

Недяк С.П., Шаропин Ю.Б.

**Лабораторный практикум по микроконтроллерам
семейства Cortex-M**

**Методическое пособие по проведению работ на отладочных
платах фирмы "Миландр"**

2017

Министерство образования и науки

**Томский университет систем управления и радиоэлектроники
(ТУСУР)**

Кафедра электронных средств автоматизации и управления (ЭСАУ)

Недяк С.П., Шаропин Ю.Б.

**Лабораторный практикум по микроконтроллерам
семейства Cortex-M**

Методическое пособие по проведению работ
на отладочных платах фирмы «Миландр»

Томск 2017

Недяк С.П., Шаропин Ю.Б.

Лабораторный практикум по микроконтроллерам семейства Cortex-M. Методическое пособие по проведению работ на отладочных платах фирмы «Миландр». - Томск: Томск. гос. ун-т систем упр. и радиоэлектро-ники, 2017. - 110 с.

Методическое пособие предназначено для студентов старших курсов, изучающих микропроцессорную технику, также оно будет полезно и всем желающим освоить программирование МК. Методическое пособие посвящено практическому изучению микроконтроллеров с архитектурой Cortex-M3. В первой части описана методика изучения ассемблера семейства процессоров Cortex-M3, во второй - методика изучения периферийных устройств микроконтроллеров фирмы «Миландр» с ядром Cortex-M3. При написании данного пособия использовалась интегрированная среда разработки IAR Embedded Workbench for ARM ver. 6.30, впоследствии добавлена информация по работе со средой разработки Keil MDK-ARM ver 5.10.

Методическое пособие написано в рамках курса *«Микропроцессорные системы автоматизации и управления»* читаемого авторами в Томском государственном университете систем управления и автоматизации на кафедре электронных средств автоматизации и управления для студентов специальности *«Автоматизация технологических процессов и производств»*.

© Недяк С.П., Шаропин Ю.Б., 2017.

Напечатано в 22:40:33, 25.04.2017 . Версия 0.1.77

Оглавление

Предисловие.....	10
Общие указания к выполнению лабораторных работ.....	12
Введение.....	13
1 Информационные ресурсы.....	14
2 Общий ход работы.....	16
3 Основные термины и определения.....	16
4 Знакомство с лабораторным инструментарием. Лабораторная № 0.....	19
4.1 Содержание работы.....	19
4.2 Краткое описание лабораторного инструментария.....	20
4.3 Меры безопасности при работе с бескорпусной отладочной платой.....	27
4.4 Контрольные вопросы.....	27
Часть I. Процессор Cortex-M3. Программирование на ассемблере.....	28
1 Когда используется ассемблер.....	28
2 Создание и компиляция первого проекта в среде IAR. Написание простейшего модуля на языке Assembler.	
Лабораторная работа № 1.....	30
2.1 Создание нового проекта.....	30
2.2 Разработка первой программы для микроконтроллера.....	32
2.3 Указания к выполнению лабораторной работы №1.....	40
3 Создание и компиляция первого проекта в среде KEIL. Написание простейшего модуля на языке Assembler.	
Лабораторная работа № 2.....	41
3.1 Введение.....	41
3.2 Создание нового проекта в среде Keil.....	41
3.3 Разработка простейшей программы для микроконтроллера.....	41
3.4 Заключение.....	46
3.5 Требования к содержанию отчёта.....	46
3.6 Контрольные вопросы.....	46
4 Интерфейс Си и ассемблера.	
Лабораторная работа № 3.....	47
4.1 Введение.....	47
4.2 Содержание работы.....	47
4.3 Обмен данными через параметры функций.....	47
4.4 Обмен данными через общую область памяти. Глобальные переменные в Си-модуле.....	55
4.5 Обмен данными через общую область памяти. Глобальные переменные в ассемблерном модуле.....	60
4.6 Заключение.....	62
4.7 Контрольные вопросы.....	63
5 Исследование битовых полей машинного кода с помощью дизассемблера.	
Лабораторная работа № 4.....	64
5.1 Введение.....	64
5.2 Содержание работы.....	65
5.3 Выполнение работы.....	65
5.4 Заключение.....	73
5.5 Содержание отчёта по лабораторной работе.....	73

6 Исследование условного исполнения группы команд. Лабораторная работа № 5.....	74
6.1 Введение.....	74
6.2 Содержание работы.....	75
6.3 Выполнение работы.....	75
6.4 О побочных возможностях внутрисхемной отладки.....	78
6.5 Вопросы для самопроверки.....	78
6.6 Содержание отчёта по лабораторной работе.....	78
7 Макросредства языка Assembler. Лабораторная работа № 6.....	79
7.1 Введение.....	79
7.2 Содержание работы.....	80
7.3 Выполнение работы.....	81
7.4 Контрольные вопросы.....	84
7.5 Содержание отчёта по лабораторной работе.....	84
8 Задачи для любителей поупражнять свои мозги.....	85
9 Литература.....	85
Часть II. Ввод-вывод в МК «Миландр».....	86
1 Общие теоретические замечания.....	86
1.1 Порядок работы с блоками ввода-вывода МК.....	86
1.2 Стандартная библиотека ввода-вывода и Стандартный программный интерфейс микроконтроллеров ARM® Cortex™.....	86
1.3 Описание демонстрационного проекта MDR32F9Qx_Demo.....	88
1.4 Общие требования к содержанию отчета.....	91
1.5 Литература.....	91
2 Пользовательский ввод-вывод информации в малых вычислительных системах. Лабораторная работа № 7	92
2.1 Ввод-вывод двоичной информации.....	92
2.2 Вывод символьной информации.....	93
2.3 Ввод информации.....	94
2.4 Задания	94
2.5 Контрольные вопросы.....	94
2.6 Литература для изучения.....	95
3 Таймеры-счетчики. Лабораторная работа № 8.....	96
3.1 Ход работы	96
3.2 Контрольные вопросы.....	96
4 Аналоговый ввод-вывод. Лабораторная работа № 9.....	97
4.1 Работа с АЦП.....	97
4.2 Работа с ЦАП.....	97
4.3 Работа с компаратором.....	98
5 Последовательный обмен данными. Лабораторная работа №10.....	99
5.1 Краткий обзор последовательных «стандартных» интерфейсов МК.....	99
5.2 Контроллер UART.....	100
5.3 Контроллер I2C.....	100
5.4 Контроллер SSP (SPI).....	101

5.5 Контроллер CAN.....	101
Оформление и документирование программного кода.....	103
1 Стиль кодирования.....	103
2 Документирование ПО. Doxygen.....	103
2.1 Литература для изучения.....	106
Для заметок, найденных ошибок, пожеланий.....	107

Предисловие

Сегодня микроконтроллеры являются неотъемлемым компонентом почти всех сфер автоматизации и средств связи. На рынке представлен достаточно широкий спектр микроконтроллеров (МК) для самых различных областей применения: от брелока для ключей до какого-либо блока управления на спутнике связи или на самолёте. Это очень бурно развивающийся сегмент рынка электронных компонентов и один из самых больших. По оценкам специалистов ТУСУРа литература по микроконтроллерам устаревает примерно за 5-6 лет¹. К сегодняшнему дню уже устарело и само пособие. Но в нём, с нашей точки зрения, есть абсолютно точная фраза, и мы позволим себе её процитировать:

«Каждый автор старается ориентировать свой труд (книгу) на определённый круг читателей. Конечно, книга – это, в какой-то мере, зеркало его научных интересов, знаний и способности методически точно и хорошо донести до читателя суть предмета, заинтересовать его изложением предметной области материала. Поэтому, видимо, нет такой единой книги или учебника, по которому можно изучить МП (микропроцессоры) или научиться разрабатывать на МП устройства и системы. Процесс изучения МП включает такие разделы знаний как «Основы электроники», «Информатика», «Программирование», «Средства автоматизации проектирования САПР» и другие, которые охватывают, порой, различные области знаний. Хорошее освоение дисциплины курса возможно только при использовании нескольких учебных пособий или книг, так как они дополняют друг друга, позволяют посмотреть на один и тот же вопрос с разных сторон.»

В задачу вуза входит подготовка специалиста, способного работать по избранной профессии долгие годы. Это будет возможно только при условии непрерывного самообразования и после окончания университета.

При всём многообразии МК и программного обеспечения к ним, есть нечто общее в освоении новых программно-аппаратных средств – это приёмы (методика) самообразования. Именно на этом сделан акцент в данном пособии.

