Efficient coding for embedded applications (Эффективное кодирование для встраиваемых приложений)

- Выбор типов данных.
- Управление размещением данных и функций в памяти.
- Контроль оптимизации компилятора.
- Содействие генерации хорошего кода.

Выбор типов данных

Для эффективной обработки данных следует учитывать используемые типы данных и наиболее эффективное размещение переменных.

ИСПОЛЬЗОВАНИЕ ЭФФЕКТИВНЫХ ТИПОВ ДАННЫХ

Следует тщательно продумывать используемые типы данных, поскольку это может сильно повлиять на размер кода и скорость кода.

- По возможности используйте ***int*** или ***long*** вместо ***char*** или ***short***, чтобы избежать расширения знака или нулевого расширения. В частности, индексы цикла всегда должны быть целыми или длинными, чтобы минимизировать генерацию кода. Кроме того, в режиме ***Thumb*** доступ через указатель стека (SP) ограничен 32-битными типами данных, что дополнительно подчеркивает преимущества использования одного из этих типов данных.
- Используйте беззнаковые типы данных, если ваше приложение действительно не требует значений со знаком.
- В 32-битном режиме помните о стоимости использования 64-битных типов данных, таких как ***double*** и ***long long***.
- Битовые поля и упакованные структуры генерируют большой и медленный код.
- Использование типов с плавающей запятой в микропроцессоре без математического сопроцессора неэффективно как с точки зрения размера кода, так и скорости выполнения.
- Объявление указателя на константные данные сообщает вызывающей функции, что указанные данные не изменятся, что открывается для лучшей оптимизации.

Для получения информации о представлении поддерживаемых типов данных, указателей и типов структур см. Главу Представление данных.

ТИПЫ ПЛАВАЮЩЕЙ ТОЧКИ

Использование типов с плавающей запятой в микропроцессоре без математического сопроцессора неэффективно как с точки зрения размера кода, так и скорости выполнения. Поэтому вам следует подумать о замене кода, использующего операции с плавающей запятой, на код, использующий целые числа, поскольку он более эффективен.

Компилятор поддерживает три формата с плавающей запятой - 16, 32 и 64 бит. 32-битный тип с плавающей запятой более эффективен с точки зрения размера кода и скорости выполнения. 64-битный формат double поддерживает более высокую точность и большие числа. 16-битный формат в основном полезен в некоторых конкретных ситуациях.

В компиляторе тип ***float*** с плавающей запятой всегда использует 32-битный формат, а тип ***double*** всегда использует 64-битный формат.

Если приложению не требуется дополнительная точность, которую обеспечивают 64-разрядные числа с плавающей запятой, мы рекомендуем вместо этого использовать 32-разрядные числа с плавающей запятой.

По умолчанию *константа с плавающей запятой* в исходном коде рассматривается как имеющая тип ***double***. Это может привести к тому, что невинно выглядящие выражения будут оцениваться с двойной точностью. В приведенном ниже примере a преобразуется из числа с плавающей запятой в число типа ***double***, добавляется константа типа ***double 1.0***, и результат конвертируется обратно в число с плавающей запятой:

```
double Test(float a)
{
  return a + 1.0;
}
```

Чтобы рассматривать константу с плавающей запятой как число с плавающей запятой, а не как двойное, добавьте к ней суффикс f, например:

```
double Test(float a)
{
  return a + 1.0f;
}
```

Для получения дополнительной информации о типах с плавающей запятой см. Основные типы данных - типы с плавающей запятой, стр. 372.

ВЫРАВНИВАНИЕ ЭЛЕМЕНТОВ КОНСТРУКЦИИ

Некоторые ядра Arm требуют, чтобы при доступе к данным в памяти эти данные были выровнены. Каждый элемент в структуре должен быть выровнен в соответствии с требованиями указанного типа. Это означает, что компилятору может потребоваться вставить байты заполнения, чтобы обеспечить правильное выравнивание.

Бывают ситуации, когда это может быть проблемой:

- Существуют внешние требования, например, протоколы сетевой связи обычно указываются в терминах типов данных без промежуточных отступов.
- Вам необходимо сохранить память данных.

Для получения информации о требованиях к выравниванию см. Выравнивание, стр. 365.

Используйте директиву ***#pragma pack*** или атрибут типа данных ***__packed*** для более плотной компоновки структуры. Недостатком является то, что каждый доступ к невыровненному элементу в структуре будет использовать больше кода.

Как вариант, напишите свои собственные настраиваемые функции для упаковки и распаковки структур. Это более переносимый способ, который не будет производить больше кода, кроме ваших функций. Недостатком является необходимость двух представлений данных структуры - упакованных и распакованных.

Для получения дополнительной информации о директиве #pragma pack см. Pack, стр. 416.

АНОНИМНЫЕ СТРУКТУРЫ И СОЮЗЫ

Когда структура или объединение объявляются без имени, они становятся анонимными. В результате его члены будут видны только в окружающей области.

