УТВЕРЖДЕН

КПДА.96901-01 33 01-ЛУ

КОМПЛЕКТ РАЗРАБОТЧИКА ДЛЯ ЗОСРВ «НЕЙТРИНО»

Руководство программиста

КПДА.96901-01 33 01

Листов 15

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл.	

2020

СОДЕРЖАНИЕ

СПИСОК ИСПОЛЬЗУЕМЫХ СОКРАЩЕНИЙ

ЗОСРВ – защищенная операционная система реального времени

ELF – Executable and Linkable Format

1. НАЗНАЧЕНИЕ И УСЛОВИЯ ПРИМЕНЕНИЯ

1.1. Назначение и функции программы

Программа предназначена для компиляции и компоновки программ, написанных на языках Си и C++, предназначенных для функционирования в среде исполнения ЗОСРВ «Нейтрино».

Основные функции, выполняемые программой:

1) Препроцессорная обработка программ, обеспечивающая выполнение директив препроцессора C/C++;

2) Генерация кодов Ассемблера, обеспечивающая преобразование (компиляцию) текста программы, написанной на языках C/C++, в платформо-зависимые коды Ассемблера;

3) Генерация объектного кода, обеспечивающая преобразование (компиляцию) кодов Ассемблера в объектный код соответствующей микропроцессорной архитектуры;

4) Компоновка (редактирование связей), обеспечивающая создание исполняемых модулей в формате ELF.

5) Символьная отладка прикладных программ для ЗОСРВ «Нейтрино».

1.2. Условия для выполнения программы

Условия для выполнения программы указаны в разделе 2 «Условия применения» документа «Комплект разработчика для ЗОСРВ «Нейтрино». Описание применения. Часть 1» КПДА.96901-01 31 01.

Перед использованием программы необходимо выполнить установку и проверку корректности установки из дистрибутива в соответствии с инструкцией, поставляемой в составе дистрибутива.

2. ХАРАКТЕРИСТИКИ ПРОГРАММЫ

Программа включает модифицированные для работы с двоичным форматом объектных кодов ЗОСРВ «Нейтрино» программные компоненты из состава следующих продуктов с открытым исходным текстом:

1) Версии GCC – 4.8.3, 8.3.0;
2) Версия GDB – 7.6.1;
3) Версии GNU Binutils – 2.24, 2.32.

Сведения о поддерживаемых целевых архитектурах ЗОСРВ «Нейтрино», для которых поддерживается генерация и компоновка объектного кода, приведены в разделе 3 «Состав инструментальных средств» документа «Комплект разработчика для ЗОСРВ «Нейтрино». Описание применения. Часть 1» КПДА.96901-01 31 01.

3. ОБРАЩЕНИЕ К ПРОГРАММЕ

3.1. Общие принципы

Для того чтобы создать собственную программу, необходимо сформировать ее исходный текст на языке C или C++. Файл или файлы с исходным текстом сначала обрабатываются препроцессором, затем результат обрабатывается соответственно компилятором C или C++. В результате получается объектный файл. В этот объектный файл компоновщик (редактор связей или "линкер") включает объектные файлы из библиотек и в результате получается исполняемый файл — программа.

Итак, для получения приложения исходный текст обрабатывается следующим конвейером:

Препроцессор — Компилятор — Компоновщик

Эту цепочку удобнее всего вызывать командой kcc. Удобство обеспечивается тем, что у этой команды есть конфигурационные файлы, задающие по умолчанию оптимальную комбинацию опций для каждого из инструментов конвейера и каждой целевой аппаратной архитектуры.

Для примера рассмотрим классическую программу, выводящую на терминал фразу "Здравствуй, Нейтрино!". Для этого создадим текстовый файл с именем hello.c, например:

```
$ cd /tmp
$ touch hello.c
```

Наполним файл hello.c текстом на языке C при помощи любого текстового редактора как показано в листинге:

```
#include <stdio.h>
main()
{
    printf("Здравствуй, Нейтрино!\n");
}
```

Функция main() — единственная обязательная функция программы, написанной на языке C или C++. Она является точкой входа в первый поток процесса (исполняющейся на ЭВМ программы), т. е. местом, с которого программа начинает выполняться. Остальные функции, в том числе функции создания новых потоков процесса, вызываются уже изнутри функции main().

Строго говоря, директива препроцессора #include <stdio.h> не требуется, поскольку компилятор применяет ее автоматически, потому что создание программы на языке Си без заголовочного файла stdio.h не возможно.

Для того чтобы получить программу hello для целевой системы с архитектурой Intel x86, следует выполнить команду:

```
kcc -Vgcc_ntox86 hello.c -o hello
```

Получить полный список целевых платформ, для которых можно сгенерировать приложение, можно командой

```
kcc -V
```

Например, gcc_ntoppcbe — для целевых систем PowerPC BigEndian, а gcc_ntomipsle — для MIPS LittleEndian.