Видимо, мы не очень ошибёмся, если скажем, что сегодня для разработчика основным источником информации является интернет. А умение отыскивать в нём необходимую информацию является одним из важнейших профессиональных качеств, которое появляется вместе с базовыми знаниями по предмету. В интернете можно найти самые свежие сведения по микроконтроллерам, а также отследить тенденции их развития. Последнее часто имеет решающее значение при разработке концепции автоматизации того или иного объекта.

Как практикующие программисты мы считаем, что методичка в ее традиционном понимании уже не отвечает современным требованиям, поскольку информация в ней должна обновляться синхронно с обновлением информации на сайтах разработчиков микроконтроллеров. В бумажном варианте сделать это невозможно.

Мы надеемся, что со временем это методическое пособие превратится в электронный, постоянно обновляемый ресурс для студентов и способных школьников. Нужен хороший букварь по микроконтроллерам, позволяющий быстро и с минимальными затратами знакомиться с достаточно нетривиальными вещами. Насколько нам известно, пока такого учебника нет.

В 2011г. фирма-разработчик микроэлектроники «Миландр» безвозмездно предоставила ТУСУРу на кафедру ЭСАУ восемь отладочных комплектов с МК архитектурой Cortex-M3 (32-разрядное RISC-ядро) и два с архитектурой TMS320C546 (16-разрядное DSP-ядро). В основном по этой причине с 2012 г. линейка микроконтроллеров фирмы "Миландр" принята в качестве базовой в учебной работе на каф. ЭСАУ и весь лабораторный практикум выполняется на отладочных комплектах фирмы "Миландр".

Перед нами ставится задача, чтобы по окончанию курса МПСАУ студент, получив в

¹ Мартышевский Ю.В., Кормилиев В.А. Лабораторный практикум по микропроцессорам. Методическое пособие по проведению работ – Томск: ТУСУР, 1998 – 123 с.

распоряжение компьютер, подключенный к интернету и отладочный комплект фирмы "Миландр", через час был готов решать любую задачу по автоматизации. То есть, он должен уметь: устанавливать и настраивать среду разработки и другое инструментальное ПО, создавать и настраивать проект, конфигурировать микроконтроллер под данную задачу, хорошо владеть приемами программирования и отладки.

При написании данного методического пособия авторам пришлось преодолеть свое убеждение о вреде методичек, которое заключается в том, что привыкнув к «разжёвыванию» материала, студент поощряется в лени, и собственно привыкает работать в таком режиме. Всего через два года никаких методичек у вас, господа студенты — будущие инженеры, не будет!

Ведение

Данное методическое пособие предназначено для выполнения лабораторных работ с использованием отладочных плат фирмы «Миландр» серии *1986BExx* и сред разработки *IAR Embedded Workbench IDE for ARM* и *Keil MDK-ARM*.

Методическое пособие написано в рамках курса «Микропроцессорные средства систем автоматизации и управления» специальности 220301 – *Автоматизация технологических процессов и производств (в приборостроении)*, но оно будет полезно всем желающим освоить программирование МК семейства Cortex-M3 независимо от производителя, и в принципе, интегрированной среды разработки. Кроме того, данное пособие окажется полезным для изучения всего семейства процессоров **Cortex-M(0,1,3,4)**, т.к. разработчик архитектуры процессора Cortex-M, британская компания **ARM** (Advanced RISC Machine), реализовало принцип программной совместимости во всем семействе Cortex-M.

Авторы искренне надеются, что студенты, получали свои оценки по базовым предметам честно, и уже умеют писать программы на языке C/C++, знают азы электроники и владеют компьютером на уровне квалифицированного пользователя. Последнее крайне важно так, как современный специалист работает в быстро меняющемся информационном окружении, и ему просто необходимо владеть навыками работы с разнообразными информационными ресурсами: WEB, FTP, SMB, SVN, почта.... Все эти ресурсы широко используются в курсе МПСАУ и коротко описаны в данном пособии.

Методическое пособие состоит из двух основных частей. В первой части изучается процессор Cortex-M3, даются примеры использования языка ассемблер, способы исследования кода с помощью встроенного дизассемблера, а также приемы написания эффективных (быстродействующих) программ. Вторая часть посвящена работе с периферийными модулями МК 1986BExx и связана с курсовым проектированием, для которого выполняется поэтапная практическая реализация элементов программного кода курсового проекта.

В разработке первой части методического пособия, кроме преподавателя Недяка Сергея Павловича, самое активное участие принимали студенты 539гр. ТУСУРа, обучавшиеся на кафедре ЭСАУ в 2012-2013гг Бушуева Анастасия, Климов Владимир, Павлов Дмитрий, Лигачев Сергей. Частично оно составлено из их отчетов по лабораторным работам.

В основу второй части, преподавателем Шаропиным Юрием Борисовичем, положен принцип: *вся информация по МК в первоисточнике, в спецификации на серию МК **spec_seriya_1986BE9x.pdf** (Версия 3.2.0 от 20.09.2012), вся теория в книгах предыдущих курсов блока электроника, блока программирование и лекций по текущему предмету, в методическом пособии только методология, некоторые базовые понятия, ссылки, описание действий, задания и вопросы.*

Уважаемый читатель, если Вы нашли в методическом пособии какие-либо ошибки, неточности или у Вас есть пожелания по улучшению содержания, просим вас написать по одному из адресов: mpsau@iit.tusur.ru или nediak.serg@yandex.ru.

*Для студентов есть уникальная возможность потренироваться в использовании системы контроля версий *Subversion* на примере редактирования и исправления ошибок в данном методическом пособии. Оно находится по адресу <svn://esau.tusur.ru/labs/Metod>. Доступ для записи одинаков для всех: *student*, *iit*. Единственная просьба оставляйте в сообщении указание на ошибку и свои фио-группу. Кроме этого методическое пособие находится в системе *Redmine* по адресу <http://esau.tusur.ru:8085/issues/700>.*

1 Информационные ресурсы

Для организации учебной и преподавательской работы с информацией на современном уровне на кафедре авторским коллективом (Шаропин Ю.Б., Алтухов Ю.А., Карелин А.Е., Недяк С.П.) внедрён ряд информационных сервисов. Выбор этих сервисов был основан на принципе: *учиться и работать с реальными, настоящими средствами и инструментами, с которыми большинству студентов предстоит встретиться через пару лет, на первом месте работы после получения диплома.*

Информационный сервер. Для хранения информации по предметам для студентов и преподавателей организован информационный сервер. В локальной сети его имя «SV2», IP-адрес в локальной сети: 192.168.77.178. Сервер поддерживает два протокола доступа: SMB (работает только в локальной сети) и FTP, который работает и в локальной сети и глобальной.

В тексте методички будем указывать только относительные пути к файлу (директории), без указания способа (протокола) доступа.

Доступ из глобальной сети ftp://esau.tusur.ru/далее_относительный_адрес. Доступ из локальной сети ftp://192.168.77.178/далее_относительный_адрес или в подсети UNDERGROUND заходить на сервер SV2 в папку «exchange». Доступ по прямому IP в файловом менеджере FAR: **net: 192.168.77.178** или **net: sv2**.

Система сопровождения проектирования и разработки призвана обеспечить: рабочее WEB-пространство для постановки и решения задач проектирования совместно со студентами, контроль над ходом выполнения работы, обмен сообщениями в режиме форума, обмен файлами, генерацию отчетов активности по проекту. В курсе используется интернет-приложение **Redmine**, работающие на кафедральном сервере. Доступ в глобальной сети: <http://esau.tusur.ru:8085>, доступ в локальной сети: <http://192.168.77.177>. Программное обеспечение свободное, исходный код открыт. Документации по использованию **Redmine** не требуется, оно не сложнее чем использование, например сайта «в контакте», интерфейс русифицирован.

Система контроля версий программного кода. Данное клиент-серверное ПО предназначено для управления версиями программного кода. Для студента оно обеспечивает: поэтапное сохранение изменений программного кода, удаленное централизованное или распределенное хранение кода, доступ к коду с любого компьютера подключенному к Интернет, возможность просмотра всей истории создания кода, что также очень полезно преподавателю и обеспечивает контроль над ходом разработки программного кода и дает возможность оценить не только результат в конце проектирования, но и весь ход работы. В курсе МПСАУ используется система **Subversion**, которая также установлена на сервере кафедры. Программное обеспечение свободное, исходный код открыт. Документация по использованию **Subversion** имеется на русском языке.

Адрес доступа svn://esau.tusur.ru/labs/<Папка_с_репозиторием>. Логин и пароль доступа к описанным ресурсам уточнять у преподавателя или администратора сети.

