ОПТИМИЗИРУЮЩИЙ КОМПИЛЯТОР

для операционной системы «Нейтрино»

Руководство пользователя

Листов 47

2021

АННОТАЦИЯ

Данный документ содержит руководство пользователя для оптимизирующего компилятора языков C/C++. Руководство включает информацию, необходимую для запуска компилятора — режимы исполнения, параметры, опции, входные и выходные данные.

Компилятор генерирует нативный код для ЗОСРВ «Нейтрино». Сам компилятор функционирует на платформе Linux x86-64 (amd64) в режиме кросс-компиляции.

СОДЕРЖАНИЕ

1. НАЗНАЧЕНИЕ И УСЛОВИЯ ПРИМЕНЕНИЯ ПРОГРАММЫ

Программа компилятор предназначена для автоматического преобразования алгоритма, текст которого написан на языке высокого уровня, в эквивалентный ему алгоритм, представленный в виде последовательности машинных команд. Компилятор осуществляет компиляцию текстов программ на языках программирования C/C++ в файлы двоичного кода целевой платформы.

Для языка C:

- номинальная совместимость с gcc-5.5.0;
- стандарт C90 (ANSI/ISO 9899:1990) поддержан полностью;
- стандарт C99 (ISO/IEC 9899:1999 as modified by Technical Corrigenda 1 through 3) поддержан полностью;
- стандарт C11 (ISO/IEC 9899:2011) поддержан полностью, за исключением необязательного расширения _atomic.

По умолчанию включен режим -std=gnu11 (язык C11 с gnu-расширениями).

Для языка C++:

- номинальная совместимость с g++-5.5.0;
- библиотека libstdc++ от gcc-5.5.0;
- технология zero cost exceptions (0eh);
- стандарт C++03 (ISO/IEC 14882:2003) поддержан полностью;
- стандарт C++11 (ISO/IEC 14882:2011) поддержан полностью;
- стандарт C++14 (ISO/IEC 14882:2014) поддержан полностью.

По умолчанию включен режим -std=gnu++98 (язык C++03 с gnu-расширениями)

2. ХАРАКТЕРИСТИКИ ПРОГРАММЫ

Компилятор — программа, преобразующая алгоритм, текст которого написан на языке высокого уровня, в эквивалентный ему алгоритм, представленный в виде последовательности машинных команд.

Компилятор может формировать код различной эффективности. Критерием эффективности является время исполнения кода программы. При этом, чем меньше время исполнения программы, тем более эффективным считается ее код.

В процессе работы компилятор может выполнять над алгоритмом программы различные преобразования, которые могут привести к формированию более эффективного кода. Назовем их оптимизирующими преобразованиями, или оптимизациями. Вообще говоря, применение оптимизаций вовсе не гарантирует повышение эффективности. Всё зависит от исходного алгоритма. Тем не менее, правильный подбор оптимизирующих преобразований может серьезно уменьшить время исполнения задачи.

Компилятор производит оптимизации не произвольным образом, а в составе некоторого согласованного пакета оптимизаций. Предусмотрено несколько таких пакетов или уровней оптимизаций. Существует начальный уровень простейших оптимизаций, который выполняется всегда. Код, полученный с использованием начального уровня оптимизаций, называется неоптимизированным. Оптимизированным называется код, полученный с применением уровня оптимизаций выше начального. Компилятор называется оптимизирующим, если он способен создавать оптимизированный код. Критерием правильности работы Компилятора является работа откомпилированного кода в соответствии с алгоритмом исходной программы.

Для конечного пользователя работа Компилятора представляется как единый и неделимый процесс. Однако в действительности это процесс состоит из последовательных стадий (фаз компиляции), исполняемых в автоматическом режиме. С помощью специальных опций программист имеет возможность

завершить процесс компиляции после любой стадии исполнения. Более того, в некоторых случаях имеется возможность начать процесс компиляции, минуя некоторые начальные стадии работы.

Рассмотрим фазы компиляции.

Первой фазой компилятора является препроцессор. Любая достаточно сложная программа работает в некотором программном окружении. Информация об этом окружении содержится в первую очередь в заголовочных файлах системных библиотек. Кроме того, если программа состоит из нескольких модулей, такая информация содержится в заголовочных файлах этих модулей. Препроцессор обеспечивает корректное подключение заголовочных файлов к исходной программе. На выходе препроцессора формируется преобразованный текст исходной программы, в котором раскрыты все макроопределения, удалены комментарии и фрагменты, на которые распространилось действие так называемой условной трансляции. Такой текст полностью готов к следующей стадии компиляции.

На второй фазе компиляции происходит непосредственное преобразование алгоритма, написанного на языке высокого уровня, в алгоритм, выраженный в терминах команд микропроцессора. Назовем эту фазу транслятор. На этой фазе происходят все оптимизирующие преобразования. Кроме исходного текста, на высоких уровнях оптимизации транслятор может использовать дополнительную информацию о программе (профиль исполнения и др.). На выходе транслятор формирует последовательность команд микропроцессора, представленных в виде двоичного кода.

Следует иметь в виду, что хотя работа препроцессора и транслятора описаны, как два независимых процесса, управляемых своими наборами параметров, реально они реализованы как единая программа.

Результатом работы транслятора является файл объектного кода. Полученный код неполон и не может быть непосредственно исполнен на микропроцессоре.

Для этого требуется объединить его со стартовыми файлами, системными библиотеками и, возможно, кодом других модулей. В результате этих действий образуется готовый к запуску на микропроцессоре файл исполняемого кода. Следует отметить, что далеко не всегда целью работы компилятора является создание исполняемого кода. Весьма часто целью работы является создание библиотечных файлов или архивов, в которые с помощью специальных программ-архиваторов объединяются несколько файлов объектного кода.

Завершающей фазой является редактирование связей или линковка. Для этого компилятор вызывает штатно установленную в системе программу — редактор связей. Компилятор автоматически передает редактору связей соответствующие параметры, заданные в командной строке, а также те, что необходимо задать по умолчанию. Таким образом, для получения исполняемого кода программисту необязательно явно вызывать редактор связей — компилятор все сделает сам. Кроме формирования файла исполняемого кода, редактор связей может формировать динамические библиотеки.

Компилятор может работать в различных режимах. Можно выделить несколько основных параметров формирования кода, различные сочетания которых могут характеризовать тот или иной режим. Следует помнить, что при сборке готовой программы все объектные файлы и библиотеки должны быть откомпилированы в одном режиме. Исключения из этого правила будут оговорены особо.

Режимы компиляции:

- основные параметры формирования кода;
- позиционно-независимый код;
- оптимизированный код;
- код с отладочной информацией;
- размер адреса в память;
- защищенный режим.

2.1. Позиционно-независимый код

Отличие позиционно-независимого кода (position-independent code, PIC) от «обычного» состоит в следующем. В «обычном» или статическом коде ссылка на глобальный объект (переменную или функцию) представляет собой литерал, содержащий непосредственный адрес этого объекта. Иными словами, адрес объекта задается статически во время линковки программы. Но для этого необходимо, чтобы все модули, из которых состоит программа, были собраны в один исполняемый код. Как следствие, при изменении одного из модулей необходимо пересобрать весь код программы.

В позиционно-независимом коде адреса всех объектов собраны в специальной таблице (Global Offset Table, сокращенно GOT), а в качестве литеральной ссылки на объект выступает индекс по таблице GOT. При обращении к объекту сначала считывается её адрес из таблицы, а потом идет обращение по считанному адресу. При этом код одного модуля не имеет жесткой привязки к коду другого модуля, т.к. содержимое таблицы GOT формируется не в момент линковки, а в момент загрузки программы. Как следствие, программа может состоять не из одного, а из нескольких файлов, откомпилированных независимо друг от друга, и изменения в одном модуле не потребуют перекомпиляции всей программы.

2.2. Оптимизированный код

Данный признак характеризует не столько сам код, сколько способ его получения. Оптимизации дают возможность более полно использовать потенциал аппаратуры. Вследствие этого многие инструкции микропроцессора могут содержаться только в оптимизированном коде.

Введена иерархия уровней оптимизации — каждому уровню оптимизации присвоен свой номер. Уровень с большим номером включает в себя уровень с меньшим номером, базовый уровень имеет номер 0 (таблица 1).

