правило, это включает в себя обеспечение того, чтобы каждый глобальный (статически связанный) символ C / C ++ получил свое надлежащее значение инициализации перед вызовом основной функции.

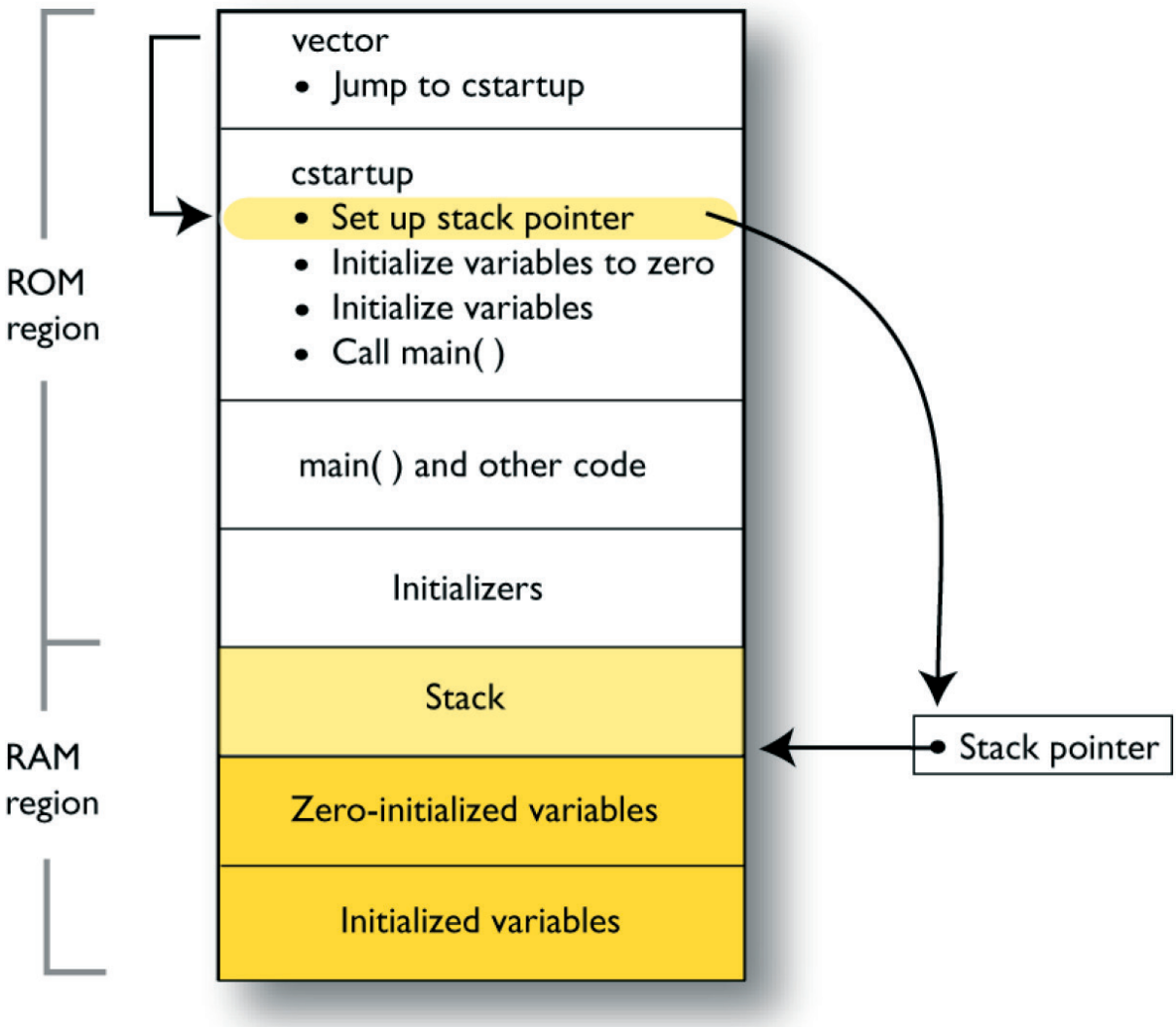
• Инициализация приложения.

Это полностью зависит от вашего приложения. Он может включать настройку ядра ОСРВ и запуск начальных задач для приложения, управляемого ОСРВ. Для простого приложения это может включать в себя настройку различных прерываний, инициализацию связи, инициализацию устройств и т. д.

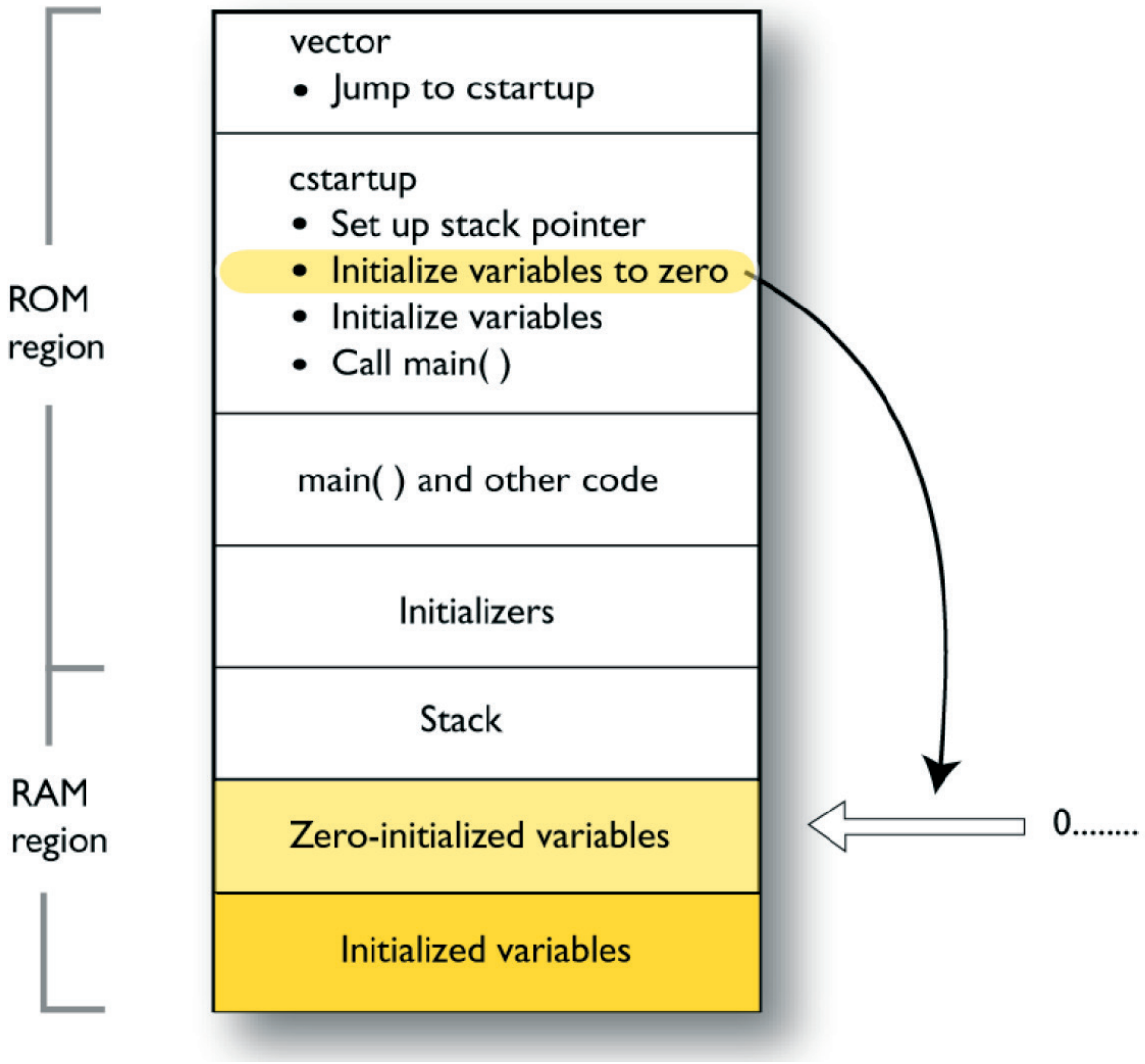
Для системы на основе ПЗУ / флэш-памяти константы и функции уже помещены в ПЗУ. Компоновщик уже разделил доступную RAM на различные области для переменных, стека, кучи и т. д. Все символы, помещенные в RAM, должны быть инициализированы перед вызовом основной функции.

Следующая последовательность иллюстраций дает упрощенный обзор различных этапов инициализации.

1. Когда приложение запускается, код запуска системы сначала выполняет аппаратную инициализацию, такую как инициализация указателя стека, чтобы он указывал на конец предопределенной области стека:

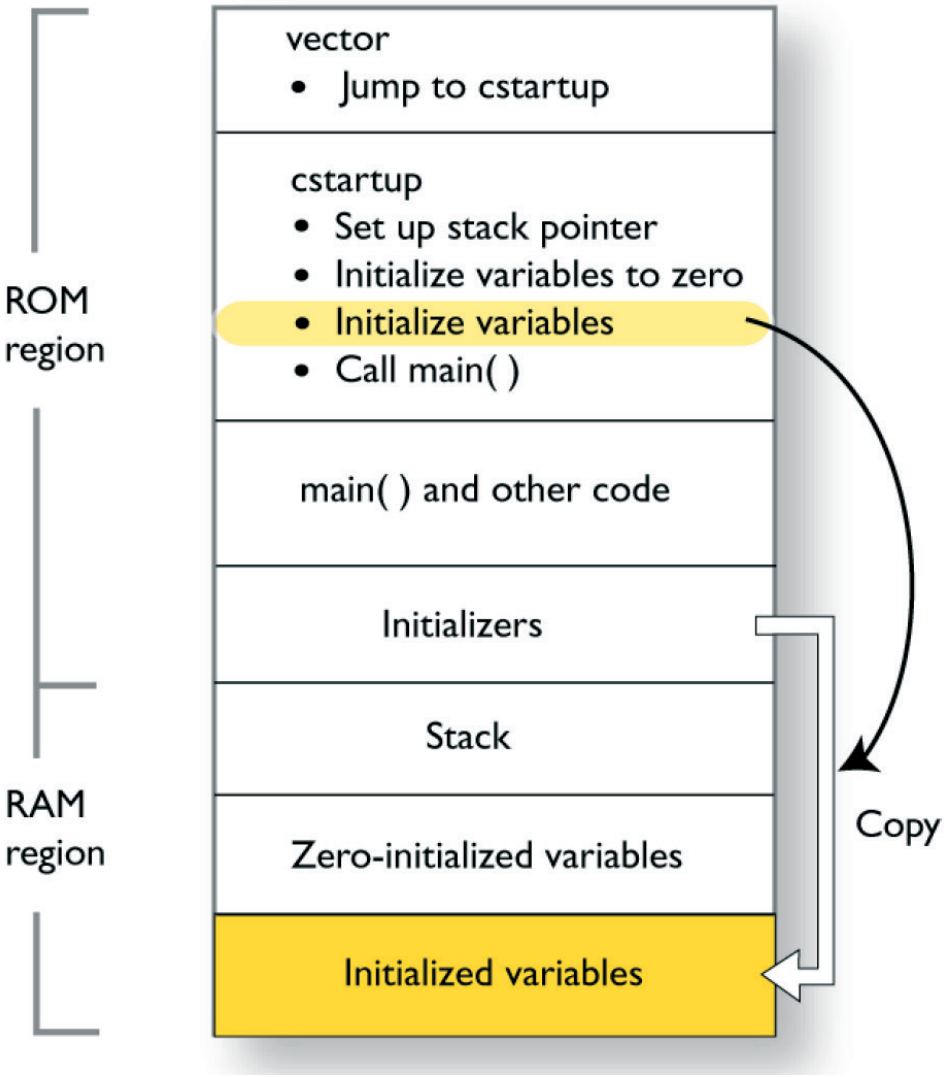


2 Затем память, которая должна быть инициализирована нулями, очищается, другими словами, заполняется нулями:



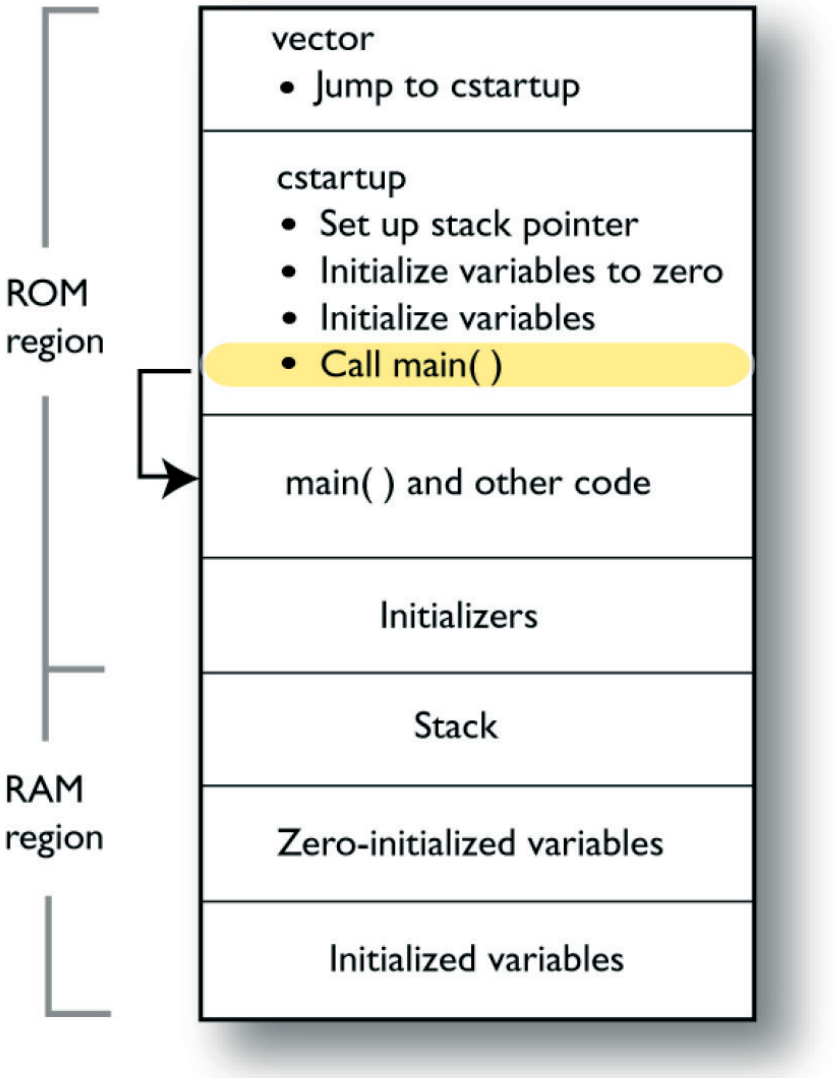
Обычно это данные, называемые данными с нулевой инициализацией - переменные, объявленные, например, как ***int i = 0;***

3 Для инициализированных данных данные, объявленные, например, как ***int i = 6;*** инициализаторы копируются из ПЗУ в ОЗУ



Затем создаются динамически инициализированные статические объекты, такие как объекты C ++.

4 Наконец, вызывается функция main:



Для получения дополнительной информации о каждом этапе см. Запуск и завершение работы системы, стр. 149. Для получения дополнительной информации об инициализации данных см. Инициализация при запуске системы, стр. 102.

ЭТАП ВЫПОЛНЕНИЯ

Программное обеспечение встроенного приложения обычно реализуется в виде цикла, который либо управляется прерываниями, либо использует опрос для управления внешним взаимодействием или внутренними событиями. Для системы, управляемой прерываниями, прерывания обычно инициализируются в начале функции main.

В системе, работающей в режиме реального времени и где скорость отклика имеет решающее значение, может потребоваться многозадачная система. Это означает, что ваше прикладное программное обеспечение должно быть дополнено операционной системой реального времени (RTOS). В этом случае ОСРВ и различные задачи также должны быть инициализированы в начале функции main.

ФАЗА ЗАВЕРШЕНИЯ

Обычно выполнение встроенного приложения никогда не должно заканчиваться. Если это так, вы должны определить правильное конечное поведение.

Чтобы завершить приложение управляемым способом, либо вызовите одну из функций стандартной библиотеки C ***exit, _Exit, quick_exit*** или ***abort***, либо вернитесь из main. Если вы вернетесь из main, выполняется функция выхода, что означает, что вызываются деструкторы C ++ для статических и глобальных переменных (только C ++) и все открытые файлы закрываются.

Конечно, в случае неправильной логики программы приложение может завершиться неконтролируемым и ненормальным образом - сбой системы.

Для получения дополнительной информации об этом см. Завершение работы системы, стр. 152.

Создание приложений (Building applications) - обзор

В интерфейсе командной строки следующая строка компилирует исходный файл ***myfile.c*** в объектный файл ***myfile.o*** с использованием настроек по умолчанию:

```
iccarm myfile.c
```

Вы также должны указать некоторые важные параметры, см. Базовая конфигурация проекта, стр. 69.