В таблице 1 перечислены все необходимые файлы документации по лабораторному курсу и доступ к ним с информационных ресурсов кафедры. Естественно, самую новую информацию нужно брать с сайта производителя <http://milandr.ru>.

Таблица 1. - *Техническая документация по курсу лабораторных работ*

Описание	Адрес расположения
Спецификация по МК 1986BE9x	_For_Students\MPSSAU\Milandr\Микроконтроллеры и микропроцессоры\1986\spec_seriya_1986BE9x.pdf
Описание найденных ошибок в реализации МК	Там же: <i>spec_seriya_1986BE9x_errata_2.pdf</i>
Спецификация на отладочную плату 1986BE93У	_For_Students\MPSSAU\Milandr\Отладочная плата\Hardware\Отладочная плата 1986BE93У\1986EvBrd_48_tех_описание.pdf
Принципиальные схемы на отладочную плату 1986BE93У	_For_Students\MPSSAU\Milandr\Отладочная плата\Hardware\Отладочная плата 1986BE93У\Печатная плата 1986EvBrd_48\1986EvBrd_48_схема.pdf
Спецификация на отладочную плату 1986BE91Т	_For_Students\MPSSAU\Milandr\Отладочная плата\Hardware\Отладочная плата 1986BE91Т\1986EvBrd_tех_описание.pdf
Принципиальные схемы отладочной платы 1986BE91Т	Там же: 1986BE91_dk_sch_1.pdf 1986BE91_dk_sch_2.pdf
Исходные коды по первой части ЛР	_For_Students\MPSSAU\Labs\Labs_Asm
Дополнительная информация по второй части ЛР	_For_Students\MPSSAU\Labs\Labs_IO
Стандартная библиотека ввода-вывода и примеры программ	_For_Students\MPSSAU\Milandr\Программное обеспечение\Library\Example_Projects\Examples\
Примеры программ для различных отладочных плат	MDR1901VC1\ MDR1986VE1T\ MDR1986VE3\ MDR1986VE9x\ IDE\ iar_arm keil Phyton_CodeMaster
Файлы конфигурации для различных сред разработки	
Библиотеки CMSIS, Ввода-вывода и Программная документация.	Libraries\ CMSIS\ MDR32F9Qx_StdPeriph_Driver\ MDR32F9Qx_stdperiph_lib.chm MDR32F9Qx_USB_Library.pdf
Демонстрационный проект для отладочных плат	svn://esau.tusur.ru/_labs/MDR32F9Qx_Demo
Проект мигания светодиодом	svn://esau.tusur.ru/_labs/laba0/Trunk/test1

2 Общие указания к выполнению лабораторных работ

Обобщая опыт преподавания, можно сделать некоторые наблюдения. Одно из них заключается в том, что студенты очень редко описывают (протоколируют) реальный ход выполнения работы. Любая работа, как правило, связана со значительным числом неудачных попыток. Их детальное описание как раз и составляют наибольшую ценность отчёта по лабораторной работе, поскольку в противном случае эти же самые ошибки будут повторяться следующими поколениями студентов, вашими младшими коллегами, и на их устранение будет тратиться такое же количество рабочего времени.

Второе наблюдение заключается в том, что копирование сторонних источников за последние годы стало настолько привычным и даже рутинным делом, что возврат к нормальной вузовской проверке полученных знаний поначалу воспринимается как нечто необычное. Мы в своей преподавательской практике придерживаемся консервативных позиций — учим по-старому, т. е. по-честному, и проверяем тщательно, по крайней мере, стремимся это делать. Весь драматизм событий по этой части зафиксирован в <http://esau.tusur.ru:8085/issues/802>. Выпускникам нашей кафедры рекомендуем ознакомиться, чтобы учесть ошибки предыдущих поколений.

В пособии приведены, по сути, *примеры* выполнения лабораторных работ. В процессе обучения каждый студент получает своё маленькое индивидуальное задание для проведения своей первой самостоятельной исследовательской (лабораторной) работы. Причём, задания специально недоопределены. Как показывает производственный опыт, многие проблемы, возникающие на завершающей стадии серьёзных проектов, закладываются ещё на стадии формулировки технического задания. Поэтому умение детально анализировать задачу также важно, как и умение писать программы.

2.1 Общий ход работы

1. Подготовка и допуск к лабораторной работе. Освоение теории по указанной в методическом пособии литературе, до начала лабораторной работы.
2. Запись в тетрадь вопросов по теории. Обсуждение непонятных вопросов с преподавателем и коллегами.
3. Освоение материала из спецификации (документации) на МК.
4. Зарисовка структурных-схем изучаемых блоков.
5. Запись в тетрадь вопросов по документации. Обсуждение с преподавателем и коллегами.
6. Изучение исходных кодов на изучаемый блок МК.
7. Запись в тетрадь вопросов по исходным кодам. Обсуждение с преподавателем и коллегами.
8. Изучение работы программ на отладочной плате, в режиме пошагового выполнения и выполнения с контрольными точками. Выяснение непонятных моментов. Обсуждение с преподавателем и коллегами.
9. Запись осциллограмм, измерение временных параметров выполнения участков программ, контроль над размерами функции, стека, оперативной памяти.
10. Оформление отчета и исходных кодов.
11. Подготовка ответов на контрольные вопросы.

2.2 Оформление и документирование программного кода

Данный раздел больше относится к проектированию ПО, т.к. как именно на этом этапе Вам придется создавать свой программный код, но к этому этапу нужно быть готовым! Его нужно было бы начать с такого базового понятия, как **Парадигма программирования (ПП)**, т.к. оформление и документирование кода это не самые первые этапы в жизненном цикле программы. **ПП** — это система идей и понятий, определяющих стиль написания компьютерных программ. Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером². Для лучшего понимания этого важного вопроса отсылаю вас к «библии программиста» [1].

2.2.1 Стиль кодирования

Стиль кодирования (оформление исходного кода) или стандарт оформления кода (стандарт кодирования) (*англ. coding standards, coding convention или programming style*) — набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования. Наличие общего стиля кодирования облегчает понимание и поддержание исходного кода, написанного более чем одним программистом, а также упрощает взаимодействие нескольких человек при разработке программного обеспечения³.

Стиль кодирования обуславливается парадигмой программирования (или стилем программирования МК, мы Вам рекомендуем освоить парадигму «*автоматного программирования*» [10]. Он очень эффективен для создания программ встраиваемой вычислительной техники, где, как правило, алгоритм работы устройства **достаточно хорошо** представляется *конечным автоматом*.

Для освоения приемлемого для настоящей работы стиля кодирования, и получения желаемой оценки, вам нужно использовать источники [3,4,5].

Господа студенты, нам с вами для лучшего взаимопонимания придется пользоваться один стилем!

2.2.2 Документирование ПО - Doxygen

Doxygen — это кроссплатформенная система документирования исходных текстов, которая поддерживает C++, Си, Objective-C, Python, Java, IDL, PHP, C#, Фортран, VHDL и, частично, D.

Doxygen генерирует документацию на основе набора исходных текстов и также может быть настроен для извлечения структуры программы из недокументированных исходных кодов. Возможно составление графов зависимостей программных объектов, диаграмм классов и исходных кодов с гиперссылками.

Doxygen имеет встроенную поддержку генерации документации в формате HTML, LATEX, man, RTF и XML. Также вывод может быть легко сконвертирован в CHM, PostScript, PDF⁴.

С целью сокращения объема данного методического пособия исходные тексты здесь приведены без надлежащего оформления. Все исходные тексты программ в лабораторных работах, предъявляемых на сдачу, должны соответствовать требованиям к оформлению программной документации. Лучше если вы сразу привыкните пользоваться Doxygen-ом.

Doxygen уже широко используемый продукт. Оформление кода по правилам этой программы создает особый стиль оформления, который использован и в библиотеки CMSIS и в библиотеки стандартных периферийных устройств. На рис.1 показан хелп-файл

2 [https://ru.wikipedia.org/wiki/Парадигма программирования](https://ru.wikipedia.org/wiki/Парадигма_программирования)

3 [https://ru.wikipedia.org/wiki/Стиль кодирования](https://ru.wikipedia.org/wiki/Стиль_кодирования)

4 <https://ru.wikipedia.org/wiki/Doxygen>

MDR32F9Qx_Standard_Peripherals_Library.chm, созданный Doxygen-ом по исходным текстам CMSIS и библиотеки стандартных периферийных модулей версии 1.3.0. ф. "Миландр".