Т а б л и ц а 1 — Уровни оптимизации

Уровень оптимизации	Описание оптимизаций
O0	Начальный уровень. Сохраняет соответствие между исходным текстом программы и ее двоичным кодом, что позволяет формировать отладочную информацию для символьного отладчика
O1	Локальные оптимизации потока данных и управления, не требующие аппаратной поддержки, выполняются в рамках линейного участка
O2	Внутрипроцедурные оптимизации потока данных и управления, использующие все аппаратные возможности, цикловые оптимизации, слабо увеличивающие размер кода, минимальные межпроцедурные оптимизации
O3	Все межпроцедурные оптимизации, возможность оптимизации в режиме «вся программа», все цикловые оптимизации

Допускается формирование кода программы из объектов, полученных с разными уровнями оптимизации.

2.3. Код с отладочной информацией

Отладчик или дебагер (от анг. Debugger) — утилита, позволяющая выполнять трассировку процесса выполнения двоичного кода алгоритма с привязкой исполняемых команд к исходному тексту программы. Для корректной работы отладчика необходимо наличие специально сформированной отладочной информации. Кроме того, необходимо, чтобы двоичный код программы однозначно соответствовал конструкциям входного текста. Компилятор на высоких уровнях оптимизации столь сильно преобразует алгоритм исходной программы, что такой однозначности достигнуть не удается. Поэтому отлаживать с помощью отладчика можно только неоптимизированный код программы.

Допускается формирование кода программы из объектов, полученных как с отладочной информацией, так и без нее.

2.4. Размер адреса в память

В компиляторе реализована работа с разными типами адресов.

Для платформы «Эльбрус» компилятор по умолчанию формирует код с размером адреса 64 бита. Для совместимости со старыми программами, для которых этот параметр критичен, компилятор может формировать код с адресами размером 32 бита. Код с 32-х и 64-х битной адресацией не имеет принципиальных отличий, кроме номинального размера указателя и, как следствие, возможности обратиться к большему объему памяти.

Код с размером адреса 128 бит реализует защищенный режим программирования.

Допускается формирование кода программы из объектов, имеющих только одинаковый размер адреса в память. Для платформы SPARC V9 компилятор формирует код с размером адреса 32 и 64 бита.

2.5. Код для защищенного режима

Данный режим реализован только для платформы «Эльбрус».

Под защищенным режимом понимается исполнение кода программы с использованием аппаратной поддержки разделения контекста и контроля доступа к данным. Код такой программы должен быть соответствующим образом сгенерирован с максимальным сохранением исходных свойств входного языка высокого уровня и, быть может, желаний разработчика программы по ее структуризации. Для поддержки защищенного режима в аппаратуре микропроцессора «Эльбрус» имеются специальные средства. Подробнее они описаны в системе команд, здесь упомянем лишь, что при обращении в память в защищенном режиме в качестве указателя используется специальная структура размером 128 бит. Кроме собственно адреса объекта в памяти, она содержит дополнительную информацию (размер объекта, смещение внутри объекта и др.). Это дает возможность аппаратуре контролировать ошибки при обращении в память (например, выход за границу массива).

2.6. Автоматическое распараллеливание программ

Включение режима компиляции для автоматического распараллеливания программ на общей памяти осуществляется опцией `-fautopar`.

При автоматическом распараллеливании на этапе исполнения программы циклические участки программы, возможность распараллеливания которых была определена в результате фазы анализа оптимизирующего компилятора, распадаются на несколько параллельных потоков, исполняемых на разных ядрах микропроцессора. Взаимодействие и синхронизация потоков происходит с помощью специальной библиотеки поддержки автоматического распараллеливания. Число потоков задается статически с помощью опции `-fthreads=N`, где `N` — число потоков.

2.6.1. Распараллеливание на распределенной памяти

В отличие от распараллеливания на общую память, работа одного приложения в режиме распределенной памяти требует серьезных накладных расходов в виде регулярной пересылки данных с одной машины на другую по каналу обмена данными. Этот факт приводит к серьезным ограничениям на эффективную применимость автоматического распараллеливания.

Поскольку решение о распараллеливании оптимизирующий компилятор должен принимать по результатам анализа исходного кода, рассмотрению подлежат горячие по профилю гнезда циклов, не содержащих вызовов процедур, ввиду ресурсных ограничений на работу анализа.

Вначале проводится цикловой анализ зависимостей по памяти для всех операций внутри гнезда циклов. Такая работа проводится и при автоматическом распараллеливании на общую память. Пользователю рекомендуется:

- использовать выстроенные массивы для хранения циклически обрабатываемых данных;

- использовать модификатор `restrict` для обозначения неконфликтующих указателей;

- не использовать одновременной нелинейной работы с индексами обрабатываемых массивов.

После проведения анализа теоретической возможности распараллелить гнездо циклов, компилятор проводит оценку эффективности работы.

Главными характеристиками гнезда циклов для принятия решения о распараллеливании являются:

- оценка времени работы гнезда циклов;

- оценка объема данных, требуемых для пересылки.

Для того, чтобы оценки были точными, необходимо использовать двухфазный режим компиляции со сбором динамического профиля.

Компилятор старается распознать все обращения к массивам внутри гнезда, оценить диапазоны изменения индексов в этих массивах и тем самым определить объем обрабатываемых данных.

Распараллеливание гнезда считается эффективным, если оценка времени работы после распараллеливания с учетом накладных расходов меньше времени работы исходного цикла. Поскольку темп вычислительной обработки данных, как правило, существенно превышает темп передачи данных, компилятором используется следующий простой критерий — размерность пересылаемых данных должна быть строго меньше, чем размерность обрабатываемых данных

Приведем примеры:

1) умножение квадратных матриц.

Для умножения двух матриц размера NxN требуется $2N^3$ арифметических операций. При распараллеливании на K узлов потребуется разрезать одну из матриц на K полос, передать полосы на вычислительные узлы, растиражировать вторую матрицу на все узлы, провести на узлах параллельное умножение матриц размера N/KxN и NxN, и собрать полосы обратно на головной узел. Итого необходимо провести $(2+K)*N^2$ пересылок.

При K<<N положительный эффект от распараллеливания C_1*N^3 арифметических операций превысит накладные расходы от передачи C_2*N^2 байт данных. Компилятор примет здесь верное решение о распараллеливании, так как 2 строго меньше 3;

2) транспонирование квадратной матрицы.

Распараллеливание этой задачи на распределенную память неэффективно, так как для ее решения на одном узле требуется N^2 чтений и столько же записей. Для распараллеливания потребуется переслать матрицу на удаленные узлы и затем собрать ее обратно, что сохранит количество чтений и записей на головном узле, то есть не позволит сократить астрономическое время работы распараллеленной программы. Компилятор откажется от распараллеливания т.к. 2=2, и требуемое строгое неравенство не выполняется.

2.7. Код для сбора профильной информации

Профиль исполнения задачи представляет собой набор статистических данных. Это счетчики исполнения команд, счетчики итераций циклов и др. На основании этой информации можно вычислить вероятности переходов, выделить так называемые «горячие» (наиболее часто повторяющиеся) участки задачи и т.п.

Компиляция с использованием профиля позволяет получить более эффективный код. Получение и использование профиля состоит из следующих шагов:

1) получение профиля. Вначале программа компилируется в специальном режиме — должна быть подана опция `-fprofile-generate`. При этом в код встраиваются специальные инструментальные средства, позволяющие получить статистику исполнения отдельных участков программы. Полученный код запускается на исполнение. В результате исполнения программы получается файл профиля (по умолчанию `eprof.sum`);

2) оптимизация программы с использованием профильной информации. Для этого нужно второй раз запустить компилятор, при этом опции запуска должны

быть точно такими же, как при первом запуске. Единственное исключение — вместо опции формирования профиля должна быть подана опция его использования `-fprofile-use`. Полученный код готов к исполнению.

Пример работы компилятора с использованием профильной информации:

```
lcc -fprofile-generate –O ttt.c –o ttt
lcc -fprofile-use –O ttt.c –o ttt
./ttt
```

3. ОБРАЩЕНИЕ К ПРОГРАММЕ

Обращение к компилятору, вне зависимости от целевой платформы, имеет следующий вид:

`lcc <опции> <входные файлы> …`

В зависимости от режима запуска компилятор требует определенного набора входных параметров или опций. Однако, вне зависимости от остальных параметров, при задании опции `--help` Компилятор распечатает полный список своих опций и завершит исполнение. Опции и входные файлы могут следовать в любом порядке.