В командной строке для запуска компоновщика можно использовать следующую строку:

```
ilinkarm myfile.o myfile2.o -o a.out --config my_configfile.icf
```

В этом примере ***myfile.o*** и ***myfile2.o*** - объектные файлы, а ***my_configfile.icf*** - файл конфигурации компоновщика. Параметр -o указывает имя выходного файла.

Примечание. По умолчанию приложение запускается с метки __iar_program_start. Вы можете использовать параметр командной строки --entry, чтобы изменить это.

При сборке проекта IAR Embedded Workbench IDE может создавать обширную информацию о сборке в окне сообщений сборки. Эта информация может быть полезна, например, в качестве основы для создания командных файлов для построения в командной строке. Вы можете скопировать информацию и вставить ее в текстовый файл. Чтобы активировать подробную информацию о сборке, щелкните правой кнопкой мыши в окне «Сообщения сборки (Build messages)» и выберите «Все (All)» в контекстном меню.

Базовая конфигурация проекта (Basic project configuration)

В этом разделе представлен обзор основных настроек, необходимых для создания наилучшего кода для используемого вами устройства Arm. Вы можете указать параметры либо из интерфейса командной строки, либо в среде IDE. В командной строке вы должны указать каждую опцию отдельно, но если вы используете IDE, многие опции будут установлены автоматически в зависимости от ваших настроек некоторых из основных опций.

Вам необходимо произвести настройки для:

- Конфигурация процессора, то есть вариант процессора, режим ЦП, VFP и арифметика с плавающей запятой, а также порядок байтов.
- Настройки оптимизации.
- Среда выполнения, см. Настройка среды выполнения, стр. 133.

• Настройку конфигурации ILINK см. В главе «Компоновка приложения».

В дополнение к этим настройкам вы можете использовать множество других опций и настроек для еще большей точной настройки результата. Для получения информации о том, как установить параметры и список всех доступных параметров, см. Главы Параметры компилятора, параметры компоновщика и Руководство по управлению проектами и построению IDE для Arm, соответственно.

КОНФИГУРАЦИЯ ПРОЦЕССОРА 32-БИТНОГО РЕЖИМА

Чтобы компилятор генерировал оптимальный код, вы должны настроить его для используемого ядра Arm.

Вариант процессора

Компилятор IAR C / C ++ для Arm поддерживает большинство 32-битных ядер и устройств Arm. Все поддерживаемые ядра поддерживают инструкции Thumb и 64-битные инструкции умножения. Объектный код, который генерирует компилятор, не всегда двоично совместим между ядрами, поэтому очень важно указать компилятору параметр процессора. Ядро по умолчанию - Cortex-M3.

Режим выполнения должен быть 32-битным. Для получения информации о настройке варианта процессора см. Руководство по управлению проектами и построению IDE для Arm.

Используйте параметр ***--cpu***, чтобы указать ядро Arm. Для получения информации о синтаксисе см. --Arm, стр. 281 и --thumb, стр. 320.

VFP и арифметика с плавающей запятой

Если вы используете ядро Arm, которое содержит сопроцессор с векторной плавающей запятой (VFP), вы можете использовать параметр ***--fpu*** для генерации кода, который выполняет операции с плавающей запятой с использованием сопроцессора, вместо использования программных библиотечных процедур с плавающей запятой.

Информацию о настройке параметра FPU в среде IDE см. В Руководстве по управлению проектами и построению IDE для Arm.

Используйте параметр --fpu, чтобы указать ядро Arm. Информацию о синтаксисе см. --Fpu, стр. 294.

Порядок байтов

Компилятор поддерживает порядок байтов с прямым и обратным порядком байтов. Все пользовательские и библиотечные модули в вашем приложении должны использовать один и тот же порядок байтов.

См. Руководство по управлению проектами и построению IDE для Arm для получения информации о настройке параметра Endian mode в среде IDE.

Используйте параметр ***--endian***, чтобы указать порядок байтов для вашего проекта. Информацию о синтаксисе см. --Endian, стр. 292.

КОНФИГУРАЦИЯ ПРОЦЕССОРА В 64-БИТНОМ РЕЖИМЕ

Чтобы компилятор генерировал оптимальный код, вы должны настроить его для используемого ядра Arm.

Вариант процессора

Выберите 64-битное ядро Armv8-A, которое поддерживает компилятор IAR C / C ++ для Arm. Объектный код, который генерирует компилятор, не всегда дво-

ично совместим между ядрами, поэтому очень важно указать компилятору параметр процессора.

Режим выполнения должен быть 64-битным. Для получения информации о настройке варианта процессора см. Руководство по управлению проектами и построению IDE для Arm.

Используйте параметр ***--cpu***, чтобы указать ядро Arm. Для получения информации о синтаксисе см. --Cpu, стр. 282, и --aarch64, стр. 279.

Модель данных

Выберите модель данных для использования для сгенерированного кода, ILP32 или LP64.

Для получения информации о настройке параметра модели данных см. Руководство по управлению и построению проектов IDE для Arm.

Используйте параметр ***--abi***, чтобы указать модель данных. Информацию о синтаксисе см. --Abi, стр. 279.

ОПТИМИЗАЦИЯ СКОРОСТИ И РАЗМЕРА

Оптимизатор компилятора выполняет, среди прочего, устранение мертвого кода, распространение констант, встраивание, удаление общих подвыражений, статическую кластеризацию, планирование инструкций и снижение точности. Он также выполняет оптимизацию цикла, например, разворачивание и исключение переменных индукции.

Вы можете выбирать между несколькими уровнями оптимизации, а для самого высокого уровня вы можете выбирать между различными целями оптимизации - размером, скоростью или сбалансированностью. Большинство оптимизаций делают приложение меньше и быстрее. Однако, когда это не так, компилятор использует выбранную цель оптимизации, чтобы решить, как выполнить оптимизацию.

Уровень и цель оптимизации могут быть указаны для всего приложения, для отдельных файлов и для отдельных функций. Кроме того, можно отключить некоторые индивидуальные оптимизации, такие как встраивание функций.

Информацию об оптимизации компилятора и дополнительную информацию об эффективных методах кодирования см. В главе «Эффективное кодирование для встроенных приложений».

Хранилище данных (Data storage)

- Введение
- Хранение автоматических переменных и параметров.
- Динамическая память в куче

Вступление

32-битное ядро Arm может адресовать 4 ГБ непрерывной памяти в диапазоне от 0x0 до 0xFFFF’FFFF. 64-битное ядро Arm может адресовать 16 эксбибайт непрерывной памяти в диапазоне от 0x0 до 0xFFFF’FFFF’FFFF’FFFF. В диапазон памяти могут быть помещены различные типы физической памяти. Типичное приложение будет иметь как постоянную память (ROM), так и память для чтения / записи (RAM). Кроме того, некоторые части диапазона памяти содержат регистры управления процессором и периферийные устройства.

РАЗЛИЧНЫЕ СПОСОБЫ ХРАНЕНИЯ ДАННЫХ

В типичном приложении данные могут храниться в памяти тремя различными способами:

• Автоматические переменные.

Все переменные, которые являются локальными для функции, кроме тех, которые объявлены статическими, хранятся либо в регистрах, либо в стеке. Эти переменные можно использовать, пока выполняется функция. Когда функция возвращается к вызывающей стороне, пространство памяти больше не действует. Для получения дополнительной информации см. Хранение автоматических переменных и параметров, стр. 74.

• Глобальные переменные, статические переменные модуля и локальные переменные объявлены статическими.

В этом случае память выделяется раз и навсегда. Слово статический в этом контексте означает, что объем памяти, выделенной для этого типа переменных, не изменяется во время работы приложения. Ядро Arm имеет одно адресное пространство, а компилятор поддерживает полную адресацию памяти.

• Динамически распределяемые данные

Приложение может размещать данные в куче, где данные остаются действительными до тех пор, пока они не будут явным образом переданы обратно в систему приложением. Этот тип памяти полезен, когда количество объектов неизвестно до выполнения приложения.

Примечание. Существуют потенциальные риски, связанные с использованием динамически выделяемых данных в системах с ограниченным объемом памяти или системах, которые, как ожидается, будут работать в течение длительного времени. Дополнительные сведения см. В разделе Динамическая память в куче, стр. 75.

Хранение автоматических переменных и параметров

Переменные, которые определены внутри функции - а не объявлены статическими - в стандарте С называются автоматическими переменными. Некоторые из этих переменных помещаются в регистры процессора, а остальные помещаются в стек. С семантической точки зрения это эквивалентно. Основные отличия заключаются в том, что доступ к регистрам происходит быстрее и требуется меньше памяти по сравнению с тем, когда переменные находятся в стеке.

Автоматические переменные могут существовать только до тех пор, пока функция выполняется - когда функция возвращается, память, выделенная в стеке, освобождается.

СТЕК

Стек может содержать:

• Локальные переменные и параметры не хранящиеся в регистрах.

• Временные результаты выражений

• Возвращаемое значение функции (если оно не передается в регистры)

• Состояние процессора во время прерываний

• Регистры процессора, которые должны быть восстановлены перед возвратом функции (регистры сохранения вызываемого объекта).

• Канарейки, используемые в функциях, защищенных стеком. См. Раздел Защита стека, стр. 92.

Стек - это фиксированный блок памяти, разделенный на две части. Первая часть содержит выделенную память, используемую функцией, которая вызвала текущую функцию, и функцией, которая ее вызвала, и т. д. Вторая часть содержит свободную память, которую можно выделить. Граница между двумя областями называется вершиной стека и представлена указателем стека, который представляет собой специальный регистр процессора. Память выделяется в стеке путем перемещения указателя стека.

Функция никогда не должна ссылаться на память в области стека, которая содержит свободную память. Причина в том, что в случае прерывания вызываемая функция прерывания может выделить, изменить и, конечно же, освободить память в стеке.

См. Также разделы «Особенности стека», стр. 216 и Настройка памяти стека, стр. 118.

Преимущества

Основное преимущество стека заключается в том, что функции в разных частях программы могут использовать одно и то же пространство памяти для хранения своих данных. В отличие от кучи, стек никогда не станет фрагментированным или не будет страдать от утечек памяти.

Функция может вызывать себя прямо или косвенно - рекурсивная функция - и каждый вызов может сохранять свои собственные данные в стеке.

Потенциальные проблемы

Способ работы стека делает невозможным хранение данных, которые должны жить после возврата из функции. Следующая функция демонстрирует типичную ошибку программирования. Он возвращает указатель на переменную x, переменную, которая перестает существовать, когда функция возвращается.

```
int *MyFunction()
{
  int x;
  /* Do something here. Сделай что-нибудь здесь. */
  return &x; /* Incorrect Неверно */
}
```

Другая проблема - это риск нехватки места в стеке. Это произойдет, когда одна функция вызывает другую, которая, в свою очередь, вызывает третью и т. д., И сумма использования стека каждой функцией больше, чем размер стека. Риск выше, если в стеке хранятся большие объекты данных или когда используются рекурсивные функции.

Динамическая память в куче

Память для объектов, выделенных в куче, будет жить до тех пор, пока объекты не будут явно освобождены. Этот тип памяти полезен для приложений, в которых объем данных неизвестен до времени выполнения.

В C память выделяется с помощью стандартной библиотечной функции ***malloc*** или одной из связанных функций ***calloc*** и ***realloc***. Память снова освобождается, используя ***free***.

В C ++ специальное ключевое слово ***new*** выделяет память и запускает конструкторы. Память, выделенная с помощью ***new***, должна быть освобождена с помощью ключевого слова ***delete***. Для получения информации о том, как установить размер кучи памяти, см. Настройка кучи памяти, стр. 118.

ПОТЕНЦИАЛЬНЫЕ ПРОБЛЕМЫ

Приложения, использующие объекты данных, размещенные в куче, должны быть тщательно спроектированы, поскольку легко попасть в ситуацию, когда невозможно выделить объекты в куче.

Куча может быть исчерпана, если ваше приложение использует слишком много памяти. Он также может заполниться, если память, которая больше не используется, не была освобождена.

Для каждого выделенного блока памяти требуется несколько байтов данных для административных целей. Для приложений, которые выделяют большое количество небольших блоков, эти административные издержки могут быть значительными.

Также существует проблема фрагментации; это означает кучу, в которой небольшие участки свободной памяти разделены памятью, используемой выделенными объектами. Невозможно выделить новый объект, если ни одна часть свободной памяти не является достаточно большой для объекта, даже если сумма размеров свободной памяти превышает размер объекта.

К сожалению, фрагментация имеет тенденцию увеличиваться по мере выделения и освобождения памяти. По этой причине приложения, предназначенные для работы в течение длительного времени, должны избегать использования памяти, выделенной в куче.

Функции

- Расширения, связанные с функциями
- 32-битный Arm и Thumb код
- 64-битный код A64
- Выполнение в ОЗУ
- Функции прерывания для устройств Cortex-M
- Функции прерывания для устройств Arm7 / 9/11, Cortex-A и Cortex-R
- Функции-исключения для 64-битного режима.
- Встраивание функций
- Защита стека
- Интерфейс TrustZone

Расширения, связанные с функциями

Помимо поддержки Standard C, компилятор предоставляет несколько расширений для написания функций на C. С их помощью вы можете:

- Сгенерировать код для 32-битных режимов ЦП Arm и Thumb.
- Сгенерировать код для набора инструкций A64.
- Выполнить функции в ОЗУ.
- Записать функций прерывания для различных устройств.
- Встроить управляющие функции

- Содействовать оптимизации функций
- Получить доступ к аппаратным функциям.
- Создать функции интерфейса для TrustZone

Компилятор использует параметры компилятора, расширенные ключевые слова, директивы pragma и внутренние функции для поддержки этого.

Для получения дополнительной информации об оптимизации см. «Эффективное кодирование для встроенных приложений», стр. 237. Информацию о доступных встроенных функциях для доступа к аппаратным операциям см. В главе «Внутренние функции».

32-битный Arm и Thumb код

В 32-битном режиме компилятор IAR C / C ++ для Arm может генерировать код либо для 32-битного Arm, либо для 16-битного набора команд Thumb или Thumb2. Используйте параметр ***--cpu_mode***, а также параметры ***--arm*** или ***--thumb***, чтобы указать, какой набор инструкций следует использовать для вашего проекта. Для отдельных функций можно переопределить настройку проекта, используя расширенные ключевые слова ***__arm*** и ***__thumb***. Вы можете свободно смешивать код Arm и Thumb в одном приложении.

При выполнении вызовов функций компилятор всегда пытается сгенерировать наиболее эффективную доступную инструкцию или последовательность инструкций на языке ассемблера. В результате для размещения кода можно использовать 4 ГБ постоянной памяти в диапазоне 0x0-0xFFFF'FFFF. На каждый модуль кода установлено ограничение в 4 Мбайта.

Размер всех указателей кода - 4 байта. Существуют ограничения на неявное и явное приведение указателей кода к указателям данных или целочисленным типам или наоборот. Для получения дополнительной информации об ограничениях см. Типы указателей, стр. 374.

В главе «Интерфейс языка ассемблера» сгенерированный код изучается более подробно при описании вызова функций C из языка ассемблера и наоборот.

64-битный код A64

В 64-битном режиме компилятор IAR C / C ++ для Arm может генерировать код для набора инструкций A64. Используйте параметр ***--cpu_mode***, а также параметры ***--aarch64*** или ***--abi***, чтобы указать, какой набор инструкций следует использовать для вашего проекта.

При выполнении вызовов функций компилятор всегда пытается сгенерировать наиболее эффективную доступную инструкцию или последовательность инструкций на языке ассемблера. В результате для размещения кода можно использовать 16 эксбибайт (Exbibytes. 1 EiB = 2^{60} байт) непрерывной памяти в диапазоне 0x0-0xFFFF'FFFF'FFFF'FFFF. Ограничение на один кодовый модуль составляет 64 Мбайта.

Размер указателей кода составляет 4 или 8 байтов, в зависимости от модели данных. Существуют ограничения на неявное и явное приведение указателей кода к указателям данных или целочисленным типам или наоборот. Для получения дополнительной информации об ограничениях см. Типы указателей, стр. 374.

В главе «Интерфейс языка ассемблера» сгенерированный код изучается более подробно при описании вызова функций C из языка ассемблера и наоборот.

Выполнение в ОЗУ

Ключевое слово ***__ramfunc*** заставляет функцию выполняться в ОЗУ. Другими словами, он помещает функцию в раздел с атрибутами чтения / записи. Функция копируется из ПЗУ в ОЗУ при запуске системы, как и любая инициализированная переменная, см. Запуск и завершение работы системы, стр. 149.