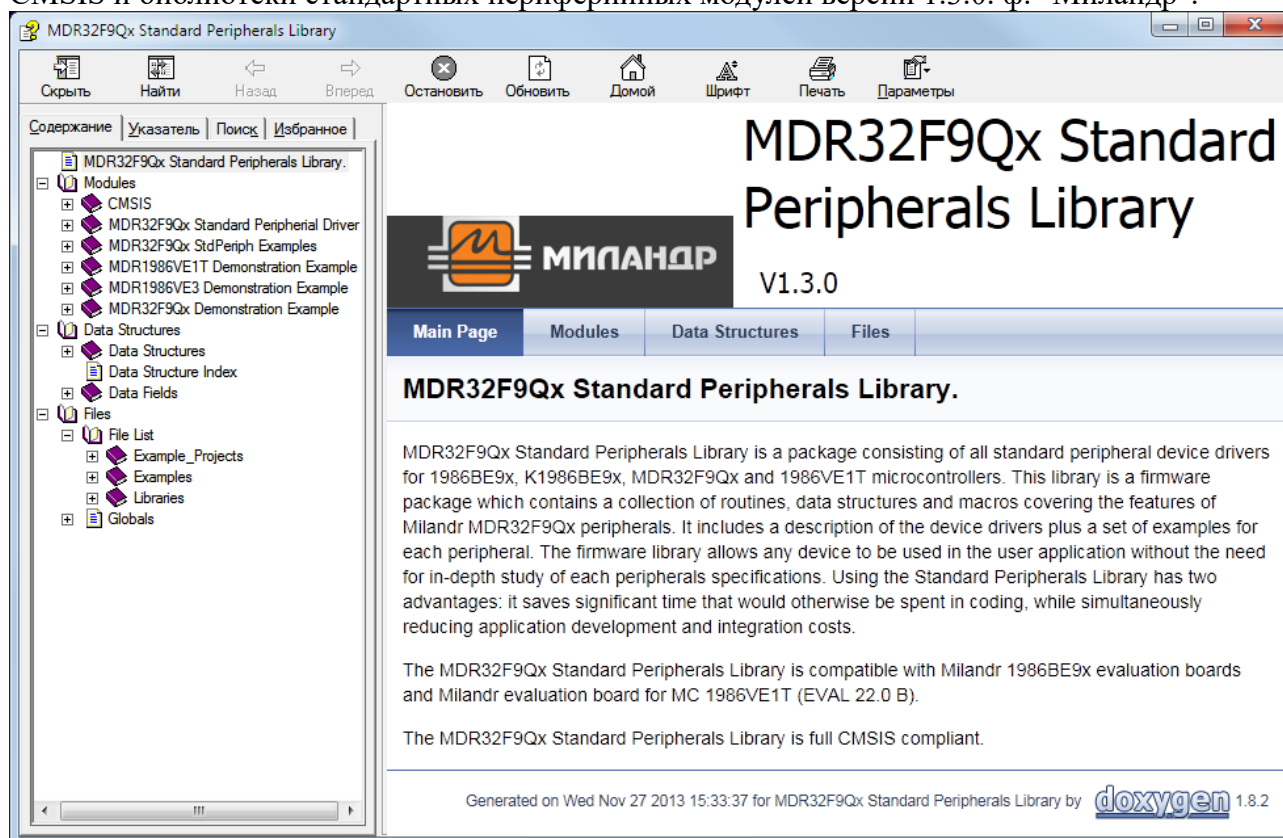


Рисунок 1 Хелп-файл к библиотеке стандартных периферийных модулей

В нашем курсе МПСАУ освоение Doxygen, и использование информационных ресурсов (SVN, Redmine), являются обязательными элементами дисциплины, и прописаны в рабочей программе.

По сгенерированному Doxygen-ом файлу легко **ознакомиться с чужим исходным кодом** и понять, как эта чужая программа устроена. Кто сталкивался с подобной задачей, тот знает, что это далеко не просто, особенно, если программа большая.

Заметьте, что все комментарии в описанных библиотечных модулях на английском языке. И это неспроста: по поводу выбора языка для комментариев есть очень поучительная фраза в учебном пособии Е.О.Степанова и С.В.Чирикова [4.] на стр.23: *«Все комментарии рекомендуется писать на английском языке. Программистов, не владеющих английским, в природе не существует, а вот компиляторы, не поддерживающие кириллицу, к сожалению, встречаются часто.»*

Ну не так уж и часто, поэтому до недавнего времени эта фраза как-то не воспринималась всерьёз. Пока однажды не встала задача адаптировать Си-код в среду MatLAB. В этой среде есть встроенный Си-компилятор, так вот он не совсем адекватно реагировал на кириллицу в комментариях. Чтобы это, в конце концов, понять было затрачено много рабочего времени.

Авторы упомянутого учебного пособия работают в том же учебном заведении, что и известный автор учебников по языку Си/Си++ Павловская Татьяна Александровна. В учебном пособии много толковых советов по выбору стиля программирования. Мы в своей работе стремимся их придерживаться. Следовательно, для наших студентов оно обязательно для ознакомления и использования.

Санкт-Петербургский государственный институт точной механики и оптики (бывший

ЛИТМО, военмех), где работают упомянутые авторы, знаменит тем, что США вводили против него санкции, когда ещё бомбили Югославию. Это явный индикатор того, что работать там до недавнего времени умели, а реальная наука, видимо, ещё не окончательно умерла.

2.3 Литература для изучения

1. Непейвода Н.Н. Скопин И.Н. **Основания программирования**. - Москва — Ижевск: Институт компьютерных исследований, - 2003, 868 стр.
2. Если Вы программируете на C\C++, в основе Вашего стиля кодирования лежит? URL: <http://habrahabr.ru/post/116819/>.
3. Сацкий Сергей. **Стандарт кодирования программ на языке C++**. <http://satsky.spb.ru/articles/CodingStandard/CodingStandard.php>.- 2005г.
4. Степанов Е.О., Чириков С.В. **Стиль программирования на C++**. - 2001 – с. URL: www.ict.edu.ru/ft/001718//style.pdf.
5. **Правила оформления исходного текста программ**. - Сайт томского железнодорожного техникума URL: http://www.tgdt.edu.ru/docs/08/recomend/text_pro.html.
6. Павловская Т.А. **C/C++. Программирование на языке высокого уровня**. — СПб.: Питер, 2001-2011. - 461 с.
7. Козаченко В.Ф. **Эффективный метод программной реализации дискретных управляющих автоматов во встроенных системах управления** // "Энергосбережение" №7/2005, - URL: www.motorcontrol.ru/publications/state_mashine.pdf.
8. Ковязин Р. **Выбор технологии программирования встроенных систем** // Компоненты и технологии. 2005 № 1 — с. 23
9. Парфенов В.В., Терехов А.Н. **RTST - технология программирования встроенных систем реального времени** // URL: http://www.math.spbu.ru/user/ant/all_articles/048_Terekhov_Parfenov_RTST.pdf
10. **Сборник статей и ссылок по теме «Автоматы»** - Национальный Исследовательский Университет Информационных Технологий, Механики и Оптики, Кафедра «Технологии программирования». URL:<http://is.ifmo.ru/progeny>.
11. Шалыто А.А. **Switch-технология. Алгоритмизация и программирование задач логического управления**. СПб.: Наука, 1998. 628 с.
12. Виктор Тимофеев. **Как писать программы без ошибок** http://www.pic24.ru/doku.php/osa/articles/encoding_without_errors. - ноябрь 2009 г.

3 Основные термины и определения

Встраиваемые вычислительные системы (Embedded systems) - **Встраиваемая система** (**встроенная система**, англ. embedded system) — специализированная микропроцессорная система управления, концепция разработки которой заключается в том, что такая система будет работать, будучи встроенной непосредственно в устройство, которым она управляет.

То есть устройство строится на базе встроенного компьютера, который в то же время не воспринимается пользователем устройства как компьютер (так как не имеет обычного монитора и клавиатуры, не отображает привычной ОС и другого ПО)⁵.

Микроконтроллер (англ. Micro Controller Unit, MCU) — микросхема, предназначенная для управления электронными устройствами. Типичный микроконтроллер сочетает на одном кристалле функции процессора и периферийных устройств, содержит ОЗУ и (или) ПЗУ. По сути, это однокристалльный компьютер, способный выполнять простые задачи⁶. Устаревшее, но используемое название «однокристалльная ЭВМ».