При запуске Компилятора в случае успешного завершения образуются выходные данные, и формируется нулевой код возврата. Если в процессе исполнения были обнаружены ошибки, выходных данных не образуется и формируется ненулевой код возврата.

Набор опций компилятора зависит от целевой платформы.

3.1. Опции компилятора

В таблице 2 приведены наиболее используемые опции.

Т а б л и ц а 2 — Опции компилятора

Опции управления процессом компиляции	
-c	Компилировать до объектного файла
-S	Компилировать до ассемблерного файла
-o<file> (-o <file>)	Сохранять результат работы в файле <file>
-v (--verbose)	Выводить строки запуска компонент и их verbose-выдачи
Опции режима компиляции целевой архитектуры	
-mcpu=<cpu>	Формировать код для системы команд <cpu>
Для e90s:	
-m32	Компилировать в режиме 32 бит (режим по умолчанию)
-m64	Компилировать в режиме 64 бит
Для e2k:	
-mptr32	Компилировать в режиме 32 бит
-mptr64	Компилировать в режиме 64 бит (режим по умолчанию)
-mptr128	Компилировать в защищённом режиме

Продолжение таблицы 2

Опции управления входным языком	
-ansi (--ansi)	Компилировать в режиме строгого ANSI C
-fgnu (-fno-gnu)	Включить (выключить) поддержку GNU-расширений
-C++	Компилировать в режиме C++
-traditional	Компилировать в режиме совместимости с K&R
-traditional-cpp	Компилировать с использованием старого стиля препроцессирования
-fpermissive	Разрешать устаревшие конструкции C++
-std=<standard>	Использовать стандарт C/C++ <standard>
-fexceptions (-fhandle-exceptions)	Включить поддержку обработки исключительных ситуаций языка C++ (по умолчанию)
-fno-exceptions (-fno-handle-exceptions)	Обратная опция к -fexceptions
Опции управления препроцессированием	
-I <dir> (-I<dir>)	Добавить директорию <dir> к списку директорий для поиска заголовочных файлов
-idirafter <dir>	Добавить директорию <dir> в конец списка директорий для поиска заголовочных файлов
-D<name[=value]>	Задать макроопределение для препроцессора
-U<name>	Отменить макроопределение для препроцессора
-E	Только препроцессировать
-P	Только препроцессировать; не генерировать директивы #line
-C	Сохранять комментарии в препроцессированном тексте
-M	Генерировать список зависимостей для программы make и печатать их на экран, компиляцию не запускать
-MD	Генерировать список зависимостей для программы make и сохранять их в файл
-MM	Опция аналогична -M, но не формировать зависимости от системных файлов
-MMD	Опция аналогична -MD, но не формировать зависимости от системных файлов
-MT <name>	При использовании опции -M, -MM, -MD или -MMD: использовать имя цели <name>
-MF <file>	При использовании опции -MD или -MMD: сохранять зависимости в файле <file>
-MP	При использовании опции -M, -MM, -MD или -MMD: генерировать специальные цели на случай удаления требуемых файлов
-dM	При использовании опции -E: печатать список макроопределений, полученных к концу компиляции
-H	Печатать имена подключаемых заголовочных файлов
-undef	Не создавать дополнительных предопределенных макроопределений
-nostdinc	Не пользоваться стандартными путями поиска заголовочных файлов

Продолжение таблицы 2

-nostdinc++	Не пользоваться стандартными путями поиска заголовочных файлов C++
Опции управления режимом компиляции	
-fPIC (-fpic)	Генерировать позиционно-независимый код
-fPIE (-fpie)	Генерировать позиционно-независимый код, подходящий только для исполняемых файлов
-pg	Генерировать код программы для получения профиля для дальнейшего его использования программой gprof
Опции отладочной информации	
-g0	Отключить генерацию отладочной информации
-g	Генерировать отладочную информацию
Дополнительные опции	
-fkernel	Задать режим сборки ядра ОС Linux
Опции для работы с предупреждениями	
-W	Включить дополнительные предупреждения
-w	Подавлять все предупреждения и замечания компилятора
Базовые опции оптимизаций	
-O0	Компилировать с уровнем оптимизации O0
-O1	Компилировать с уровнем оптимизации O1
-O2	Компилировать с уровнем оптимизации O2
-O3	Компилировать с уровнем оптимизации O3
-Os (-Osize)	Использовать стратегию оптимизации с учетом размера генерируемого кода
-Obase	Компилировать в режиме «вся программа» (см. fwhole)
-O	Компилировать с уровнем оптимизации по умолчанию (-O3 -Osize)
Опции профилирования	
-fprofile-use[=<file>]	Загрузить данные профилирования из файла <file> (по умолчанию eprof.sum)
-fprofile-use-ext[=<file>]	Использовать профиль, собранный с одновременным профилированием библиотечных процедур
-fprofile-generate[=<path>]	Генерировать код программы для получения профиля в каталоге <path> (по умолчанию в текущем каталоге)
-fprofile-generate-ext[=<path>]	Генерировать инструментальный код программы для его одновременного профилирования с библиотечными процедурами
-fprofile-subst= <old_module_name1>: <new_module_name1>,...	Заменить имя считанного из профиля файла <old_module_name> на указанное имя <new_module_name>
-fprofile-strict-names	Выдать ошибку при отсутствии в профиле информации для файла, поданного в командной строке

Продолжение таблицы 2

Опции режима "вся программа"	
-fwhole	Включить режим компиляции "вся программа", в том числе и для помодульной сборки
Расширенные опции управления процессом компиляции	
-B<prefix> (-B <prefix>)	Добавить префикс поиска исполняемых компонент, заголовочных файлов, библиотек и crt-модулей
-Wa,<opt1>,<opt2>...	Передать опции <opt1> <opt2> ... напрямую ассемблеру
-Wc,<opt1>,<opt2>...	Передать опции <opt1> <opt2> ... напрямую компилятору
-Wl,<opt1>,<opt2>...	Передать опции <opt1> <opt2> ... напрямую линкеру
-Wp,<opt1>,<opt2>...	Передать опции <opt1> <opt2> ... напрямую препроцессору
-Xlinker <opt1>,<opt2>...	Эквивалентно опции -Wl,<opt1>,<opt2>...
-x<lang> (-x <lang>)	Считать, что последующие файлы содержат исходный код на языке <lang>
Опции управления линковкой	
-dynamic	Использовать динамическую линковку
-frpath-internal (-fno-rpath-internal)	Включить (выключить) пути до внутренних библиотек в пути поиска динамических библиотек в момент исполнения
-L<dir> (-L <dir>)	Добавить <dir> к списку директорий для поиска библиотек
-l<name> (-l <name>)	Линковать с библиотекой <name>
-nodefaultlibs	Не линковать с библиотеками по умолчанию
-nodefaultlibs++	Не линковать с библиотеками C++ по умолчанию
-nostartfiles	Не линковать со стартовыми модулями по умолчанию
-nostdlib	Эквивалентно комбинации опций -nodefaultlibs и -nostartfiles
-rdynamic	Вывести все символы в динамическую таблицу символов
-shared (--shared)	Создать динамическую библиотеку
-static	Использовать статическую линковку
-shared-libgcc	Аварийное завершение на этапе компиляции (для контроля совместимости с gcc)
-static-libgcc	Опция игнорируется (для совместимости с gcc)
-symbolic	Передать в линкер опцию -Bsymbolic
Опции, которые передаются линкеру (подробное описание см. [1])	
-e<entry> (-e <entry>)	Задать стартовый адрес программы
-h<soname> (-h <soname>)	Задать внутреннее имя динамической библиотеки
-pie	Создать перемещаемый выходной файл
-R<path> (-R <path>)	Использовать только символы из входного файла
-r	Создать перемещаемый выходной файл
-s	Вырезать все символы из выходного файла
-T<script> (-T <script>)	Использовать скрипт линкера
-u<symname> (-u <symname>)	Считать символ изначально неопределенным
-z<keyword> (-z <keyword>)	Разные опции, см. [1]
Поддерживаемые переменные окружения	
TMPDIR	Настройка каталога для временных файлов
CPATH	Дополнительные пути поиска заголовочных файлов

Продолжение таблицы 2

C_INCLUDE_PATH	Дополнительные пути поиска заголовочных файлов языка C
CPLUS_INCLUDE_PATH	Дополнительные пути поиска заголовочных файлов языка C++
DEPENDENCIES_OUTPUT	Значение должно иметь формат "file[name]"; то же, что и -MMD -MF file -MT name
SUNPRO_DEPENDENCIES	Значение должно иметь формат "file[name]"; то же, что и -MD -MF file -MT name
LIBRARY_PATH	Дополнительные пути поиска библиотек
LC_MESSAGES	Определяет язык выдачи сообщений

3.2. Программное окружение

Любая программа работает в некотором системном окружении. Общепринято, что информация о системном окружении и режимах трансляции передается от компилятора программе через предустановленные макроопределения. Ниже приведены имена и значения основных предопределенных макроопределений компилятора и описаны случаи, при которых они выставляются.