Ключевое слово указывается перед типом возвращаемого значения:

__ramfunc void foo (void);

Если функция, объявленная ***__ramfunc***, пытается получить доступ к ПЗУ, компилятор выдаст предупреждение. Если вся область памяти, используемая для кода и констант, отключена - например, когда стирается вся флэш-память, - могут использоваться только функции и данные, хранящиеся в ОЗУ. Прерывания должны быть запрещены, если вектор прерывания и процедуры обслуживания прерывания также не хранятся в ОЗУ.

Строковых литералов и других констант можно избежать, используя инициализированные переменные. Например, следующие строки:

```
__ramfunc void test()
{
  /* myc: initializer in ROM  инициализирована в ПЗУ*/
  const int myc[] = { 10, 20 };
  /* string literal in ROM  строковый литерал в ПЗУ*/
  msg(«Hello»);
}
```

можно переписать на:

```
__ramfunc void test()
{
  /* myc: initialized by cstartup инициализирована cstartup*/
  static int myc[] = { 10, 20 };
  /* hello: initialized by cstartup * инициализирована cstartup/
  static char hello[] = «Hello»;
  msg(hello);
}
```

Для получения дополнительной информации см. Код инициализации - копирование ПЗУ в ОЗУ, стр. 121.

Функции прерывания для устройств Cortex-M

Cortex-M имеет другой механизм прерывания, чем предыдущие архитектуры Arm, что означает, что примитивы, предоставляемые компилятором, также отличаются.

ПРЕРЫВАНИЯ ДЛЯ КОРТЕКС-М

В Cortex-M подпрограмма обслуживания прерывания входит и возвращается так же, как и обычная функция, что означает, что никаких специальных ключе-

вых слов не требуется. Следовательно, ключевые слова ***__irq***, ***__fiq*** и ***__nested*** недоступны при компиляции для Cortex-M.

Эти имена функций исключения определены в ***cstartup_M.c*** и ***cstartup_M.s.*** На них ссылается код вектора исключения библиотеки:

NMI_Handler
HardFault_Handler
MemManage_Handler
BusFault_Handler
UsageFault_Handler
SVC_Handler
DebugMon_Handler
PendSV_Handler
SysTick_Handler

Таблица векторов реализована в виде массива. Он всегда должен иметь имя ***__vector_table***, потому что отладчик C-SPY ищет этот символ при определении расположения векторной таблицы.

Предопределенные функции исключения определены как слабые символы. Слабый символ включается компоновщиком только до тех пор, пока не будет найден повторяющийся символ. Если с тем же именем определен другой символ, он будет иметь приоритет. Поэтому ваше приложение может просто определить свою собственную функцию исключения, просто определив ее с использованием правильного имени из списка выше. Если вам нужны другие прерывания или другие обработчики исключений, вы должны сделать копию файла ***cstartup_M.c*** или ***cstartup_M.s*** и внести соответствующее добавление в таблицу векторов.

Встроенные функции ***__get_CPSR*** и ***__set_CPSR*** недоступны при компиляции для Cortex-M. Вместо этого, если вам нужно получить или установить значения тех или иных регистров, вы можете использовать встроенный ассемблер. Для получения дополнительной информации см. Передача значений между C и объектами ассемблера, стр. 254.

Функции прерывания для устройств Arm7 / 9/11, Cortex-A и Cortex-R

Компилятор IAR C / C ++ для Arm предоставляет следующие примитивы, связанные с написанием функций прерывания для устройств Arm7 / 9/11, Cortex-A и Cortex-R:

- Расширенные ключевые слова: ***__irq, __fiq, __nested***,
- Встроенные функции: ***__enable_interrupt, __disable_interrupt, __get_interrupt_state, __set_interrupt_state***

Примечание: Cortex-M имеет другой механизм прерывания, чем другие устройства Arm, и для этих устройств доступен другой набор примитивов. Для получения дополнительной информации см. Функции прерывания для устройств Cortex-M, стр. 80.

ФУНКЦИИ ПРЕРЫВАНИЙ

Во встроенных системах использование прерываний - это метод немедленной обработки внешних событий, например, обнаружения нажатия кнопки.

Процедуры обслуживания прерывания

Как правило, когда в коде происходит прерывание, ядро немедленно прекращает выполнение кода, которое оно запускает, и вместо этого начинает выполнять процедуру обработки прерывания. Важно, чтобы среда прерванной функции была восстановлена после обработки прерывания - это включает значения регистров процессора и регистра состояния процессора. Это позволяет продолжить выполнение исходного кода после того, как код, обработавший прерывание, был выполнен.

Компилятор поддерживает прерывания, программные прерывания и быстрые прерывания. Для каждого типа прерывания может быть написана процедура прерывания.

Все функции прерывания должны компилироваться в режиме Arm - если вы используете режим Thumb, используйте расширенное ключевое слово ***__arm*** или директиву ***#pragma type_attribute = __ arm***, чтобы переопределить поведение по умолчанию. Это не применимо к устройствам Cortex-M.

Векторы прерываний и таблица векторов прерываний

Каждая процедура прерывания связана с векторным адресом / инструкцией в таблице векторов исключений, которая указана в документации по ядрам Arm. Вектор прерывания - это адрес в таблице векторов исключений. Для ядер Arm таблица векторов исключений начинается с адреса 0x0.

По умолчанию таблица векторов заполняется обработчиком прерывания по умолчанию, который повторяется бесконечно. Для каждого источника прерывания, у которого нет явной подпрограммы обслуживания прерывания, будет вызван обработчик прерывания по умолчанию. Если вы напишете свою собственную процедуру обслуживания для определенного вектора, эта процедура переопределит обработчик прерывания по умолчанию.

Определение функции прерывания - пример

Чтобы определить функцию прерывания, можно использовать ключевое слово ***__irq*** или ***__fiq***. Например:

```
__irq __arm void IRQ_Handler (void)
{
/* Do something  Сделай что-нибудь */
}
```

Для получения дополнительной информации о таблице векторов прерываний см. Документацию по ядрам Arm.

Примечание. Функция прерывания должна иметь возвращаемый тип void, и она не может указывать какие-либо параметры.

Прерывание и функции-члены C ++

Только статические функции-члены могут быть функциями прерывания. Когда вызывается нестатическая функция-член, она должна применяться к объекту. Когда происходит прерывание и вызывается функция прерывания, нет объекта, к которому можно было бы применить функцию-член.

УСТАНОВКА ИСКЛЮЧИТЕЛЬНЫХ ФУНКЦИЙ

Все функции прерывания и программные обработчики прерываний должны быть установлены в таблице векторов. Это делается на языке ассемблера в файле запуска системы ***cstartup.s***.

Реализация по умолчанию таблицы векторов исключений Arm в стандартной библиотеке времени выполнения переходит к предопределенным функциям, реализующим бесконечный цикл. Таким образом, любое исключение, возникающее для события, не обрабатываемого вашим приложением, будет попадать в бесконечный цикл (B.).

Предопределенные функции определены как слабые символы. Слабый символ включается компоновщиком только до тех пор, пока не будет найден повторяющийся символ. Если с тем же именем определен другой символ, он будет иметь приоритет. Поэтому ваше приложение может просто определить свою собственную функцию исключения, просто определив ее с использованием правильного имени.

Эти имена функций исключения определены в cstartup.s и упоминаются кодом вектора исключения библиотеки:

```
Undefined_Handler
SVC_Handler
Prefetch_Handler
Abort_Handler
IRQ_Handler
FIQ_Handler
```

Чтобы реализовать собственный обработчик исключений, определите функцию, используя соответствующее имя функции исключения из списка выше.

Например, чтобы добавить функцию прерывания в C, достаточно определить функцию прерывания с именем IRQ_Handler:

```
__irq __arm void IRQ_Handler()
{
}
```

Функция прерывания должна иметь связь C, подробнее см. Соглашение о вызовах, стр. 183.

Если вы используете C ++, функция прерывания могла бы выглядеть, например, так:

```
extern "C"
{
  __irq __arm void IRQ_Handler(void);
}
__irq __arm void IRQ_Handler(void)
{
}
```

Никаких других изменений не требуется.

ПРЕРЫВАНИЯ И БЫСТРЫЕ ПРЕРЫВАНИЯ

С функциями прерывания и быстрого прерывания легко справиться, поскольку они не принимают параметры и не имеют возвращаемого значения. Используйте любое из этих ключевых слов:

• Чтобы объявить функцию прерывания, используйте расширенное ключевое слово ***__irq*** или директиву ***#pragma type_attribute = __ irq***. Для получения информации о синтаксисе см. ***_Irq***, стр. 388 и ***type_attribute***, стр. 422, соответственно.

• Чтобы объявить функцию быстрого прерывания, используйте расширенное ключевое слово ***__fiq*** или директиву ***#pragma type_attribute = __ fiq***. Для получения информации о синтаксисе см. ***__fiq***, стр. 388, и type***_attribute***, стр. 422, соответственно.

Примечание. Функция прерывания (irq) и функция быстрого прерывания (fiq) должны иметь тип возврата void и не могут иметь никаких параметров. Функция программного прерывания (swi или svc) может иметь параметры и возвращаемые значения. По умолчанию только четыре регистра R0 – R3 могут использоваться для параметров, и только регистры R0 – R1 могут использоваться для возвращаемых значений.

Вложенные прерывания

Прерывания автоматически блокируются ядром Arm до входа в обработчик прерывания. Если обработчик прерывания повторно разрешает прерывания, вызывает функции и возникает другое прерывание, то адрес возврата прерванной функции - сохраненный в LR - перезаписывается при взятии второго IRQ. Кроме того, содержимое SPSR будет уничтожено при втором прерывании. Само по себе ключевое слово ***__irq*** не сохраняет и не восстанавливает LR и SPSR. Чтобы обработчик прерывания выполнял необходимые действия при обработке вложенных прерываний, в дополнение к ***__irq*** необходимо использовать ключевое слово ***__nested***. Пролог функции - последовательность входа в функцию - которую компилятор генерирует для вложенных обработчиков прерываний, переключится из режима IRQ в системный режим. Убедитесь, что настроены и стек IRQ, и системный стек. Если вы используете файл ***cstartup.s*** по умолчанию, оба стека настроены правильно.

Созданные компилятором обработчики прерываний, допускающие вложенные прерывания, поддерживаются только для прерываний IRQ. Прерывания FIQ предназначены для быстрого обслуживания, что в большинстве случаев означает, что накладные расходы вложенных прерываний будут слишком высокими.

В этом примере показано, как использовать вложенные прерывания с контроллером векторных прерываний Arm (VIC):

```
__irq __nested __arm void interrupt_handler(void)
{
  void (*interrupt_task)();
  unsigned int vector;
  /* Get interrupt vector. Получите вектор прерывания. */
  vector = VICVectAddr;
  interrupt_task = (void(*)()) vector;
 /* Allow other IRQ interrupts to be serviced. Разрешить обслуживание других прерываний IRQ. */
  __enable_interrupt();
  /* Execute the task associated with this interrupt. */
/*Выполните задачу, связанную с этим прерыванием. */
  (*interrupt_task)();
}
```

Примечание. Ключевое слово __nested требует, чтобы режим процессора был либо пользовательским, либо системным.

ПРОГРАММНЫЕ ПРЕРЫВАНИЯ

Функции программного прерывания немного сложнее других функций прерывания в том смысле, что им нужен программный обработчик прерывания (диспетчер), они вызываются (вызываются) из запущенного прикладного программного обеспечения, принимают аргументы и имеют возвращаемые значения. Здесь описаны механизмы для вызова функции программного прерывания и то, как обработчик программного прерывания отправляет вызов фактической функции программного прерывания.

Вызов функции программного прерывания

Чтобы вызвать функцию программного прерывания из исходного кода вашего приложения, используется инструкция ассемблера ***SVC #immed***, где immed - это целое число, которое в данном руководстве называется номером программного прерывания или ***svc_number.*** Компилятор предоставляет простой способ неявно сгенерировать эту инструкцию из исходного кода C / C ++, используя ключевое слово ***__svc*** и директиву ***#pragma svc_number*** при объявлении функции.

Например, функцию ***__svc*** можно объявить так:

```
#pragma svc_number = 0x23
__svc int svc_function (int a, int b);
```

В этом случае будет сгенерирована инструкция ассемблера ***SVC 0x23*** там, где вызывается функция.

Функции программного прерывания следуют тому же соглашению о вызовах в отношении параметров и возвращаемых значений, что и обычные функции, за исключением использования стека, см. Соглашение о вызовах, стр. 183.

Для получения дополнительной информации см. ***__svc***, стр. 395, и ***svc_number***, стр. 422, соответственно.

Программный обработчик прерываний и функции

Обработчик прерывания, например ***SVC_Handler***, работает как диспетчер для функций программного прерывания. Он вызывается из вектора прерывания и отвечает за получение номера программного прерывания, а затем за вызов соответствующей функции программного прерывания. ***SVC_Handler*** должен быть написан на ассемблере, поскольку нет способа получить номер программного прерывания из исходного кода C / C ++.

Функции программного прерывания

Функции программного прерывания могут быть написаны на C или C ++. Используйте ключевое слово ***__svc*** в определении функции, чтобы компилятор сгенерировал возвращаемую последовательность, подходящую для конкретной функции программного прерывания. Директива #pragma svc_number не требуется в определении функции прерывания.

Для получения дополнительной информации см. ***__svc*** , стр. 395.

Настройка указателя стека программных прерываний

Если в вашем приложении будут использоваться программные прерывания, тогда должен быть установлен указатель стека программных прерываний (***SVC_STACK***) и должно быть выделено некоторое пространство для стека. Указатель ***SVC_STACK*** можно настроить вместе с другими стеками в файле ***cstartup.s***. В качестве примера см. Настройку указателя стека прерываний. Соответствующее пространство для указателя ***SVC_STACK*** устанавливается в файле конфигурации компоновщика, см. Настройка памяти стека, стр. 118.

ПРЕРВАННОЕ ВЫПОЛНЕНИЕ

Функция прерывания вызывается при возникновении внешнего события. Обычно он вызывается сразу во время выполнения другой функции. Когда функция прерывания завершает выполнение, она возвращается к исходной функции. Крайне важно, чтобы среда прерванной функции была восстановлена - это включает значение регистров процессора и регистр состояния процессора.

При возникновении прерывания выполняются следующие действия:

- Рабочий режим изменяется в соответствии с конкретным исключением.
- Адрес инструкции, следующей за инструкцией по вводу исключения, сохраняется в R14 нового режима.
- Старое значение CPSR сохраняется в SPSR нового режима.
- Запросы прерывания запрещаются установкой бита 7 CPSR и, если исключение является быстрым прерыванием, дальнейшие быстрые прерывания запрещаются установкой бита 6 CPSR
- ПК принудительно начинает выполнение по соответствующему адресу вектора.

Например, если произойдет прерывание для вектора 0x18, процессор начнет выполнять код по адресу 0x18. Область памяти, которая используется как начальная область для прерываний, называется таблицей векторов прерываний. Содержимое вектора прерывания обычно представляет собой инструкцию перехода к подпрограмме прерывания.

Примечание. Если функция прерывания разрешает прерывания, следует предполагать, что специальные регистры процессора, необходимые для возврата из процедуры обработки прерывания, уничтожены. По этой причине они должны быть сохранены программой обработки прерывания для восстановления до ее возврата. Это обрабатывается автоматически, если используется ключевое слово ***__nested****.*

Функции-исключения для 64-битного режима

Компилятор предоставляет следующие примитивы, связанные с написанием функций исключения для 64-битного режима:

- Расширенные ключевые слова ***__exception, __nested*** и ***__svc***.
- Встроенные функции ***__enable_interrupt*** и ***__disable_interrupt***.
- Имена специальных функций ***Synchronous_Handler_A64, Error_Handler_A64, IRQ_Handler_A64*** и ***FIQ_Handler_A64***

ФУНКЦИИ-ИСКЛЮЧЕНИЯ

Функция исключения используется для обработки событий внешнего прерывания или внутренних исключений. Когда возникает исключение, код, выполняемый в ядре, останавливается, и вместо него начинает выполняться код исключения. Важно, чтобы среда исключенного кода была восстановлена после обработки исключения - это включает значения регистров процессора, регистров состояния и т. д. Затем выполнение может продолжаться, как если бы исключение не произошло.

__exception - это атрибут типа функции, который определяет функцию исключения. Он должен иметь значение void в качестве возвращаемого значения и не может иметь параметров. Все использованные регистры сохраняются при входе и восстанавливаются при выходе. Он возвращается с инструкцией ERET.

```
__exception void func (void)
{
/* Do something  Сделай что-нибудь */
}
```

ИСКЛЮЧЕНИЯ И ФУНКЦИИ ЧЛЕНОВ C ++

Только статические функции-члены могут быть функциями исключения. Когда вызывается нестатическая функция-член, она должна применяться к объекту. Когда возникает исключение и вызывается функция исключения, нет объекта, к которому можно было бы применить функцию-член.

ТАБЛИЦА ВЕКТОРОВ ИСКЛЮЧЕНИЙ

Компилятор IAR C / C ++ использует одну и ту же таблицу векторов исключений для трех уровней исключений EL1, EL2 и EL3. Таблица векторов исключений начинается с определенного компоновщиком символа ***__eevector***. Он имеет 16 векторов, каждый размером 128 байт.