Контроллер. 1) Изделие для автоматизации и управления. 2) Микросхема или часть микросхемы реализующая отдельную функцию или задачу управления.

Отладочная, Оценочная (Evaluation) или демонстрационная плата (Board) — электронный модуль, как правило, в бескорпусном изготовлении, содержащий минимально необходимый набор микросхем для разработки ПО для МК.

Структурная электрическая схема — вид схемы, определяемый ЕСКД, как документ, определяющий основные функциональные части изделия, их назначения и взаимосвязи.

Интегрированная среда разработки, ИСР (англ. IDE, Integrated development environment) — система программных средств, используемая программистами для разработки программного обеспечения (ПО).

Обычно, среда разработки включает в себя:

- 1) текстовый редактор,
- 2) компилятор и/или интерпретатор,
- 3) средства автоматизации сборки,
- 4) отладчик⁷.

Сходный термин **Инструментальная среда**, используемая как инструмент разработчика ПО.

JTAG - (сокращение от англ. Joint Test Action Group; произносится «джей-таг») — название рабочей группы по разработке стандарта IEEE 1149. Позднее это сокращение стало прочно ассоциироваться с разработанным этой группой специализированным аппаратным интерфейсом на базе стандарта IEEE 1149.1. Официальное название стандарта **Standard Test Access Port and Boundary-Scan Architecture**. Интерфейс предназначен для подключения сложных цифровых микросхем или устройств уровня печатной платы к стандартной аппаратуре тестирования и отладки.

На текущий момент интерфейс стал промышленным стандартом. Практически все сколько-нибудь сложные цифровые микросхемы оснащаются этим интерфейсом для:

- выходного контроля микросхем при производстве;
- тестирования собранных печатных плат;
- прошивки микросхем с памятью;
- отладочных работ при проектировании аппаратуры и программного обеспечения.

Метод тестирования, реализованный в стандарте, получил название Boundary Scan

5 Встраиваемые вычислительные системы. Материал из Википедии. URL https://ru.wikipedia.org/wiki/Встраиваемые_вычислительные_системы

6 Микроконтроллер. Материал из Википедии. URL: <https://ru.wikipedia.org/wiki/Микроконтроллер>

7 JTAG. Материал из Википедии. URL: <https://ru.wikipedia.org/wiki/JTAG>

(граничное сканирование). Название отражает первоначальную идею процесса: в микросхеме выделяются функциональные блоки, входы которых можно отсоединить от остальной схемы, подать заданные комбинации сигналов и оценить состояние выходов каждого блока. Весь процесс производится специальными командами по интерфейсу JTAG, при этом никакого физического вмешательства не требуется. Разработан стандартный язык управления данным процессом — Boundary Scan Description Language (BSDL) [Википедия].

J-Link - это JTAG эмулятор с питанием от шины USB, поддерживающий большое количество ядер CPU. Основанный на 32-разрядном RISC CPU, он может с высокой скоростью обмениваться данными со всеми поддерживаемыми CPU. J-Link используется в десятках тысяч мест по всему миру для целей разработки и производства (программирования flash памяти). Поддержка **J-Link** интегрирована в большинство профессиональных IDE, таких как IAR, Keil, Rowley и многие другие. Наряду с OEM версиями (такими как IAR J-Link, ATMEL SAM-Ice и другими) были проданы более чем 60000 экземпляров J-Links, что позволяет говорить о J-Link как наиболее популярном эмуляторе для ARM ядер и, де-факто, промышленном стандарте⁸.

SWD - Serial Wire Debug. Двухпроводной отладочный порт, относительно новая альтернатива 20-выводному интерфейсу JTAG.

Регистры общего назначения -РОН (General Purpose Registers - **GPR**) — Программно-доступные регистры процессора для временного хранения операндов, является самой быстрой (и самой маленькой) памятью вычислительной системы.

Регистры специальных функций (Special Function Registers -**SFR**) — регистры управления и состояния периферийных модулей МК и регистры ЦП.

Центральное процессорное устройство (ЦПУ) — процессор электронный блок, либо интегральная схема (микропроцессор), исполняющая машинные инструкции (код программ), главная часть аппаратного обеспечения компьютера или программируемого логического контроллера. **Пример Cortex-M3.**

Ядро (процессорное) - часть микропроцессора, содержащая основные его функциональные блоки. **Пример CM3Core.**

4 Знакомство с лабораторным инструментарием.

Лабораторная № 0

Целью работы является ознакомление с отладочными платами и инструментами разработки для микроконтроллеров 1986BE91T, 1986BE92Y, MDR32F9Q2I, 1986BE93Y, 1901BЦ1T. Перечень отладочных плат приведен в таблице 2.

4.1 Содержание работы

Ход работы

1. Ознакомиться с техническим описанием на плату по фирменной документации.
2. Ознакомиться с принципиальной схемой на плату по фирменной документации.
3. Установить и ознакомиться со средой разработки IAR Embedded Workbench (Keil MDK-ARM).
4. Открыть проект мигания светодиодом (аналог программы «привет мир» для МК) или демонстрационный проект в среде разработки см. таблицу 1.
5. Включить отладочную плату, выданную преподавателем. Записать происходящее на ЖК-экране и светодиодах.
6. Подключить J-link к компьютеру (USB) и отладочной плате к разъему JTAG-A.
7. Скомпилировать открытый проект и «залить» его в память МК. Запустить программу

⁸ Сайт фирмы Терраэлектроника. Описание Продукта J-link компании Segger
http://www.terraelectronica.ru/catalog_info.php?ID=838&CODE=556624

на выполнение в пошаговом режиме (Step), и в режиме прогона (Run).

Содержание отчета

1. Описание платы по структурной схеме, составленной Вами из имеющейся принципиальной схемы. Указать идентификационный номер отладочной платы.
2. Примеры участков кода программы, которые непосредственно управляют устройствами отображения информации.
3. Описать происходящее при отладке в пошаговом режиме и в режиме прогона.
4. Описание ошибок сделанных при выполнении работы.
5. Выводы.

Таблица 2 - Список отладочных плат для МК "Миландр" с процессорным ядром Cortex-M3

Наименование устройств	Кол-во	Спецификация на отладочную плату
Отладочный комплект для МК 1986BE91T	4	1986EvBrd_tех_описание.pdf
Отладочный комплект для МК MDR32F9Q2I	1	1986EvBrd_64_tех_описание.pdf
Отладочная комплект для МК 1986BE93У	1	1986EvBrd_48_tех_описание.pdf
Отладочный комплект для двухъядерного (Cortex-M3 + TMS320) микроконтроллера 1901ВЦ1Т	2	Ex_1901VC1F_rev.pdf (схема принципиальная)

Документация на отладочные комплекты находится на сервере по адресу: [\ For Students\MPSSAU\Milandr\Отладочная плата\](#) или на сайте производителя <http://milandr.ru>.

4.2 Краткое описание лабораторного инструментария

4.2.1 Аппаратура

Современный минимально необходимый инструментарий программиста встраиваемых систем достаточно прост:

1. **Инструментальная машина:** персональный компьютер или ноутбук, под управлением Windows. К сожалению, описанные в этой работе IDE пока работают только под Windows. Сторонникам проекта GNU и Linux-систем просьба не огорчаться, для Вас конечно же также существует ряд программных продуктов для работы с МК ARM, например: **GNU Tools for ARM Embedded Processors** (<https://launchpad.net/gcc-arm-embedded>)⁹.
2. **Целевая плата**, для которой будет разрабатываться ПО, в нашем случае это любая из перечисленных отладочных плат. Внешний вид отладочной платы для МК 1986BE91T показан на рисунке 2.
3. **JTAG - адаптер**, который обеспечит доступ к ресурсам МК, его внешний вид показан

⁹ Описание работы со «свободным» инструментарием ждет своих авторов.

на рисунке 3.