Независимо от версии компилятора предопределённые макросы можно посмотреть следующим образом:

```
lcc -dM -E -xc /dev/null
```

Данный запуск покажет все установленные на конец препроцессирования макросы. Поскольку мы подаём пустой файл, то это означает, что мы увидим все предустановленные макросы в режиме Си. Если нужен режим Си++, то вместо `-xc` подаём `-xc++`. Если подадим опцию `-mptr128`, то в печати появятся макросы, которые взводятся по `-mptr128`, если подадим `-O2`, то появятся макросы, которые взводятся для режима с оптимизациями и т.п.

Предопределённый макрос эквивалентен тому, что в в самом начале каждого компилируемого файла из командной строки появляется соответствующая директива `#define`. Или, что то же самое, в компиляцию подаются соответствующие опции `-D`. По правилам языков Си/Си++ при наличии конструкции типа `#define XXX` (т.е. задаётся имя макроса, но не задаётся его значение) макрос устанавливается в значение `1`. То же самое касается

и предопределённых макросов. То есть, если в описании просто задано имя макроса, то в соответствии с правилами значение макроса будет выставлено в 1. За исключением случаев, когда это явным образом оговорено.

Макросы, зависящие от целевой операционной системы.

Макросы, специфичные для `target-os linux`:

`__linux`, `__linux__`, `linux` последний взводится только при отсутствии режима `-ansi`;

`__SIZE_TYPE__=unsigned long`;

`__PTRDIFF_TYPE__=long`;

`__WCHAR_TYPE__=int`;

`__pic__`, `__PIC__` взводятся одновременно в режиме `-fPIC` или `-fpic`;

`_LP64`, `__LP64__` для e2k в режиме `-mptr64`, для e90s в режиме `-m64`;

`__LONG_DOUBLE_128__` для e90s в режиме `-m32` (исторический рудимент, связанный с развитием поддержки `long double` на `sparc-linux`).

Макросы, специфичные для целевой ЗОСРВ «Нейтрино»:

`__QNX__`, `__QNXNTO__`;

`__SIZE_TYPE__= unsigned long`;

Макросы, определяющие характеристики базовых типов.

Данные макросы введены для совместимости с компилятором `gcc`, и используются, в том числе, и в стандартных заголовочных файлах `limits.h` и `float.h`

Макросы, задающие характеристики целочисленных типов:

`__SCHAR_MAX__=127`;

`__SHRT_MAX__=32767`;

`__INT_MAX__=2147483647`;

__LONG_MAX__=<val> для multicore, e2k в режиме -mptr32, e90s в режиме -m32 <val> равно 2147483647L, в остальных случаях — 9223372036854775807L;

__LONG_LONG_MAX__=9223372036854775807LL;

__CHAR_BIT__=8.

Макросы, задающие характеристики типа float:

__FLT_RADIX__=2;

__FLT_EVAL_METHOD__=0;

__FLT_MANT_DIG__=24;

__FLT_DIG__=6;

__FLT_MIN_EXP__=(-125);

__FLT_MIN_10_EXP__=(-37);

__FLT_MAX_EXP__=128;

__FLT_MAX_10_EXP__=38;

__FLT_MIN__=1.175494351E-38F;

__FLT_MAX__=3.402823466E+38F;

__FLT_EPSILON__=1.192092896E-07F;

__FLT_DENORM_MIN__=1.40129846e-45F;

__FLT_HAS_INFINITY__=1;

__FLT_HAS_QUIET_NAN__=1;

Макросы, задающие характеристики типа double:

__DBL_MANT_DIG__=53;

__DBL_DIG__=15;

__DBL_MIN_EXP__=(-1021);

__DBL_MIN_10_EXP__=(-307);

__DBL_MAX_EXP__=1024;

```
__DBL_MAX_10_EXP__=308;
__DBL_MIN__=2.2250738585072014E-308;
__DBL_MAX__=1.7976931348623157E+308;
__DBL_EPSILON__=2.2204460492503131E-16;
__DBL_DENORM_MIN__=4.9406564584124654e-324;
__DBL_HAS_INFINITY__=1;
__DBL_HAS_QUIET_NAN__=1.
```

Макросы, задающие характеристики типа `long double` на e2k (взводятся только на e2k):

```
__LDBL_MANT_DIG__=64;
__LDBL_DIG__=18;
__LDBL_MIN_EXP__=(-16381);
__LDBL_MIN_10_EXP__=(-4931);
__LDBL_MAX_EXP__=16384;
__LDBL_MAX_10_EXP__=4932;
__LDBL_MIN__=3.362103143112093506262 7E-4932L;
__LDBL_MAX__=1.189731495357231765021 3E+4932L;
__LDBL_EPSILON__=1.0842021724855044340075E-19L;
__LDBL_DENORM_MIN__=3.6451995318824746025 3e-4951L;
__LDBL_HAS_INFINITY__=1;
__LDBL_HAS_QUIET_NAN__=1;
__DECIMAL_DIG__=21.
```

Макросы, задающие характеристики типа `long double` на e90s (взводятся только на e90s):

```
__LDBL_MANT_DIG__=113;
__LDBL_DIG__=33;
```

```
__LDBL_MIN_EXP__=(-16381);
__LDBL_MIN_10_EXP__=(-4931);
__LDBL_MAX_EXP__=16384;
__LDBL_MAX_10_EXP__=4932;
__LDBL_MIN__=3.36210314311209350626267781732175260 3E-
4932L;
__LDBL_MAX__=1.18973149535723176508575932662800701 6E+493
2L;
__LDBL_EPSILON__=1.92592994438723585305597794258492 7319E
-34L;
__LDBL_DENORM_MIN__=6.47517511943802511092443895822 76465
5e-4966L;
__LDBL_HAS_INFINITY__=1;
__LDBL_HAS_QUIET_NAN__=1;
__DECIMAL_DIG__=36.
```

Прочие макросы, которые взводятся всегда.

Оговоримся, что подразумевается под "всегда". Есть макросы, которые взводятся только для e2k, но не взводятся для e90s. В этом смысле по отношению к e2k оно означает "всегда". Есть макросы, которые взводятся всегда, но в зависимости от целевой архитектуры значение макроса может меняться. Но по отношению к тому, что макрос присутствует или отсутствует, оно означает "всегда".

Макросы, которые взводятся всегда:

`__unix`, `__unix__`, `unix` последний взводится только при отсутствии режима `-ansi`;

`__ELF__`;

__LONG_DOUBLE__=<val> для e2k <val> равно 80, для e90s 128;

__LCC__=123 соответствует первым двум числам версии компилятора;

__LCC_MINOR__=<val> значение равно последнему числу в номере версии компилятора. Например, для версии компилятора 1.23.18 <val> равно 18;

_LITTLE_ENDIAN только для e2k;

__SIGNED_CHARS__;

__PRAGMA_REDEFINE_EXTNAME;

__STDC__;

__DATE__=<val> выставляется в строковый литерал с датой компиляции, например "Feb 13 2014";

__TIME__=<val> выставляется в строковый литерал с временем компиляции, например "13:48:01";

__FILE__=<val> выставляется в строковый литерал с именем файла, соответствующего точке использования макроса, например t.c;

__LINE__=<val> выставляется в номер строки, соответствующей точке использования макроса, например 10;

__EDG__;

__EDG_VERSION__=414;

__EDG_SIZE_TYPE__=unsigned long;

__EDG_PTRDIFF_TYPE__=long.

Прочие макросы, которые взводятся в зависимости от поданных опций:

__STRICT_ANSI__ в режиме -ansi;

__OPTIMIZE__ в режимах с оптимизациями;

__FAST_MATH__ в режиме -ffast-math;

__EXCEPTIONS в режиме -fexceptions;

_REENTRANT в режиме -pthreads;

__VIS=<val> только для e90s в режиме -mvis:

- в режиме -mcpu=ultrasparc <val> равно 0x100,

- в режиме -mcpu=ultrasparc3 — 0x200,

- в режиме -mcpu=r1000 — 0x300.