Компилятор IAR C / C ++ определяет только векторы для исключений, которые не изменяют уровень исключения и которые используют SP для текущего уровня исключения (смещения 0x200, 0x280, 0x300 и 0x380). Эти четыре определенных вектора имеют имена ***Synchronous_Handler_A64, Error_Handler_A64, IRQ_Handler_A64 и FIQ_Handler_A64***. У них есть реализация по умолчанию, которую можно переопределить, определив функцию ***__exception*** с одним из этих имен. Если функция слишком велика для размещения в векторе, компилятор выдаст ошибку. В этом случае функцию нельзя использовать напрямую как функцию исключения. Вместо этого вы должны:

1 Напишите модуль ассемблера, который начинается с глобального символа, например ***ee***. Символ должен перейти к функции исключения.

2 Отредактируйте файл конфигурации компоновщика. Замените место в директиве для соответствующей функции исключения (например, ***Synchronous_Handler_A64***) на место в адресе ***synchronous_evector {symbol ee} ;***.

По умолчанию таблица векторов исключений размещается по адресу 2048. Чтобы разместить ее по другому адресу, используйте один из следующих методов:

• Используйте параметр компоновщика ***--config_def***, чтобы установить символ конфигурации компоновщика

__Exception_table_address, например:

--config_def __Exception_table_address = 4096

• Отредактируйте файл конфигурации компоновщика, который использует проект.

Таблица исключений должна быть выровнена по 2 Кбайта.

ВЛОЖЕННЫЕ ИСКЛЮЧИТЕЛЬНЫЕ ФУНКЦИИ

Функция исключения может быть вложенной. Это также сохраняет системный регистр ELR_EL1 на входе. При выходе из функции прерывания блокируются, и все сохраненные регистры восстанавливаются. Пример:

```
#include <intrinsics.h>
__exception __nested void func(void)
{
    // All used registers + ELR_EL1 have been saved. SPSR_EL1 and ESR_EL1 can be saved/used.
    // Все использованные регистры + ELR_EL1 были сохранены. SPSR_EL1 и ESR_EL1
    // можно сохранять / использовать.
    __enable_interrupt();
    // Do stuf f Делай что-нибудь
    __disable_interrupt();
    // The possibly changed SPSR_EL1 and ESR_EL1 can be restored.
    // Возможно измененные SPSR_EL1 и ESR_EL1 могут быть восстановлены.
    // // At exit, interrupts will be disabled and then all used
    // registers are restored. Then ERET is executed.
    // При выходе прерывания будут отключены, а затем все использованные
    // регистры будут восстановлены. Затем выполняется ERET.
}
```

ФУНКЦИИ, ОПРЕДЕЛЯЕМЫЕ СУПЕРВИЗОРОМ

Функция, определенная с использованием атрибута типа функции ***__svc***, может иметь возвращаемые значения и принимать параметры. Он сохраняет те же регистры, что и при обычном вызове функции, и возвращается с инструкцией ***ERET***. Функция, определяемая SVC, обрабатывает синхронные исключения.

```
__svc void func(void)
{
   /* Do something */
}
```

См. Пример ниже.

Вызов супервизора

SVC - это инструкция A64, которая выполняет вызов супервизора, то есть исключение. Это обрабатывается синхронным вектором исключений. Компилятор IAR C / C ++ поддерживает замену обычной инструкции вызова, используемой для вызова функции, инструкцией SVC, используя директиву ***pragma svc_number*** перед любым объявлением или определением функции. Предоставленный номер будет сохранен в системном регистре ***ESR_EL1***.

```
#pragma svc_number = 23
__svc int Synchronous_Handler_A64(int i)
{
  return i;
}

void f()
{
  int i = Synchronous_Handler_A64(5); // Will use an SVC Будет использовать SVC
}
```

Предполагаемое использование функций SVC - позволить коду выполняться на более низком уровне исключения, вызывая код на более высоком уровне исключения.

```
// User code Код пользователя
#pragma svc_number = 1
int svc1(int);
#pragma svc_number = 2
int svc2(int);
int main(void)
{
  svc1(1);
}
// Supervisor code Код супервайзера
__svc int Synchronous_Handler_A64(int a)
{
  // Get syndrome: AARCH64 SVC Синдром Get: AARCH64 SVC
  long long nr = 0;
  __asm("MRS %x0, ESR_EL1\n" : "=r"(nr));
  int ec = (nr >> 26) & 0x3F;
  if (ec != 0x15)
    return -1;
  // Get SVC number. Получить номер SVC
  nr &= 0xFF'FFFF;
  return nr + a;
}
```

Функции, объявленные с помощью ***#pragma svc_number***, не обязательно должны использовать одну и ту же сигнатуру функции. Если используются разные подписи, ***Synchronous_Handler_A64*** должен быть написан на языке ассемблера как трамплин для различных вызывающих обработчиков, чтобы правильно передавать параметры и обрабатывать возвращаемые значения.

АДРЕС СБРОСА

По умолчанию предполагается, что адрес сброса находится на адресе 0. Чтобы разместить его на другом адресе, используйте один из следующих методов:

• Используйте параметр компоновщика ***--config_def***, чтобы задать символ конфигурации компоновщика.

__Reset_address, например: ***--config_def __Reset_address = 4096***

• Отредактируйте файл конфигурации компоновщика, который использует проект.

Встраивание функций

Встраивание функций означает, что функция, определение которой известно во время компиляции, интегрируется в тело вызывающей функции, чтобы устранить накладные расходы на вызов функции. Эта оптимизация, которая выполняется на уровне оптимизации High, обычно сокращает время выполнения, но может увеличить размер кода. Полученный код может стать труднее отлаживать. То, как происходит ли встраивание на самом деле, зависит от эвристики компилятора.

Компилятор эвристически решает, какие функции встраивать. Различные эвристики используются при оптимизации скорости, размера или при балансировании между размером и скоростью. Обычно размер кода не увеличивается при оптимизации по размеру.

СЕМАНТИКА В СРАВНЕНИИ C и C ++

В C ++ все определения конкретной встроенной функции в отдельных единицах перевода должны быть точно такими же. Если функция не встроена в одну или несколько единиц трансляции, то одно из определений из этих единиц трансляции будет использоваться в качестве реализации функции.

В C вы должны вручную выбрать одну единицу перевода, которая включает не встроенную версию встроенной функции. Вы делаете это, явно объявляя функцию как extern в этой единице перевода. Если вы объявите функцию как extern в нескольких единицах перевода, компоновщик выдаст ошибку множественного определения. Кроме того, в C встроенные функции не могут ссылаться на статические переменные или функции.

Например:

```
// In a header file. В заголовочном файле.
static int sX;
inline void F(void)
{
  //static int sY; // Cannot refer to statics. Не может ссылаться на static
  //sX;          // Cannot refer to statics.
}
// In one source file. В одном исходном файле.
// Declare this F as the non-inlined version to use.
// Объявите эту F как версию без встроенного кода для использования.
extern inline void F();
```

ОСОБЕННОСТИ УПРАВЛЕНИЯ ФУНКЦИЕЙ INLINING

Есть несколько механизмов для управления встраиванием функций:

• Ключевое слово inline сообщает компилятору, что функция, определенная сразу после директивы, должна быть встроена.

Если вы компилируете свою функцию в режиме C или C ++, ключевое слово будет интерпретироваться в соответствии с его определением в Стандартном C или Стандартном C ++ соответственно.

Основное различие в семантике заключается в том, что в стандарте C вы не можете (как правило) просто предоставить встроенное определение в файле заголовка. Вы должны предоставить внешнее определение в одном из модулей компиляции, обозначив встроенное определение как внешнее в этом модуле компиляции.

• ***# pragma inline*** похожа на ключевое слово ***inline***, но с той разницей, что компилятор всегда использует встроенную семантику C ++.

Используя встроенную директиву ***#pragma***, вы также можете отключить эвристику компилятора, чтобы принудительно или полностью отключить встраивание. Для получения дополнительной информации см. Стр. 411.

• ***--use_c ++ _ inline*** заставляет компилятор использовать семантику C ++ при компиляции стандартного файла исходного кода C.

• ***--no_inline, #pragma optimize = no_inline*** и ***#pragma inline =*** никогда не отключать встраивание функций. По умолчанию встраивание функций включено на уровне оптимизации High.

Компилятор может встроить функцию только в том случае, если определение известно. Обычно это ограничивается текущей единицей перевода. Однако, когда используется опция компилятора ***--mfc*** для многофайловой компиляции, компилятор может встроить определения из всех единиц перевода в многофайловой единице компиляции. Для получения дополнительной информации см. Многофайловые модули компиляции, стр. 245.

Дополнительные сведения об оптимизации встраивания функций см. В разделе Встраивание функций, стр. 248.

Защита стека

В программном обеспечении переполнение буфера стека происходит, когда программа выполняет запись в адрес памяти в стеке вызовов программы за пределами предполагаемой структуры данных, которая обычно является буфером фиксированной длины. Результатом почти всегда является повреждение ближайших данных, и это может даже изменить функцию, к которой нужно вернуться. Если это сделано преднамеренно, это часто называют разбиванием (рюмки) стека. Одним из методов защиты от переполнения буфера стека является использование канареек стека, названных по аналогии с использованием канареек в угольных шахтах.

ЗАЩИТА СТЕКА В КОМПИЛЯТОРЕ IAR C / C ++

Компилятор IAR C / C ++ для Arm поддерживает защиту стека.

Чтобы включить защиту стека для функций, которые, как считается, в ней нуждаются, используйте параметр компилятора ***--stack_protection***. Для получения дополнительной информации см. --Stack_protection, стр. 318.

Реализация защиты стека IAR Systems использует эвристику, чтобы определить, нуждается ли функция в защите стека или нет. Если какая-либо определенная локальная переменная имеет тип массива или тип структуры, который содержит член типа массива, функции потребуется защита стека. Кроме того, если адрес какой-либо локальной переменной распространяется за пределы функции, такая функция также потребует защиты стека.

Если функции требуется защита стека, локальные переменные сортируются, чтобы позволить переменным с типом массива быть помещенными как можно

выше в блоке стека функций. После этих переменных помещается канареечный элемент. Канарейка инициализируется при входе в функцию. Значение инициализации берется из глобальной переменной ***__stack_chk_guard***. При выходе из функции код проверяет, что канареечный элемент все еще содержит исходное значение. В противном случае вызывается функция ***__stack_chk_fail***.

ИСПОЛЬЗОВАНИЕ ЗАЩИТЫ СТЕКА В ПРИЛОЖЕНИИ

Чтобы использовать защиту стека, вы должны определить эти объекты в своем приложении:

• ***extern uint32_t __stack_chk_guard*** Глобальная переменная ***__stack_chk_guard*** должна быть инициализирована перед первым использованием. Если значение инициализации рандомизировано, оно будет более безопасным.

• ***__interwork __nounwind __noreturn void __stack_chk_fail (void)*** Назначение функции ***__stack_chk_fail*** - уведомить о проблеме и затем завершить приложение.

Примечание: адрес возврата из этой функции будет указывать на функцию, которая завершилась ошибкой.

Примечание переводчика: Stack Canary

Канарейки (canary) — это известные значения, которые помещаются между буфером и управляющими данными в стеке для мониторинга переполнения буфера. После переполнения буфера, первыми подлежащими повреждению данными, как правило, будет канарейка. Таким образом, значение канарейки будет проверяться и в случае неудачной проверки сигнализировать о переполнении буфера. Существует три типа канареек:

Terminator. Канарейки построены из нулевых терминаторов, CR, LF и -1. В результате злоумышленник должен написать нулевой символ перед записью адреса возврата, чтобы избежать изменения канарейки. Это предотвращает атаки с использованием strcpy() и других методов, которые возвращаются при копировании нулевого символа, в то время как нежелательным результатом является известность канарейки.

Random. Генерируются случайным образом. Обычно случайная канарейка генерируется при инициализации программы и сохраняется в глобальной переменной. Эта переменная обычно дополняется неотображенными страницами, поэтому попытка ее чтения с использованием любых уловок, использующих ошибки для чтения из ОЗУ, вызывает ошибку сегментации, завершающую программу.

Random XOR. Случайные канареи, которые ксорятся с контрольных данными. Таким образом, как только канарейка или контрольные данные будут засорены, значение канарейки будет неправильным. Имеют те же уязвимости, что и случайные канарейки, за исключением того, что метод «чтения из стека» для получения канарейки немного сложнее. Атакующий должен получить канарейку, алгоритм и контрольные данные, чтобы заново сгенерировать исходную канарейку, необходимую для подделки защиты.

Файл ***stack_protection.c*** в каталоге ***arm \ src \ lib \ runtime*** может использоваться как шаблон для ***__stack_chk_guard*** и ***__stack_chk_fail***.

Интерфейс TrustZone

TrustZone for Arm v8-M (32-разрядный режим) нуждается в некоторой поддержке компилятора для создания безопасного интерфейса между безопасным и незащищенным кодом. Для этого есть два атрибута типа функции, которые управляют генерацией кода:

__cmse_nonsecure_entry и ***__cmse_nonsecure_call***. Для получения дополнительной информации см. Arm TrustZone®, стр. 232.

Примечание. Поддержка TrustZone автоматическая в 64-битном режиме.

Компоновка с помощью ILINK

- Компоновка - обзор
- Модули и разделы
- Подробное описание процесса компоновки.
- Размещение кода и данных - файл конфигурации компоновщика.
- Инициализация при запуске системы.
- Анализ использования стека.

Компоновка - обзор

IAR ILINK Linker - это мощный и гибкий программный инструмент для использования при разработке встраиваемых приложений. Он одинаково хорошо подходит для компоновки небольших однофайловых программ на абсолютном ассемблере, а также для компоновки больших, перемещаемых, многомодульных программ на C / C ++ или смешанных программ на C / C ++ и ассемблере.

Компоновщик объединяет один или несколько перемещаемых объектных файлов, созданных компилятором или ассемблером IAR Systems, с выбранными частями одной или нескольких объектных библиотек для создания исполняемого изображения в стандартном для отрасли формате ***Executable and Linking Format (ELF)***.

Компоновщик автоматически загрузит только те библиотечные модули - пользовательские библиотеки и варианты стандартных библиотек C или C ++ - которые действительно необходимы приложению, которое вы связываете. Кроме того, компоновщик удаляет повторяющиеся разделы и разделы, которые не требуются.

ILINK может связывать как Arm так и Thumb код, а также их комбинацию. Автоматически вставляя дополнительные инструкции (виниры), ILINK гарантирует, что пункт назначения будет достигнут для любых вызовов и ответвлений, и что состояние процессора переключается при необходимости. Для получения дополнительной информации о том, как создавать виниры, см. Виниры, стр. 123.

Компоновщик использует ***configuration file*** (файл конфигурации), в котором вы можете указать отдельные местоположения для областей кода и данных карты памяти вашей целевой системы. Этот файл также поддерживает автоматическую обработку фазы инициализации приложения, что означает инициализацию областей глобальных переменных и областей кода путем копирования инициализаторов и, возможно, их распаковки.

Конечный результат, созданный ILINK, представляет собой абсолютный объектный файл, содержащий исполняемое изображение в формате ELF (включая DWARF для отладочной информации). Файл можно загрузить в C-SPY или любой другой совместимый отладчик, поддерживающий ELF / DWARF, или его можно сохранить в EPROM или флэш-памяти.

Для обработки файлов ELF включены различные инструменты. Для получения информации о включенных утилитах см. Специальные инструменты ELF, стр. 52.

Модули и разделы (Modules and sections)

Каждый перемещаемый объектный файл содержит один модуль, который состоит из:

- Несколько разделов кода или данных.

• Атрибуты среды выполнения, указывающие различные типы информации, например версию среды выполнения.

• При желании отладочная информация в формате DWARF.

• Таблица символов всех глобальных символов и всех используемых внешних символов.

Примечание. В библиотеке каждый модуль (исходный файл) должен содержать только одну единственную функцию. Это важно, если вы хотите заменить функцию в библиотеке функцией в собственном приложении. Компоновщик включает модули, только если на них ссылается остальная часть приложения. Если компоновщик включает в себя библиотечный модуль, который содержит несколько функций, потому что одна функция упоминается, а другая функция в этом модуле должна быть переопределена функцией, определенной вашим приложением, компоновщик выдает ошибку «повторяющиеся определения».

Section (Раздел) - это логический объект, содержащий фрагмент данных или кода, который должен быть размещен в физическом месте в памяти. Раздел может состоять из нескольких фрагментов раздела, обычно по одному для каждой переменной или функции (символов). Раздел можно разместить как в RAM, так и в ROM. В обычном встроенном приложении разделы, помещенные в оперативную память, не имеют содержимого, они только занимают место.