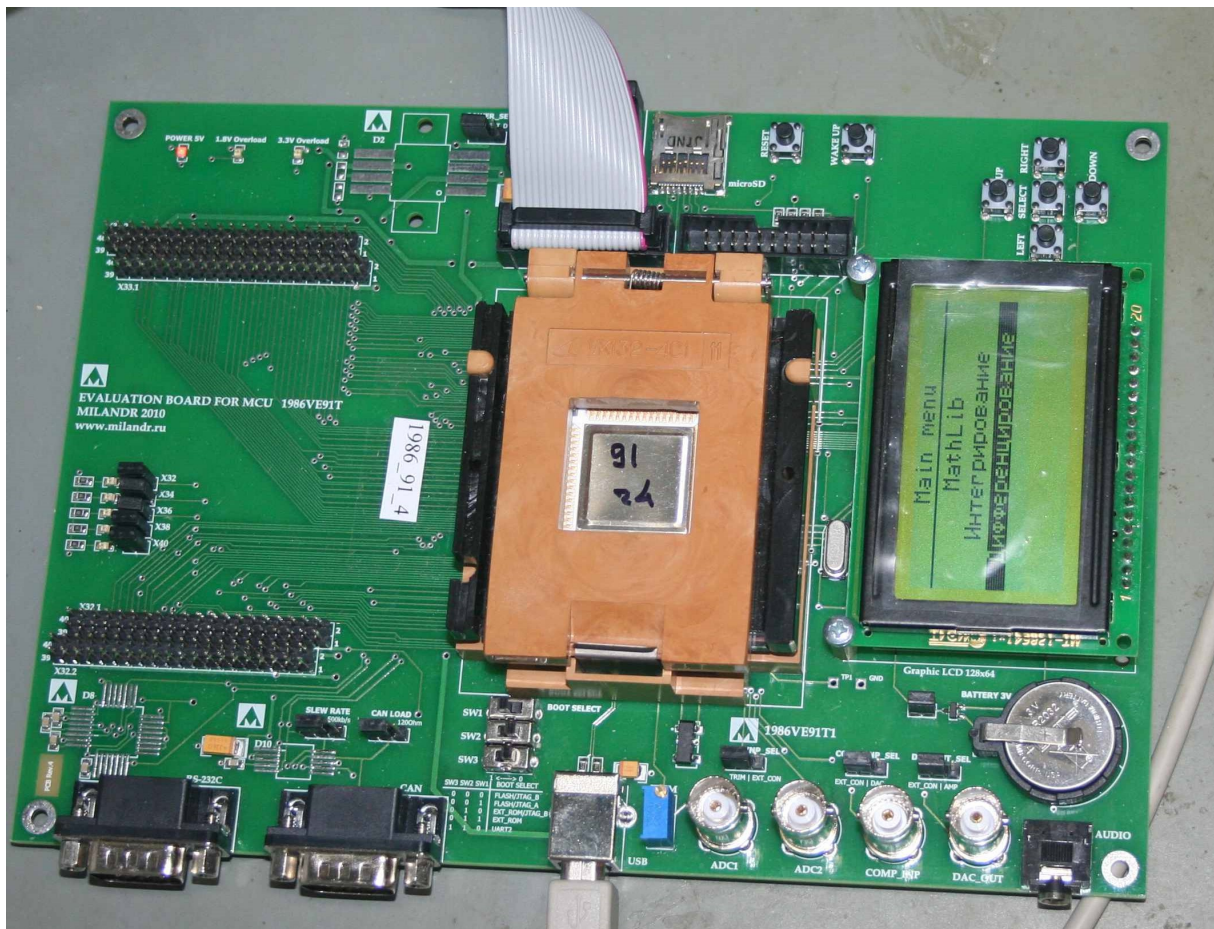


Рисунок 2 Внешний вид отладочной платы для МК 1986VE91T

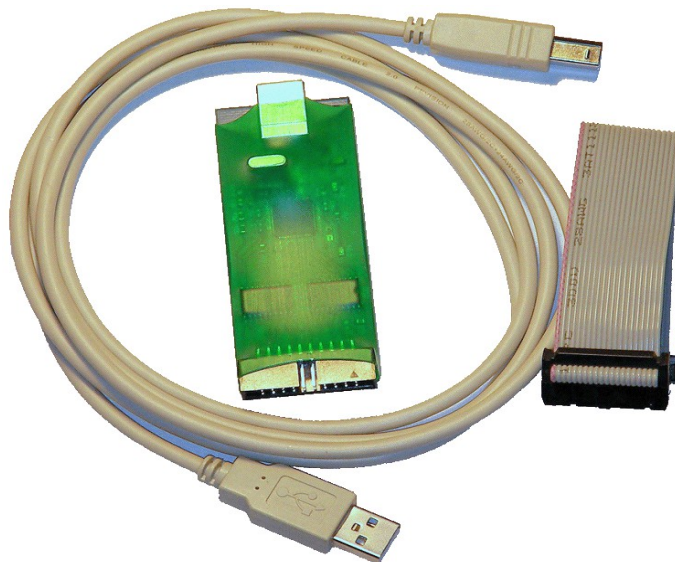


Рисунок 3— JTAG - эмулятор MT-Link аналог J-Link¹⁰

¹⁰ Фото взято с сайта starter-kit.ru

4.2.2 Среда разработки программ для МК

Для создания простейших программ, минимально необходимым набором **инструментального ПО** являются: текстовый редактор, транслятор ассемблерного кода и симулятор для отладки. И такого набора на заре развития вычислительной техники вполне хватало. Но с развитием МК, с ростом объема оперативной памяти и памяти программ и широчайшим распространением МК в различных областях техники, минимального набора стало не хватать. Для создания «больших» программ и повторного использования уже отлаженного кода, в виде библиотек, появились редакторы связей (линковщики, компановщики), появились отладчики, и более совершенные трансляторы и, наконец, стало возможным и обоснованным применение компиляторов (примерно с середины 90-х прошлого века), появился диалект Embedded C/C++. И все эти средства для удобства использования стали объединять в один программный продукт - так появились интегрированные среды разработки (IDE) и целая отрасль разработки ПО. Одними из лидеров в этой области являются фирмы IAR Systems (iar.com) и Keil (keil.com).

Немецкая фирма Keil разрабатывает и поставляет среды разработки для платформ: ARM, 8085, 251, C166, JTAG-отладчики и отладочные платы для них. Нас интересует платформа ARM, состав и варианты поставки среды разработки Keil-MDK представлены в таблице 3.

Таблица 3 Состав и версии среды разработки Keil- MDK¹¹

Компоненты IDE	Microcontroller Development Kit Editions				
	MDK-Professional	MDK-Standard	MDK-CortexM	MDK-Basic	MDK-Lite
µVision IDE	+	+	+	+	+
RealView C/C++ Compiler	+	+	+	256KB	32KB
RealView Macro Assembler	+	+	+	+	+
RealView Linker	+	+	+	256KB	32KB
RealView Utilities	+	+	+	+	+
ARM Standard Run-Time Library	+	+	+	+	+
ARM MicroLib Run-Time Library	+	+	+	+	+
µVision Debugger	+	+	+	256KB	32KB
Поддерживаемые процессоры ARM					
Cortex-M0, M0+, M1, M3, M4	+	+	+	+	+
Cortex-R4	+	+	-	+	+
ARM7, ARM9	+	+	-	+	+
SecurCore	+	+	-	+	+

¹¹ Таблица взята из «желпа» к MDK-Lite Ver.5.10.0. ARM Development Tools rev. from august 2013.

В нашей работе мы будем пользоваться оценочными версиями ПО, которые предоставляются разработчиками бесплатно. Эти версии имеют определенные ограничения. Для версии MDK-Lite <https://www.keil.com/demo/eval/arm.htm>: ограничение на размер компилируемого и компоновемого кода 32 килобайта.

Компания IAR, также бесплатно предлагает для ознакомления две версии своего продукта: версию evolution с полным функционалом и ограничением времени использования 30 дней и версию kickstart (в имени дистрибутива есть буквы KS) с ограничением на размер генерируемого исполняемого кода -32 кбайт), но без ограничения времени использования (<http://supp.iar.com/Download/SW/?item=EWARM-EVAL>).

Для студенческих нужд размера кода в 32КБ более чем достаточно. Поэтому мы рекомендуем загрузить и установить именно эту версию.

Следует отметить, что компания Keil является официальным партнером ARM, а сама Keil-MDK является совместной разработкой Keil и ARM (<http://www.arm.com/products/tools/software-tools/index.php>). Так ядро IDE (компилятор, линковщик, ассемблер и ряд утилит) собственная разработка ARM, от Keil-а только оболочка (µVision IDE) и отладчик.

Шведская компания IAR на рынке инструментария для встраиваемых систем с 1983 года. Она охватывает огромный перечень вычислительных платформ (8051, AVR, H8, MSP430, SuperH, MAXQ, STM8, ColdFire...). Практически все поставляемое ПО, является собственной разработкой этой компании. Именно такой подход позволил данной компании охватить чуть ли ни все вычислительные платформы для встраиваемых систем.

В нашей работе мы будем пользоваться IAR Embedded Workbench for ARM ver 6.30. Состав IDE IAR показан на рисунке 4 в виде блок-схемы создания ПО для встраиваемых систем.