Пример

В этом примере к членам анонимного объединения можно получить доступ в функции F без явного указания имени объединения:

```
struct S
{
  char mTag;
  union
  {
    long mL;
    float mF;
  };
} St;
void F(void)
{
  St.mL = 5;
}
```

Имена членов должны быть уникальными в окружающей области. Также допускается наличие анонимной структуры или объединения в области видимости файла в качестве глобальной, внешней или статической переменной. Это может, например, использоваться для объявления регистров ввода-вывода, как в этом примере:

```
__no_init volatile
union
{
  unsigned char IOPORT;
  struct
  {
    unsigned char way: 1;
    unsigned char out: 1;
  };
} @ 0x1000;
/* The variables are used here. Здесь используются переменные. */
void Test(void)
{
  IOPORT = 0;
  way = 1;
  out = 1;
}
```

Это объявляет байт регистра ввода-вывода IOPORT по адресу 0x1000. В регистре ввода-вывода объявлено 2 бита, Way и Out - как внутренняя структура, так и внешнее объединение анонимны.

Анонимные структуры и объединения реализуются в виде объектов, названных по имени первого поля, с префиксом _A_ для размещения имени в части ре-

ализации пространства имен. В этом примере анонимное объединение будет реализовано через объект с именем _A_IOPORT.

Управление размещением данных и функций в памяти

Компилятор предоставляет различные механизмы для управления размещением функций и объектов данных в памяти. Чтобы эффективно использовать память, вы должны быть знакомы с этими механизмами и знать, какой из них лучше всего подходит для различных ситуаций. Вы можете использовать:

• Оператор @ и директива ***#pragma location*** для абсолютного размещения.

Используя оператор @ или директиву ***#pragma location***, вы можете размещать отдельные глобальные и статические переменные по абсолютным адресам. Обратите внимание, что это обозначение невозможно использовать для абсолютного размещения отдельных функций. Для получения дополнительной информации см. Размещение данных в абсолютном местоположении, стр. 241.

• Оператор @ и директива ***#pragma location*** для размещения раздела.

Используя оператор @ или директиву ***#pragma location***, вы можете размещать отдельные функции, переменные и константы в именованных разделах. Размещение этих разделов затем можно контролировать с помощью директив компоновщика. Для получения дополнительной информации см. Размещение данных и функций в разделах, стр. 242.

• Оператор @ и директива ***#pragma location*** для размещения регистров.

Используйте оператор @ или директиву ***#pragma location*** для размещения отдельных глобальных и статических переменных в регистрах. Переменные должны быть объявлены ***__no_init***. Это полезно для отдельных объектов данных, которые должны находиться в определенном регистре.

• Используя параметр ***--section***, вы можете установить сегмент по умолчанию для функций, переменных и констант в конкретном модуле. Для получения дополнительной информации см. --Section, стр. 316.

РАЗМЕЩЕНИЕ ДАННЫХ В АБСОЛЮТНОМ МЕСТЕ

Оператор @ или директива ***#pragma location*** может использоваться для размещения глобальных и статических переменных по абсолютным адресам.

Чтобы разместить переменную по абсолютному адресу, аргумент оператора @ и директивы ***#pragma location*** должен быть буквальным числом, представляющим фактический адрес. Абсолютное местоположение должно соответствовать требованию выравнивания для переменной, которая должна быть расположена.

*Примечание. Все объявления переменных **__no_init**, размещенные по абсолютному адресу, являются предварительными определениями. Предварительно определенные переменные сохраняются в выводе компилятора только в том случае, если они необходимы в компилируемом модуле. Такие переменные будут определены во всех модулях, в которых они используются, и будут работать до тех пор, пока они определены одинаково. Рекомендуется размещать все такие объявления в файлах заголовков, которые включены во все модули, использующие переменные.*

Другие переменные, помещенные в абсолютный адрес, используют обычное различие между объявлением и определением. Для этих переменных вы должны предоставить определение только в одном модуле, обычно с инициализатором.

Другие модули могут ссылаться на переменную с помощью объявления extern с явным адресом или без него.

Примеры

В этом примере объявленная переменная __no_init размещается по абсолютному адресу. Это полезно для взаимодействия между несколькими процессами, приложениями и т. д.:

```
__no_init volatile char alpha @ 0xFF2000;/* OK */
```

Следующий пример содержит два объявленных объекта const. Первый не инициализируется, а второй инициализируется определенным значением. (Первый случай полезен для параметров конфигурации, потому что они доступны из внешнего интерфейса.) Оба объекта помещаются в ПЗУ. Обратите внимание, что во втором случае компилятор не обязан фактически читать из переменной, потому что значение известно.

```
#pragma location=0xFF2002
__no_init const int beta; /* OK */
const int gamma @ 0xFF2004 = 3; /* OK */
```

В первом случае значение не инициализируется компилятором - значение должно быть установлено другими способами. Типичное использование - для конфигураций, где значения загружаются в ПЗУ отдельно, или для регистров специальных функций, которые доступны только для чтения.

```
__no_init int epsilon @ 0xFF2007; /* Error, misaligned. Ошибка, несовпадение. */
```

Замечания по C ++

В C ++ константные переменные в области модуля являются статическими (локальными для модуля), тогда как в C они являются глобальными. Это означает, что каждый модуль, объявляющий определенную константную переменную, будет содержать отдельную переменную с этим именем. Если вы связываете приложение с несколькими такими модулями, все из которых содержат (через файл заголовка), например, объявление:

```
volatile const __no_init int x @ 0x100;       /* Bad in C++ Плохо в C ++ */
```

компоновщик сообщит, что по адресу 0x100 расположено более одной переменной.