В результате будет получена исполняемая программа с именем hello. Её необходимо любым способом скопировать в целевую систему ЗОСРВ «Нейтрино». Например, для этого могут быть использованы сетевые средства (NFS, CIFS, SFTP, FTP) или съемный физический носитель.

После копирования в целевую систему ЗОСРВ «Нейтрино» необходимо убедиться в наличии у неё выставленных битов разрешения исполнения и, при необходимости, выставить эти биты. Например:

```
$ chmod a+x hello
```

Запустим программу. Для этого нужно либо, чтобы путь к ней содержался в переменной окружения PATH, либо необходимо указать ее относительное или абсолютное путевое имя:

```
./hello
```

Каталог "точка" (`.`) — это текущий каталог, т. е. имя `./hello` обозначает файл `hello` в текущем каталоге.

Примечание

Можно добавить в переменную `PATH` текущий каталог, чтобы командный интерпретатор всегда сначала искал исполняемый файл в нем и только потом искал в остальных каталогах:

```
PATH=.:$PATH
```

Если данную команду поместить в файл `$HOME/.profile`, то она будет автоматически выполняться при каждом запуске интерпретатора.

3.2. Использование утилиты make

Компиляцию и сборку приложения, состоящего из нескольких файлов, удобно выполнять с использованием программы make.

Рассмотрим пример. Пусть проект включает следующие файлы:

- aaa.c;
- bbb.c;
- ccc.c;
- defs.h.

Пусть один из файлов с расширением «.c» содержит функцию mail(), остальные файлы с расширением «.c» содержат вспомогательные функции, а файл defs.h содержит объявления вспомогательных функций и необходимые макроопределения.

Для того чтобы из этих четырех файлов получить программу **program**, следует выполнить следующие команды:

```
kcc –c aaa.c
kcc –c bbb.c
kcc –c ccc.c
kcc –o program aaa.o bbb.o ccc.o
```

Выполнение такой цепочки команд называют *сборкой проекта*.

Программа make дает возможность написать конфигурационный файл, описывающий зависимости между файлами проекта и то, какую последовательность команд надо выполнить, чтобы пересобрать проект при внесении изменений в отдельные файлы исходных текстов, не затрагивая при этом те файлы, на которые изменения не повлияли. Таким файлом управления сборкой проекта является `Makefile`, а анализируется этот файл утилитой **make**. Точнее, при запуске утилита **make** ищет сначала файл с именем `makefile`, затем, если не нашла, — файл с именем `Makefile`. В принципе, можно задать любое имя, указав его утилите **make** с помощью опции **–f**.

Примечание

Интегрированные среды разработки также используют утилиту **make**, скрывая это от программиста за графическим интерфейсом.

Простейший файл `Makefile` содержит только два типа синтаксических конструкций — цели и макросы.

Цель — это имя файла, который следует сгенерировать. Описание цели имеет такой вид:

```
ЦЕЛЬ: ЗАВИСИМОСТЬ-1 ЗАВИСИМОСТЬ-2 ... ЗАВИСИМОСТЬ-N
      КОМАНДА-1
      КОМАНДА-1
      ...
```

КОМАНДА-N

ЦЕЛЬ-*i* — непустой список файлов, которые предполагается создать. ЗАВИСИМОСТЬ-*i* — список файлов, из которых строится цель. В качестве зависимости может использоваться другая цель. Имя цели и список зависимостей записываются в одну строку и разделяются двоеточием. Они составляют *заголовок цели*. Ниже следует список *команд*, каждая команда — в отдельной строке.

Примечание

Каждая команда ОБЯЗАТЕЛЬНО начинается с символа табуляции.

Утилиту **make** можно запускать или с указанием имени конкретной цели, или без имени цели. Если цель не указана, то **make** строит первую цель, заданную в файле Makefile, имя которой не начинается с точки. Построение цели заключается в выполнении команд, заданных для цели. Перед тем как строить цель, **make** сравнивает время последней модификации цели и зависимостей. Если какая-либо из зависимостей была изменена после последней сборки цели, то цель пересобирается. Если какая-то зависимость сама является целью (т. е. "подцелью"), то сначала собирается "подцель".

Итак, с учетом сказанного, напишем файл управления сборкой нашего проекта с именем mf1:

```
program: aaa.o bbb.o ccc.o
      kcc -o program aaa.o bbb.o ccc.o
aaa.o : aaa.c defs.h
      kcc –c aaa.c

bbb.o : bbb.c defs.h
      kcc –c bbb.c

ccc.o : ccc.c defs.h
      kcc –c ccc.c
```

Выполним сборку:

make –f mf1

Примечание

Если исходный код программы **program** не изменялся со времени последней компиляции, утилита **make** не будет выполнять сборку, а выведет сообщение:

```
make: 'program' is up to date
```

В рассматриваемом примере, поскольку утилита **make** вызвана без указания цели, будет вызвана первая цель — program. Для цели program заданы три зависимости, каждая из которых является подцелью. Утилита **make** сначала соберет подцели, а затем и цель. Можно задавать произвольную цель, например:

```
make –f mf1 ccc.o
```

В этом случае будет собрана только цель ccc.o, т. к. среди ее зависимостей нет подцелей.