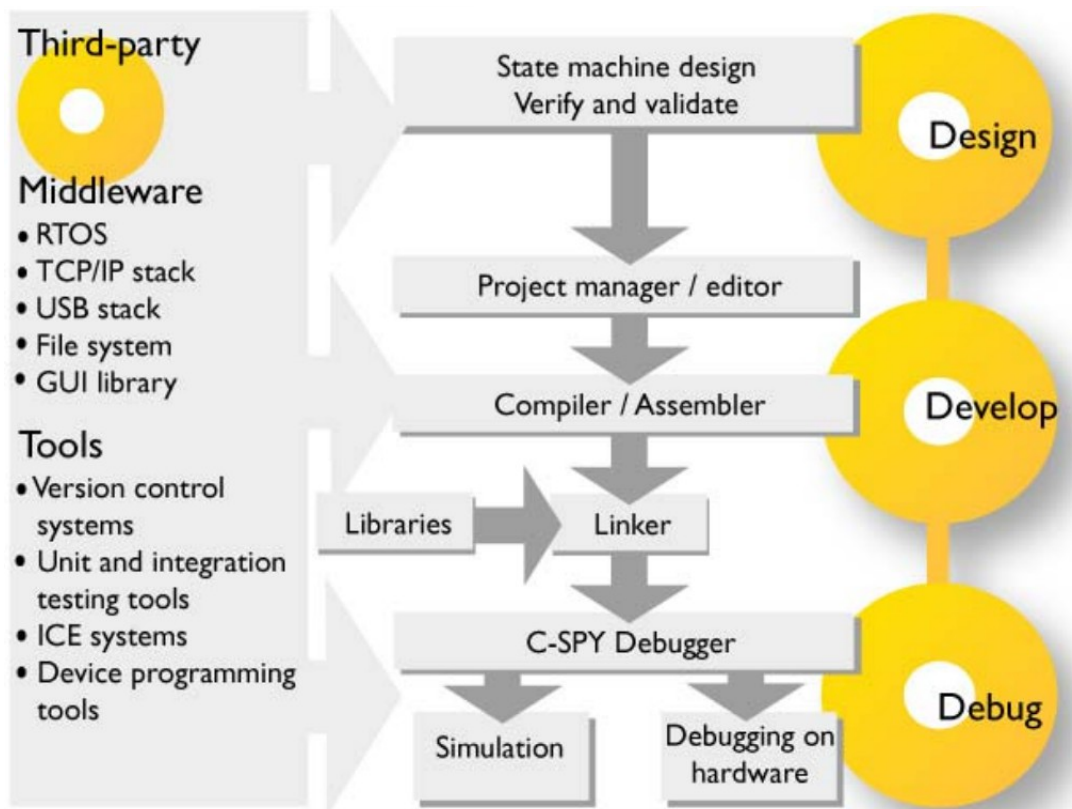


Рисунок 4 Состав IDE IAR и ход разработки ПО¹²

12 Getting Started with IAR Embedded Workbench Third edition: April 2012. Доступ IAR IDE-> Help → Getting Started... Имя файла: EW_GettingStarted.ENU.pdf.

Данный рисунок очень информативен и познавателен. Процесс создания ПО для МК начинается с проектирования (Design). На этом этапе принимаются основные решения — разрабатывается *архитектура* ПО, принимаются решения о составе и взаимосвязи компонентов, об использовании готовых библиотек сторонних организаций (Third-party), о применении межплатформенного программного обеспечения (middleware): операционные системы реального времени (RTOS), стеков протоколов (TCP-IP, USB,...), файловых систем, графических библиотек (GUI) и т. д. Также на этом этапе выбираются средства разработки. Исходными данными для выполнения этого этапа является техническое задание.

Следующий этап — разработка (develop), воплощение в коде проекта ПО. Правильнее, этот этап назвать кодирование. На этом этапе используется менеджер (конфигуратор) проекта, редактор, компилятор, ассемблер и компоновщик, который и формирует программу («прошивку») для МК.

И последний этап изображенный на рисунке 4 - отладка, он выполняется с помощью отладчика C-SPY и может быть выполнен как в режиме симулирования аппаратных ресурсов, так и непосредственно в целевом устройстве, в микроконтроллере.

В профессиональной деятельности программиста, представление о которой вы можете получить из стандарта на профессию¹³, кроме IDE в обязательном порядке применяется и ряд других программных и аппаратных инструментов: системы контроля версий, и коллективного ведения проектов (Redmine, ICE Systems), осциллографы и логические анализаторы... Также мы настоятельно рекомендуем посмотреть на требования к программистам-разработчикам МК-систем (или встраиваемых систем), публикуемых в объявлениях о вакансиях.

Описанный рисунок значительно идеализирует процесс разработки ПО. В действительности, он не такой простой и «линейный», содержит ряд обратных связей, т. е. является итеративным. Дорисуйте эти обратные связи в своем отчете по ЛР.

4.2.2.1 Keil-MDK vs IAR Embedded

Коротко описывая данные среды разработки, мы невольно начинаем их сравнивать — так, что же лучше? На изучение чего в первую очередь стоит тратить время?

На второй вопрос мы ответим так: человек, основательно изучив любую IDE, легко (в течении пару часов) освоит любую другую. Основательно — это значит, нужно знать и понимать назначение и взаимосвязи всех компонентов среды разработки.

Ответ на первый вопрос потребует от нас хотя бы беглого объективного сравнительного анализа. Для этого составим таблицу 4. с основными характеристиками и компонентами сред разработки.

Таблица 4 Краткое сравнение IAR и Keil

Характеристика	IAR Embedded	Keil-MDK
Языки	C/C++	C/C++
Диалекты (стандарты языка)	C89 C99 Embedded C++ Extended Embedded C++	C90 C99 C++ 2003
Оптимизация кода	+	+
Стандартные библиотеки языка C	+	+
Контроль размера стека	+	?

13 Профессиональный стандарт «Программист». - 2013 - 19 с. [Электронный ресурс] URL: <http://www.tusur.ru/export/sites/ru.tusur.new/ru/education/documents/federal/13.15.doc>.

Характеристика	IAR Embedded	Keil-MDK
Поддержка RTOS	12 разных RTOS (CMX-RTX, FreeRTOS, SafeRTOS ...) ¹⁴	Keil-RTX
Поддержка правил MISRA (Motor Industry Software Reliability Association)	MISRA-C 1998 MISRA-C 2004	В стороннем пакете PC-Lint
Статический анализ кода	В сторонних пакетах.	В сторонних пакетах PC-Lint и Parasoft C++test.
Анализ кода в процессе его выполнения (Динамический анализ кода)	Пакет C-RUN Начиная с версии 7.20	-
Сертификация и проверка соответствию стандартам	Сертификация на безопасность по стандартам IEC 61508 и ISO 26262 экспертной организацией TÜV SÜD.	Проверка соответствия стандарту C компанией Plum Hall. Проверка расширений языка, качество кода и ядра RTOS без внешнего аудита.
Поддержка МК "Миландр"	-	Не полная

В этом сравнении мы затронули очень важный вопрос о **надежности ПО** для встраиваемых систем, т.к. микроконтроллеры очень часто используются в ответственных и особо **критичных для жизни** областях (авиация, космос, медицина, управление техпроцессами...). Этому вопросу посвящены другие дисциплины «**Встраиваемые системы для ответственных применений**» и «**Встраиваемые системы для транспорта**», изучаемые в следующем семестре.

Процесс установки рассмотренных IDE не представляет трудностей: запустите скачанный exe-файл на выполнение и следуйте подсказкам программы-инсталлятора. Главное запомнить в какую директорию Вы поставили среду разработки — в дальнейшем это пригодится. Установленная IDE требует настройки с целью возможности работы с МК Миландр. Это связано с тем, что в дистрибутивы этих сред разработки пока, что не входит поддержка МК ф. "Миландр".

В следующих разделах объяснено, как настроить среды разработки, чтобы они «видели» и могли работать с МК производства "Миландр".