У каждого раздела есть имя и атрибут типа, который определяет содержимое. Атрибут type используется (вместе с именем) для выбора разделов для конфигурации ILINK.

Основное назначение атрибутов разделов - различать разделы, которые могут быть помещены в ПЗУ, и разделы, которые должны быть помещены в ОЗУ:

ro | readonly	ROM sections
rw | readwrite	RAM sections

В каждой категории разделы можно разделить на те, которые содержат код, и те, которые содержат данные, в результате чего можно выделить четыре основные категории:

ro code	Normal code	Нормальный код
ro data	Constants	Константы
rw code	Code copied to RAM	Код скопированный в RAM
rw data	Variables	Переменные

Данные ***readwrite*** также имеют подкатегорию ***zi | zeroinit*** - для разделов, которые инициализируются нулем при запуске приложения.

Примечание. В дополнение к этим типам разделов - разделам, которые содержат код и данные, являющиеся частью вашего приложения, - конечный объектный файл будет содержать многие другие типы разделов, например разделы, содержащие отладочную информацию или другой тип метаинформации.

Section (Раздел) - это наименьший элемент, который можно связать, но, если возможно, ILINK может исключить меньшие элементы - фрагменты раздела - из окончательного приложения. Для получения дополнительной информации см. Сохранение модулей, стр. 117 и Сохранение символов и разделов, стр. 117.

Во время компиляции данные и функции размещаются в разных разделах. Во время связывания одна из наиболее важных функций компоновщика - назначать адреса различным разделам, используемым приложением.

У инструментов сборки IAR есть много предопределенных имен разделов. Дополнительные сведения о каждом разделе см. В разделе «Справочник по разделам».

Вы можете группировать разделы для размещения с помощью блоков. См. Директиву define block, стр. 505.

Подробно о процессе Компоновки

Перемещаемые модули в объектных файлах и библиотеках, созданные компилятором и ассемблером IAR, не могут быть выполнены как есть. Чтобы стать исполняемым приложением, они должны быть связаны.

Примечание. Модули, созданные набором инструментов от другого поставщика, также могут быть включены в сборку, если модуль соответствует стандарту AEABI (Arm Embedded Application Binary Interface). Имейте в виду, что для этого также может потребоваться библиотека служебных программ компилятора от того же поставщика.

Компоновщик используется для процесса связывания. Обычно он выполняет следующую процедуру (обратите внимание, что некоторые из шагов могут быть отключены параметрами командной строки или директивами в файле конфигурации компоновщика):

• Определите, какие модули включить в приложение. Модули, представленные в объектных файлах, всегда включаются. Модуль в файл библиотеки включается только в том случае, если он предоставляет определение глобального символа, на который ссылается включенный модуль.

• Выберите, какие файлы стандартной библиотеки использовать. Выбор основан на атрибутах включенных модулей. Эти библиотеки затем используются для удовлетворения любых еще не определенных неопределенных символов.

• Обрабатывать символы с более чем одним определением. Если существует более одного неслабого определения, выдается ошибка. В противном случае выбирается одно из определений (неслабое, если оно есть), а остальные подавляются. Слабые определения обычно используются для встроенных и шаблонных функций. Если вам нужно переопределить некоторые неслабые определения из библиотечного модуля, вы должны убедиться, что библиотечный модуль не включен (обычно путем предоставления альтернативных определений для всех символов, которые ваше приложение использует в этом библиотечном модуле).

• Определите, какие разделы / фрагменты разделов из включенных модулей включить в приложение. Включаются только те разделы / фрагменты разделов, которые действительно необходимы приложению. Есть несколько способов определить, какие разделы / фрагменты разделов необходимы, например, с помощью атрибута объекта ***__root***, директивы ***#pragma required*** и директивы компоновщика ***keep***. В случае дублирования разделов включается только один.

• При необходимости организуйте инициализацию инициализированных переменных и кода в ОЗУ. Директива инициализации заставляет компоновщик создавать дополнительные разделы, чтобы разрешить копирование из ПЗУ в ОЗУ. Каждый раздел, который будет инициализирован копированием, разделен на два раздела: один для части ПЗУ, а другой - для части ОЗУ. Если ручная инициализация не используется, компоновщик также подстраивает код запуска для выполнения инициализации.

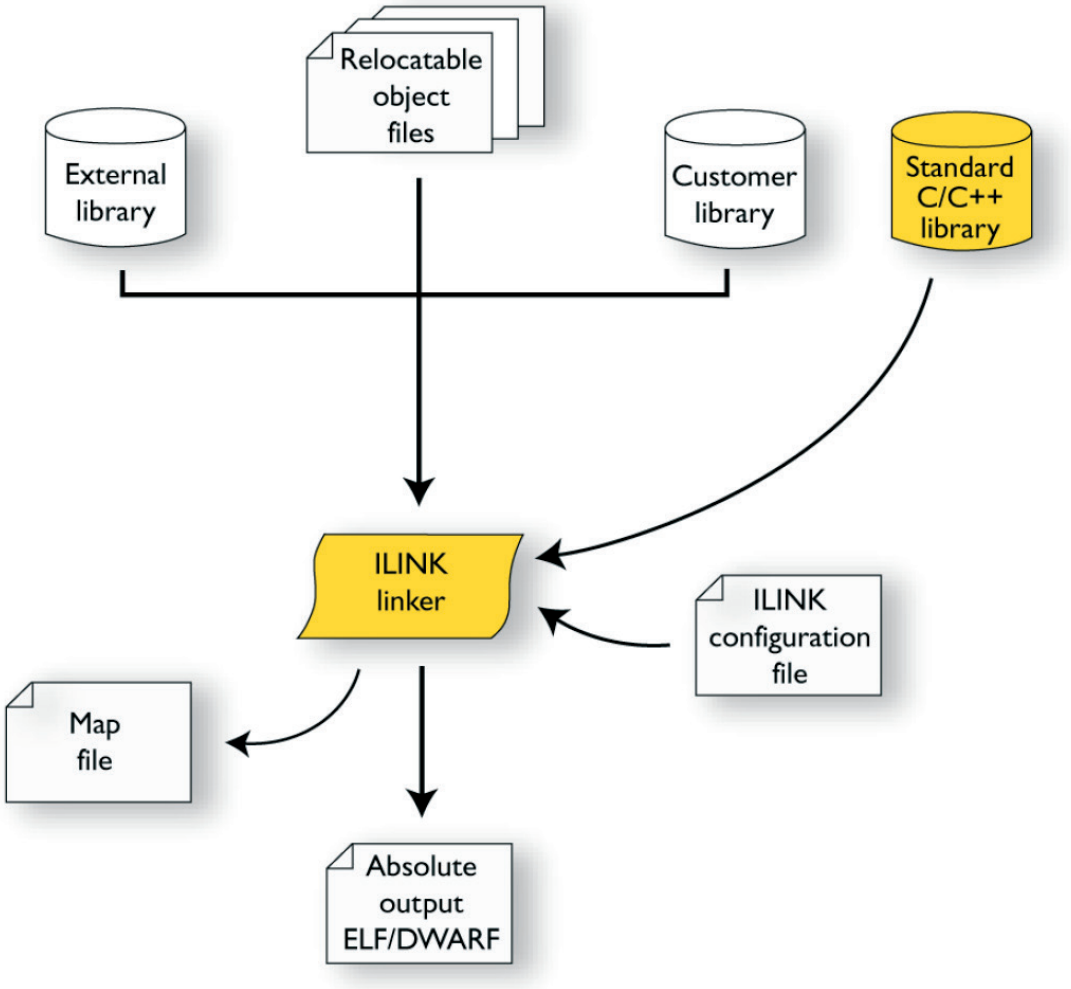
• Определите, где разместить каждый раздел в соответствии с директивами размещения разделов в файле конфигурации компоновщика. Разделы, которые должны быть инициализированы копированием, появляются дважды в сопо-

ставлении с директивами размещения, один раз для части ПЗУ и один раз для части RAM, с разными атрибутами. Во время размещения компоновщик также добавляет все необходимые виниры, чтобы ссылка на код достигла места назначения или переключать режимы ЦП.

• Создайте абсолютный файл, содержащий исполняемый образ и любую предоставленную отладочную информацию. Содержимое каждого необходимого раздела в перемещаемых входных файлах рассчитывается с использованием информации о перемещении, содержащейся в его файле, и адресов, определенных при размещении разделов. Этот процесс может привести к одному или нескольким сбоям перемещения, если некоторые из требований для конкретного раздела не выполняются, например, если размещение привело к тому, что адрес назначения для команды перехода относительно ПК оказался вне диапазона для этой инструкции.

• При желании создайте файл карты, в котором перечислены результаты размещения раздела, адрес каждого глобального символа и, наконец, сводка использования памяти для каждого модуля и библиотеки.

На этой иллюстрации показан процесс связывания:



Во время связывания ILINK может выдавать сообщения об ошибках и журналы на ***stdout*** и ***stderr***. Сообщения журнала полезны для понимания того, почему приложение было связано именно так. Например, почему был включен модуль или раздел (или фрагмент раздела).

Примечание. Чтобы увидеть фактическое содержимое объектного файла ELF, используйте ielfdumparm. См. The IAR ELF Dumper — ielfdump, стр. 551.

Размещение кода и данных - файл конфигурации компоновщика

Размещение разделов в памяти выполняется компоновщиком IAR ILINK. Он использует файл конфигурации компоновщика, в котором вы можете определить, как ILINK должен обрабатывать каждый раздел и как они должны быть помещены в доступную память.

Типичный файл конфигурации компоновщика содержит определения:

- Доступная адресуемая память
- Населенные регионы этой памяти
- Как обрабатывать разделы ввода
- Созданные разделы
- Как разместить разделы в доступных регионах.

Файл состоит из последовательности декларативных директив. Это означает, что процесс связывания будет регулироваться всеми директивами одновременно.

Чтобы использовать один и тот же исходный код с разными производными, просто перестройте код с соответствующим файлом конфигурации.

ПРОСТОЙ ПРИМЕР КОНФИГУРАЦИОННОГО ФАЙЛА

Предположим, что простая 32-разрядная архитектура имеет следующие требования к памяти:

- Имеется 4 Гбайт адресуемой памяти.
- Имеется ПЗУ в диапазоне адресов 0x0000–0x10000.
- Имеется оперативная память в диапазоне 0x20000–0x30000.
- Стек имеет выравнивание 8.
- Код запуска системы должен располагаться по фиксированному адресу.

Простой файл конфигурации для этой предполагаемой архитектуры может выглядеть так:

```
/* The memory space denoting the maximum possible amount of addressable memory */
define memory Mem with size = 4G;
/* Memory regions in an address space */
define region ROM = Mem:[from 0x00000 size 0x10000];
define region RAM = Mem:[from 0x20000 size 0x10000];
/* Create a stack */
define block STACK with size = 0x1000, alignment = 8 { };
/* Handle initialization */
initialize by copy { readwrite }; /* Initialize RW sections */
/* Place startup code at a fixed address */
place at start of ROM { readonly section .cstartup };
/* Place code and data */
place in ROM { readonly }; /* Place constants and initializers in
                              ROM: .rodata and .data_init    */
place in RAM { readwrite,  /* Place .data, .bss, and .noinit */
               block STACK }; /* and STACK                   */
```

Этот файл конфигурации определяет одну адресуемую память Mem с максимальным объемом памяти 4 ГБ. Кроме того, он определяет область ROM и область RAM в Mem, а именно ROM и RAM. Каждый регион имеет размер 64 Кбайт.

Затем файл создает пустой блок с именем STACK размером 4 Кбайта, в котором будет размещаться стек приложения. Создание блока (***block***) - это основной метод, который вы можете использовать для детального управления размещением, размером и т. д. Его можно использовать для группировки разделов, а также, как в этом примере, для указания размера и размещения области памяти.

Затем файл определяет, как обрабатывать инициализацию переменных, разделов типа чтения/записи (***readwrite)***. В этом примере инициализаторы помещаются в ПЗУ и копируются при запуске приложения в область ОЗУ. По умолчанию ILINK может сжимать инициализаторы, если это представляется целесообразным.

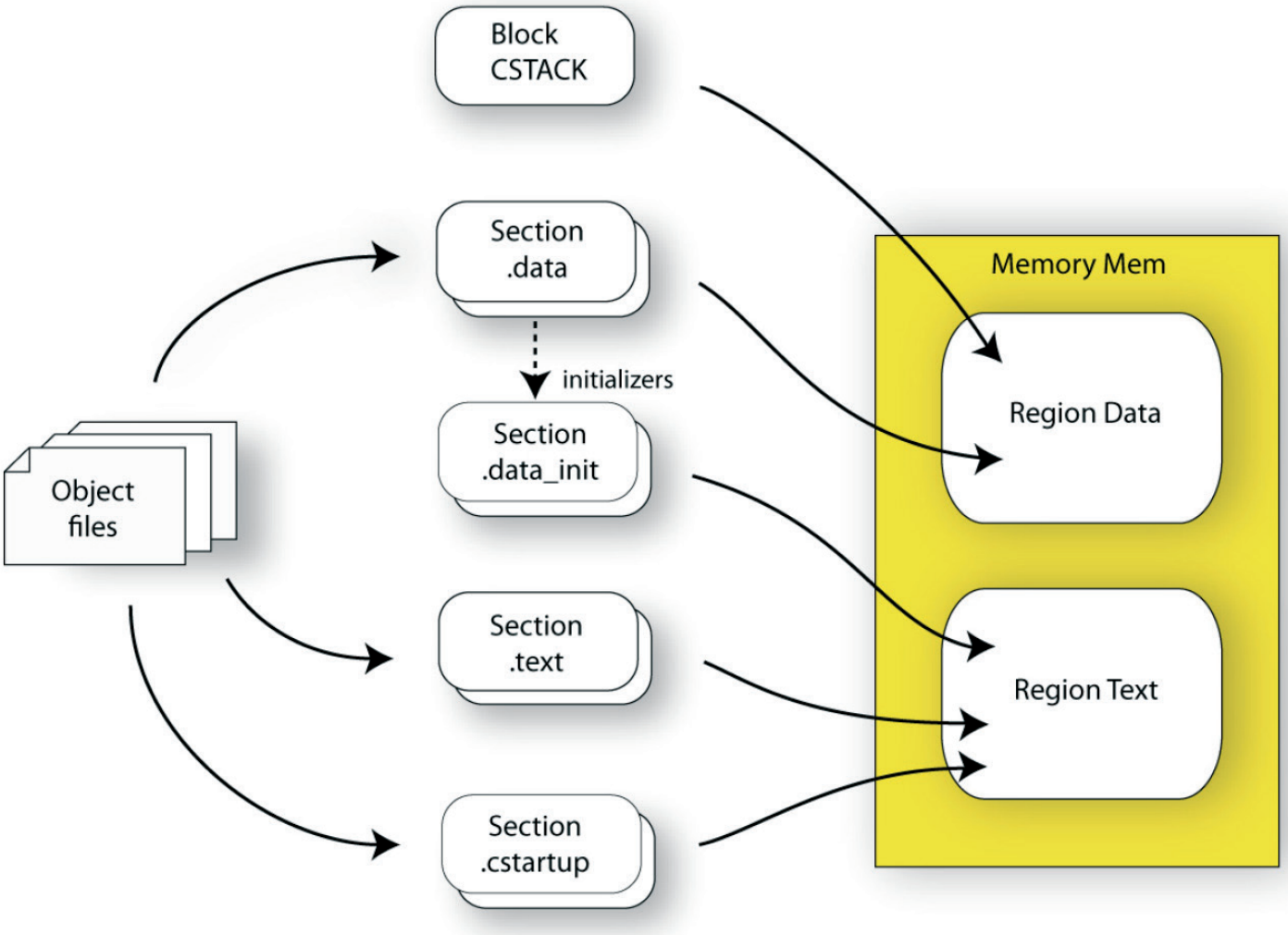
Последняя часть файла конфигурации обрабатывает фактическое размещение всех разделов в доступных регионах. Во-первых, код запуска, определенный как находящийся в разделе ***(readonly)*** section ***.cstartup***, помещается в начало области ПЗУ, то есть по адресу ***0x10000***.

Примечание. Часть внутри {} называется ***section selection*** *(выбором раздела), и она выбирает разделы, к которым должна применяться директива. Затем остальные разделы только для чтения помещаются в область ПЗУ.*

Примечание. Выбор раздела ***{readonly section .cstartup}*** *имеет приоритет над более общим выбором раздела* ***{readonly}.***

Наконец, разделы чтения / записи (readwrite) и блок STACK помещаются в область RAM.

На этой иллюстрации схематично показано, как приложение размещается в памяти:



В дополнение к этим стандартным директивам файл конфигурации может содержать директивы, определяющие, как:

- Сопоставьте память, к которой можно обращаться разными способами.
- Обрабатывать условные директивы.
- Создавайте символы со значениями, которые можно использовать в приложении.
- Более подробно выбирать разделы, к которым должна применяться директива.
- Более подробно инициализировать код и данные.