Чтобы избежать этой проблемы и сделать процесс одинаковым в C и C ++, вы должны объявить эти переменные extern, например:

```
/* The extern keyword makes x public. */
/* Ключевое слово extern делает x общедоступным.  */
extern volatile const __no_init int x @ 0x100;
```

Примечание. Статические переменные-члены C ++ можно разместить по абсолютному адресу, как и любую другую статическую переменную.

РАЗМЕЩЕНИЕ ДАННЫХ И ФУНКЦИЙ В РАЗДЕЛАХ (IN SECTIONS)

Следующий метод может использоваться для размещения данных или функций в именованных разделах, отличных от заданных по умолчанию:

• Оператор @ или директива ***#pragma location*** может использоваться для размещения отдельных переменных или отдельных функций в именованных разделах. Именованный раздел может быть заранее определенным или определяемым пользователем.

• Параметр ***--section*** может использоваться для размещения переменных и функций, которые являются частями всего модуля компиляции, в именованные разделы.

Статические переменные-члены C ++ могут быть помещены в именованные разделы, как и любые другие статические переменные.

Если вы используете свои собственные разделы, в дополнение к предопределенным разделам, разделы также должны быть определены в файле конфигурации компоновщика.

Примечание: будьте осторожны при явном размещении переменной или функции в предопределенном разделе, отличном от того, который используется по умолчанию. Это полезно в некоторых ситуациях, но неправильное размещение может привести к чему угодно - от сообщений об ошибках во время компиляции до связывания с неисправным приложением. Внимательно рассмотрите обстоятельства - могут быть строгие требования к объявлению и использованию функции или переменной.

Расположение разделов можно контролировать из файла конфигурации компоновщика.

Дополнительные сведения о разделах см. В разделе «Справочник по разделам».

Примеры размещения переменных в именованных разделах

В следующих примерах объект данных помещается в определяемый пользователем раздел. Обратите внимание, что вы, как всегда, должны убедиться, что раздел помещен в соответствующую область памяти при связывании.

```
__no_init int alpha @ "MY_NOINIT";      /* OK */
#pragma location="MY_CONSTANTS"
const int beta = 42;                    /* OK */
const int gamma @ "MY_CONSTANTS" = 17;  /* OK */
int theta @ "MY_ZEROS";                 /* OK */
int phi @ "MY_INITED" = 4711;           /* OK */
```

Компоновщик обычно подбирает правильный тип инициализации для каждой переменной. Если вы хотите контролировать или подавлять автоматическую инициализацию, вы можете использовать директивы ***initialize*** and ***do not initialize*** в файле конфигурации компоновщика.

Примеры размещения функций в именованных разделах

```
void f(void) @ "MY_FUNCTIONS";
void g(void) @ "MY_FUNCTIONS"
{
}
#pragma location="MY_FUNCTIONS"
void h(void);
```

РАЗМЕЩЕНИЕ ДАННЫХ В РЕГИСТРАХ (32-БИТНЫЙ РЕЖИМ)

В 32-битном режиме оператор @ или директива ***#pragma location*** можно использовать для размещения глобальных и статических переменных в регистре.

Чтобы поместить переменную в регистр, аргумент оператора @ и директивы ***#pragma location*** должен быть идентификатором, который соответствует регистру ядра Arm в диапазоне R4 – R11 (R9 нельзя указать в сочетании с командой ***--rwpi*** вариант линии).

Переменная может быть помещена в регистр только в том случае, если она объявлена как ***__no_init***, имеет область видимости файла и ее размер составляет четыре байта. Переменная, помещенная в регистр, не имеет адреса памяти, поэтому оператор адреса ***&*** не может использоваться.

В модуле, в котором переменная помещается в регистр, указанный регистр будет использоваться только для доступа к этой переменной. Значение переменной сохраняется при вызовах функций в другие модули, потому что регистры R4 – R11 сохраняются вызываемым пользователем и, как таковые, восстанавливаются при возврате выполнения. Однако значение переменной, помещенной в регистр, не всегда сохраняется должным образом:

• В обработчике исключений или подпрограмме обратного вызова библиотеки (например, в функции компаратора, переданной в qsort) значение может не сохраняться. Значение будет сохранено, если параметр командной строки ***--lock_regs*** используется для блокировки реестра во всех модулях приложения, включая модули библиотеки.