На самом деле утилита **make** знает, что если в качестве цели задан объектный файл (т. е. файл с расширением .o) и существует одноименный С-файл, то необходимо вызывать компилятор с опцией **–c**. Следующий листинг (файл mf2) показывает более краткую запись файла построения:

```
program: aaa.o bbb.o ccc.o
      kcc -o program aaa.o bbb.o ccc.o

aaa.o bbb.o ccc.o: defs.h
```

Обратите внимание, что в данном листинге один и тот же список объектных файлов в трех строчках записан три раза. Чтобы этого не делать, **make** поддерживает макросы, имеющие следующий синтаксис:

ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ

ЗНАЧЕНИЕ может являться произвольной последовательностью символов, включая пробелы и обращения к значениям ранее определенных переменных. После

объявления переменной мы можем обратиться к ней так: `$(ПЕРЕМЕННАЯ)`. Запишем файл построения `mf3` с учетом указанных возможностей make:

```
OBJS=aaa.o bbb.o ccc.o
program: $(OBJS)
      kcc -o program $(OBJS)
$(OBJS): defs.h
```

Некоторые переменные уже предопределены. К ним относятся, например:

- `AR=ar` — программа-архиватор;
- `AS=as` — ассемблер.
- `CC=cc` — компилятор С(отсутствует в КР)
- `CXX=g++` — компилятор С++;
- `CPP=cpp` — препроцессор;
- `LEX=lex` — лексический анализатор С-программ;
- `RM=rm -f` — команда удаления файлов.

Есть также специальные встроенные макросы, значение которых изменяется в процессе выполнения утилиты **make**:

- **$@** — имя текущей цели;
- **$?** — список зависимостей, более свежих, чем цель;
- **$^** — полный список зависимостей для данной цели;
- **$*** — имя цели без расширения.

Примечание

Полный перечень всех предопределенных макропеременных и зависимостей можно получить с помощью команды:

make –p </dev/null 2>/dev/null

Добавим служебные переменные в рассматриваемый пример (получим файл `mf4`):

```
OBJS=aaa.o bbb.o ccc.o
program: $(OBJS)
      $(CC) -o $@ $(OBJS)
$(OBJS): defs.h
```

Для удобства и корректности, необходимо добавить служебную цель, позволяющую удалять результаты компиляции. Такая цель имеет общепринятое название — `clean` (получим файл `mf5`):

```
OBJS=aaa.o bbb.o ccc.o
program: $(OBJS)
      $(CC) -o $@ $(OBJS)
$(OBJS): defs.h
clean:
      $(RM) program $(OBJS)
```

Иногда добавляют и другие цели, например, `install`.

Поскольку имена целей задаются фактически произвольно, могут возникать некорректные совпадения имен целей и имен файлов проекта. Для обработки возникающих при этом конфликтов предназначена специальная цель `.PHONY`, в качестве зависимостей которой указывают имена служебных целей. Проиллюстрируем это в файле `mf6`:

```
OBJS=aaa.o bbb.o ccc.o
 .PHONY: clean
program: $(OBJS)
      $(CC) -o $@ $(OBJS)
$(OBJS): defs.h
clean:
      $(RM) program $(OBJS)
```

В файле Makefile могут присутствовать комментарии. Началом комментария считается символ #, а окончанием — конец строки.

4. ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ

4.1. Входные данные

Входные данные должны соответствовать следующим требованиям:

1) Входные данные на языке Си – стандарту ISO/IEC 9899:2011 «Information technology – Programming languages – C».
2) Входные данные на языке C++ – стандарту ISO/IEC 14882:2014 «Information technology – Programming languages – C++».

При разработке приложений для ЗОСРВ «Нейтрино» на языке Си рекомендуется дополнительно выполнять требования стандарта MISRA C:2012 “Guidelines for the use of the C language in critical systems”.

При разработке приложений для ЗОСРВ «Нейтрино» на языке Си рекомендуется дополнительно выполнять требования следующих стандартов:

1) MISRA C++:2008 «Guidelines for the use of the C++ language in critical systems».
2) AUTOSAR «Guidelines for the use of the C++14 language in critical and safety-related systems».

4.2. Выходные данные

Выходные данные компиляции представляют собой файлы, которые имеют формат 32-битного ELF и соответствуют стандарту комитета Tool Interface Standard (TIS) «Executable and Linking Format (ELF). Specification Version 1.2». Данные файлы могут использоваться в среде исполнения ЗОСРВ «Нейтрино» на ЭВМ с соответствующей процессорной архитектурой.

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц) в докум.	№ документа	Входящий № сопроводительного докум. и дата	Подпись	Дата
Изм.	Измененных	Замененных	новых	аннулированных					