Дополнительные сведения и примеры настройки файла конфигурации компоновщика см. В главе «Связывание приложения».

Дополнительные сведения о файле конфигурации компоновщика см. В главе Файл конфигурации компоновщика.

Инициализация при запуске системы

В стандарте C все статические переменные - переменные, которые выделяются по фиксированному адресу памяти - должны быть инициализированы системой времени выполнения известным значением при запуске приложения. Это значение либо явное значение, присвоенное переменной, либо, если значение не задано, оно сбрасывается до нуля. В компиляторе есть исключения из этого правила, например, переменные, объявленные ***__no_init***, которые вообще не инициализируются.

Компилятор генерирует определенный тип раздела для каждого типа инициализации переменной:

Categories of declared data	Source	Section type	Section name	Section content
Zero-initialized data	`int i;`	Read/write data, zero-init	`.bss`	None
Zero-initialized data	`int i = 0;`	Read/write data, zero-init	`.bss`	None
Initialized data (non-zero)	`int i = 6;`	Read/write data	`.data`	The initializer
Non-initialized data	`__no_init int i;`	Read/write data, zero-init	`.noinit`	None
Constants	`const int i = 6;`	Read-only data	`.rodata`	The constant
Code	`__ramfunc void myfunc() {}`	Read/write code	`.textrw`	The code

Table 3: Sections holding initialized data

Примечание. Кластеризация статических переменных может группировать инициализированные нулем переменные вместе с инициализированными данными в .data. Компилятор может решить поместить константы в раздел .text, чтобы избежать загрузки адреса константы из таблицы констант.

Для получения информации обо всех поддерживаемых разделах см. Раздел Section reference.

ПРОЦЕСС ИНИЦИАЛИЗАЦИИ

Инициализацией данных совместно занимается ILINK и код запуска системы.

Чтобы настроить инициализацию переменных, необходимо учитывать следующие вопросы:

• Разделы, которые должны быть инициализированы нулями или не инициализированы вообще (***__no_init***), автоматически обрабатываются ILINK.

• Разделы, которые должны быть инициализированы, за исключением разделов с нулевой инициализацией, должны быть перечислены в ***initialize*** directive (директиве инициализации).

Обычно во время связывания секция, которая должна быть инициализирована, разделяется на две секции, где исходная инициализированная секция сохраняет имя. Содержимое помещается в новый раздел инициализатора, который получит исходное имя с суффиксом ***_init***. Инициализаторы должны быть помещены в ПЗУ, а инициализированные разделы в ОЗУ с помощью директив размещения. Самый распространенный пример - это ***.data*** section (раздел), который компоновщик разбивает на ***.data*** и ***.data_init***.

• Разделы, содержащие константы, не должны инициализироваться - их следует размещать только во флэш-памяти / ПЗУ.

В файле конфигурации компоновщика это может выглядеть так:

```
/* Handle initialization */
initialize by copy { readwrite }; /* Initialize RW sections */
/* Place startup code at a fixed address */
place at start of ROM { readonly section .cstartup };
/* Place code and data */
place in ROM { readonly }; /* Place constants and initializers in
                           ROM: .rodata and .data_init    */
place in RAM { readwrite,  /* Place .data, .bss, and .noinit */
               block STACK }; /* and STACK                  */
```

Примечание. Когда используются сжатые инициализаторы (см. Директиву инициализации, стр. 511), разделы содержимого (то есть разделы с суффиксом ***_init****) не указываются как отдельные разделы в файле карты. Вместо этого они объединяются в агрегаты «initializer bytes (байтов инициализатора)». Вы можете разместить разделы содержимого обычным способом в файле конфигурации компоновщика, однако это влияет на размещение - и, возможно, количество - агрегатов «байтов инициализатора».*

Дополнительные сведения и примеры настройки инициализации см. В разделе Рекомендации по связыванию, стр. 113.

ДИНАМИЧЕСКАЯ ИНИЦИАЛИЗАЦИЯ C ++

Компилятор помещает указатели подпрограмм для выполнения динамической инициализации C ++ в разделы типа раздела ***ELF SHT_PREINIT_ARRAY*** и ***SHT_INIT_ARRAY***. По умолчанию компоновщик помещает их в созданный компоновщиком блок, гарантируя, что все разделы типа ***SHT_PREINIT_ARRAY*** помещаются перед разделами типа S***HT_INIT_ARRAY***. Если были включены какие-либо такие разделы, также будет включен код для вызова подпрограмм.

Блоки, созданные компоновщиком, создаются только в том случае, если конфигурация компоновщика не содержит шаблонов селектора секций для типов секций ***preinit_array*** и ***init_array***. Эффект блоков, созданных компоновщиком, будет очень похож на то, что происходит, если файл конфигурации компоновщика содержит это:

```
define block SHT$$PREINIT_ARRAY { preinit_array };
define block SHT$$INIT_ARRAY { init_array };
define block CPP_INIT with fixed order { block
                                SHT$$PREINIT_ARRAY,
                                block SHT$$INIT_ARRAY };
```

Если вы поместите это в файл конфигурации компоновщика, вы также должны упомянуть блок ***CPP_INIT*** в одной из директив размещения раздела. Если вы хотите выбрать место размещения блока, созданного компоновщиком, вы можете использовать селектор раздела с именем "***.init_array***".

См. Также переключатели секций, стр. 519.

Анализ использования стека

В этом разделе описывается, как выполнить анализ использования стека с помощью компоновщика.

В каталоге arm \ src вы можете найти пример проекта, демонстрирующий анализ использования стека.

ВВЕДЕНИЕ В АНАЛИЗ ИСПОЛЬЗОВАНИЯ СТЕКОВ

При правильных обстоятельствах компоновщик может точно рассчитать максимальное использование стека для каждого графа вызовов, начиная с запуска программы, функций прерывания, задач и т. д. (Каждая функция, которая не вызывается из другой функции, другими словами, корень).

Если вы включите анализ использования стека, в файл карты компоновщика будет добавлена глава об использовании стека, в которой для каждого корня графа вызовов будет указана конкретная цепочка вызовов, что приводит к максимальной глубине стека.

Анализ будет точным только в том случае, если есть точная информация об использовании стека для каждой функции в приложении.

В общем, компилятор будет генерировать эту информацию для каждой функции C, но если в вашем приложении есть косвенные вызовы - вызовы с использованием указателей на функции, вы должны предоставить список возможных функций, которые могут быть вызваны из каждой вызывающей функции.

Если вы используете файл управления использованием стека, вы также можете предоставить информацию об использовании стека для функций в модулях, которые не имеют информации об использовании стека.

Вы можете использовать директиву ***check that*** в вашем файле управления использованием стека, чтобы проверить, что использование стека, вычисленное компоновщиком, не превышает выделенное вами пространство стека.

ПРОВЕДЕНИЕ АНАЛИЗА ИСПОЛЬЗОВАНИЯ СТЕКА

1 Включите анализ использования стека:

В среде IDE выберите

Project>Options>Linker>Advanced>Enable stack usage analysis.

В командной строке используйте параметр компоновщика --enable_stack_usage.

См. --Enable_stack_usage, стр. 337.

2 Включите файл карты компоновщика:

В среде IDE выберите

Project>Options>Linker>List>Generate linker map file.

В командной строке используйте параметр компоновщика ***--map***

3 Скомпонуйте свой проект.

Примечание. Компоновщик будет выдавать предупреждения, связанные с использованием стека при определенных обстоятельствах, см. Ситуации, когда появляются предупреждения, стр. 110.

4 Просмотрите файл карты компоновщика, который теперь содержит главу об использовании стека со сводкой использования стека для каждого корня графа вызовов. Для получения дополнительной информации см. Результат анализа - содержимое файла карты, стр. 106.

5 Для получения дополнительных сведений проанализируйте журнал графика вызовов, см. Журнал графика вызовов, стр. 110.

Примечание. В анализе есть ограничения и источники неточности, см. Ограничения, стр. 109.

Возможно, вам потребуется указать дополнительную информацию компоновщику, чтобы получить более представительный результат. См. Раздел Указание дополнительной информации об использовании стека, стр.

В среде IDE выберите

Project>Options>Linker>Advanced>Enable stack usage analysis>Control file.

В командной строке используйте параметр компоновщика ***--stack_usage_control***.

См. --Stack_usage_control, стр. 357.

6 Чтобы добавить автоматическую проверку того, что вы выделили достаточно памяти для стека, используйте директиву ***check that*** в файле конфигурации компоновщика. Например, предполагая, что блок стека называется MY_STACK, вы можете написать так:

```
check that size(block MY_STACK) >=maxstack("Program entry")
                        + totalstack("interrupt") + 100;
```

При связывании компоновщик выдает ошибку, если проверка не удалась. В этом примере будет выдана ошибка, если сумма следующего превышает размер блока MY_STACK:

• Максимальное использование стека в категории «Запись программы» (основная программа).

• Сумма каждого отдельного максимального использования стека в категории прерывание (при условии, что всем подпрограммам обработки прерываний требуется пространство одновременно).

• Запас безопасности 100 байт (для учета использования стека, не видимого для анализа).

См. Также директиву "Проверьте эту директиву" на стр. 523 и "Особенности стека" на стр. 216.

РЕЗУЛЬТАТ АНАЛИЗА - СОДЕРЖАНИЕ ФАЙЛА MAP

Когда включен анализ использования стека, файл карты компоновщика содержит главу об использовании стека со сводкой использования стека для каждой

корневой категории графа вызовов и перечисляет цепочку вызовов, которая приводит к максимальной глубине стека для каждого корня графа вызовов. Это пример того, как может выглядеть глава об использовании стека в файле карты:

```
**************************************************************************
    *** STACK USAGE
    ***
    Call Graph Root Category  Max Use  Total Use
    ------------------------  -------  ---------
    interrupt                 104      136
    Program entry             168      168
Program entry
    "__iar_program_start": 0x000085ac
    Maximum call chain                 168 bytes
        "__iar_program_start"          0
        "__cmain"                      0
        "main"                         8
        "printf"                       24
        "_PrintfTiny"                  56
        "_Prout"                       16
        "putchar"                      16
        "__write"                      0
        "__dwrite"                     0
        "__iar_sh_stdout"              24
        "__iar_get_ttio"               24
        "__iar_lookup_ttioh"           0
interrupt
    "FaultHandler": 0x00008434
    Maximum call chain                 32 bytes
        "FaultHandler"                 32
interrupt
    "IRQHandler": 0x00008424
    Maximum call chain                     104 bytes
        "IRQHandler"                       24
        "do_something" in suexample.o [1]  80
```

Сводка содержит глубину самой глубокой цепочки вызовов в каждой категории, а также сумму глубин самой глубокой цепочки вызовов в этой категории.

Каждый корень графа вызовов принадлежит к корневой категории графа вызовов, чтобы обеспечить удобные вычисления для проверки этих директив.

УКАЗАНИЕ ДОПОЛНИТЕЛЬНОЙ ИНФОРМАЦИИ ОБ ИСПОЛЬЗОВАНИИ СТЕКА

Чтобы указать дополнительную информацию об использовании стека, вы можете использовать либо файл управления использованием стека (***suc***), в котором вы указываете директивы управления использованием стека, либо аннотируйте исходный код.

Вы можете:

• Указать полную информацию об использовании стека (корневую категорию графа вызовов, использование стека и возможные вызовы) для функции, используя функцию директивы управления использованием стека. Обычно вы делаете это, если информация об использовании стека отсутствует, например, в модуле ассемблера. В вашем ***suc*** файле вы можете, например, написать так:

```
function MyFunc: 32,
calls MyFunc2,
calls MyFunc3, MyFunc4: 16;
function [interrupt] MyInterruptHandler: 44;
```

См. Также функциональную директиву, стр. 538.

• Исключить определенные функции из анализа использования стека с помощью директивы управления использованием стека ***exclude***. В вашем ***suc*** файле вы можете, например, написать так:

```
exclude MyFunc5, MyFunc6;
```

См. Также директиву exclude, стр. 538.

• Указать список возможных адресатов для косвенных вызовов в функции, используя возможные вызовы директивы управления использованием стека. Используйте это для функций, которые, как известно, выполняют косвенные вызовы, и где вы точно знаете, какие функции могут быть вызваны в этом конкретном приложении. В вашем ***suc*** файле вы можете, например, написать так:

```
possible calls MyFunc7: MyFunc8, MyFunc9;
```

Если информация о том, какие функции могут быть вызваны, доступна во время компиляции, рассмотрите возможность использования вместо этого директивы ***#pragma calls***.

См. Также директиву о возможных вызовах на стр. 540 и о вызовах на стр. 403.

• Указать, что функции являются корнями графа вызовов, включая необязательную корневую категорию графа вызовов, с помощью директивы управления использованием стека или директивы ***#pragma call_graph_root***. В вашем ***suc*** файле вы можете, например, написать так:

```
call graph root [task]: MyFunc10, MyFunc11;
```

Если ваши функции прерывания еще не были определены компилятором как корни графа вызовов, вы должны сделать это вручную. Вы можете сделать это либо с помощью директивы ***#pragma call_graph_root*** в исходном коде, либо указав директиву в вашем ***suc*** файле, например:

```
call graph root [interrupt]: Irq1Handler, Irq2Handler;
```

См. Также корневую директиву графа вызовов, стр. 538 и call_graph_root, стр. 404.

• Указать максимальное количество итераций в любом из циклов в гнезде рекурсии, членом которого является функция.

В вашем ***suc*** файле вы можете, например, написать так:

```
max recursion depth MyFunc12: 10;
```

• Выборочно подавить предупреждение о неупомянутых функциях, на которые ссылается модуль, для которого вы предоставили информацию об использовании стека в файле управления использованием стека. Используйте директиву ***no calls from*** в вашем ***suc*** файле, например, следующим образом:

```
no calls from [file.o] to MyFunc13, MyFunc14;
```

• Вместо указания информации об использовании стека для модулей ассемблера в файле управления использованием стека, вы можете аннотировать источник ассемблера информацией о кадре вызова.

Для получения дополнительной информации см. Руководство пользователя ассемблера IAR для Arm. Для получения дополнительной информации см. Главу Файл управления использованием стека.

ОГРАНИЧЕНИЯ

Помимо отсутствующей или неверной информации об использовании стека, есть и другие источники неточности в анализе:

• Компоновщик не всегда может идентифицировать все функции в объектных модулях, в которых отсутствует информация об использовании стека. В частности, это может быть проблема с объектными модулями, написанными на языке ассемблера или созданными инструментами, отличными от IAR. Вы можете предоставить информацию об использовании стека для таких модулей, используя файл управления использованием стека, а для модулей языка ассемблера вы также можете аннотировать исходный код ассемблера директивами CFI, чтобы предоставить информацию об использовании стека. См. Руководство пользователя ассемблера IAR для Arm.

• Если вы используете встроенный ассемблер для изменения размера кадра или для выполнения вызовов функций, это не будет отражено в анализе.

• Дополнительное пространство, используемое другими источниками (процессором, операционной системой и т. д.), Не учитывается.

• Исходный код, использующий исключения, не поддерживается.

• Если вы используете другие формы вызовов функций, например программные прерывания, они не будут отражены в графе вызовов.

• Использование многофайловой компиляции (***--mfc***) может помешать использованию файла управления использованием стека для определения свойств локальных для модуля функций в задействованных файлах.

Примечание. Анализ использования стека дает наихудший результат. Программа может и не попасть в максимальную цепочку вызовов по замыслу или по стечению обстоятельств. В частности, набор возможных мест назначения для вызова виртуальной функции в C ++ может иногда включать в себя реализации рассматриваемой функции, которые фактически не могут быть вызваны из этой точки кода.

Анализ использования стека является лишь дополнением к фактическим измерениям. Если результат важен, необходимо провести независимую проверку результатов анализа.

СИТУАЦИИ, ВЫДАЮЩИЕ ПРЕДУПРЕЖДЕНИЯ

Когда в компоновщике включен анализ использования стека, предупреждения будут генерироваться в следующих случаях:

• Есть функция без информации об использовании стека.

• В приложении есть сайт косвенного вызова, для которого не предоставлен список возможных вызываемых функций.

• Нет известных косвенных вызовов, но есть невызванная функция, которая, как известно, не является корнем графа вызовов.

• Приложение содержит рекурсию (цикл в графе вызовов), для которой не указана максимальная глубина рекурсии или которая имеет форму, для которой компоновщик не может рассчитать надежную оценку использования стека.

• Есть вызовы функции, объявленной как корень графа вызовов.

• Вы использовали файл управления использованием стека для предоставления информации об использовании стека для функций в модуле, который не имеет такой информации, и есть функции, на которые ссылается этот модуль, которые не были упомянуты как вызываемые в файле управления использованием стека.

