


Азбука халтурщика-ARMатурщика




учебный курс по микроконтроллерам Cortex-Mx:
Миландр 1986BE, STM32F, LPC21xx

(сорупаста) Понятов Д.А. <dponyatov@gmail.com>, ИКП СГАУ

14 июля 2014 г.

Оглавление


I	Обзор семейства микроконтроллеров Cortex-Mx	2
1	Архитектура ARM	3
II	Программное обеспечение	12
2	Рабочая среда разработчика встраиваемых систем	13
	LP1: Установка Debian GNU/Linux	16
	LP2: Установка Git	17
	LP3: Установка GNU toolchain	19
	LP4: Установка утилит GnuWin32	22
	LP5: Установка MinGW	23
	LP6: Редактирование системной переменной Windows \$PATH	23
	LP7: Установка Java	26
	LP8: Установка IDE  ECLIPSE	27
	LP9: Установка симулятора QEMU	32
	LP10: Установка системы верстки документации L^AT_EX	33

3	Первые шаги	35
	LP11: Создание нового проекта в  ECLIPSE	36
	LP12: Создание Makefile	37
	LP13: Hello World	44
	LP14: Настройка отладчика в  ECLIPSE	49
4	Система управления версиями Git	57
5	Интегрированная среда разработки ECLIPSE	59
6	Пакет кросс-компиляции GNU toolchain	61
III	Отладка	62
IV	Встраиваемый C^{++}	63
V	RTOS	64
VI	Автоматное программирование /фреймворк QuantumLeaps/	65
VII	Разработка и изготовление железа	66
7	САПР KiCAD	67



8	Инструмент и оборудование	68
9	Технологии изготовления плат и монтажа	69
VIII Подготовка документации		70
10	DocBook	71
11	Л ^A T _E X	72
Литература		74

Лабораторные работы

LP1: Установка Debian GNU/Linux	16
Установка ПО	16
LP2: Установка Git	17
LP3: Установка GNU toolchain	19
LP4: Установка утилит GnuWin32	22

ЛР5: Установка MinGW	23
ЛР6: Редактирование системной переменной Windows \$PATH	23
ЛР7: Установка Java	26
ЛР8: Установка IDE  ECLIPSE	27
ЛР9: Установка симулятора QEMU	32
ЛР10: Установка системы верстки документации L ^A T _E X	33

Первые шаги 35

ЛР11: Создание нового проекта в  ECLIPSE	36
ЛР12: Создание Makefile	37
ЛР13: Hello World	44
ЛР14: Настройка отладчика в  ECLIPSE	49

Часть I

Обзор семейства микроконтроллеров Cortex-Mx

Глава 1

Архитектура ARM

¹

Архитектура ARM (Advanced RISC Machine, Acorn RISC Machine, усовершенствованная RISC-машина) — семейство лицензируемых 32-битных и 64-битных микропроцессорных ядер разработки компании ARM Limited.

Среди лицензиатов: практически все заметные разработчики цифровых электронных компонентов. Многие лицензиаты делают собственные версии ядер на базе ARM.

Значимые семейства процессоров: ARM7, ARM9, ARM11 и Cortex.

В 2007 году около 98% из более чем миллиарда мобильных телефонов, продаваемых ежегодно, были оснащены по крайней мере одним процессором ARM. По состоянию на 2009 на процессоры ARM приходилось до 90% всех встроенных 32-разрядных процессоров. Процессоры ARM широко используются в потребительской электронике — в том числе КПК, мобильных телефонах, цифровых носителях и плеерах, портативных игровых консолях, калькуляторах и компьютерных периферийных устройствах, таких как

¹копиюнаста: <http://ru.wikipedia.org/wiki/ARM>

жесткие диски или маршрутизаторы.

Эти процессоры имеют низкое энергопотребление, поэтому находят широкое применение во встраиваемых системах и преобладают на рынке мобильных устройств, для которых данный фактор немаловажен.

В настоящее время значимыми являются несколько семейств процессоров ARM:

- ARM7 (с тактовой частотой до 60-72 МГц), предназначенные, например, для недорогих мобильных телефонов и встраиваемых решений средней производительности. В настоящее время активно вытесняется новым семейством Cortex.
- ARM9, ARM11 (с частотами до 1 ГГц) для продвинутых телефонов, карманных компьютеров и встраиваемых решений высокой производительности.
- Cortex A — новое семейство процессоров на смену ARM9 и ARM11.
- Cortex M — новое семейство процессоров на смену ARM7, также призванное занять новую для ARM нишу встраиваемых решений низкой производительности. В семействе присутствуют три значимых ядра: Cortex-M0, Cortex-M3 и Cortex-M4.

Архитектура Существует спецификация архитектуры ARM Cortex, которая разграничивает все типы опций, которые поддерживает ARM, так как детали реализации каждого типа процессора могут отличаться. Архитектура развивалась с течением времени, и начиная с ARMv7 были определены 3 профиля:

- A (application) для устройств, требующих высокой производительности (смартфоны, планшеты)
- R (real time) для приложений, работающих в реальном времени,
- M (microcontroller) для микроконтроллеров и недорогих встраиваемых устройств.

Режимы процессора Процессор может находиться в одном из следующих рабочих режимов:

- User mode — обычный режим выполнения программ. В этом режиме выполняется большинство программ.
- Fast Interrupt (FIQ) — режим быстрого прерывания (меньшее время срабатывания)
- Interrupt (IRQ) — основной режим прерывания.
- System mode — защищённый режим для использования операционной системой.
- Abort mode — режим, в который процессор переходит при возникновении ошибки доступа к памяти (доступ к данным или к инструкции на этапе prefetch конвейера).
- Supervisor mode — привилегированный пользовательский режим.
- Undefined mode — режим, в который процессор входит при попытке выполнить неизвестную ему инструкцию.

Переключение режима процессора происходит при возникновении соответствующего исключения, или же модификацией регистра статуса.

Набор команд ARM Режим, в котором исполняется 32-битный набор команд.

Набор команд Thumb Для улучшения плотности кода процессоры, начиная с ARM7TDMI, снабжены режимом Thumb. В этом режиме процессор выполняет альтернативный набор 16-битных команд. Большинство из этих 16-разрядных команд переводятся в нормальные команды ARM. Уменьшение длины команды достигается за счет сокрытия некоторых операндов и ограничения возможностей адресации по сравнению с режимом полного набора команд ARM.

В режиме Thumb меньшие коды операций обладают меньшей функциональностью. Например, только ветвления могут быть условными, и многие коды операций имеют ограничение на доступ только к половине главных регистров процессора. Более короткие коды операций в целом дают большую плотность

кода, хотя некоторые операции требуют дополнительных команд. В ситуациях, когда порт памяти или ширина шины ограничены 16 битами, более короткие коды операций режима Thumb становятся гораздо производительнее по сравнению с обычным 32-битным ARM кодом, так как меньший программный код придется загружать в процессор при ограниченной пропускной способности памяти.

Аппаратные средства типа Game Boy Advance, как правило, имеют небольшой объём оперативной памяти доступной с полным 32-битным информационным каналом. Но большинство операций выполняется через 16-битный или более узкий информационный канал. В этом случае имеет смысл использовать Thumb код и вручную оптимизировать некоторые тяжелые участки кода, используя переключение в режим ARM.

Набор команд Thumb2 Thumb2 — технология, стартовавшая с ARM1156 core, анонсированного в 2003 году. Он расширяет ограниченный 16-битный набор команд Thumb дополнительными 32-битными командами, чтобы задать набору команд дополнительную ширину. Цель Thumb2 — достичь плотности кода как у Thumb, и производительности как у набора команд ARM на 32 битах. Можно сказать, что в ARMv7 эта цель была достигнута.

Thumb2 расширяет как команды ARM, так и команды Thumb ещё большим количеством команд, включая управление битовым полем, табличное ветвление, условное исполнение. Новый язык «Unified Assembly Language» (UAL) поддерживает создание команд как для ARM, так и для Thumb из одного и того же исходного кода. Версии Thumb на ARMv7 выглядят как код ARM. Это требует осторожности и использования новой команды if-then, которая поддерживает исполнение до 4 последовательных команд испытываемого состояния. Во время компиляции в ARM код она игнорируется, но во время компиляции в код Thumb2 генерирует команды.

Набор команд Jazelle Jazelle — это технология, которая позволяет байткоду Java исполняться прямо в архитектуре ARM в качестве 3-го состояния исполнения (и набора команд) наряду с обычными командами ARM и режимом Thumb. Поддержка технологии Jazelle обозначается буквой «J» в названии процессора — например, ARMv5TEJ. Данная технология поддерживается начиная с архитектуры ARMv6, хотя новые ядра содержат лишь ограниченные реализации, которые не поддерживают аппаратного ускорения.