_OPENMP в режиме -fopenmp;

__MMX__ только для e2k в режимах -mmmx, -msse, -msse2, -msse3, -mssse3;

__SSE__ только для e2k в режимах -msse, -msse2, -msse3, -mssse3;

__SSE2__ только для e2k в режимах -msse2, -msse3, -mssse3;

__SSE3__ только для e2k в режиме -msse3, -mssse3;

__SSSE3__ только для e2k в режиме -mssse3;

__SSE4_1__ только для e2k в режиме -msse4.1;

__SSE4_2__ только для e2k в режиме -msse4.2;

__SSE4__ только для e2k в режиме -msse4;

__AVX__ только для e2k в режиме -mavx;

__3dNOW__ только для e2k в режиме -m3dnow;

__SSE4A__ только для e2k в режиме -msse4a;

__FMA4__ только для e2k в режиме -mfma4;

__XOP__ только для e2k в режиме -mxop;

__PCLMUL__ только для e2k в режиме -mpclmul;

__RDRND__ только для e2k в режиме -mrdrnd;

__BMI__ только для e2k в режиме -mbmi;

__TBM__ только для e2k в режиме -mtbm;

__ABM__ только для e2k в режиме -mabm;

__F16C__ только для e2k в режиме -mf16c;

__POPCNT__ только для e2k в режиме -mpopcnt;

__USER_LABEL_PREFIX__ в режиме -fgnu. Значение макроса равно пустышке (а не 1, как это бы следовало из общих правил);

__REGISTER_PREFIX__ в режиме -fgnu. Значение макроса равно пустышке (а не 1, как это бы следовало из общих правил);

__NO_INLINE__ в режиме -fgnu в случае если не используются оптимизации и не используется профилирование;

_GNU_SOURCE в режиме C++ -fgnu;

__GXX_WEAK__ в режиме C++ -fgnu;

__DEPRECATED в режиме C++ -fgnu -Wdeprecated;

__STDC_VERSION__=<val> в режиме языка C99 <val> равно 199901L, в остальных случаях 199409L;

__STDC_HOSTED__ в режиме языка C99;

__cplusplus=<val> в режиме C++ -fgnu <val> равно 1, в режиме C++ -fno-gnu 199711L;

_WCHAR_T в режиме языка C++. Макрос означает наличие поддержки ключевого слова "wchar_t";

_BOOL в режиме языка C++. Макрос означает наличие поддержки ключевого слова "bool";

__ARRAY_OPERATORS в режиме языка C++. Макрос означает наличие поддержки операторов new[] и delete[];

__RTTI в режиме языка C++ в режиме -frtti. Макрос означает наличие поддержки RTTI (Run-Time Type Information);

__PLACEMENT_DELETE в режиме языка C++. Макрос означает наличие поддержки оператора "placement delete";

__EDG_RUNTIME_USES_NAMESPACES в режиме языка C++;

__EDG_IA64_ABI в режиме языка C++. Макрос означает использование IA64_ABI при работе с Си++;

__EDG_IA64_ABI_USE_INT_STATIC_INIT_GUARD в режиме языка C++;

__NO_LONG_LONG в режиме -ansi -fno-gnu. Макрос означает отсутствие поддержки типа "long long";

__EDG_TYPE_TRAITS_ENABLED в режиме языка C++ в режиме -fno-gnu;

__GNUC__=5 в режиме -fgnu;

__GNUC_MINOR__=5 в режиме -fgnu;

__GNUC_PATCHLEVEL__=0 в режиме -fgnu;

__GNUG__=5 в режиме C++ -fgnu;

__VERSION__="5.5" в режиме -fgnu.

3.3. Ассемблерные вставки

Формат ассемблерной вставки:

```
asm ( "ассемблерный код"
      :  список выходных операндов
      :  список входных операндов
      :  список используемых регистров  );
```

Изначально gnu-вставка подразумевает, что компилятор её внутренностей не видит. То есть вставка представляет собой чёрный ящик, для которого описаны входящие и выходящие операнды и регистры, которые используются во вставке, но явно по операндам этого вычислить нельзя. А потому содержимое ассемблерной вставки никак компилятором не может быть оптимизировано.

Компилятор lcc устроен так, что в большинстве случаев он "видит" содержимое вставки и может его оптимизировать. Но для совместимости с gcc следует писать вставки в предположении, что для компилятора они являются чёрным ящиком.

Если вставка описана как asm volatile (<всё остальное>), то это означает, что компилятор не будет менять местоположение вставки относительно прочих инструкций языка, а так же не будет её удалять, если она находится в недостижимых по исполнению ветках

Списки входных и выходных операндов представляют собой последовательность описателей, разделенных запятой. Описатель представляет собой конструкцию:

`"constraint" (expr)`

`expr` — это выражение языка C, которое есть именующее выражение (`lvalue`) для выходных операндов и выражение для входных**.** Конструкция `expr` должна быть заключена в круглые скобки.

`constraint` представляет собой набор символов, обозначающих типы операндов и способы их модификации. Конструкция `constraint` заключается в кавычки.

В качестве типов операндов допустимо использовать следующие символы:

`r` — регистр;

`g` — то же, что и `r`;

`i` — константа (целочисленная, плавающая, адресная);

`I` — 5-битный литерал.

Для выходных операндов перед символом типа должен стоять символ `"="`, если в операнд идёт только запись, или символ `"+"`, если в операнд идёт запись, но при этом его значение ещё и подаётся на вход ассемблерной вставки. Если внутри ассемблерной вставки после записи в операнд есть ещё и его чтения (так называемый early clobbered operand), то после `"="` или `"+"` нужно ставить символ `"&"`.

Список используемых регистров есть разделённый запятыми список имён регистров, заключённых в кавычки. В качестве регистров могут использоваться только регистры общего назначения.

Ассемблерный код представляет собой заключенную в кавычки строку, содержащую текст на языке ассемблер. Внутри этого текста обращение к входному или выходному операнду идёт посредством конструкций `%N`, где `N = 0,1,2,..` — это номер по объединенному списку операндов. Этот список имеет сквозную нумерацию — сначала индексируются выходные операнды, а затем входные.

Если внутри ассемблерного кода нужно использовать символ процента, то нужно для этого написать %%.

Если для ассемблерной вставки список используемых регистров пуст, его и предшествующий ему символ ":" можно не указывать. Если при этом так же пуст список входных операндов, с ним поступают аналогично.

Примеры:

1) Запись в переменную a числа 1:

```
asm ( "adds 0, 1, %0" : "=r" (a) );
```

Поле constraint "=r" в описателе означает, что выходной операнд ассемблерной вставки будет являться регистром. Это совсем не означает, что переменная a будет на регистре. В случае если a находится в памяти, компилятор выделит регистр для ассемблерной вставки, а потом перепишет результат в память.

2) Запись в переменную a суммы величин b и c:

```
asm ( "adds %1, %2, %0"
      : "=r" (a)
      : "r" (b), "ri" (c) );
```

В качестве первого входного операнда (b) мы разрешаем использовать только регистр, если b представляет собой константу (например, по результатам оптимизаций), то она перед ассемблерной вставки будет перекинута на регистр.

В качестве второго (c) мы разрешаем использовать регистр или константу.

3) Запись величины a*b в глобальный регистр %g3:

```
asm ("adds 0, %0, %%g3"
        :
        : "ri" (a*b)
        : "%g3");
```

Выходных операндов у вставки нет, а потому список пустой.

В качестве входного значения результат умножения будет перекинут на регистр (или использоваться в виде константы, если компилятор что-то оптимизировал).

Вставка изменяет значение регистра %g3, а потому этот регистр пишем в список используемых регистров.

3.4. Задание целевой архитектуры

Компилятор может генерировать код для исполнения на микропроцессорах с различной архитектурой. По умолчанию, компилятор генерирует код, совместимый со всеми вычислительными комплексами с архитектурой «Эльбрус». Если задать опцию —mcpu=elbrus-2c+, компилятор создаст код, исполняемый только на ВК с процессором (процессорами) «Эльбрус-2С+». Если задать опцию −march=<arch>, компилятор создаст код, исполняемый только на ВК с соответствующей архитектурой процессоров, возможные значения <arch> см. ниже.