• В обработчике быстрого прерывания значение переменной в R8 – R11 не сохраняется извне обработчика, потому что эти регистры хранятся в банках.

• Функция ***longjmp*** и исключения C ++ могут восстанавливать переменные, помещенные в регистры, до старых значений, в отличие от других переменных со статической продолжительностью хранения, которые не восстанавливаются.

Компоновщик не запрещает модулям помещать разные переменные в один и тот же регистр. Переменные в разных модулях могут быть помещены в один и тот же регистр, а другой модуль может использовать регистр для других целей.

Примечание. Переменная, помещенная в регистр, должна быть определена во включаемом файле, который будет включен в каждый модуль, использующий эту переменную. Неиспользованное определение в модуле приведет к тому, что регистр не будет использоваться в этом модуле.

Управление оптимизацией компилятора

Компилятор выполняет множество преобразований в вашем приложении, чтобы сгенерировать наилучший возможный код. Примерами таких преобразований являются сохранение значений в регистрах вместо памяти, удаление лишнего кода, переупорядочение вычислений в более эффективном порядке и замена арифметических операций более дешевыми операциями.

Компоновщик также следует рассматривать как неотъемлемую часть системы компиляции, потому что некоторые оптимизации выполняются компоновщиком. Например, все неиспользуемые функции и переменные удаляются и не включаются в окончательный вывод.

ОБЪЕМ ВЫПОЛНЕННЫХ ОПТИМИЗАЦИЙ

Вы можете решить, следует ли проводить оптимизацию для всего приложения или для отдельных файлов. По умолчанию для всего проекта используются одни и те же типы оптимизации, но вам следует рассмотреть возможность использования разных настроек оптимизации для отдельных файлов. Например, поместите код, который должен выполняться быстро, в отдельный файл и скомпилировать его для минимального времени выполнения, а остальную часть кода - для минимального размера кода. Это даст небольшую программу, которая все еще достаточно быстра там, где это важно.

Вы также можете исключить отдельные функции из выполненных оптимизаций. Директива ***#pragma optimize*** позволяет либо снизить уровень оптимизации, либо указать другой тип оптимизации, которую необходимо выполнить. См. Optimize, стр. 415, для получения информации о директиве pragma.

МНОГОФАЙЛОВЫЕ ПОДБОРКИ

Помимо применения различных оптимизаций к разным исходным файлам или даже функциям, вы также можете решить, из чего состоит модуль компиляции - из одного или нескольких файлов исходного кода.

По умолчанию единица компиляции состоит из одного исходного файла, но вы также можете использовать многофайловую компиляцию для создания нескольких исходных файлов в единице компиляции. Преимущество заключается в том, что межпроцедурные оптимизации, такие как встраивание и перекрестный переход, требуют большего количества исходного кода для работы. В идеале все приложение должно быть скомпилировано как одна единица компиляции. Однако для больших приложений это нецелесообразно из-за ограничений ресурсов на главном компьютере. Для получения дополнительной информации см. --Mfc, стр. 299.

Примечание. Создается только один объектный файл, поэтому все символы будут частью этого объектного файла.

Если все приложение компилируется как одна единица компиляции, полезно заставить компилятор также отбрасывать неиспользуемые общедоступные функции и переменные перед выполнением межпроцедурных оптимизаций. Это ограничивает область оптимизации фактически используемыми функциями и переменными. Для получения дополнительной информации см. --Discard_ unused_publics, стр. 289.

УРОВНИ ОПТИМИЗАЦИИ

Компилятор поддерживает разные уровни оптимизации. В этой таблице перечислены оптимизации, которые обычно выполняются на каждом уровне:

None (Best debug support)	**Нет** (Лучшая поддержка отладки)
Variables live through their entire scope	Переменные действуют во всей своей области
Dead code elimination	Устранение мертвого кода
Redundant label elimination	Удаление избыточной метки
Redundant branch elimination	Устранение избыточной ветви

Low Низкая

То же, что и выше, но переменные существуют только до тех пор, пока они необходимы, не обязательно во всей их области видимости.

Medium	**Средняя**. То же, что и выше, и:
Live-dead analysis and optimization	Живой-мертвый анализ и оптимизация
Dead code elimination	Устранение мертвого кода
Redundant label elimination	Удаление избыточной метки
Redundant branch elimination	Устранение избыточной ветви
Code hoisting	Подъем кода
Peephole optimization	Оптимизация глазка
Some register content analysis and optimization	Контент-анализ и оптимизация некоторых регистров
Common subexpression elimination	Исключение общего подвыражения
Code motion	Код движения
Static clustering	Статическая кластеризация

High (Balanced)	**Высокая** (сбалансированный) То же, что и выше, и:
Instruction scheduling	Планирование инструкций
Cross jumping	Кросс-джампинг
Advanced register content analysis and optimization	Расширенный анализ и оптимизация содержимого реестра
Loop unrolling	Развертывание петли
Function inlining	Встраивание функций
Type-based alias analysis	Анализ псевдонимов на основе типов

Примечание. Некоторые из выполненных оптимизаций можно включить или отключить по отдельности. Дополнительные сведения см. В разделе Преобразования с включенной тонкой настройкой, стр. 247.