ARMv8 и набор команд ARM 64 бит В конце 2011 года была опубликована новая версия архитектуры, ARMv8. В ней появилось определение архитектуры AArch64, в которой исполняется 64-битный набор команд A64. Поддержка 32-битных команд получила название A32 и исполняется на архитектурах AArch32. Инструкции Thumb поддерживаются в режиме T32, только при использовании 32-битных архитектур. Допускается исполнение 32-битных приложений в 64-битной ОС, и запуск виртуализированной 32-битной ОС при помощи 64-битного гипервизора.[47] Applied Micro, AMD, Broadcom, Calxeda, HiSilicon, Samsung, STM и другие заявили о планах по использованию ARMv8. Ядра Cortex-A53 и Cortex-A57, поддерживающие ARMv8, были представлены компанией ARM 30 октября 2012 года.[48]

Как AArch32, так и AArch64, поддерживают VFPv3, VFPv4 и advanced SIMD (NEON). Также добавлены криптографические инструкции для работы с AES, SHA-1 и SHA-256.

Условное исполнение Одним из существенных отличий архитектуры ARM от других архитектур ЦПУ является так называемая предикация — возможность условного исполнения команд. Под «условным исполнением» здесь понимается то, что команда будет выполнена или проигнорирована в зависимости от текущего состояния флагов состояния процессора.

В то время как для других архитектур таким свойством, как правило, обладают только команды условных переходов, в архитектуру ARM была заложена возможность условного исполнения практически любой команды. Это было достигнуто добавлением в коды их инструкций особого 4-битового поля (предиката). Одно из его значений зарезервировано на то, что инструкция должна быть выполнена безусловно, а остальные кодируют то или иное сочетание условий (флагов). С одной стороны, с учётом ограниченности общей длины инструкции, это сократило число бит, доступных для кодирования смещения в командах обращения к памяти, но с другой — позволило избавляться от инструкций ветвления при генерации кода для небольших if-блоков.

Пример, обычно рассматриваемый для иллюстрации — основанный на вычитании алгоритм Евклида. В языке C он выглядит так:

алгоритм Евклида

```
1 while ( i != j ) {
```

```

2     if (i > j)
3         i -= j;
4     else
5         j -= i;
6 }

```

А на ассемблере ARM — так:

```

1 loop CMP Ri, Rj; set condition "NE" if (i != j),
2           ; "GT" if (i > j),
3           ; or "LT" if (i < j)
4     SUBGT Ri, Ri, Rj ; if "GT" (greater than), i = i-j;
5     SUBLT Rj, Rj, Ri ; if "LT" (less than), j = j-i;
6     BNE loop ; if "NE" (not equal), then loop

```

Из кода видно, что использование предикации позволило полностью избежать ветвления в операторах `else` и `then`. Заметим, что если `Ri` и `Rj` равны, то ни одна из `SUB` инструкций не будет выполнена, полностью убирая необходимость в ветке, реализующей проверку `while` при каждом начале цикла, что могло быть реализовано, например, при помощи инструкции `SUBLE` (меньше либо равно).

Один из способов, которым уплотнённый (Thumb) код достигает большей экономии объёма — это именно удаление 4-битового предиката из всех инструкций, кроме ветвлений.

Другие особенности Другая особенность набора команд это возможность соединять сдвиги и вращения в инструкции «обработки информации» (арифметическую, логическую, движение регистр-регистр) так, что, например выражение `C:`

```

1 a += (j << 2);

```

может быть преобразовано в команду из одного слова и одного цикла в ARM:

Это приводит к тому, что типичные программы ARM становятся плотнее, чем обычно, с меньшим доступом к памяти. Таким образом, конвейер используется гораздо более эффективно. Даже несмотря на то, что ARM работает на скоростях, которые многие бы сочли низкими, он довольно-таки легко конкурирует с многими более сложными архитектурами ЦПУ.

ARM процессор также имеет некоторые особенности, редко встречающиеся в других архитектурах RISC — такие, как адресация относительно счетчика команд (на самом деле счетчик команд ARM является одним из 16 регистров), а также пре- и пост-инкрементные режимы адресации.

Другая особенность, которую стоит отметить, это то, что некоторые ранние ARM процессоры (до ARM7TDMI), например, не имеют команд для хранения 2-байтных чисел. Таким образом, строго говоря, для них невозможно сгенерировать эффективный код, который бы вел себя так, как ожидается от объектов C, типа `volatile int16_t`.

Сопроцессоры Архитектура предоставляет способ расширения набора команд, используя сопроцессоры, которые могут быть адресованы, используя MCR, MRC, MRRC, MCRR и похожие команды. Пространство сопроцессора логически разбито на 16 сопроцессоров с номерами от 0 до 15, причем 15-й зарезервирован для некоторых типичных функций управления, типа управления кэш-памятью и операции блока управления памятью (на процессорах, в которых они есть).

В машинах на основе ARM периферийные устройства обычно подсоединяются к процессору путем сопоставления их физических регистров в памяти ARM или в памяти сопроцессора, или путем присоединения к шинам, которые в свою очередь подсоединяются к процессору. Доступ к сопроцессорам имеет большее время ожидания, поэтому некоторые периферийные устройства проектируются для доступа в обоих направлениях. В остальных случаях разработчики чипов лишь пользуются механизмом интеграции сопроцессора. Например, движок обработки изображений должен состоять из малого ядра ARM7TDMI, совмещенного с сопроцессором, который поддерживает примитивные операции по обработке элементарных кодировок HDTV.

Усовершенствованный SIMD (NEON) Расширение усовершенствованного SIMD, также называемое технологией NEON — это комбинированный 64- и 128-битный набор команд SIMD (single instruction multiple data), который обеспечивает стандартизованное ускорение для медиа приложений и приложений обработки сигнала. NEON может выполнять декодирование аудио формата mp3 на частоте процессора в 10 МГц, и может работать с речевым кодеком GSM AMR (adaptive multi-rate) на частоте более 13МГц. Он обладает внушительным набором команд, отдельными регистровыми файлами, и независимой системой исполнения на аппаратном уровне. NEON поддерживает 8-, 16-, 32-, 64-битную информацию целого типа, одинарной точности и с плавающей запятой, и работает в операциях SIMD по обработке аудио и видео (графика и игры). В NEON SIMD поддерживает до 16 операций единовременно. VFP

Технология VFP (Vector Floating Point, вектора чисел с плавающей запятой) — расширение сопроцессора в архитектуре ARM. Она производит низкозатратные вычисления над числами с плавающей запятой одинарной/двойной точности, в полной мере соответствующие стандарту ANSI/IEEE Std 754—1985 Standard for Binary Floating-Point Arithmetic. VFP производит вычисления с плавающей запятой, подходящие для широкого спектра приложений — например, для КПК, смартфонов, сжатие звука, трёхмерной графики и цифрового звука, а также принтеров и телеприставок. Архитектура VFP также поддерживает исполнение коротких векторных команд. Но, поскольку процессор выполняет операции последовательно над каждым элементом вектора, то VFP нельзя назвать истинным SIMD набором инструкций. Этот режим может быть полезен в графике и приложениях обработки сигнала, так как он позволяет уменьшить размер кода и выработку команд.

Другие сопроцессоры с плавающей запятой и/или SIMD, находящиеся в ARM процессорах включают в себя FPA, FPE, iwMMXt. Они обеспечивают ту же функциональность, что и VFP, но не совместимы с ним на уровне опкодов.

Отладка Все современные процессоры ARM включают аппаратные средства отладки, так как без них отладчики ПО не смогли бы выполнить самые базовые операции типа остановки, отступа, установка контрольных точек после перезагрузки.

Архитектура ARMv7 определяет базовые средства отладки на архитектурном уровне. К ним относятся

точки останова, точки просмотра и выполнение команд в режиме отладки. Такие средства были также доступны с модулем отладки EmbeddedICE. Поддерживаются оба режима — остановки и обзора. Реальный транспортный механизм, который используется для доступа к средствам отладки, не специфицирован архитектурно, но реализация, как правило, включает поддержку JTAG.

Существует отдельная архитектура отладки «с обзором ядра», которая не требуется архитектурно процессорами ARMv7.

Регистры ARM предоставляет 31 регистр общего назначения разрядностью 32 бит. В зависимости от режима и состояния процессора пользователь имеет доступ только к строго определённым набору регистров. В ARM state разработчику постоянно доступны 17 регистров:

- 13 регистров общего назначения (**R0..R12**).
- Stack Pointer (**R13**) — содержит указатель стека выполняемой программы.
- Link register (**R14**) — содержит адрес возврата в инструкциях ветвления.
- Program Counter (**R15**) — биты [31:1] содержат адрес выполняемой инструкции.
- Current Program Status Register (**RCPSR**) — содержит флаги, описывающие текущее состояние процессора. Модифицируется при выполнении многих инструкций: логических, арифметических, и др.

Во всех режимах, кроме User mode и System mode, доступен также Saved Program Status Register (**SPSR**). После возникновения исключения регистр **CPSR** сохраняется в **SPSR**. Тем самым фиксируется состояние процессора (режим, состояние; флаги арифметических, логических операций, разрешения прерываний) на момент непосредственно перед прерыванием.

Часть II

Программное обеспечение

Глава 2

Рабочая среда разработчика встраиваемых систем

- Операционная система с набором типовых утилит

Для Windows требуется дополнительно установить несколько модулей из пакета **GnuWin32**, чтобы обеспечить минимальную совместимость с UNIX-средой. Установка **GnuWin32** описана в ЛР??.

Установка Linux описана в ЛР1.