ЖУРНАЛ ВЫЗОВОВ

Чтобы помочь вам интерпретировать результаты анализа использования стека, есть опция вывода журнала, которая создает простое текстовое представление графа вызовов (***--log call_graph***)

Пример вывода:

```
Program entry:
  0 __iar_program_start [168]
    0 __cmain [168]
      0 __iar_data_init3 [16]
        8 __iar_zero_init3 [8]
          16 - [0]
        8 __iar_copy_init3 [8]
          16 - [0]
      0 __low_level_init [0]
      0 main [168]
        8 printf [160]
          32 _PrintfTiny [136]
            88 _Prout [80]
              104 putchar [64]
                120 __write [48]
                  120 __dwrite [48]
                    120 __iar_sh_stdout [48]
                      144 __iar_get_ttio [24]
                        168 __iar_lookup_ttioh [0]
                    120 __iar_sh_write [24]
                      144 - [0]
            88 __aeabi_uidiv [0]
              88 __aeabi_idiv0 [0]
            88 strlen [0]
  0 exit [8]
    0 _exit [8]
      0 __exit [8]
        0 __iar_close_ttio [8]
          8 __iar_lookup_ttioh [0] ***
  0 __exit [8] ***
```

Каждая строка состоит из этой информации:

• Использование стека в момент вызова функции.

• Имя функции или одиночный ’-’ для обозначения использования в функции в точке без вызова функции (обычно в листовой функции).

• Использование стека в самой глубокой цепочке вызовов с этой точки. Если такое значение не может быть вычислено, вместо него выводится «[---]». «***» отмечает функции, которые уже были показаны.

ВЫВОД ГРАФИКА XML

Компоновщик также может создавать файл графа вызовов в формате XML. Этот файл содержит по одному узлу для каждой функции в вашем приложении с информацией об использовании стека и вызовах, относящейся к этой функции. Он предназначен для использования в инструментах постобработки и не особо удобочитаем.

Дополнительные сведения об используемом формате XML см. В файле callGraph.txt в установке продукта.

Компоновка вашего приложения

- Рекомендации по Компоновке
- Советы по устранению неполадок.
- Проверка согласованности модуля.
- Оптимизация компоновщика.

Рекомендации по Компоновке

Прежде чем вы сможете связать свое приложение, вы должны настроить конфигурацию, требуемую ILINK. Как правило, необходимо учитывать:

ВЫБОР ФАЙЛА КОНФИГУРАЦИИ ЛИНКЕРА

Каталог config содержит готовые шаблоны для файлов конфигурации компоновщика (* .icf) для всех поддерживаемых ядер.

Файлы содержат информацию, требуемую ILINK. Единственное изменение, если оно есть, обычно необходимо внести в поставляемый файл конфигурации, - это настроить начальный и конечный адреса каждой области так, чтобы они соответствовали карте памяти целевой системы. Если, например, ваше приложение использует дополнительную внешнюю оперативную память, вы также должны добавить сведения об области внешней оперативной памяти.

Для некоторых устройств автоматически выбираются файлы конфигурации для конкретных устройств.

Чтобы отредактировать файл конфигурации компоновщика, используйте редактор в среде IDE или любой другой подходящий редактор. Либо выберите Project>Options>Linker и нажмите кнопку «Изменить» на странице «Конфигурация», чтобы открыть специальный редактор файла конфигурации компоновщика.

Не изменяйте исходный файл шаблона. Мы рекомендуем вам сделать копию в рабочем каталоге и вместо этого изменить копию. Если вы используете редактор файла конфигурации компоновщика в среде IDE, среда IDE сделает для вас копию.

Каждый проект в среде IDE должен иметь ссылку на один и только один файл конфигурации компоновщика. Этот файл можно редактировать, но для боль-

шинства проектов достаточно настроить жизненно важные параметры в Project> Options> Linker> Config.

ОПРЕДЕЛЕНИЕ СОБСТВЕННЫХ ОБЛАСТЕЙ ПАМЯТИ

Выбранный вами файл конфигурации по умолчанию имеет предварительно определенные области ПЗУ и ОЗУ. Этот пример будет использоваться в качестве отправной точки для всех дальнейших примеров в этой главе. Обратите внимание, что все примеры относятся к 32-битному режиму, если не указано иное.

```
/* Define the addressable memory */
define memory Mem with size = 4G;

/* Define a region named ROM with start address 0 and to be 64 Kbytes large */
define region ROM = Mem:[from 0 size 0x10000];

/* Define a region named RAM with start address 0x20000 and to be 64 Kbytes large */
define region RAM = Mem:[from 0x20000 size 0x10000];
```

Каждое определение региона должно быть адаптировано к реальному оборудованию.

Чтобы узнать, какая часть каждой памяти была заполнена кодом и данными после связывания, просмотрите сводку памяти в файле карты (параметр командной строки ***--map***).

Добавление дополнительного региона

Чтобы добавить дополнительный регион, используйте директиву define region, например:

```
/* Define a 2nd ROM region to start at address 0x80000 and to be 128 Kbytes large */
define region ROM2 = Mem:[from 0x80000 size 0x20000];
```

Объединение разных областей в один регион

Если регион состоит из нескольких областей, используйте выражение региона, чтобы объединить разные области в одну, например:

```
/* Define the 2nd ROM region to have two areas. The first with
the start address 0x80000 and 128 Kbytes large, and the 2nd with
the start address 0xC0000 and 32 Kbytes large */
/* Определите 2-ю область ПЗУ, чтобы иметь две области.
Первую с начальным адресом 0x80000 и размером 128 Кбайт,
а вторую с начальным адресом 0xC0000 и размером 32 Кбайт */

define region ROM2    =    Mem:[from 0x80000 size 0x20000]
                           | Mem:[from 0xC0000 size 0x08000];
```

или эквивалентно

```
define region ROM2    =    Mem:[from 0x80000 to 0xC7FFF]
                           –Mem:[from 0xA0000 to 0xBFFFF];
```

РАЗМЕЩЕНИЕ СЕКЦИЙ

Выбранный вами файл конфигурации по умолчанию помещает все предопределенные разделы в память, но бывают ситуации, когда вы можете захотеть изменить это. Например, если вы хотите разместить секцию, содержащую постоянные символы, в области ***CONSTANT***, а не в месте по умолчанию. В этом случае используйте ***place*** в директиве, например:

```
/* Place sections with readonly content in the ROM region */
/* Поместите разделы с содержимым только для чтения в область ROM  */

place in ROM {readonly};

/* Place the constant symbols in the CONSTANT region */
/* Поместите символы констант в область CONSTANT */

place in CONSTANT {readonly section .rodata};
```

Примечание. Размещение раздела, используемого инструментами сборки IAR, в другой памяти, которая использует другой способ обращения к его содержимому, не удастся.

Чтобы узнать результат каждой директивы размещения после связывания, просмотрите сводку размещения в файле карты (параметр командной строки --map).

Размещение раздела по определенному адресу в памяти

Чтобы разместить раздел по определенному адресу в памяти, используйте директиву place at, например:

```
/* Place section .vectors at address 0 Поместите раздел .vectors по адресу 0 */
place at address Mem:0x0 {readonly section .vectors};
```

Размещение раздела первым или последним в регионе

Размещение раздела первым или последним в регионе аналогично, например:

```
/* Place section .vectors at start of ROM Поместите раздел .vectors в начало ПЗУ */
place at start of ROM {readonly section .vectors};
```

Объявление и размещение своих разделов

Чтобы объявить новые разделы - в дополнение к тем, которые используются инструментами сборки IAR - для хранения определенных частей вашего кода или данных, используйте механизмы в компиляторе и ассемблере. Например:

```
/* Place a variable in that section. Поместите переменную в этот раздел. */
const short MyVariable @ "MYOWNSECTION" = 0xF0F0;
```

Это соответствующий пример на языке ассемблера:

```
    name  createSection
          section MYOWNSECTION:CONST        ; Create a section,   Создайте раздел
                                            ; and fill it with    и заполните его
          dc16    0xF0F0                    ; constant bytes.     байтовой константой
    end
```

Чтобы разместить свой новый раздел, исходное место в ПЗУ ***{readonly}***; директивы достаточно.

Однако, чтобы явно разместить раздел ***MyOwnSection***, обновите файл конфигурации компоновщика, указав место в директиве, например:

```
/* Place MyOwnSection in the ROM region Поместите MyOwnSection в область ROM */
place in ROM {readonly section MyOwnSection};
```

РЕЗЕРВИРОВАНИЕ МЕСТА В ОЗУ

Часто приложение должно иметь пустую неинициализированную область памяти, которая будет использоваться для временного хранения, например, куча или стек. Проще всего добиться этого во время компоновки.

Вы должны создать блок с указанным размером, а затем поместить его в память.

В файле конфигурации компоновщика это может выглядеть так:

```
define block TempStorage with size = 0x1000, alignment = 4 { };
place in RAM { block TempStorage };
```

Чтобы получить начало выделенной памяти из приложения, исходный код может выглядеть следующим образом:

```
/* Define a section for temporary storage. Определите раздел для временного хранения. */
#pragma section = "TempStorage"
char *GetTempStorageStartAddress()
{
  /* Return start address of section TempStorage. */
  /* Вернуть начальный адрес раздела TempStorage. */
  return __section_begin("TempStorage");
}
```

СОХРАНЕНИЕ МОДУЛЕЙ

Если модуль связан как объектный файл, он всегда сохраняется. То есть он будет способствовать связанному приложению. Однако, если модуль является частью библиотеки, он включается только в том случае, если на него символически ссылаются из других частей приложения. Это верно, даже если библиотечный модуль содержит корневой символ. Чтобы гарантировать, что такой библиотечный модуль всегда включен, используйте ***iarchive*** для извлечения модуля из библиотеки, см. Инструмент ***IAR Archive - iarchive***, стр. 545.

Для получения информации о включенных и исключенных модулях просмотрите файл журнала (параметр командной строки ***--log modules***).

Для получения дополнительной информации о модулях см. Модули и разделы, стр. 96.

СОХРАНЕНИЕ СИМВОЛОВ И РАЗДЕЛОВ

По умолчанию ILINK удаляет все разделы, фрагменты разделов и глобальные символы, которые не нужны приложению. Чтобы сохранить символ, который кажется ненужным - или фактически фрагмент раздела, в котором он определен - вы можете либо использовать корневой атрибут символа в исходном коде C / C ++ или ассемблера, либо использовать параметр ILINK ***--keep*** (хранить). Чтобы сохранить разделы на основе имен атрибутов или имен объектов, используйте директиву ***keep*** в файле конфигурации компоновщика.

Чтобы ILINK не исключал разделы и фрагменты разделов, используйте параметры командной строки ***--no_remove*** или ***--no_fragments*** соответственно.

Для получения информации о включенных и исключенных символах и разделах просмотрите файл журнала (параметр командной строки ***--log sections***).

Для получения дополнительной информации о процедуре связывания для сохранения символов и разделов см. Процесс связывания, стр. 62.

ЗАПУСК ПРИЛОЖЕНИЯ В 32-БИТНОМ РЕЖИМЕ

По умолчанию точка начала выполнения приложения определяется меткой ***__iar_program_start***, которая указывает на начало файла cstartup.s. Метка также передается через ELF любому используемому отладчику.

Чтобы изменить начальную точку приложения на другую метку, используйте параметр ILINK ***--entry***, см. ***--entry***, стр. 338.

ЗАПУСК ПРИЛОЖЕНИЯ В 64-БИТНОМ РЕЖИМЕ

Точка, в которой приложение начинает выполнение, определяется меткой ***__Reset_address*** (она определяет, где запускается модуль ***cstartup***). Метка ***__iar_program_start*** находится по тому же адресу. Эта метка также передается через ELF любому используемому отладчику.

Для получения информации о том, как изменить адрес сброса, см. Адрес сброса, стр. 90.

НАСТРОЙКА ПАМЯТИ СТЕКА

Размер блока CSTACK определяется в файле конфигурации компоновщика. Чтобы изменить выделенный объем памяти, измените определение блока для CSTACK:

```
define block CSTACK with size = 0x2000, alignment = 8{ };
```

Укажите размер, подходящий для вашего приложения. Обратите внимание, что в 64-битном режиме выравнивание стека равно 16.

Дополнительные сведения о стеке см. В разделе Особенности стека, стр. 216.

НАСТРОЙКА ПАМЯТИ КУЧИ

Размер кучи определяется в файле конфигурации компоновщика как блок:

```
define block HEAP with size = 0x1000, alignment = 8{ };
place in RAM {block HEAP};
```

Укажите размер, подходящий для вашего приложения. Если вы используете кучу, вы должны выделить для нее не менее 50 байтов. Обратите внимание, что в 64-битном режиме выравнивание кучи равно 16.

УСТАНОВКА ATEXIT LIMIT

По умолчанию функция atexit может вызываться из вашего приложения максимум 32 раза. Чтобы увеличить или уменьшить это число, добавьте строку в файл конфигурации. Например, чтобы вместо этого зарезервировать место на 10 вызовов, напишите:

```
define symbol __iar_maximum_atexit_calls = 10;
```

ИЗМЕНЕНИЕ ИНИЦИАЛИЗАЦИИ ПО УМОЛЧАНИЮ

По умолчанию инициализация памяти выполняется при запуске приложения. ILINK настраивает процесс инициализации и выбирает подходящий метод упаковки. Если процесс инициализации по умолчанию не подходит для вашего приложения и вы хотите более точный контроль над процессом инициализации, доступны следующие альтернативы:

- Подавление инициализации
- Выбор алгоритма упаковки.
- Ручная инициализация.
- Код инициализации - копирование ПЗУ в ОЗУ.

Для получения информации о выполненных инициализациях просмотрите файл журнала (параметр командной строки ***--log initialization***).

Подавление инициализации

Если вы не хотите, чтобы компоновщик организовывал инициализацию путем копирования для некоторых или всех разделов, убедитесь, что эти разделы не соответствуют шаблону в директиве ***initialize by copy***, или используйте предложение ***except***, чтобы исключить их из сопоставления. Если вам вообще не нужна инициализация копированием, вы можете полностью опустить директиву ***initialize by copy*** (инициализации копированием).

Это может быть полезно, если ваше приложение или только ваши переменные загружаются в оперативную память каким-либо другим механизмом перед запуском приложения.

Выбор алгоритма упаковки

Чтобы переопределить алгоритм упаковки по умолчанию, напишите, например:

```
initialize by copy with packing = lz77 { readwrite };
```

Дополнительные сведения о доступных алгоритмах упаковки см. ***initialize directive*** (директиве инициализации), стр. 511.

Ручная инициализация

В обычном случае директива ***initialize by copy*** используется для того, чтобы компоновщик упорядочил инициализацию путем копирования - с упаковкой или без упаковки - разделов с содержимым при запуске приложения. Компоновщик достигает этого, логически создавая раздел инициализации для каждого такого раздела, удерживая его содержимое и превращая исходный раздел в раздел без контента. Затем компоновщик добавляет элементы таблицы в таблицу инициализации, чтобы инициализация выполнялась при запуске приложения. Вы можете использовать ***initialize manually***, чтобы подавить создание элементов таблицы, чтобы контролировать, когда и как эти элементы копируются. Это полезно для наложений, но также и в других случаях.

Для разделов без содержимого (разделы с нулевой инициализацией) ситуация обратная. Компоновщик обеспечивает нулевую инициализацию всех таких разделов при запуске приложения, за исключением тех, которые упомянуты в директиве ***do not initialize***.

Простой пример копирования с автоматической блокировкой

Предположим, что у вас есть несколько инициализированных переменных в MYSECTION. Если вы добавите эту директиву в файл конфигурации компоновщика:

```
initialize manually { section MYSECTION };
```

вы можете использовать этот пример исходного кода для инициализации раздела:

```
#pragma section = "MYSECTION"
#pragma section = "MYSECTION_init"
void DoInit()
{
  char * from = __section_begin("MYSECTION_init");
  char * to   = __section_begin("MYSECTION");
  memcpy(to, from, __section_size("MYSECTION"));
}
```

Этот фрагмент исходного кода использует тот факт, что если вы используете ***__section_begin*** (и связанные операторы) с именем раздела, компоновщик для этих разделов создает автоматический блок.

Примечание. Автоматические блоки отменяют обычный процесс выбора раздела и заставляют все, что соответствует имени раздела, формировать один блок.

Пример с явными блоками

Предположим, что вместо ручной инициализации переменных в определенном разделе она нужна для всех инициализированных переменных из конкретной библиотеки. В этом случае вы должны создать явные блоки как для переменных, так и для содержимого. Подобно этому:

```
initialize manually               { section .data       object mylib.a };

define block MYBLOCK              { section .data       object mylib.a };
define block MYBLOCK_init         { section .data_init  object mylib.a };
```

Вы также должны разместить два новых блока, используя одну из директив размещения разделов, блок MYBLOCK в RAM и блок MYBLOCK_init в ROM.

Затем вы можете инициализировать разделы, используя тот же исходный код, что и в предыдущем примере, только с MYBLOCK вместо MYSECTION.