Высокий уровень оптимизации может привести к увеличению времени компиляции, а также, скорее всего, затруднит отладку, поскольку менее ясно, как сгенерированный код соотносится с исходным кодом. Например, на низком, среднем и высоком уровнях оптимизации переменные не охватывают всю свою область действия, что означает, что регистры процессора, используемые для хранения переменных, могут быть повторно использованы сразу после их последнего использования. Из-за этого окно C-SPY Watch может не отображать значение переменной во всей своей области или даже иногда отображать неправильное значение. В любое время, если вы испытываете трудности при отладке кода, попробуйте снизить уровень оптимизации.

СКОРОСТЬ ПРОТИВ РАЗМЕРА

На высоком уровне оптимизации компилятор балансирует между оптимизацией размера и скорости. Однако можно явно настроить оптимизацию либо для размера, либо для скорости. Они различаются только используемыми порогами: скорость будет зависеть от размера, а размер - от размера.

Если вы используете уровень оптимизации Высокая скорость, параметр компилятора ***--no_size_constraints*** ослабляет обычные ограничения для увеличения размера кода и позволяет выполнять более агрессивные оптимизации.

Вы можете выбрать цель оптимизации для каждого модуля или даже отдельных функций, используя параметры командной строки и директивы pragma (см. -O, стр. 309 и optimize, стр. 415). Для небольшого встроенного приложения это позволяет достичь приемлемой скорости при минимальном размере кода: обычно только несколько мест в приложении должны быть быстрыми, например, наиболее часто выполняемые внутренние циклы или обработчики прерываний.

Вместо того, чтобы компилировать все приложение с оптимизацией High (Balanced), вы можете использовать High (Size) в целом, но переопределить это, чтобы получить оптимизацию High (Speed) только для тех функций, где приложение должно быть быстрым.

Примечание. Из-за непредсказуемого способа взаимодействия различных оптимизаций, когда одна оптимизация может включать другие оптимизации, иногда функция становится меньше при компиляции с оптимизацией High (Speed), чем при использовании High (Size). Кроме того, использование многофайловой компиляции (см. --Mfc, стр. 299) может обеспечить множество оптимизаций для повышения как скорости, так и размера. Рекомендуется поэкспериментировать с различными настройками оптимизации, чтобы выбрать лучшие для своего проекта.

ПРЕОБРАЗОВАНИЯ С ТОЧНОЙ НАСТРОЙКОЙ

На каждом уровне оптимизации вы можете отключить некоторые преобразования по отдельности. Чтобы отключить преобразование, используйте либо соответствующий параметр, например, параметр командной строки ***--no_inline***, либо его эквивалент во встраивании функций IDE, либо директиву ***#pragma optimize***. Эти преобразования можно отключить индивидуально:

- Common subexpression elimination (Исключение общих подвыражений).
- Loop unrolling (Раскрутка петли).
- Function inlining (Встраивание функций).
- Code motion (Код движения).
- Type-based alias analysis (Анализ псевдонимов на основе типов).
- Static clustering (Статическая кластеризация).
- Instruction scheduling (Планирование инструкций).

Исключение общего подвыражения

Избыточная переоценка общих подвыражений по умолчанию устраняется на уровнях оптимизации Средний и Высокий. Эта оптимизация обычно сокращает как размер кода, так и время выполнения. Однако полученный код может быть сложно отладить.

Примечание. Этот параметр не действует на уровнях оптимизации «Нет» и «Низкий».

Для получения дополнительной информации о параметре командной строки см. --No_cse, стр. 301.

Развертывание петли

Развертывание цикла означает, что тело кода цикла, количество итераций которого может быть определено во время компиляции, дублируется. Развертывание цикла снижает накладные расходы на цикл, амортизируя его за несколько итераций.

Эта оптимизация наиболее эффективна для небольших циклов, где служебные данные цикла могут составлять значительную часть всего тела цикла.

Развертывание цикла, которое может выполняться на уровне оптимизации High, обычно сокращает время выполнения, но увеличивает размер кода. Полученный код также может быть трудно отладить.

Компилятор эвристически решает, какие циклы развернуть. Развертываться будут только относительно небольшие петли, в которых заметно снижение накладных расходов. Различные эвристики используются при оптимизации скорости, размера или при балансировании между размером и скоростью.

Примечание. Этот параметр не действует на уровнях оптимизации «Нет», «Низкий» и «Средний».

Чтобы отключить развертывание цикла, используйте параметр командной строки --no_unroll, см. --No_unroll, стр. 308.

Встраивание функций

Встраивание функций означает, что функция, определение которой известно во время компиляции, интегрируется в тело вызывающего, чтобы устранить накладные расходы на вызов. Эта оптимизация обычно сокращает время выполнения, но может увеличить размер кода.

Для получения дополнительной информации см. Встраивание функций, стр. 90.