- Система управления версиями (**VCS**)

VCS предназначены для хранения полной истории изменений файлов проекта, и позволяют получить выгрузку проекта на любой момент времени, вести несколько веток разработки, получить историю изменений конкретного файла, или сравнить две версии файла ([diff](#)).

Установка VCS Git описана в ЛР2.

- Текстовый редактор или интегрированная среда разработки (IDE)

Редактирование текстов программ и скриптов сборки (компиляции) с цветовой подсветкой синтаксиса (в зависимости от языка файла), [автодополнением](#) и вызовом программ-утилит нажатием сочетаний клавиш. Также включает различные вспомогательные функции, например отладочный интерфейс и отображение объектов программ.

Установка IDE ECLIPSE описана в ЛР8.

- Тулчайн

Пакет кросс-компилятора, ассемблера, линкера и других утилит типа make, objdump,.. для получения прошивок из исходных текстов программ.

Установка GNU toolchain описана в ЛР??.

- ПО для программатора, JTAG-адаптера

Загрузка полученной прошивки в целевое устройство, редактирование памяти, внутрисхемная отладка в процессе работы устройства, прямое изменение сигналов на выводах процессора (граничное сканирование и тестирование железа).

Установка ПО для адаптеров ST-Link ЛР??, Segger J-Link ЛР??.

- Симулятор для отладки программ без железа

Симулятор может использоваться как ограниченная замена реального железа для начального обучения, и для отладки программ, не завязанных на работу железа.

Установка QEMU ЛР9.

- Система верстки документации

Для документирования проектов и написания руководств нужна система верстки документации, выполняющая трансляцию текстов программ и файлов документации в выходной формат, чаще всего **.pdf** и **.html**.

Установка L^AT_EX ЛР10.


ЛР1: Установка Debian GNU/Linux

ЛР2: Установка Git

Создадим рабочий каталог, установим систему контроля версий Git⁴ и получим локальную копию проекта этой книги, содержащий кроме текста для издательской системы L^AT_EX еще и исходные коды библиотек, примеры кода и т.п., которые вы захотите использовать в своих проектах.

 +  <http://git-scm.com/download/win>

Запуститься закачка установочного пакета scm-git (**Git-1.9.4-preview20140611.exe**), после его загрузки запустите установщик,


Welcome  Next

GNU GPL  Next

Select components  Windows Explorer Integration  Simple Context Menu  Git GUI here  Next

Use Git and optional Unix tools from the Command Prompt  Next

Use OpenSSH  Next

Checkout Windows-style  Next

Extracting files...

Completing Setup  ☐ View ReleaseNotes  Finish

Проверим что Git правильно установился:

 +  cmd

```
1 C:\Documents and Settings\pda>git --version
2 git version 1.9.4.msysgit.0
```

Первое, что вам следует сделать после установки Gita —указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Gite содержит эту информацию, и она включена в коммиты, передаваемые вами:

```
1 C:\Documents and Settings\pda>git config --global user.name "Vasya Pupkin"
2 C:\Documents and Settings\pda>git config --global user.email no@mail.com
3 C:\Documents and Settings\pda>git config --global push.default simple
```

Эти настройки достаточно сделать только один раз, поскольку в этом случае Git будет использовать эти данные для всего, что вы делаете. Если для каких-то отдельных проектов вы хотите указать другое имя или электронную почту, можно выполнить эту же команду без параметра `--global` в каталоге с нужным проектом.

Создаем каталог **D:/ARM** и выгружаем текущую копию этой книги из репозитория <https://github.com/ponyatov/CortexMx>, создавая свой собственный локальный [репозиторий проекта](#).

 +  cmd











```
1 C:\Documents and Settings\pda>D:
2 D:\>mkdir \ARM
3 D:\>cd \ARM
4 D:\ARM>git clone --depth=1 https://github.com/ponyatov/CortexMx.git book
```

ЛР3: Установка GNU toolchain

Самая важная часть — ставим GCC toolchain (набор инструментов) для процессоров ARM, собранный для **\$TARGET = arm-none-eabi**. Вариантов сборок для разработки для ARM под Windows много, есть и такие дистрибутивы как **CooCox IDE**, включающие полный комплект ПО одним пакетом. Ограничимся установкой варианта сборки под названием Yagarto:

  <http://sourceforge.net/projects/yagarto/> 

Запускаем скачанный инсталлятор.

Welcome 
License  
Choose Components  Add YAGARTO to PATH 
Destination folder  
Start Menu Folder  
Installation Complete  

Ygà поставилась, теперь можно проверить что доступны базовые утилиты:

Ассемблер

```
1 C:\Documents and Settings\pda>arm-none-eabi-as --version
2 GNU assembler (GNU Binutils) 2.23.1
3 Copyright 2012 Free Software Foundation, Inc.
4 This program is free software; you may redistribute it under the terms of
5 the GNU General Public License version 3 or later.
6 This program has absolutely no warranty.
7 This assembler was configured for a target of 'arm-none-eabi'.
```

Линкер

```
1 C:\Documents and Settings\pda>arm-none-eabi-ld --version
2 GNU ld (GNU Binutils) 2.23.1
```

Утилиты для работы с объектными файлами в формате ELF

```
1 C:\Documents and Settings\pda>arm-none-eabi-objdump --version
2 GNU objdump (GNU Binutils) 2.23.1
```

```
1 C:\Documents and Settings\pda>arm-none-eabi-objcopy --version
2 GNU objcopy (GNU Binutils) 2.23.1
```

Препроцессор (не компилятор C^{++})

```
1 C:\Documents and Settings\pda>arm-none-eabi-cpp --version
2 arm-none-eabi-cpp (GCC) 4.7.2
3 Copyright (C) 2012 Free Software Foundation, Inc.
4 This is free software; see the source for copying conditions. There is NO
5 warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Компилятор Си

```
1 C:\Documents and Settings\pda>arm-none-eabi-gcc --version
2 arm-none-eabi-gcc (GCC) 4.7.2
```

Компилятор C^{++}


```
1 C:\Documents and Settings\pda>arm-none-eabi-g++ --version
2 arm-none-eabi-g++ (GCC) 4.7.2
```



Утилита Make



```
1 C:\Documents and Settings\pda>make --version
2 "make" is not internal or external command.
3
4 C:\Documents and Settings\pda>arm-none-eabi-make --version
5 "make" is not internal or external command.
6
7 C:\Documents and Settings\pda>dir D:\ARM\Yaga\bin\*make*
8 Volume D has no label.
9 Serial #: 6588-9778
10
11 Direcorey contents D:\ARM\Yaga\bin
12
13 File not found
```

Упс, а **make** почему-то в комплект не включили ☹. Придется его ставить отдельно в ЛР4.




ЛР4: Установка утилит GnuWin32

Для совместимости скриптов придется поставить несколько пакетов из **GnuWin32**:

 +  <http://gnuwin32.sourceforge.net/packages.html>

 +  >> coreutils >>  + > Setup

 +  >> wget >>  + > Setup

 +  >> gnu make >>  + > Setup

coreutils-5.3.0.exe основные UNIX-утилиты типа rm ls , собранные под win32

Welcome >> Next

License >> Accept >> Next

Folder >> D:/ARM/GnuWin32 >> Next

Components >> Next

Start Menu >> GnuWin32/CoreUtils >> Next

Select Additional >> Next

Ready to Install >> Next

Completing >> Finish

Аналогично ставим:

make-3.81.exe утилита make

wget-1.11.4.exe консольная утилита загрузки файлов по HTTP/FTP

grep-2.5.4.exe утилита поиска строк в файлах и stdin/stdout потоке

ЛР5: Установка MinGW

Под Windows x64 обнаружилась проблема — если в **Makefile** используется перенаправление вывода команды в файл типа `objdump -xd program.o > program.o.dump` или маркеры сцепления, при выполнении **make** (из пакета **GnuWin32**) завершается с ошибкой

```
make: Interrupt/Exception caught (code = 0xc00000fd, addr = 0x4227d3).
```

Для обхода этой проблемы был найден способ — поставить пакет **MinGW**: это GNU тулчейн для нативной компиляции программ под win32. За счет небольшой потери объема диска получаем решение проблемы с **make** на Windows x64, и заодно получаем возможность компилировать простые вспомогательные утилиты на Си/C⁺.

Кроме того, нужно стараться писать [портабельный](#) код максимально независимый от платформы¹, а тестировать платформо-независимость можно, компилируя один и тот же код одновременно и для микроконтроллера, и для выполнения под Windows.

Для совместимости поставим 32-битную версию **MinGW**.



ЛР6: Редактирование системной переменной Windows \$PATH

Чтобы утилиты **GnuWin32** были доступны, нужно прописать переменную пользователя \$PATH в системном окружении.

¹чтобы можно было по необходимости легко и быстро перенести ваш проект на любой другой микроконтроллер, или использовать одни и те же куски кода как на МК, так и на ПК, например процедуры кодирования/декодирования данных или реализации протоколов обмена данными

Пуск > Настройка > Панель управления > Система > Дополнительно > Переменные среды

Переменные среды > переменные пользователя > Создать/Изменить

Имя переменной > PATH

Значение переменной > добавить в начало D:/ARM/GnuWin32/bin;D:/MinGW/bin;D:/ARM/Yaga/bin;..

Ok > Ok > Ok


Проверяем:

```
1 C:\Documents and Settings\pda>ls -la
2 total 3111
3 drwxr-xr-x   29 pda      user          0 Jul  4 14:03 .
4 drwxr-xr-x    9 pda      user          0 Oct  8 2013 ..
5 -rw-r--r--    1 pda      user       5242 May 22 14:29 .bash_history
6 drwxr-xr-x    2 pda      user          0 May 23 2013 .borland
7 drwxr-xr-x   18 pda      user          0 Sep  4 2013 .ccache
8 drwxr-xr-x    3 pda      user          0 Mar 26 2013 .eclipse
```

```
1 C:\Documents and Settings\pda>wget --version
2 GNU Wget 1.7
3
4 Copyright (C) 1995, 1996, 1997, 1998, 2000, 2001 Free Software Foundation, Inc.
5 This program is distributed in the hope that it will be useful,
6 but WITHOUT ANY WARRANTY; without even the implied warranty of
7 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
8 GNU General Public License for more details.
9
10 Originally written by Hrvoje Niksic <hniksic@arsdigita.com>.
```

```
1 C:\Documents and Settings\pda>mingw32-make --version
2 GNU Make 3.82.90
3 Built for i686-pc-mingw32
4 Copyright (C) 1988-2012 Free Software Foundation, Inc.
5 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
6 This is free software: you are free to change and redistribute it.
7 There is NO WARRANTY, to the extent permitted by law.
```

ЛР7: Установка Java

Для работы IDE  ECLIPSE требуется установленная Java:

 +  <http://www.oracle.com/technetwork/java/javase/downloads/>

- Минимальный вариант — ставим только Java Runtime:

Java Platform, Standard Edition >> JRE >> Download >> Accept License >> **jre-8u5-windows-i586.exe**

jre-8u5-windows-i586.exe >> Welcome >> ☒ Change destination folder >> Install

Destination folder >> **D:/Java/jre8** >> Next >> Installing >> Close

- Если вы планируете параллельно еще и осваивать язык Java — ставим Java SE JDK:

Java Platform, Standard Edition >> JDK >> Download >> Accept License >> **jdk-8u5-windows-i586.exe**

jdk-8u5-windows-i586.exe >> Welcome >> Next

Install to: **D:/Java/jdk8** >> Next

JRE Destination folder >> Install to: **D:/Java/jre8** >> Next

Java SE Development Kit 8 Update 5 Successfully Installed >> Close

ЛР8: Установка IDE ☰ECLIPSE

Для работы IDE ☰ECLIPSE требуется установленная Java ЛР7.

Для установки доступны два варианта:


1. **Eclipse Standard** базовый вариант среды, в ЛР рассмотрен именно он для иллюстрации ручной установки расширений
2. **Eclipse IDE for C/C++ Developers** вариант сборки уже включает расширение CDT, поэтому в следующий раз рекомендуем сразу качать его, это упростит и сэкономит немного времени на установку рабочей среды

☰ + R <http://www.eclipse.org/downloads/>

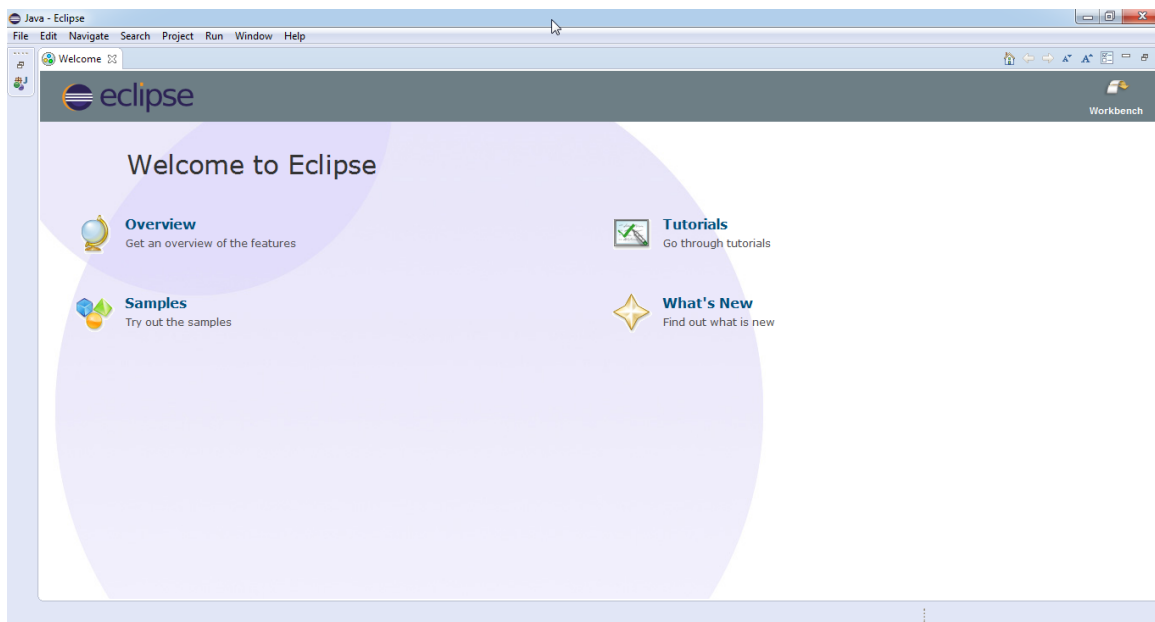
Eclipse Standard >> Windows 32 Bit >> Download >> **eclipse-standard-luna-R-win32.zip**

Перетащите каталог **eclipse** из архива в **D:/ARM** и создайте удобным для вас способом ссылку на **D:/ARM/eclipse/eclipse.exe**.



Workspace — рабочий каталог, в котором создаются каталоги отдельных проектов, типа **D:/WORK**. Eclipse создаст в нем служебный каталог **.metadata**, и поместит в него служебную информацию, относящуюся сразу ко всем проектам. Как побочный эффект, если в workspace уже есть какой-то каталог, можно создать новый проект (например **book**), и в левой части рабочей области  ECLIPSE в окне **Project Explorer** появится дерево файлов **book/***.

D:/ARM/eclipse/eclipse.exe >> Workspace >> D:/ARM >> Use as default >> OK



Проверяем наличие обновлений

Help >> Check for Updates >> Details >> No updates found >> OK

В базовом варианте Eclipse поддерживает только Java, поэтому нужно установить расширение для работы с C/C++: **CDT**.

Проект **CDT** предоставляет полнофункциональную интегрированную среду для разработки на Си и C⁺⁺. Поддерживаются: управление проектами и компиляцией для различных тулчейнов, стандартная сборка через **make**, навигация по исходным текстам, различные инструменты для работы с исходным текстом, такие как иерархия типов, граф вызовов, браузер подключаемых файлов, браузер макроопределений, редактор кода с подсветкой синтаксиса, сворачивание синтаксических структур (фолдинг) и гипертекстовая навигация, рефакторинг и генерация кода, средства визуальной отладки, включающие просмотр памяти, регистров и дизассемблер.

 +  <http://www.eclipse.org/cdt/downloads.php>

Выделить и скопировать в буфер обмена ссылку

p2 software repository: <http://download.eclipse.org/tools/cdt/releases/8.4>.

Добавляем сетевое хранилище пакетов для  ECLIPSE:

 ECLIPSE >> Help >> Install New Software >> Work with >> Add

Name >> CDT

Location >> <http://download.eclipse.org/tools/cdt/releases/8.4>

OK

Выбрать (если оно не выbralось само) хранилище  Work with: >> CDT, и в дереве выбора пакетов выбрать:

CDT

- └ CDT Main Features
 - └ ☒ C/C++ Development Tools
- └ CDT Optional Features
 - └ ☒ C/C++ C99 LR Parser
 - └ ☒ C/C++ GCC Cross Compiler Support
 - └ ☒ C/C++ GDB Hardware Debugging

Next > Next > Licenses > Accept > Finish

После установки пакетов появится окно с запросом перезапуска  ECLIPSE.


Аналогично ставим плагин GNU ARM Eclipse:

Help > Install > Work with > Add
 Name > GNU ARM plugin
 Location > <http://sourceforge.net/projects/gnuarmecclipse/files/Eclipse/updates/>

GNU ARM C/C++ Cross Development Tools

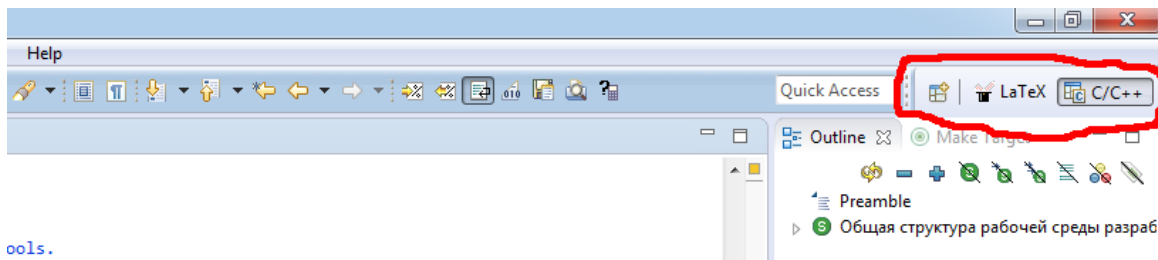
- └ ☒ Cross Compiler Support
- └ ☒ Generic Cortex-M Project Template
- └ ☒ STM32Fx Project Templates
- └ ☒ OpenOCD Debugging Support

Warning: You install unsigned content > Ok

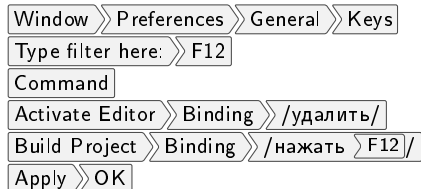
В  ECLIPSE есть так называемые [перспективы](#) (perspective) — это переключаемые режимы отображения рабочего набора окон, настроенные под тип работы. По умолчанию запускается перспектива **Java**. Нас интересует перспектива **C/C++**:

Window > Open Perspective > Other > C/C++ > Ok

Также перспективу можно переключить кнопкой на панели в правом верхнем углу:



Для настройки привычных вам клавиш можно сразу зайти в глобальные настройки среды и поменять привязку клавиш:



ЛР9: Установка симулятора QEMU

Нередко в практике разработчика возникают ситуации, когда программное обеспечение (ПО) для микроконтроллера приходится писать в отсутствии под рукой аппаратной платформы.

Например, печатная плата устройства отдана на подготовку к производству, а времени ждать готовое устройство для тестирования на нем программного обеспечения нет.

В таких случаях для оценки работоспособности ПО можно воспользоваться программным симулятором целевого микроконтроллера.

Для интегрированной среды разработки \oplus ECLIPSE CDT в качестве программного симулятора микроконтроллеров ARM можно использовать симулятор (или виртуальную машину,если быть точным) **qemu-arm** с интерфейсом командной строки:



Добавьте **D:/ARM/qemu** в системную переменную **\$PATH** (ЛР6).