Следует помнить, что вместе могут быть слинкованы объекты, полученные для какой-то одной архитектуры, и объекты, полученные в совместимом режиме. Смешивать при линковке объекты, полученные для разных архитектур нельзя. Приведем опции режима компиляции целевой архитектуры:

-march=<cpu> — формировать код для архитектуры <arch>. Параметр опции может принимать следующие значения: native, elbrus-v2, elbrus-v3, elbrus-v4, elbrus-v5, elbrus-v6;

-mptr32 — компилировать в режиме 32 бит;

-mptr64 — компилировать в режиме 64 бит (режим по-умолчанию);

-mptr128 — компилировать в защищённом режиме;

3.5. Особенности компиляции в защищенном режиме

Для компиляции программ в защищенном режиме (-mptr128) для ВК на основе процессора 2С+ используется режим статического связывания библиотек -static. Это связано с исправлением ошибки стека данных в режиме динамического связывания для остальных режимов сборки (-mptr32, -mptr64).

4. ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ

4.1. Входной язык

Входными данными для компилятора являются файлы, содержащие текст программы на языках C/C++.

Компилятор поддерживает следующие стандарты:

- язык C поддерживает стандарт ISO/IEC 9899 : 1990 (E);
- язык C++ поддерживает стандарт ISO/IEC 14882 : 1998 (E).

Кроме того, компилятор поддерживает расширение языков C/C++, реализованное в компиляторах семейства GNU (так называемое GNU-расширение). Следует отметить, что в разных версиях GNU-компиляторов расширение языка может отличаться. Полное описание GNU-расширения языка дается в приложении 1. В текущей версии компилятора остается нереализованным только одно расширение — вложенные функции.

4.2. Выходной файл

В качестве выходных данных компилятора, в зависимости от поданных опций, могут быть:

- препроцессированный текст на языках C/C++ (опция `-E`);
- текст на языке ассемблер (опция `-S`);
- файл объектного кода (опция `-c`);
- динамическая библиотека (опция `-shared`);
- исполняемый файл (по умолчанию).

5. СООБЩЕНИЯ

В процессе работы компилятор может выдавать два типа сообщений. Аварийные сообщения информируют о том, что в процессе компиляции были обнаружены ошибки. Источником ошибок может быть как некорректный текст исходной программы, так и неверные параметры запуска Компилятора. В любом случае, работа Компилятора будет аварийно завершена. При обнаружении аварийных сообщений программисту следует устранить причину ошибки и повторить запуск Компилятора.

Информационные сообщения или предупреждения информируют об обнаружении каких-либо нестандартных ситуаций, при этом процесс компиляции не прерывается. При обнаружении предупреждений программист может по своему усмотрению либо принять сообщение к сведению и продолжить работу либо устранить причину нестандартной ситуации и повторить запуск Компилятора.

Все сообщения имеют стандартную форму. Печатается имя файла, номер строки, тип сообщения (ошибка или предупреждение), и текст информационного сообщения. На следующей строке печатается некорректный текст программы.

Пример сообщения об ошибке:

```
    "ooo.c", строка 6: ошибка: идентификатор "retu" не
определен
    retu rn 3.14;
```

Пример предупреждения:

```
    "ooo.c", строка 4: предупреждение: переменная "xxx" была
    определена, но не использована
    int xxx = 1;
```

5.1. Язык сообщения

По умолчанию, все сообщения должны выдаваться по-русски. Однако если выставлена переменная окружения `LC_MESSAGES`, сообщения могут выдаваться на английском языке. Проверить установки можно приказом:

```
/opt/mcst/bin/lcc -print-recognized-environment
```

6. ПРОГРАММНЫЙ ИНТЕРФЕЙС ПРИЛОЖЕНИЙ ДЛЯ ВК «ЭЛЬБРУС»

6.1. Представление данных

Аппаратно поддерживается работа со следующими форматами данных:

- байт (SB) — беззнаковое целое размера 8 разрядов.

- знаковый байт (UB) — целое со знаком размера 8 разрядов (знаковый разряд и 7 значащих разрядов).

- полслова (UH) — беззнаковое целое размера 16 разрядов.

- знаковые полслова (SH) — целое со знаком размера 8 разрядов (знаковый разряд и 15 значащих разрядов).

- слово (UW) — беззнаковое целое размера 32 разряда.

- знаковое слово (SW) — целое со знаком размера 32 разряда (знаковый разряд и 31 значащий разряд).

- двойное слово (UD) — беззнаковое целое размера 64 разряда.

- знаковое двойное слово (SD) — целое со знаком размера 64 разряда (знаковый разряд и 63 значащих разрядов).

- квадро слово (NQ) — числовое значение размера 128 разрядов.

- вещественное число (FW) — вещественное значение в формате IEEE single precision.

- вещественное число удвоенной точности (FD) — вещественное значение в формате IEEE double precision.

- расширенное вещественное число (FX) — вещественное значение в формате IEEE double-extended precision.

- двойной дескриптор (DD) — адресное значение размера 64 разряда.

- квадро дескриптор (DQ) — адресное значение размера 128 разрядов.

Упаковка меньших форматов в большие соответствует little endian. Ниже. для числа `0xF4F3F2F1` проиллюстрировано соответствие нумерации битов и байтов на рис. 1.

Правило упаковки значения

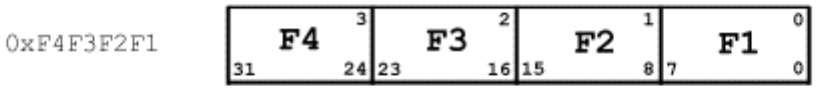

Рис. 1

На рис. 1 указан F1 — младший значащий байт, содержащий младшую часть значения. Цифры сверху показывают порядок байтов, цифры снизу — нумерацию битов.

При работе программы данные располагаются как в памяти, так и на рабочих регистрах. Рабочие регистры имеют следующие форматы:

- регистр (SR) — 32-х разрядный регистр;
- двойной регистр (DR) — 64-х разрядный регистр;
- расширенный регистр (XR) — 80-и разрядный регистр;
- квадро регистр (QR) — 128-и разрядный регистр.

Правило соответствия вложенности и нумерации рабочих регистров продемонстрировано на следующем рис. 2.

Правило соответствия вложенности рабочих регистров

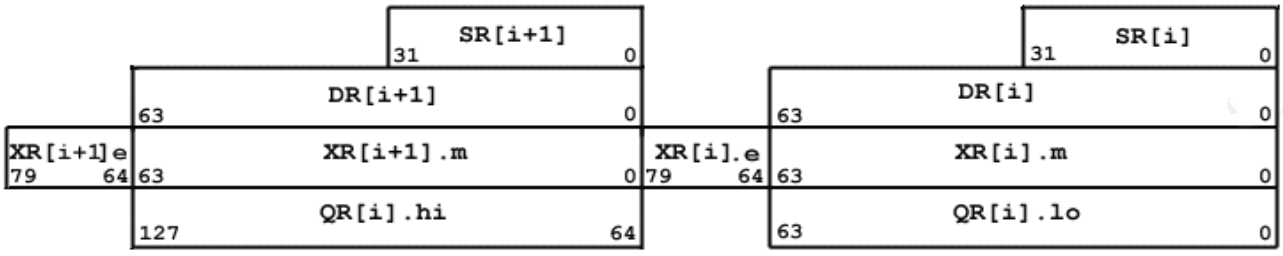

Рис. 2

Где SR[i] — регистр с номером i, под которым подразумевается четное число (I = 2*n). Соответственно, SR[i+1] — регистр со следующим, нечетным номером. XR[i].m — поле расширенного регистра, содержащее 64 младшие разряды

(мантисса). XR[i].e — поле расширенного регистра, содержащее 16 старших разрядов (экспонента). QR[i].lo — младшая часть квадро регистра, QR[i].hi — старшая часть.

Одинарный регистр наложен на двойной с совпадением младших разрядов. Старшая часть регистра операциям, работающим в формате 32 недоступна. Двойные регистры накладываются на квадро регистры таким образом, что регистр с четным номером соответствует младшей половине охватывающего квадро регистра, а регистр с нечетным номером — старшей половине. Старшая часть расширенного регистра не наложена ни на какие регистры, и доступна только подмножеству операций вещественной арифметики расширенной точности. Младшая половина доступна как двойной регистр.