Чтобы отключить встраивание функций, используйте параметр командной строки --no_inline, см. --No_inline, стр. 302.

Код движения

Вычисление инвариантных к циклам выражений и общих подвыражений перемещено, чтобы избежать повторной повторной оценки. Эта оптимизация, которая выполняется на уровне оптимизации Средний и выше, обычно сокращает размер кода и время выполнения. Однако полученный код может быть трудно отладить.

Примечание. Этот параметр не действует на уровнях оптимизации ниже среднего.

Для получения дополнительной информации о параметре командной строки см. --No_code_motion, стр. 300.

Анализ псевдонимов на основе типов

Когда два или более указателя ссылаются на одно и то же место в памяти, эти указатели называются псевдонимами друг для друга. Существование псевдонимов затрудняет оптимизацию, поскольку во время компиляции не обязательно известно, изменяется ли конкретное значение.

Оптимизация анализа псевдонимов на основе типов предполагает, что все обращения к объекту выполняются с использованием его объявленного типа или типа char. Это предположение позволяет компилятору определять, могут ли указатели ссылаться на одно и то же место в памяти или нет.

Анализ псевдонимов на основе типов выполняется на уровне оптимизации Высокий. Для кода приложения, соответствующего стандартному коду приложения C или C ++, эта оптимизация может уменьшить размер кода и время выполнения. Однако нестандартный код C или C ++ может привести к тому, что компилятор создаст код, который приведет к неожиданному поведению. Следовательно, эту оптимизацию можно отключить.

Примечание. Этот параметр не действует на уровнях оптимизации «Нет», «Низкий» и «Средний».

Для получения дополнительной информации о параметре командной строки см. --No_tbaa, стр. 306.

Пример

```
short F(short *p1, long *p2)
{
  *p2 = 0;
  *p1 = 1;
  return *p2;
}
```

При анализе псевдонимов на основе типов предполагается, что доступ на запись к короткому номеру, на который указывает p1, не может повлиять на длинное значение, на которое указывает p2. Следовательно, во время компиляции известно, что эта функция возвращает 0. Однако в нестандартном коде C или C ++ эти указатели могут перекрывать друг друга, будучи частью одного объединения. Если вы используете явное приведение типов, вы также можете заставить указатели разных типов указывать на одно и то же место в памяти.

Статическая кластеризация

Когда статическая кластеризация включена, статические и глобальные переменные, которые определены в одном модуле, расположены так, что переменные, к которым осуществляется доступ в одной функции, хранятся рядом друг с другом. Это позволяет компилятору использовать один и тот же базовый указатель для нескольких обращений.

Примечание. Этот параметр не действует на уровнях оптимизации «Нет» и «Низкий».

Для получения дополнительной информации о параметре командной строки см. --No_clustering, стр. 300.

Планирование инструкций

В компиляторе есть планировщик инструкций для повышения производительности сгенерированного кода. Для достижения этой цели планировщик изменяет инструкции, чтобы минимизировать количество остановок конвейера, возникающих из-за конфликтов ресурсов внутри микропроцессора.

Для получения дополнительной информации о параметре командной строки см. --No_scheduling, стр. 305.

Содействие генерации хорошего кода

В этом разделе содержатся советы о том, как помочь компилятору сгенерировать хороший код:

- Написание исходного кода, удобного для оптимизации, стр. 250.
- Экономия места в стеке и оперативной памяти, стр. 251.
- Функциональные прототипы, стр. 251.
- Целочисленные типы и битовое отрицание, стр. 252
- Защита переменных, к которым осуществляется одновременный доступ, стр. 253.
- Доступ к регистрам специальных функций, стр. 253

• Передача значений между C и объектами ассемблера, стр. 254
• Неинициализированные переменные, стр. 254

НАПИСАНИЕ ИСТОЧНИКА ДЛЯ ОПТИМИЗАЦИИ

Ниже приводится список методов программирования, которые, при соблюдении, позволят компилятору лучше оптимизировать приложение.

• Локальные переменные - автоматические переменные и параметры - предпочтительнее статических или глобальных переменных. Причина в том, что оптимизатор должен предположить, например, что вызываемые функции могут изменять нелокальные переменные. Когда срок службы локальных переменных заканчивается, ранее занятая память может быть повторно использована. Глобально объявленные переменные будут занимать память данных в течение всего выполнения программы.

• Избегайте использования адреса локальных переменных с помощью оператора &. Это неэффективно по двум основным причинам. Во-первых, переменная должна быть помещена в память и, следовательно, не может быть помещена в регистр процессора. Это приводит к более крупному и медленному коду. Во-вторых, оптимизатор больше не может предполагать, что на локальную переменную не влияют вызовы функций.

• Локальные переменные модуля - переменные, объявленные статическими, - предпочтительнее глобальных переменных (нестатических). Также избегайте использования адреса часто используемых статических переменных.