```
1 C:\Documents and Settings\pda>qemu-system-arm -version
2 C:\Documents and Settings\pda>cat D:\ARM\qemu\stdout.txt
3 QEMU emulator version 2.0.90, Copyright (c) 2003-2008 Fabrice Bellard
```

ЛР10: Установка системы верстки документации \LaTeX

Если вы планируете писать полноценную документацию на программы и оборудование, или участвовать в доделке этой книги, вы можете установить систему верстки \LaTeX .


Для работы с \TeX требуется довольно приличное по усилиям (само)обучение [11], но оно оправдывается если вы часто пишете документацию, особенно если в ней больше 10 формул. Готовить документацию в $\text{\M\$ Word}$ — (само)убийство мозга и времени, идеология подстановочных макросов \TeX , богатый набор доп. пакетов и командный ввод формул очень доставляют.



Скачайте и установите пакет \MiKTeX :

 +  <http://miktex.org/download> > Other Downloads > Net Installer
Save as: > D:/ARM/soft/MikTeX/miktex-netsetup-2.9.4503

Загрузка дистрибутивных файлов

 miktex-netsetup-2.9.4503 > License > Accept > Далее
Task > Download > Далее

Если у вас постоянное internet-соединение: Package Set > Basic \MiKTeX > Далее

Для offline работы² Package Set > Complete \MiKTeX > Далее

Download Source > Russian Federation (ctan.uni-altai.ru) > Далее

Distribution Directory > D:/ARM/soft/MikTeX > Далее > Start > Executing > Далее > Close

Установка из ранее загруженного дистрибутива

D:/ARM/soft/MikTeX/miktex-netsetup-2.9.4503 > License > Accept > Далее
Task > Install > Далее > Basic \MiKTeX > Далее

² когда неизвестно какие пакеты понадобятся — \MiKTeX умеет их докачивать по необходимости

Install for >> Anyone/Only for user >> Далее

Install from: >> D:/ARM/soft/MikTeX >> Далее

Install to: >> D:/LaTeX/MiKTeX >> Далее

Settings

Preferred paper >> A4

Важная опция: автоматическая докачка отсутствующих пакетов Install missing packages >> Yes

Далее >> Start >> Executing >> Close

Двухступенчатая установка позволяет сначала скачать полный дистрибутив MiKTeX, а затем установить его на другой компьютер, не подключенный к Internet, или с медленным/платным каналом не дающим взять и качнуть 200 Мб.

Для удобной работы с **.tex** файлами в ECLIPSE нужно поставить дополнение **TeXlipse**:

ECLIPSE >> Help >> Install >> Work with >> Add

Name >> TeXlipse

Location >> <http://texlipse.sourceforge.net>

TeXlipse

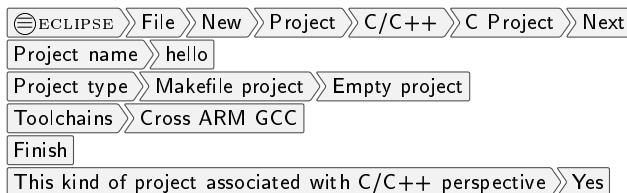
└─ ☒ TeXlipse

Глава 3

Первые шаги

ЛР11: Создание нового проекта в ECLIPSE

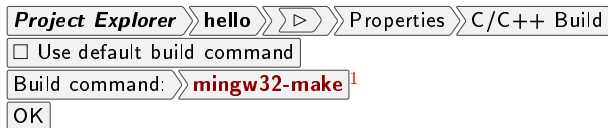
Создадим новый проект, напишем простую программу, и запустим ее в отладчике.



В окне *Project Explorer* появится пустая закладка проекта **hello**. Если вдруг там будут какие-то файлы, значит кто-то до вас уже создал проект, и что-то туда наляпал. В этом случае повторите создание, задав имя типа **hello<номер группы><FIO>** или типа того, для полной уверенности можно сначала посмотреть что в **D:/ARM** нет папки с таким именем.

Нужно сразу настроить несколько свойств проекта.

Команда-билдер для проекта — задаем явно **make**:



¹почему не просто **make** см. ЛР5

LP12: Создание Makefile

Стоит объяснить, почему при создании проекта мы выбрали тип **Makefile project**, хотя были доступны более логичные варианты типа **ARM C Project**.

Утилита **make** ведет свою историю с 70х гг. Компьютеры тогда были большими, тяжелыми, а главное медленными и с очень маленькой памятью (десятки÷сотни Кб). Компиляторам зачастую не хватало памяти, чтобы скомпилировать большую программу. Кроме того, скорость их запуска и работы была тоже черепашейей. Поэтому исходный код программы делили на модули, компилировали или ассемблировали каждый модуль по-отдельности в объектный код, а затем уже на конечном этапе с помощью линкера собирали несколько файлов объектного кода в один исполняемый файл.

Для ускорения и упрощения этого процесса и была создана утилита **make**. Чтобы не вызывать лишний раз компилятор или какой-нибудь транслятор, в файле **Makefile** прописываются зависимости между файлами. Затем запускается **make** с указанием какой файл нам нужно получить, и выполняется цепочка вызовов нужных программ.

Следует отметить, что утилита **make** используется до сих пор для сборки самых современных программных пакетов², правда в комплексе с другими средствами, обеспечивающими переносимость программ между разными ОС и автогенерацией зависимостей из исходного кода.

Для наших целей **make** используется как самое простое средство управления компиляцией проекта. В средах разработки, особенно в коммерческих, используются служебные файлы проектов, иногда бинарные, чаще текстовые, но всегда запутанные и весьма развесистые.

Если вам вдруг понадобится откомпилировать ваш проект на другом компьютере, с другой архитектурой, возможно вообще без графического интерфейса³, или вы вдруг решите попробовать работать в другой IDE — вы тут же вляпаетесь в ситуацию, когда нечем открыть файл проекта с заботливо прописанными опциями компиляции.

² типа GCC 4.9.x, ядра Linux или KDE под FreeBSD

³ например какой-нибудь удаленный сервер на процессоре 1995BM666 под раскрытым Solaris 7α4, на котором лежит криптиобиблиотека, использующая при компиляции трофейный электро-механический энкодер, существующий в единственном экземпляре ☹

```

1 # пример мейкфайла для проекта Азбука ARMатурщика
2 # лабораторная работа ЛР{labmkmake}
3 # символ # в начале -- комментарий
4
5 # пример использования переменных
6
7 # простое присваивание значения переменной
8 # обнуление переменной
9 SOMEVAR =
10 # маски временных файлов
11 TMPFILES = *.o *.elf *.hex *.objdump
12
13 # целевая платформа $TARGET, часто называют "префикс целевой платформы"
14 TARGET = arm-none-eabi
15 # целевой процессор
16 CPU = cortex-m3
17 CPUOPT = -mcpu=$(CPU) -mthumb
18
19 # переопределение переменных
20
21 # целевой процессор для запуска под gdb-симулятором ARM7TDMI
22 CPU = arm7tdmi
23 CPUOPT = -mcpu=$(CPU)
24
25 # присваивание переменной с подстановкой значений другой переменной

```