Отображение языковых типов данных на архитектурные форматы, и, соответственно, размеры занимаемых ресурсов, зависят от семантической модели.

6.1.1. Отображение целых типов

Отображение целых типов для режима 32-х разрядной адресации приведено в таблице 3.

Т а б л и ц а 3 — Отображение целых типов режима 32-х разрядной адресации

Тип	Формат представления	Отображение в памяти		Отображение на регистрах
		размер	выравнивание	
char	SB	1	1	SR
signed char				
unsigned char	UB			
short	SH	2	2	
signed short				
unsigned short	UH			
int	SW	4	4	
signed int				
enum				
unsigned int	UW			
long	SW			
unsigned long	UW			
long long	SD	8	8	DR
unsigned long long	UD			
__int128	NQ	16	16	QR

Примечание. Тип `__int128` не имеет аппаратной поддержки реализации арифметических операций.

Отображение целых типов для режима 64-х разрядной адресации и защищенного режима приведено в таблице 4.

Таблица 4 — Отображение целых типов в режиме 64-х разрядной адресации и защищенном режиме

Тип	Формат представления	Отображение в памяти		Отображение на регистрах
		размер	выравнивание	
char	SB	1	1	SR
signed char				
unsigned char	UB			
short	SH	2	2	
signed short				
unsigned short	UH			
int	SW	4	4	
signed int				
enum				
unsigned int	UW			
long	SD	8	8	DR
unsigned long	UD			
long long	SD			
unsigned long long	UD			
__int128	NQ	16	16	QR

Примечание. Тип `__int128` не имеет аппаратной поддержки реализации арифметических операций.

6.1.2. Отображение вещественных типов

Вещественные типы представляются в соответствие со стандартом вещественной арифметики ANSI/IEEE 754-1985. Аппаратно поддерживается работа с тремя форматами: простой формат (single), формат удвоенной точности (double) и расширенный формат (extended). Представление форматов вещественных данных приведено на рис. 3 – 5.

Простой вещественный формат

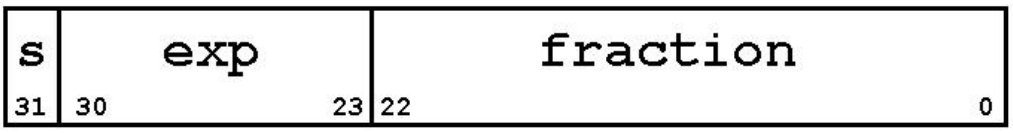

Рис. 3

Вещественный формат удвоенной точности

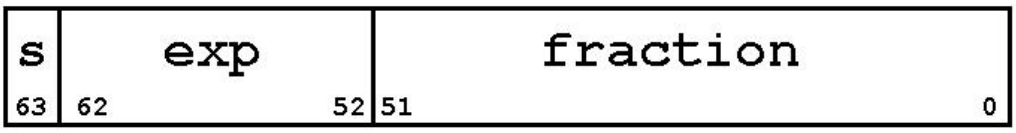

Рис. 4

Расширенный вещественный формат

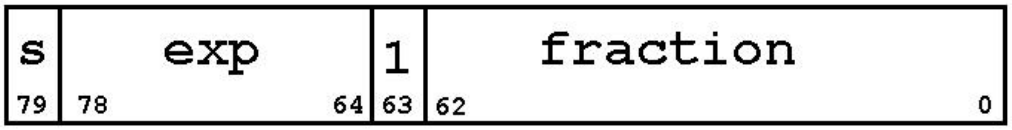

Рис. 5

Отображение вещественных типов для всех режимов приведено в таблице 5.

Т а б л и ц а 5 — Отображение вещественных типов

Тип	Формат представления	Отображение в памяти		Отображение на регистрах
		размер	выравнивание	
float	FW	4	4	SR
double	FD	8	8	DR
__float80	FX	16	16	XR
__float128	NQ	16	16	QR
long double	FX	16	16	XR

П р и м е ч а н и е . Тип `__float128` не имеет аппаратной поддержки реализации арифметических операций.

6.1.3. Отображение указательных типов

Отображение указательных типов для режима 32-х разрядной адресации приведено в таблице 6.

Т а б л и ц а 6 — Отображение указательных типов для режима 32-х разрядной адресации

Тип	Формат представления	Отображение в памяти		Отображение на регистрах
		размер	выравнивание	
any_type*	UW	4	4	SR
any_type(*)()				

Отображение указательных типов для режима 64-х разрядной адресации приведено в таблице 7.

Т а б л и ц а 7 — Отображение указательных типов для режима 64-х разрядной адресации

Тип	Формат представления	Отображение в памяти		Отображение на регистрах
		размер	выравнивание	
any_type*	UD	8	8	DR
any_type(*)()				

Отображение указательных типов для защищенного режима приведено в таблице 8.

Т а б л и ц а 8 — Отображение указательных типов для защищенного режима

Тип	Формат представления	Отображение в памяти		Отображение на регистрах
		размер	выравнивание	
any_type*	DQ	16	16	QR
any_type(*)()	DD	8	8	DR

6.1.4. Агрегатные типы

Агрегатные типы включают в себя структуры (struct), объединения (union), классы (class) и массивы. Под структурами и объединениями понимаются типы

в нотации языка C struct и union соответственно. Под классом понимаются типы языка C++ class, struct и union, которые имеют свойства, не имеющие аналогов в типах struct и union языка C.

Выравнивание объектов агрегатных типов должно соответствовать выравниванию их наиболее строго выровненных компонентов. Размер объекта агрегатного типа должен быть кратен выравниванию наиболее строго выровненного компонента. Для структур и объединений это может потребовать расширения размера пустыми полями (паддинг). Значение полей паддинга не определено.

8.1.4.1. Тип массива.

Выравнивание объекта типа массив определяется выравниванием элемента этого массива. Размер объекта типа массив равно размеру элемента массива, умноженному на число элементов массива.

8.1.4.2. Тип структуры.

Выравнивание и размер объекта типа структуры определяются правилами упаковки членов (полей) этой структуры.

Правила упаковки полей для структурных типов:

- объект типа структуры должен иметь выравнивание не хуже выравнивания наиболее строго выровненного компонента;
- поля упаковываются в структуру по порядку так, что очередное поле получает наименьшее возможное смещение от начала структуры, удовлетворяющее его выравниванию. Это может привести к возникновению внутреннего паддинга, когда требуется пропустить место для размещения очередного компонента, если текущее смещение не соответствует требуемому выравниванию;
- размер структуры должен быть увеличен, если суммарный размер всех полей, включая внутренний паддинг, не соответствует кратности выравнивания. Это приводит к возникновению хвостового паддинга.

Нижеприведенные иллюстрации поясняют действие правил упаковки полей для структур (с рис. 6 по рис. 9).

Простая маленькая структура

```
struct {
    char   c;
};
```

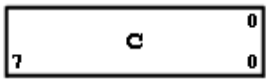

Рис. 6

Размер структуры — 1 байт. Выравнивание— 1 байт (не требуется).

Плотно упакованная структура без паддинга

```
struct {
    char    c;
    char    d;
    short   s;
    int     i;
};
```

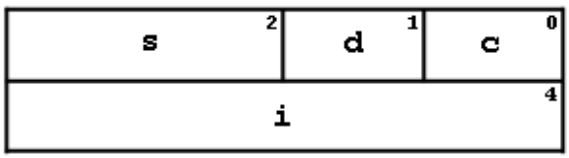

Рис. 7

Размер структуры — 8 байт. Выравнивание определяет поле `i` — 4 байта.

Структура с внутренним паддингом

```
struct {
    char    c;
    short   s;
};
```

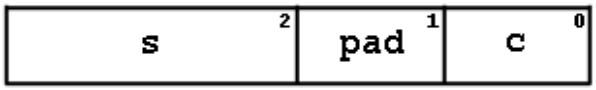

Рис. 8

Размер структуры — 4 байта. Выравнивание определяет поле `s`— 2 байта. Поле `s` не может быть размещено сразу после поля `c`, поскольку это не соответствует его выравниванию. Поэтому в байте 1 возникает поле паддинга.