• Компилятор может встраивать функции, см. Встраивание функций, стр. 248. Чтобы максимизировать эффект преобразования встраивания, рекомендуется помещать определения небольших функций, вызываемых из более чем одного модуля, в файл заголовка, а не в файл файл реализации. Как вариант, вы можете использовать многофайловую компиляцию. Для получения дополнительной информации см. Многофайловые модули компиляции, стр. 245.

• Избегайте использования встроенного ассемблера без операндов и засоренных ресурсов. Вместо этого используйте SFR или встроенные функции, если они доступны. В противном случае используйте встроенный ассемблер с операндами и заторможенными ресурсами или напишите отдельный модуль на языке ассемблера. Для получения дополнительной информации см. Смешивание C и ассемблера, стр. 169.

СОХРАНЕНИЕ МЕСТА СТЕКА И ПАМЯТИ ОЗУ

Ниже приводится список методов программирования, которые экономят память и пространство стека:

• Если пространство стека ограничено, избегайте длинных цепочек вызовов и рекурсивных функций.

• Избегайте использования больших нескалярных типов, таких как структуры, в качестве параметров или возвращаемого типа. Чтобы сэкономить место в стеке, вы должны вместо этого передавать их как указатели или, в C ++, как ссылки.

ФУНКЦИОНАЛЬНЫЕ ПРОТОТИПЫ

Можно объявлять и определять функции, используя один из двух разных стилей:

- Прототипированный
- Керниган и Ричи Си (K&R C)

Оба стиля допустимы для C, однако настоятельно рекомендуется использовать прототипный стиль и предоставлять объявление прототипа для каждой общедоступной функции в заголовке, который включается как в модуль компиляции, определяющий функцию, так и во все модули компиляции, использующие его.

Компилятор не будет выполнять проверку типов для параметров, передаваемых функциям, объявленным в стиле K&R. Использование объявлений прототипов в некоторых случаях также приведет к более эффективному коду, поскольку нет необходимости в повышении типа для этих функций.

Чтобы компилятор требовал, чтобы все определения функций использовали стиль прототипа и чтобы все общедоступные функции были объявлены перед определением, используйте параметр

Project>Options>C/C++ Compiler>Language 1>Require prototypes compiler option (--require_prototypes).

Прототипированный стиль

В объявлениях прототипированных функций необходимо указать тип для каждого параметра.

```
int Test(char, int); /* Declaration */
int Test(char ch, int i) /* Definition */
{
  return i + ch;
}
```

Стиль Керниган и Ричи

В стиле K&R - pre-Standard C - невозможно объявить функцию прототипом. Вместо этого в объявлении функции используется пустой список параметров. Кроме того, определение выглядит иначе.

Например:

```
int Test();     /* Declaration */
int Test(ch, i) /* Definition */
char ch;
int i;
{
  return i + ch;
}
```

ЦЕЛЫЕ ТИПЫ И БИТОВОЕ ОТРИЦАНИЕ

В некоторых ситуациях правила для целочисленных типов и их преобразование могут привести к путанице. На что следует обратить внимание, - это присваивания или условные выражения (тестовые выражения), включающие типы с разным размером, и логические операции, особенно отрицание битов. Здесь типы также включают типы констант.

В некоторых случаях могут быть предупреждения - например, для постоянного условного или бессмысленного сравнения - в других - просто результат, отличный от ожидаемого. При определенных обстоятельствах компилятор может выдавать предупреждения только при более высоких оптимизациях, например, если

компилятор полагается на оптимизацию для определения некоторых экземпляров константных условных выражений. В этом примере предполагается 8-битный символ, 32-битное целое число и дополнение до двух:

```
void F1(unsigned char c1)
{
  if (c1 == ~0x80)
    ;
}
```

Здесь тест всегда ложный. Справа 0x80 равно 0x00000080, а ~ 0x00000080 становится 0xFFFFFF7F. Слева c1 - это 8-битный беззнаковый символ в диапазоне 0—255, который никогда не может быть равен 0xFFFFFF7F. Кроме того, оно не может быть отрицательным, что означает, что для повышенного интегрального значения никогда не могут быть установлены самые верхние 24 бита.

ЗАЩИТА ПЕРЕМЕННЫХ С ОДНОВРЕМЕННЫМ ДОСТУПОМ

Переменные, к которым осуществляется асинхронный доступ, например, с помощью подпрограмм прерывания или кода, выполняемого в отдельных потоках, должны быть правильно помечены и иметь адекватную защиту. Единственное исключение - переменная, которая всегда доступна только для чтения.

Чтобы правильно пометить переменную, используйте ключевое слово volatile. Это, помимо прочего, сообщает компилятору, что переменную можно изменить из других потоков. В этом случае компилятор будет избегать оптимизации переменной - например, отслеживания переменной в регистрах - не будет задерживать запись в нее и будет осторожен, обращаясь к переменной только столько раз, сколько указано в исходном коде.

Дополнительные сведения о квалификаторе типа volatile и правилах доступа к изменчивым объектам см. В разделе Объявление объектов volatile, стр. 378.