```
26 # стандартные переменные, задающие команды ассемблера, компилятора и линкера
27 AS = $(TARGET)-as
28 CC = $(TARGET)-gcc
29 LD = $(TARGET)-ld
30 OBJDUMP = $(TARGET)-objdump
31 OBJCOPY = $(TARGET)-objcopy
32 SIZE = $(TARGET)-size
33 MAKE = mingw32-make
34
35 # нестандартная (?) переменная - опции оптимизации
36 OPTFLAGS = -O0
37 # опции генерации отладочной информации
38 DEBFLAGS = -g3 -ggdb
39 # стандартная переменная - флаги компилятора Си
40 CFLAGS = $(CPUOPT) $(OPTFLAGS) $(DEBFLAGS)
41 # флаги ассемблера
42 ASFLAGS = $(CPUOPT) $(DEBFLAGS)
43
44 # указание что цели all и clean являются фиктивными целями, а не файлами
45 .PHONY: all clean
46
47 # первая цель, заданная в Makefile, является целью по умолчанию
48 # и обрабатывается при вызове $(MAKE) без параметров
49
50 # стандартная цель, предусматривающая сборку всего проекта
51 all: elf.elf
52
53 elf.elf: $(CPU).ld startup.o init.o main.o
```

```

54 ____$(LD) -T $(CPU).ld -o $@ *.o && \
55 ____$(OBJDUMP) -xd $@ > $@.objdump && \
56 ____$(SIZE) $@
57
58 # стандартная цель, удаление всех временных и конечных бинарных файлов
59 clean:
60 ____rm -f $(TMPFILES)
61
62 # макро-правило: как компилировать исходные файлы в объектный код
63 # вместо % в других правилах могут подставляться любые символы, см. цель all
64 # тэг $@ заменяется на цель правила, т.е. %.o
65 # тэг $< заменяется на первый источник, т.е. %.c
66 %.o: %.c Makefile
67 ____$(CC) $(CFLAGS) -c -o $@ $<
68 ____$(OBJDUMP) -dx $@ > $@.objdump
69
70 # макро-правило: как компилировать ассемблерные файлы
71 %.o: %.S Makefile
72 ____$(AS) $(ASFLAGS) -o $@ $< && $(OBJDUMP) -dx $@ > $@.objdump

```

Обратите внимание, особенно если не используете ≡ECLIPSE — текстовый редактор должен быть настроен так, чтобы символ табуляции <TAB> не заменялся на пробелы, и отображался как 4 <пробел>а. В листинге табуляции специально выделены, т.к. *имеют синтаксическое значение*.

Этот пример **Makefile** достаточно универсален и самодостаточен для большинства проектов в этой книге. Кажущийся большой объем получился за счет использования комментариев и переменных. И те, и другие служат для документирования проекта, и повышают читаемость кода. В принципе никто не мешает⁴ написать несколько строк в **.bat**нике с явным указанием опций компиляторам, или вообще от-

⁴особенно для микроскопических объемов исходных текстов программ для контроллеров — в самом худшем случае какие-

компилировать все исходники сразу одним вызовом **gcc** с кучей опций и списком исходных файлов. Но если вам потребуется что-то изменить, куда проще и быстрее сделать это в аккуратно оформленном самодokumentированном **Makefile**.

Компилятор преобразует программу на языке программирования высокого уровня⁵ в объектный код (смесь кусочков машинного кода со служебной информацией) или в текст на языке ассемблера.

Кросс-компилятор (**gcc**) отличается от обычного компилятора тем, что генерирует код не для компьютера на котором он выполняется (хост-система, `$HOST`), а для компьютера другой архитектуры — целевой системы, `$TARGET`.

Ассемблер (**as**) преобразует человекочитаемый машинный код программы в объектный код.

Линкер (**ld**) объединяет несколько файлов объектного кода в один, и корректирует машинный код с учетом его конечного размещения в памяти целевой системы (адреса переменных, адреса переходов, размещение сегментов кода и данных в физической памяти целевой системы).

Дампер (**objdump**) позволяет получить информацию о содержимом объектных файлов, в частности значения различных служебных полей, и дизассемблированный машинный код.

Копир (**objcopy**) преобразует сегменты кода/данных из файла, полученного линкером, в формат, необходимый для ПО программатора: бинарные файлы, Intel HEX, ELF,.. загружаемые в масочное ПЗУ, FlashPROM (и EEPROM данных на МК ATmega).

Так как часто разработчики встраиваемых систем работают с разными аппаратными платформами, для команд тулчайна принято использовать префиксы типа **arm-none-eabi-**, чтобы явно отличать, какой именно (кросс-)компилятор вызывается.

Главная синтаксическая конструкция **Makefile** — блок правила, задающий зависимость между файлами и набор команд, которые нужно выполнить, если **дата модификации файла-цели старше, чем**

то жалкие сотни Кб

⁵для микроконтроллерных встраиваемых систем используются Си и C++, на более тяжелых процессорах типа Cortex-Ax свободно применяются Java, Fortran, Python, и еще стоицот языков, созданных за последние 50 лет истории IT

дата модификации одного из файлов-источников. То есть если вы измените какой-то из файлов проекта, начнут срабатывать правила, которые обновляют зависимые от него файлы.

Синтаксис:

```
<файл-цель>: [<файл-источник1> ...]  
[<tab><команда1>]  
[<tab><команда2>]  
[...]
```

Количество файлов-источников и команд может быть любое, в том числе и нулевое. Каждая команда правила отбивается слева одной табуляцией (один символ с кодом 0x09, **не пачка пробелов**). **Будьте аккуратны, редактируя Makefile во всяких блокнотах, вордпадах и прочей ереси, любящей “оптимизировать” пробелы: истинный ТАВ и 4 пробела на экране, как завещал Великий Столлман.**

Использование переменных особых комментариев не требует, обычная подстановка. Есть переменная `$$`, имеющая значение текущего файла-цели. Есть похожая переменная `$<` — имя первого файла-источника.

Если кто вдруг не знает — символ `>` в командной строке применяется для перенаправления текстового вывода любой команды в файл. Если нужно в одной строке выполнить последовательно несколько команд, используются маркеры сцепления `;&&` и `|`. Описание их применения см. любую книжку по UNIX дотсадовского уровня. В **Makefile** для простого последовательного выполнения команд⁶ рекомендуется использовать сцепку `&&`⁷.

Команды выполняются с синтаксисом: `<[путь]команда[.exe]> [параметры через пробел]`.

Команда — имя выполняемого файла, может указываться с полным путем (диск, цепочка каталогов) или без. Если путь не указан, поиск выполняемого файла проводится в списке каталогов, заданном в системной переменной `$PATH`. Под DOS и Windows исполняемые файлы имеют суффикс `.exe`, `.bat` и

⁶без передачи данных через потоки ввода/вывода

⁷следующие команды выполняются только если предыдущая завершилась без ошибок — если компиляция завершится ошибкой, незачем вызывать программатор

.com, который в командной строке обычно не указывается. Под UNIX флаг выполнимости можно поставить вообще на любой файл.

В параметрах указываются имена файлов и опции: текстовые одно- и многобуквенные имена, начинающиеся с одинарного или двойного минуса. Параметры разделяются одним или несколькими пробелами. Порядок и значение параметров зависит от команды. Параметры для команд GNU toolchain и ПО программаторов подробно описаны далее.

ЛР13: Hello World

Для начала нужно рассмотреть набор файлов минимального проекта:

- **README.txt**

Краткая информация о проекте — название, авторы, обязательно ссылки на Git-репозиторий, сайт, форум, и т.п.

- **Makefile**

Файл с описанием зависимостей между файлами, настройками проекта (в переменных) и правилами вызова компиляторов.

- **startup.S**

Стартовый код процессора, включает инициализацию системы тактирования, мапинга памяти, контроллера прерываний и минимальную инициализацию периферии. Пишется на ассемблере, т.к. на Си получается слишком сложно, синтаксически запутанно, или очень специфично для компилятора.

- **init.c**

Сишный код инициализации железа (синтаксически легче описать блоки кода, зависимые от целевого процессора).

- **main.c**

Основной код, решающий поставленную задачу.

- **\$CPU.ld**

Скрипт линкера, настраивающий генерацию выходного бинарного файла в зависимости от целевого процессора — прежде всего организация памяти, и размещение сегментов кода/данных по фактическим адресам памяти. Поэтому здесь имя файла задано через переменную, описанную в **Makefile**.

Создаем эти файлы аналогично **Makefile** в ЛР12:

ECLIPSE > **Project Explorer** > **hello** > > New > File > File name: > **НужныйФайл.xxx**

README.txt

```
1 Азбука ARMатурщика: лабораторная работа HelloWorld
2 (copy pasta) Dmitry Ponyatov <dponyatov@gmail.com>
3 https://github.com/ponyatov/CortexMx
```

startup.S

```
1 // универсальный стартовый код для любых ARM-микроконтроллеров
2 // скопирован из руководства QuantumLeaps по использованию GNU toolchain
3
4 // этот стартовый код должен быть слинкован на начало ПЗУ,
5 // которое не обязательно начинается с нулевого адреса
6
7 .text // сегмент кода
8 .thumb // Cortex-M умеет только Thumb режим
9 .code 32 // раскомментировать для ARM7TDMI (?)
10
11 .func _vector
12 .global _vector
13 _vector: // таблица векторов прерываний/исключений
14 // используется команда относительного перехода,
15 // т.к. она корректно работает при стартовом ремappingе памяти
16     B _reset // Reset // Сброс
17     B . // Undefined Instruction // Неизвестная инструкция
18     B . // Software Interrupt // Программное прерывание
```