Структура с внутренним и хвостовым паддингами

```
struct {
    char    c;
    double  d;
    short   s;
};
```

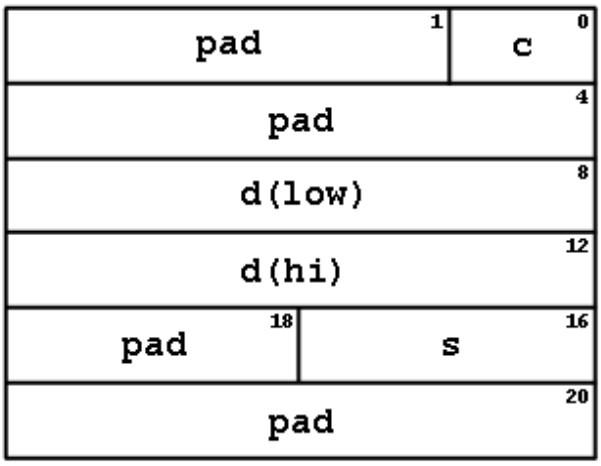

Рис. 9

Размер структуры — 24 байта. Выравнивание определяет поле `d` — 8 байт. Между полем `c` и полем `d`, в байтах с 1 по 7 из-за требования выравнивания поля `d` появляется внутренний паддинг. После распределения последнего поля `s`, размер структуры равен 18 байт, что не кратно выравниванию в 8 байт. Поэтому размер структуры увеличен до 24 байт. В байтах с 18 по 23 находится поле хвостового паддинга.

8.1.4.3. Тип объединение.

Выравнивание и размер объекта типа объединение определяются правилами упаковки членов (полей) типа объединение.

Правила упаковки полей для типа объединение:

- выравнивание объекта типа объединение должно быть не хуже, чем выравнивание поля наиболее строгим выравниванием;

- поля упаковываются в объединение так, что начала всех полей совпадают и равны началу объединения;

- размер объекта типа объединения должен быть не меньше размера максимального поля и кратен выравниванию. Если размер максимального поля

не кратен выравниванию, то размер объединения увеличивается добавлением поля хвостового паддинга;

Рис. с 11 по 13 иллюстрируют правила упаковки полей для объединений.

Объединение

```
union {
   char   c;
   shopt  s;
   int    i;
};
```

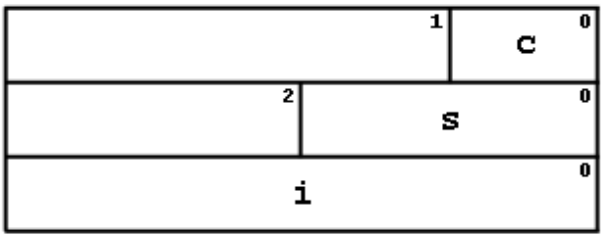

Рис. 10

Размер объединения — 4 байта. Выравнивание — 4 байта.

Объединение с хвостовым паддингом

```
union {
   short   s;
   char    c[3];
};
```

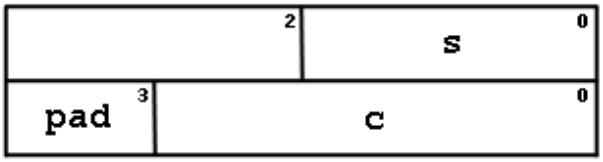

Рис. 11

Размер объединения — 4 байта. Выравнивание — 2 байта. Размер максимального поля — 3 байта, что не кратно выравниванию. Поэтому размер объединения увеличен добавлением в 3 байте поля паддинга.

8.1.4.4. Тип класс.

Для классов действуют такие же правила, как и для структур.

8.1.4.5. Битовые поля.

Битовые поля суть члены объекта типа структуры, класса или объединения с заданным в виде числа разрядов размером. Битовые поля определяются базовым типом и числом разрядов. Число разрядов не может быть больше, чем число разрядов в базовом типе. Базовым типом может быть любой целочисленный тип знаковой или беззнаковой модификации. Область памяти определенная базовым типом, в которой располагается битовое поле, называется контейнером.

Битовые поля подчиняются тем же правилам упаковки, что и не битовые поля с некоторыми добавлениями:

- битовые поля справа налево от менее значащих разрядов к более значащим;

- контейнер битового поля должен быть размещен в соответствие с правилами размещения базового типа;

- в контейнере битового поля могут размещаться другие, в том числе и не битовые поля;

- неименованные битовые поля не участвуют в определении общего выравнивания объекта;

- неименованные битовые поля ненулевой длины используются для явного задания паддинга;

- неименованные битовые поля нулевой длины используются для принудительного выравнивания последующего поля на границу, соответствующую базовому типу этого битового поля;

С рис. 12 по рис. 14 иллюстрируются правила упаковки битовых полей.

Структура с битовыми полями

```
struct {
    int i:5;
    int j:6;
    int k:7;
};
```

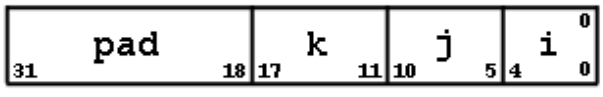

Рис. 12

Размер структуры — 4 байта. Выравнивание — 4 байта определяется базовым типом полей, который для всех один. Структура дополняется хвостовым паддингом для того, чтобы размер структуры удовлетворял кратности выравнивания.

Выравнивание в структуре с битовыми полями

```
struct {
    short     s:9;
    long long l:9;
    char      c;
    short     t:9;
    short     u:9;
    char      d;
};
```

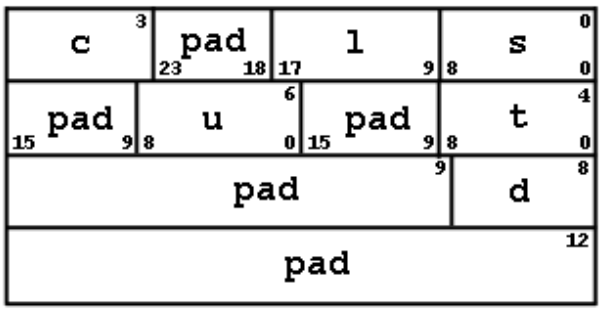

Рис. 13

Размер структуры — 16 байт. Выравнивание — 8 байт. Выравнивание определяет базовый тип битового поля `l` . Контейнер для этого поля может быть размещен начиная с нулевого байта. Но в этот контейнер уже попадает предыдущее поле `s`. Следующее поле `c` требует выравнивание на 1 байт. Следовательно, оно может быть размещено, только начиная с 24 бита. В битах с 18 по 23 остается поле внутреннего паддинга. По этой же причине возникает внутренний паддинг между полями `u` и `d`. Между полями `t` и `u` причина возникновения паддинга следующая. Если контейнер для битового поля `u` разместить, начиная с 4 байта, и поле начать размещать сразу после поля `t`, то оно не поместится в отведенный контейнер. Поэтому контейнер для этого поля размещается со следующей границы его выравнивания. После распределения всех полей размер структуры не соответствует кратности выравнивания, что требует добавления хвостового паддинга.

Структура с неименованными битовыми полями нулевой длины

```
struct {
    char   c;
    int     :0;
    char   d;
    short   :9;
    char   e;
    char    :0;
};
```

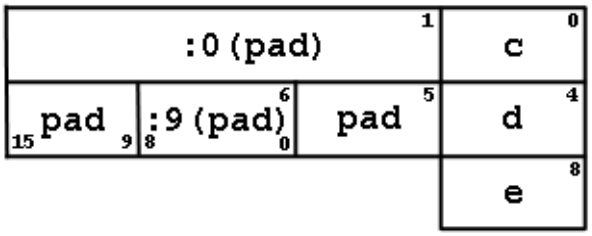

Рис. 14

Размер структуры — 9 байт. Выравнивание — 1 байт. Неименованные битовые поля не участвуют в определении выравнивания. Все остальные поля имеют тип требующий выравнивания на 1 байт. Неименованное битовое поле нулевой длины перед полем `d` требует выравнивания поля `d` в соответствие со своим базовым типом `int`, что приводит к возникновению паддинга с 1 по 3 байты. Неименованное битовое длины 9 не может быть размещено сразу после поля `d`, ибо если контейнер разместить, начиная с 4 бита, поле в него не поместится. Следующее удовлетворяющее выравниванию размещение контейнера для этого битового поля — с 6 байта. Поэтому возникает внутренний паддинг в 5 байте. Само по себе это битовое поле тоже приводит к внутреннему паддингу в 9 битов.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

МН	Машинный носитель
СП	Система программирования
ANSI	American National Standards Institute
GCC	GNU Compiler Collection
GNU	GNU's Not UNIX
LCC	Local C Compiler