ДОСТУП К СПЕЦИАЛЬНЫМ ФУНКЦИОНАЛЬНЫМ РЕГИСТРАМ

Конкретные файлы заголовков для нескольких устройств Arm включены в установку продукта IAR. Заголовочные файлы называются ***iodevice.h*** и определяют регистры специальных функций процессора (SFR).

Примечание. Каждый файл заголовка содержит одну секцию, используемую компилятором, и одну секцию, используемую ассемблером.

SFR с битовыми полями объявляются в файле заголовка. Этот пример взят из ioks32c5000a.h:

```
__no_init volatile union
{
  unsigned short mwctl2;
  struct
  {
    unsigned short edr: 1;
    unsigned short edw: 1;
    unsigned short lee: 2;
    unsigned short lemd: 2;
    unsigned short lepl: 2;
```

```
  } mwctl2bit;
} @ 0x1000;

/* By including the appropriate include file in your code,
 * it is possible to access either the whole register or any
 * individual bit (or bitfields) from C code as follows.
 * Включив соответствующий включаемый файл в свой код,
 * можно получить доступ либо ко всему реестру, либо к любому
 * отдельному биту (или битовому полю) из кода C следующим образом.
 */
void Test()
{
  /* Whole register access Полный доступ к регистру */
  mwctl2 = 0x1234;
  /* Bitfield accesses Доступ к битовому полю */
  mwctl2bit.edw  = 1;
  mwctl2bit.lepl = 3;
}
```

Вы также можете использовать файлы заголовков в качестве шаблонов при создании новых файлов заголовков для других устройств Arm.

ПРОХОДНЫЕ ЗНАЧЕНИЯ МЕЖДУ ОБЪЕКТАМИ C И СБОРКОЙ

В следующем примере показано, как вы в исходном коде C можете использовать встроенный ассемблер для установки и получения значений из специального регистра:

```
static unsigned long get_APSR( void )
{
  unsigned long value;
  asm volatile( "MRS %0, APSR" : "=r"(value) );
  return value;
}
static void set_APSR( unsigned long value)
{
  asm volatile( "MSR APSR, %0" :: "r"(value) );
}
```

Регистр общего назначения используется для получения и установки значения регистра специального назначения APSR. Тот же метод можно также использовать для доступа к другим регистрам специального назначения и конкретным инструкциям.

Чтобы узнать больше о встроенном ассемблере, см. Встроенный ассемблер, стр. 170.

НЕИНИЦИАЛИЗИРОВАННЫЕ ПЕРЕМЕННЫЕ

Обычно среда выполнения инициализирует все глобальные и статические переменные при запуске приложения.

Компилятор поддерживает объявление переменных, которые не будут инициализированы, с использованием модификатора типа ***__no_init***. Их можно указать

как ключевое слово или с помощью директивы ***#pragma object_attribute***. Компилятор помещает такие переменные в отдельный раздел.

Для ***__no_init*** ключевое слово ***const*** подразумевает, что объект доступен только для чтения, а не то, что объект хранится в постоянной памяти. Невозможно дать объекту ***__no_init*** начальное значение.

Переменные, объявленные с использованием ключевого слова ***__no_init***, могут, например, быть большими входными буферами или отображаться в специальной оперативной памяти, которая сохраняет свое содержимое, даже когда приложение выключено.

Для получения дополнительной информации см. _ _No_init, стр. 391.

Примечание. Чтобы использовать это ключевое слово, необходимо включить языковые расширения, см. -E, стр. 291. Для получения дополнительной информации см. Атрибут_объекта, стр. 414.

#pragma location

Используйте эту директиву pragma, чтобы указать:

• Местоположение (абсолютный адрес) глобальной или статической переменной, объявление которой следует за директивой pragma. Переменные должны быть объявлены __no_init.

• Идентификатор, определяющий регистр. Переменная, определенная после директивы pragma, помещается в регистр. Переменная должна быть объявлена как __no_init и иметь область видимости файла.

Строка, определяющая раздел для размещения переменной или функции, объявление которой следует за директивой pragma. Не помещайте переменные, которые обычно находятся в разных разделах, например, переменные, объявленные как __no_init, и переменные, объявленные как const, в один и тот же раздел.

```
#pragma location = 0xFFFF0400
__no_init volatile char PORT1; /* PORT1 is located at address 0xFFFF0400 */
#pragma location = R8
__no_init int TASK; /* TASK is placed in R8 */
#pragma location = "FLASH"
char PORT2; /* PORT2 is located in section FLASH */
/* A better way is to use a corresponding mechanism */
#define FLASH _Pragma("location=\"FLASH\"")
/* ... */
FLASH int i; /* i is placed in the FLASH section */
```

#pragma section

Используйте эту директиву pragma для определения имени раздела, которое может использоваться операторами раздела __section_begin, __section_end и __section_size. Все объявления разделов для определенного раздела должны иметь одинаковое выравнивание.

Примечание. Чтобы разместить переменные или функции в определенном разделе, используйте директиву #pragma location или оператор @.

```
#pragma section="MYSECTION"
```