```

19     B .           // Prefetch Abort           // Сбой предвыборки
20     B .           // Data Abort               // Сбой по данным
21     B .           // Reserved                 // зарезервировано
22     B .           // IRQ                     // Прерывание
23     B .           // FIQ                     // Быстрое прерывание
24 .endfunc
25
26 // строка копирайта, конец дополняется до границы машинного слова
27 .func _copyright
28 .global _copyright
29 _copyright:
30 .string "(c) Azbuka ARMaturschika"
31 .align 4
32 .endfunc
33
34 //.thumb_func
35 .func _reset
36 .global _reset
37 _reset:      // сюда передается управление при сбросе
38     LDR SP,=_stack_top
39     B .
40 .endfunc
41
42 .data
43 .word 0x12345678,1234
44
45 .bss
46 .comm buf,0x10,10

```

```
47
48 .end
```

init.c

```
1 init () {}
```

main.c

```
1 main () {}
```

arm7tdmi.ld

```
1 /* скрипт линкера для ARM7TDMI */
2
3 ENTRY(_vector) /* точка входа */
4
5 /* конфигурация памяти целевой системы, сильно зависит от микроконтроллера */
6
7 MEMORY
8 {
9     flash (rx) : ORIGIN = 0x00000000, LENGTH = 4K
10    ram   (rwx) : ORIGIN = 0x20000000, LENGTH = 4K
11 }
12 _stack_size = 1K; /* размер стека */
13
14 /* описание упаковки секций объектных файлов
15    в целевой файл и размещение в памяти */
16
```

```

17 SECTIONS
18 {
19     /* секция кода, обеспечиваем нужный порядок сегментов */
20     .text : {
21         startup.o(.text) /* в начало кладем таблицу векторов */
22         *(.text)          /* а потом все остальное */
23     } >flash
24     /* стек по рекомендации MISRA располагаем НИЖЕ данных т.к. растет вниз
25        (предотвращение затирания данных при переполнении стека) */
26     .stack : {
27         . = ALIGN(8);
28         _stack = .;
29         . = . + _stack_size; /* резервируем память под стек */
30         _stack_top = .;      /* создаем указатель на вершину стека */
31     } >ram
32     /* секция инициализированных данных (константы) */
33     .data : {
34         *(.data)
35     } >ram AT>flash
36     /* секция пустых данных и кучи
37        (переменные и массивы без заданных значений, динамическая память) */
38     .bss : {
39         *(.bss)
40     } >ram
41 }

```

ЛР14: Настройка отладчика в ECLIPSE

8

На сегодняшний день существуют много способов и инструментов для отладки embedded приложений, начиная с отладки “в железе” (внутрисхемная отладка) и заканчивая всякими симуляторами. У каждого метода есть свои плюсы и минусы, но поскольку мы будем писать приложения для реальных устройств, то предпочтительней реальная отладка (в железе), то есть приложение будет исполняться непосредственно микроконтроллером.

Что нам понадобится для “железной отладки” :

- ARM микроконтроллер (для симуляции необязателен)
- JTAG/SWD адаптер (для симуляции необязателен)
- GDB сервер (транслятор интерфейсов GDB/JTAG)
- GDB отладчик (имеет встроенный симулятор ARM7TDMI, используется для первых лаб)
- плагин C/C++ GDB Hardware Debugging
- плагин Eclipse Embedded Systems Register View

JTAG адаптер (он же программатор) следует выбрать тот, который поддерживает именно ваш микроконтроллер, а еще лучше, микроконтроллеры разных производителей. В моем случае (еще с давних времен у меня завалились кристаллы от Texas Instruments, ST Microelectronics, NXP, Atmel, Cypress), я сразу решил найти программатор поддерживающий имеющиеся у меня камни. Порыскав в интернетах, мой выбор пал на китайский клон знаменитого J-Link, в добавок к которому идет уйма полезных утилит от Segger Microcontroller (тут обошлось без китая ☺), облегчающие жизнь разработчику.

⁸копираста: <http://makesystem.net/?p=2146>

В этой книге также рассмотрено несколько простых вариантов JTAG-адаптеров, которые вы можете сделать сами, не обладая выдающимися знаниями в электронике и технологиях производства печатных плат.

Структура аппаратно-программного комплекта для отладки:

микроконтроллер » JTAG/SWD » адаптер » LPT/USB » GDB сервер » протокол GDB » GDB отладчик » IDE

Адаптер подключается к выводам МК с помощью колодки (JTAG) или гребенки (SWD).

К компьютеру адаптер подключается через один из распространенных интерфейсов: совсем дешевые варианты “на пяти резисторах” через порт LPT, чуть подороже через USB, совсем дорогие проф. модели могут иметь Ethernet интерфейс.

Отладчик (дебаггер, англ. debugger) — компьютерная программа, предназначенная для поиска ошибок в других программах. Отладчик позволяет выполнять пошаговую трассировку, отслеживать, устанавливать или изменять значения переменных в процессе выполнения кода, устанавливать и удалять контрольные точки или условия остановки, сопоставлять двоичный код — его исходному тексту (на основе которых можно точно определить выполняемые программой действия) и т.д. (Wiki)



Практически во все тулчейны входит утилита GDB (**arm-none-eabi-gdb**), это и есть отладчик GNU. В принципе, дебаггер выполняет два типа действий: управление исполнением программы в кристалле (через отладочный интерфейс) и вывод результатов в консоль/графическую оболочку.

При сборке тулчайна (из исходников) невозможно заранее сказать, какой набор отладочных средств будет у конечного пользователя — у типичного ембеддера⁹ запросто наберется пара-тройка различных JTAG-адаптеров, несколько демолат со встроенным адаптером, причем с разными процессорами, несколько собственных устройств с самодельными отладочными интерфейсами, и еще для комплекта пару чисто программных симуляторов ARM-ядер.

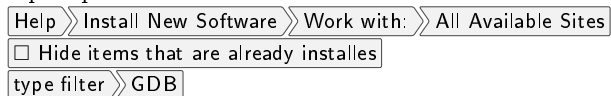
Задачу унификации интерфейсов, и подключения всего этого зоопарка к одному и тому же отладчику GDB выполняет GDB-сервер. Отладчик общается с сервером по одному и тому же унифицированному

⁹разработчика ПО под встраиваемые системы

[GDB-протоколу](#) через последовательный порт или TCP/IP соединение, а все сложности взаимодействия с железом берет на себя сервер. Это сделано (в том числе) для тех случаев, когда GDB-отладчик работает на одном ПК а GDB-сервер на другом (в соседней комнате или соседнем государстве ☺). Опять же, сервер надо выбрать тот, который поддерживает ваш JTAG-адапер или эмулятор.

GDB, как и все остальные утилиты тулчайна, работает из командной строки, что не всем удобно ☺, поэтому для начала стоит научиться им пользоваться из графической оболочки. ECLIPSE как и подобает серьезной IDE, имеет средства работы с GDB, заменяющие его консоль, имеет графические кнопки для вызова всех отладочных команд, и отображает содержимое регистров, памяти и т.п. в графических окнах. Взаимодействие ECLIPSE/GDB обеспечивает плагин **C/C++ GDB Hardware Debugging**, входящий в состав уже установленного ранее расширения **CDT**.

Проверить наличие плагина можно так:



GDB

- └ CDT Optional Features
 - └ ☒ C/C++ GDB Hardware Debugging
- └ Mobile and Device Development
 - └ ☒ C/C++ GDB Hardware Debugging

Прежде всего отключим оптимизацию кода проекта, задав в **Makefile** значение переменной **OPTFLAGS** = **-O0**.

Затем, нужно включить в проекте генерацию отладочной информации¹⁰, добавив опцию **-g[N]**.

Существуют три уровня отладочной информации:

¹⁰ имена переменных, функций и т.п. объектов программы, в т.ч. и сами строки исходного кода

1. в объектный код вставляется минимальный объем отладочной информации. Ее вполне достаточно для трассировки вызовов функций и исследования глобальных переменных, тем не менее, отсутствует информация для сопоставления выполняемого кода со строками исходного кода и информация для отслеживания локальных переменных.
2. используется по умолчанию. Помимо всей отладочной информации первого уровня он дополнительно включает данные, необходимые для сопоставления строк исходного кода с выполняемым кодом, а также имена и расположение локальных переменных.
3. помимо всей отладочной информации первого и второго уровней, включает дополнительную информацию, в частности определения макросов препроцессора.

Используем 3 уровень, изменив в **Makefile** значение переменной **DEBFLAGS = -g3 -ggdb**. Опция [-ggdb](#) задает дополнительно формат отладочной информации. Доступны форматы STABS, DWARF2 и родной формат платформы.

В файле **startup.o.dump** при этом появляются дополнительные секции с отладочной информацией, и в заголовок добавляются флаги, указывающие на ее наличие:

startup.o.objdump