Overlay example (Пример наложения)

Это простой пример наложения, в котором используется автоматическое создание блока:

```
initialize manually { section MYOVERLAY* };

define overlay MYOVERLAY { section MYOVERLAY1 };
define overlay MYOVERLAY { section MYOVERLAY2 };
```

Вы также должны разместить оверлей MYOVERLAY где-нибудь в ОЗУ. Копирование могло выглядеть так:

```
#pragma section = "MYOVERLAY"
#pragma section = "MYOVERLAY1_init"
#pragma section = "MYOVERLAY2_init"
```

```
void SwitchToOverlay1()
{
  char * from = __section_begin("MYOVERLAY1_init");
  char * to   = __section_begin("MYOVERLAY");
  memcpy(to, from, __section_size("MYOVERLAY1_init"));
}
void SwitchToOverlay2()
{
  char * from = __section_begin("MYOVERLAY2_init");
  char * to   = __section_begin("MYOVERLAY");
  memcpy(to, from, __section_size("MYOVERLAY2_init"));
}
```

Инициализация кода - копирование ПЗУ в ОЗУ

Иногда приложение копирует фрагменты кода из флэш-памяти / ПЗУ в ОЗУ. Вы можете указать компоновщику, чтобы это выполнялось автоматически при запуске приложения, или сделать это самостоятельно позже, используя методы, описанные в Инициализация вручную, стр. 119.

Вам необходимо указать разделы кода, которые следует скопировать, в директиве ***initialize by copy***. Самый простой способ - поместить соответствующие функции в определенный раздел, например ***RAMCODE***, и добавить ***section RAMCODE*** в директиву ***initialize by copy*** (инициализации путем копирования). Например:

```
initialize by copy { rw, section RAMCODE };
```

Если вам нужно разместить функции ***RAMCODE*** в каком-то определенном месте, вы должны указать их в директиве размещения, иначе они будут размещены вместе с другими разделами чтения / записи.

Если вам нужно контролировать способ и / или время копирования, вы должны вместо этого использовать директиву ***initialize manually***. См. Раздел Инициализация вручную, стр. 119.

Если функции необходимо запускать без доступа к флэш-памяти / ПЗУ, вы можете использовать ключевое слово ***__ramfunc*** при компиляции. См. Выполнение в ОЗУ, стр. 79.

Запуск всего кода из ОЗУ

Если вы хотите скопировать все приложение из ПЗУ в ОЗУ при запуске программы, используйте директиву ***initilize by copy***, например:

```
initialize by copy { readonly, readwrite };
```

Шаблон чтения-записи будет соответствовать всем статически инициализированным переменным и обеспечивать их инициализацию при запуске. Шаблон ***readonly*** (только для чтения) будет делать то же самое для всего кода и данных, доступных только для чтения, за исключением кода и данных, необходимых для инициализации.

Поскольку функция ***__low_level_init***, если она присутствует, вызывается перед инициализацией, она и все, что ей нужно, также не будут скопированы из ПЗУ в ОЗУ. В некоторых случаях - например, если содержимое ПЗУ больше не доступно для программы после запуска - вам может потребоваться избегать использования одних и тех же функций во время запуска и в остальной части кода.

Если что-либо еще не следует копировать, включите это в предложение ***except***. Это может относиться, например, к таблице векторов прерываний.

Также рекомендуется исключить возможность копирования таблицы динамической инициализации C ++ в оперативную память, поскольку она обычно читается только один раз и больше не используется. Например, вот так:

```
initialize by copy { readonly, readwrite }
    except { section .intvec,              /* Don't copy interrupt table */
             section .init_array };        /* Don't copy C++ init table */
```

ВЗАИМОДЕЙСТВИЕ МЕЖДУ ILINK И ПРИЛОЖЕНИЕМ

ILINK предоставляет параметры командной строки ***--config_def*** и ***--define_symbol*** для определения символов, которые могут использоваться для управления приложением. Вы также можете использовать символы для обозначения начала и конца непрерывной области памяти, которая определена в файле конфигурации компоновщика. Для получения дополнительной информации см. Взаимодействие между инструментами и вашим приложением, стр. 219.

Чтобы изменить ссылку с одного символа на другой, используйте параметр командной строки ILINK ***--redirect***. Это полезно, например, для перенаправления ссылки с нереализованной функции на функцию-заглушку или для выбора одной из нескольких различных реализаций определенной функции, например, как выбрать средство форматирования DLIB для функций стандартной библиотеки ***printf*** и ***scanf***.

Компилятор генерирует искаженные имена для представления сложных символов C / C ++. Если вы хотите ссылаться на эти символы из исходного кода ассемблера, вы должны использовать искаженные имена.

Для получения информации об адресах и размерах всех глобальных (статически связанных) символов просмотрите список записей в файле карты (параметр командной строки ***--map***).

Для получения дополнительной информации см. Взаимодействие между инструментами и вашим приложением, стр. 219.

ОБРАЩЕНИЕ К СТАНДАРТНОЙ БИБЛИОТЕКЕ

По умолчанию ILINK автоматически определяет, какой вариант стандартной библиотеки следует включить во время связывания. Решение основывается на сумме атрибутов времени выполнения, доступных в каждом объектном файле, и опциях библиотеки, переданных в ILINK.

Чтобы отключить автоматическое включение библиотеки, используйте параметр ***--no_library_search***. В этом случае вы должны явно указать каждый файл библиотеки, который нужно включить. Для получения информации о доступных файлах библиотеки см. Готовые библиотеки времени выполнения, стр. 141.

ПРОИЗВОДСТВО ВЫХОДНЫХ ФОРМАТОВ, ЗА ИСКЛЮЧЕНИЕМ ELF / DWARF

ILINK может создавать выходной файл только в формате ELF / DWARF. Чтобы преобразовать этот формат в формат, подходящий для программирования PROM / flash, см. Инструмент IAR ELF - ielftool, стр. 549.

VENEERS (ВИНИРЫ)

Виниры - это небольшие последовательности кода, вставленные компоновщиком для преодоления разрыва, когда инструкция вызова не достигает своего пункта назначения или не может переключиться в правильный режим.

Код для виниров может быть вставлен между любой вызывающей и вызываемой функцией. В результате некоторые регистры должны обрабатываться как временные регистры при вызове функций, включая функции, написанные на языке ассемблера. Это касается и безусловных переходов. В 32-битном режиме R12 должен рассматриваться как рабочий регистр. В 64-битном режиме и X16, и X17 должны рассматриваться как временные регистры.

Подсказки по поиску и устранению неисправностей

ILINK имеет несколько функций, которые могут помочь вам правильно управлять размещением кода и данных, например:

- Сообщения во время связывания, например, когда возникает ошибка перемещения.
- Параметр ***--log***, который переводит информацию журнала ILINK в стандартный вывод, что может быть полезно для понимания того, почему исполняемый образ стал таким, какой он есть, см. ***--log***, стр. 345
- Параметр ***--map***, который заставляет ILINK создавать файл карты памяти, который содержит результат файла конфигурации компоновщика, см. ***--map***, стр. 347.

ОШИБКИ ПЕРЕМЕЩЕНИЯ

Для каждой инструкции, которая не может быть перемещена правильно, ILINK сгенерирует ошибку перемещения. Это может произойти для инструкций, где цель находится вне досягаемости или несовместимого типа, или по многим другим причинам.

Ошибка перемещения, созданная ILINK, может выглядеть так:

```
Error[Lp002]: relocation failed: out of range or illegal value
   Kind     :   R_XXX_YYY[0x1]
   Location :   0x40000448
                "myfunc" + 0x2c
                Module:  somecode.o
                Section: 7 (.text)
                Offset:  0x2c
   Destination: 0x9000000c
                "read"
                Module:  read.o(iolib.a)
                Section: 6 (.text)
                Offset:  0x0
```

Записи сообщений описаны в этой таблице:

Точка входа сообщения	Описание
Kind Вид	Директива о перемещении, которая не удалась. Директива зависит от используемой инструкции.

Точка входа сообщения	Описание
Location Местоположение	Местоположение, в котором возникла проблема, описанное со следующей информацией: • Адрес инструкции, выраженный как в шестнадцатеричном формате, так и в виде метки со смещением. В этом примере x40000448 и «myfunc» + 0x2c. • Модуль и файл. В этом примере модуль somecode.o. • Номер раздела и название раздела. В этом примере номер раздела 7 с именем .text. • Смещение, указанное в байтах в разделе. В этом например, 0x2c.
Destination Назначение	Цель инструкции, описанная со следующими деталями: • Адрес инструкции, выраженный как шестнадцатеричным значением, так и меткой со смещением. В этом примере x9000000c и «чтение» - следовательно, без смещения. • Модуль и, если применимо, библиотека. В этом примере модуль read.o и библиотека iolib.a. • Номер раздела и название раздела. В этом примере раздел номер 6 с именем .text. • Смещение, указанное в байтах в разделе. В этом например, 0x0.

Таблица 4: Описание ошибки перемещения

Возможные решения

В этом случае расстояние от инструкции в ***myfunc*** до ***__read*** слишком велико для инструкции перехода.

Возможные решения включают обеспечение того, чтобы два раздела ***.text*** были размещены ближе друг к другу, или использование какого-либо другого механизма вызова, который может достигать необходимого расстояния. Также возможно, что ссылающаяся функция пыталась ссылаться на неправильную метку, и это вызвало ошибку диапазона.

Различные ошибки дальности имеют разные решения. Обычно решение представляет собой вариант из представленных выше, то есть изменение кода или размещения раздела.

Проверка согласованности модуля

В этом разделе представлена концепция атрибутов модели времени выполнения, механизма, используемого инструментами, предоставляемыми IAR Systems, для обеспечения совместимости модулей, связанных с приложением, другими словами, построены с использованием совместимых настроек. Инструменты используют набор предопределенных атрибутов модели времени выполнения. В дополнение к этому вы можете определить свои собственные, которые вы можете использовать, чтобы гарантировать, что несовместимые модули не используются вместе.

Примечание. В дополнение к предопределенным атрибутам совместимость также проверяется по атрибутам времени выполнения AEABI. Эти атрибуты имеют дело в основном с совместимостью объектного кода и т. д. Они отражают настройки компиляции и не настраиваются пользователем.

АТРИБУТЫ МОДЕЛИ РАБОТЫ

Атрибут времени выполнения - это пара, состоящая из именованного ключа и соответствующего ему значения. Как правило, два модуля могут быть связаны друг с другом только в том случае, если они имеют одинаковое значение для каждого ключа, который они оба определяют.

Есть одно исключение: если значение атрибута *, то этот атрибут соответствует любому значению. Причина в том, что вы можете указать это в модуле, чтобы показать, что вы учли свойство согласованности, и это гарантирует, что модуль не полагается на это свойство.

Примечание. Для предопределенных IAR атрибутов модели времени выполнения компоновщик проверяет их несколькими способами.

Пример

В этой таблице объектные файлы могут (но не обязательно) определять **Color (**цвет) и **Taste (**вкус) двух атрибутов среды выполнения:

Object file	Color	Taste
file1	blue	not defined
file2	red	not defined
file3	red	*
file4	red	spicy
file5	red	lean

Таблица 5: Пример атрибутов модели времени выполнения

В этом случае файл file1 не может быть связан ни с одним из других файлов, поскольку цвет атрибута времени выполнения не совпадает. Кроме того, file4 и file5 не могут быть связаны вместе, потому что атрибут времени выполнения вкуса не совпадает.

С другой стороны, file2 и file3 могут быть связаны друг с другом и либо с file4, либо с file5, но не с обоими.

ИСПОЛЬЗОВАНИЕ АТРИБУТОВ МОДЕЛИ РАБОТЫ

Чтобы обеспечить согласованность модуля с другими объектными файлами, используйте директиву ***#pragma rtmodel***, чтобы указать атрибуты модели времени выполнения в исходном коде C / C ++. Например, если у вас есть UART, который может работать в двух режимах, вы можете указать атрибут модели времени выполнения, например ***uart***. Для каждого режима укажите значение, например ***mode1*** и ***mode2***. Объявляйте это в каждом модуле, который предполагает, что UART находится в определенном режиме. Вот так это могло бы выглядеть в одном из модулей:

```
#pragma rtmodel = "uart", "mode1"
```

В качестве альтернативы вы также можете использовать директиву ассемблера ***rtmodel***, чтобы указать атрибуты модели времени выполнения в исходном коде ассемблера. Например:

```
rtmodel "uart", "mode1"
```

Примечание. Имена ключей, начинающиеся с двух знаков подчеркивания, зарезервированы компилятором. Для получения дополнительной информации о синтаксисе см. Rtmodel, стр. 418 и Руководство пользователя ассемблера IAR для Arm.

Во время компоновки IAR ILINK Linker проверяет согласованность модулей, гарантируя, что модули с конфликтующими атрибутами времени выполнения не будут использоваться вместе. При обнаружении конфликтов выдается ошибка.

Оптимизация компоновщика

В этом разделе содержится информация о:

- Устранении виртуальной функции, стр. 127
- Встраивание небольших функций, стр. 127.
- Объединение повторяющихся разделов, стр. 128.

УСТРАНЕНИЕ ВИРТУАЛЬНОЙ ФУНКЦИИ

Устранение виртуальных функций (VFE) - это оптимизация компоновщика, которая удаляет ненужные виртуальные функции и информацию о типах динамической среды выполнения.

Для работы исключения виртуальных функций все соответствующие модули должны предоставлять информацию о макете таблицы виртуальных функций, о том, какие виртуальные функции вызываются и для каких классов требуется информация о динамическом типе среды выполнения. Если один или несколько модулей не предоставляют эту информацию, компоновщик генерирует предупреждение и виртуальное исключение функции не выполняется.

Если вы знаете, что модули, в которых отсутствует такая информация, не выполняют никаких вызовов виртуальных функций и не определяют таблицы виртуальных функций, вы можете использовать параметр ***--vfe=forced*** компоновщика, чтобы в любом случае включить исключение виртуальных функций.

В среде IDE выберите

Project>Options>Linker>Optimizations>Perform C++ Virtual Function Elimination, чтобы включить эту оптимизацию.

*Примечание. Вы можете полностью отключить исключение виртуальных функций, используя параметр компоновщика **--no_vfe**. В этом случае для модулей, в которых отсутствует информация о VFE, предупреждение не выводится.*

Для получения дополнительной информации см. ***--vfe***, стр. 361 и ***--no_vfe***, стр. 352.

ВСТРАИВАНИЕ МАЛЫХ ФУНКЦИЙ

Встраивание малых функций - это оптимизация компоновщика, которая заменяет некоторые вызовы небольших функций телом функции. Это требует, чтобы тело помещалось в пространство инструкции, вызывающей функцию.

В среде IDE выберите

Project>Options>Linker>Optimizations>Inline small routines, чтобы включить эту оптимизацию.

Используйте параметр компоновщика ***--inline***.

ОБЪЕДИНЕНИЕ ДУБЛИРОВАННЫХ РАЗДЕЛОВ

Компоновщик может обнаруживать разделы только для чтения с идентичным содержимым и сохранять только одну копию каждого такого раздела, перенаправляя все ссылки на любой из повторяющихся разделов в сохраненный раздел.

В среде IDE выберите

Project>Options>Linker>Optimizations>Merge duplicate sections, чтобы включить эту оптимизацию.

Используйте опцию компоновщика ***--merge_duplicate_sections.***

Примечание. Эта оптимизация может привести к тому, что разные функции или константы будут иметь один и тот же адрес, поэтому, если ваше приложение зависит от разных адресов, например, при использовании адресов в качестве ключей в таблице, вам не следует включать эту оптимизацию.

Среда выполнения DLIB

- Введение в среду выполнения.
- Настройка среды выполнения.
- Дополнительная информация о среде выполнения.
- Управление многопоточной средой.

Введение в среду выполнения

Среда выполнения - это среда, в которой выполняется ваше приложение.

В этом разделе содержится информация о:

- Функциональные возможности среды выполнения, стр. 129.
- Кратко о вводе и выводе (I / O), стр. 130.
- Кратко о вводе / выводе, эмулируемом C-SPY, стр. 131
- Кратко о ретаргетинге, стр. 132.

ФУНКЦИОНАЛЬНОСТЬ СРЕДЫ РАБОТЫ

Среда выполнения DLIB поддерживает стандарты C и C ++ и состоит из:

- Стандартная библиотека C / C ++, как ее интерфейс (предоставленный в файлах системных заголовков), так и ее реализация.
- Код запуска и выхода.
- Интерфейс ввода-вывода низкого уровня для управления вводом и выводом (ввод-вывод).
- Специальная поддержка компилятора, например, функции для обработки переключателей или целочисленной арифметики.
- Поддержка аппаратных функций:
- Прямой доступ к низкоуровневым операциям процессора посредством встроенных функций, таких как функции для обработки маски прерывания.
- Регистры периферийных устройств и определения прерываний во включаемых файлах.
- Сопроцессор с векторной плавающей точкой (VFP).

Функции среды выполнения предоставляются в библиотеке времени выполнения.