```
1 startup.o:      file format elf32-littlearm
2 architecture:  armv4t, flags 0x00000011: HAS_RELOC, HAS_SYMS
3 ...
4 4 .debug_line   00000044  00000000  00000000  000000b0  2**0
5                  CONTENTS, RELOC, READONLY, DEBUGGING
6 5 .debug_info   00000044  00000000  00000000  000000f4  2**0
7                  CONTENTS, RELOC, READONLY, DEBUGGING
8 6 .debug_abbrev 00000014  00000000  00000000  00000138  2**0
9                  CONTENTS, READONLY, DEBUGGING
10 7 .debug_aranges 00000020  00000000  00000000  00000150  2**3
```


Для настройки отладочного интерфейса заходим в меню

Run >> Debug Configurations...

GDB Hardware Debugging >>> >> New

В результате открывается окно с настройками отладки.

Для начала попробуем работу нашей прошивки на встроенном в GDB программном симуляторе процессора **ARM7TDMI**.

Вкладка Main.

В поле Name, можно дать имя всей конфигурации отладки, поскольку даже для одного проекта бывают разные конфигурации отладки (скажем для отладки в RAM или Flash памяти).

Name: >> ARM7TDMI simulator

В поле Project указываем имя проекта (поскольку в нашем workspace может быть более одного проекта)

Project: >> hello

В поле C/C++ Application указываем имя *.elf файла сгенерированного после компиляции (с введенными ранее настройками для Debug прошивки) проекта и который будет использован во время отладки.

C/C++ Application: >> startup.o

Перед тем как перейти к следующей вкладке, в нижней части окна обязательно выбираем

Legacy GDB Hardware Debugging Launcher

Apply

Вкладка Debugger. Здесь мы установим связь между отладчиком и графической оболочкой, а также между отладчиком и GDB-сервером (отсюда и название вкладки).

В поле GDB Command указываем имя отладчика из тулчайна. Должен быть прописан в **\$PATH**, или можно указать полный путь

GDB Command: >> arm-none-eabi-gdb

В соответствии с идеологией ведущих разработчиков Free Software Foundation, GDB вместо собственного графического пользовательского интерфейса предоставляет возможность подключения к внешним IDE, управляющим графическим оболочкам либо использовать стандартный консольный текстовый интерфейс” (Wiki). В общем, mi (Machine Interface) это протокол общения между отладчиком и графической оболочкой.

Command Set: >> Standard (Windows)

Protocol Version: >> mi

Как ранее было сказано, общение между отладчиком и сервером осуществляется через последовательный или TCP/IP порт, поэтому в общем случае следует выбирать опции типа:

Remote Target >> ☒ Use remote target

JTAG Device: >> Generic TCP/IP

IP address: >> localhost

Port number: >> 12345

Но поскольку мы собираемся использовать встроенный симулятор ARM7, пока нужно **выключить** удаленную отладку:

☐ Use remote target

Apply

Вкладка Startup (предписания отладчику перед началом работы).

Сброс необходим для того чтобы очистить регистры ARM процессора от значений полученных в ходе предыдущей отладки (по желанию)

Reset and Delay (seconds): >> 3

Останавливаем процессор для настройки эмулятора и загрузки отлаживаемой прошивки

☒ Halt

В (пустом) текстовом поле вводим команды, выполняемые при старте отладки.

Настало время вернуться к вопросу об использовании симуляции микроконтроллеров **ARM7TDMI**. На самом деле с этой задачей запросто справляется сам GDB, если указать ему стартовые команды:

target sim

load

Из какого файла грузить прошивку

Load image » Use project binary

Из какого файла грузить отладочную информацию¹¹

Load symbols » Use project binary

Apply

Вкладка Common

Display in favorites menu » ☒ Debug

Standard Input and Output » ☒ Allocate console

☒ Launch in background

Apply

При первом запуске отладки, ⊕ECLIPSE просит разрешение на переход в режим отображения отладки (Debug perspective). Разрешаем и ставим галку “☒ больше не спрашивать“. Далее, открывается отображение многочисленных окон, каждое со своим предназначением (окна исходного кода, окно дизассемблера, окно отображения памяти и т.д.). При желании можно добавить различные окна через меню **Window** » Show View.



Первый запуск отладчика : кнопка клопа » ARM7TDMI simulator

¹¹ можно использовать отдельный .sym файл

Последующие запуски (последнего) отладчика: просто **F11**

```
1 symbol-file C:\\ARM\\book\\hello\\startup.o
2 Reading symbols from C:\\ARM\\book\\hello\\startup.o... done.
3 target sim
4 load C:\\ARM\\book\\hello\\startup.o
5 Connected to the simulator.
6 Loading section .text, size 0x50 vma 0x0
7 Start address 0x0
8 Transfer rate: 640 bits in <1 sec.
```

Глава 4

Система управления версиями Git

[6]

Установку ПО см. ЛР²

Избегайте использования бинарных файлов, по возможности генерируйте их из текстового описания на каком-нибудь макроязыке — в этом случае VCS обеспечит вам возможность получить историю или diff в человекочитаемопонимаемом виде, а не в виде набора невнятных кексов.

Рекомендую использовать Git и один из проектных хостингов типа <https://github.com/>. Установка описана в ЛР².

Глава 5

Интегрированная среда разработки



Например нажатием **F3** в **⊕ECLIPSE** можно переместиться на определение функции, на имени которой находится текстовый курсор.

Автодополнение — редактор предлагает варианты полного написания идентификаторов и ключевых слов по первым буквам и нажатию обычно **Ctrl**+**Tab** или **Ctrl**+**N**. Также автоматически расставляются закрывающие скобки, закрывающие операторы управляющих структур типа `begin/end`, и генерируются синтаксические элементы циклов при вводе ключевых слов `if/for/while`. Особенно удобно автодополнение при написании кода на ООП языках — при вводе имени класса или объекта и точки предлагается меню с именами данных и методов класса.

При вводе имени функции и скобки выводится всплывающее окно с подсказкой — определение функции с типом возвращаемого значения, типом и именами параметров.

Интерфейс IDE часто предусматривает различные вспомогательные окна, показывающие имена и свойства объектов, описанных в программе (переменные, функции, структуры,...), структуру проекта с зависимостями между файлами, блоки справки в зависимости от текущего выделенного элемента и т.п.

Часто IDE имеет встроенный графический интерфейс для отладки программ, используя для этого интерфейсные библиотеки для программатора и специальный отладочный код, добавляемый к вашей программе при компиляции. Используя аппаратный модуль отладки на целевом процессоре и отладочный код, IDE обеспечивает отображение значений и изменений регистров процессора, состояние перефери, позволяет задать точки останова в программном коде, в т.ч. условные по значению или изменению переменных или регистров железа. При использовании ОС реального времени и системы аппаратной многозадачности отображается загрузка ядер, загрузка процессора и используемые ресурсы для каждой задачи, работа планировщика, и т.п.

Для удобной работы доступно несколько бесплатных вариантов IDE, далее рассмотрим два варианта: тяжелая суперуниверсальная среда **⊕ECLIPSE**, и легкая в отношении требуемых ресурсов системы **CodeLite**.

Глава 6

Пакет кросс-компиляции GNU toolchain

Часть III

Отладка

Часть IV

Встраиваемый C^{++}

Часть V

RTOS

Часть VI

Автоматное программирование /фреймворк QuantumLeaps/

Часть VII

Разработка и изготовление железа

Глава 7

САПР KiCAD

Глава 8

Инструмент и оборудование

Глава 9

Технологии изготовления плат и монтажа

Часть VIII

Подготовка документации

Глава 10

DocBook

Глава 11

L^AT_EX

Необходимо использовать человеко-читаемые простые текстовые файлы (**plain ascii text**, кодировка по выбору, удобнее всего **utf8**) и использовать язык разметки — DocBook, а удобнее всего L^AT_EX.

Ни в коем случае не используйте для документации всякую бинарщину типа NarcoSoft Word — текстовый формат необходим для корректной и полноценной работы VCS. Исключение по необходимости — только графические файлы, подключаемые при генерации выходных файлов документации.

Эта книга написана с использованием языка разметки L^AT_EX, и транслируется в экранный **.pdf** с



помощью пакета

MiKTeX/



TeXlive.

Установка описана в ЛР10

Литература

- [1] <https://github.com/ponyatov/CortexMx> Азбука халтурщика-ARMатурщика
- [2] Ю.С. Магда Программирование и отладка C/C++ приложений для микроконтроллеров ARM. — М.: ДМК Пресс, 2012. — 168 с.: ил.
- [3] © Quantum[®]*LeaPs*
- [4] http://www.state-machine.com/arm/Building_bare-metal_ARM_with_GNU.pdf Quantum[®]*LeaPs*
Building Bare-Metal ARM Systems with GNU
- [5] <http://milandr.ru/> ЗАО «ПМК Миландр»
- [6] <http://git-scm.com/book/ru> перевод: Scott Chacon **Pro Git**
- [7] <http://habrahabr.ru/post/114239/> хабра: Quantum[®]*LeaPs* QP и диаграммы состояний в UML
- [8] <http://www.state-machine.com/> Quantum[®]*LeaPs* State Machines & Tools
- [9] <http://makesystem.net/?p=988> Изучаем ARM. Собираем свою IDE для ARM

[10] <http://makesystem.net/?p=2146> Изучаем ARM. Отладка ARM приложений в Eclipse IDE

[11] Львовский С.М. Набор и вёрстка в пакете L^AT_EX