4.2.2.2 Настройка Keil-MDK

В таблице 4 указана неполная поддержка МК-Миландр для Keil. На сайте Keil (<http://www.keil.com/dd/chip/5749.htm>) мы найдем упоминание МК-Миландр, но не найдем необходимых файлов и эти файлы не включены в дистрибутив. Все необходимые файлы для работы с конкретным кристаллом объединены в один архив и доступны на сайте производителя МК по адресу <http://ic.milandr.ru/soft/>, находим **Software pack для Keil MDK 5** (MDR32F9Qx, MDR1986VE1T, MDR1986VE3T) и скачиваем архив. Далее разархивируйте его в директорию "Папка где находится Keil"\ARM\Pack\Download\. В ней появится одноименный файл с расширением **.pack**, после инсталляции Keil, оно ассоциировано с IDE Keil и при запуске файла **.pack** появиться окно интерфейса «распаковщика» рис.5.

14 Полный список плагинов RTOS: <http://www.iar.com/Products/IAR-Embedded-Workbench/ARM>

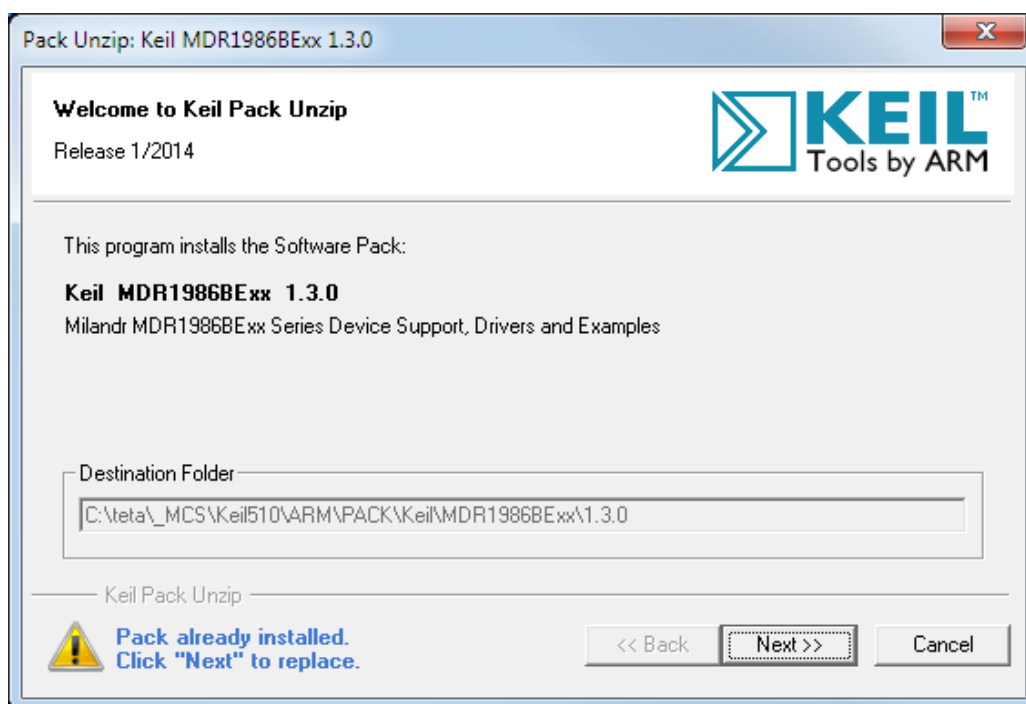


Рисунок 5 Разархивирование файлов для работы с МК-Миландр.

После этого в папке \ARM\Pack\Keil появятся файлы описания МК для среды Keil: MDR1986BExx\1.3.0\

Config\MDR32F9Qx_config.h -

Example_Projects\ - демо-проект для различных отладочных плат

Examples\ примеры работы с различными устройствами ввода-вывода МК.

IDE\ файлы необходимые для работы среды с МК-Миландр.

Libraries\ библиотеки CMSIS и ввода-вывода (см. Раздел 1.2 во второй части пособия).

Теперь можно создавать новый проект или открывать готовый для МК-Миландр.

4.2.2.3 Настройка IAR

После установки среды разработки *IAR Embedded Workbench* её также нужно адаптировать для программирования микроконтроллеров фирмы "Миландр". Для этого необходимо на сайте <http://ic.milandr.ru/soft/> скачать архив **mdr_spl_v1.4.1.pack** библиотеки стандартных периферийных устройств. После распаковки архива содержимое папки **mdr_spl_v1.4.1\IDE\iar_arm\arm** нужно скопировать в папку **arm**, где установлена *IAR*:

/src/flashloadert/Milandr - исходники загрузчика флеш;

/inc/Milandr - заголовочные файлы для 1986BE9x;

/examples/Milandr/coremark_iar - пример программы CoreMark;

/config/linker/Milandr — файл настройки линкера;

/config/flashloader/Milandr - скомпилированный загрузчик флеш-памяти с настройками;

/config/devices/Milandr - описание микроконтроллера (МК) для среды IAR.

Теперь среда разработки IAR готова для создания первой программы для микроконтроллеров производства "Миландр".

4.3 Меры безопасности при работе с бескорпусной отладочной платой

Приступая к работе с открытой печатной платой, где микросхемы и проводники, расположенные на печатной плате не защищены корпусом (open frame), нужно иметь ввиду опасность выхода из строя микросхем от статического электричества (ESD), которое почти всегда накапливается на теле человека.

Конечно ESD- защита это целая наука¹⁵ мы здесь ограничимся рядом практических рекомендаций, которые позволят безопасно эксплуатировать открытую электронику:

1. Старайтесь на лабораторные занятия одеваться «безопасно» - в хлопок.
2. Перед началом непосредственной работы с отладочной платой подойдите к батарее отопления и прикоснитесь к ней рукой для снятия статического заряда с Вашего тела. Также можно снимать заряд через заземленное оборудование, например осциллограф, прикоснувшись к «земленной» клемме рукой.

Кроме поражения от ESD, нужно иметь ввиду возможные механические повреждения, при небрежном отношении к оборудованию... будьте внимательны и аккуратны !!

Другая очень вероятная опасность для электроники это не заземленный компьютер, на корпусе которого, будет переменное напряжение 110В или не заземленный осциллограф, на корпусе которого может быть такой же потенциал. Это напряжение опасно не только для электронных компонент расположенных на плате, **но и для человека!!** Кроме не заземленного оборудования, другой опасности для человека при выполнении лабораторных работ нет.

Вопрос студентам: *откуда этот потенциал на корпусах современного оборудования?*

**ПРЕЖДЕ ЧЕМ РАБОТАТЬ С ОСЦИЛЛОГРАФОМ И ОТЛАДОЧНОЙ ПЛАТОЙ
НУЖНО УБИДИТЬСЯ ЧТО ОБА УСТРОЙСТВА ПОДКЛЮЧЕНЫ К ЗЕМЛЕ! И
ПРОВЕРИТЬ ЗАЗЕМЛЕНИЕ ИНСТРУМЕНТАЛЬНОГО КОМПЬЮТЕРА!**

4.4 Контрольные вопросы

1. Дайте определение понятию IDE.
2. Что такое компилятор и чем он отличается от транслятора?
3. Какие функции в IDE выполняет линковщик?
4. Что такое электрическая структурная схема.
5. Дорисуйте процесс разработки ПО (рис. 4) с учетом итеративности связей в этом процессе.
6. Откуда появляется потенциал переменного напряжения амплитудой 110В?
7. Какие ошибки возникают в процессе создания ПО...
8. В таблицы сравнения 4 мы пропустили некоторые очень важные вещи, какие?

¹⁵ Дмитрий Трегубов. *Новые российские стандарты в области ESD-защиты* // Компоненты и технологии. №4, 2010 – с. 12- 14. [Электронный ресурс]: http://www.kit-e.ru/assets/files/pdf/2010_04_12.pdf

Часть I. Процессор Cortex-M3. Программирование на ассемблере

Первая часть посвящена изучению процессора Cortex-M3, его системы команд, примерам использования языка ассемблер, способам исследования кода с помощью встроенного в IDE дизассемблера, а также приемам написания эффективных (быстродействующих) программ.

1 Когда используется ассемблер

Ассемблер – самый древний язык программирования. Первые программы для первых цифровых вычислительных машин писались в машинных кодах. Вместо двоичной системы счисления чаще пользуются записью программы в шестнадцатиричной системе счисления, что в прочем кардинально существа дела это не меняет. Всё равно написание программы в машинных кодах было очень трудоёмким занятием, поэтому машинный код решили заменить на представление инструкций мнемокомандами, несущими какую-то смысловую нагрузку. Имя изобретателя ассемблера, как и имя изобретателя колеса, до сих пор неизвестно, так как это изобретение было исторической необходимостью и наверняка появилось одновременно в разных институтах и лабораториях, как и колесо в разных уголках планеты.

Если отбросить многие несущественные детали, то язык ассемблер – по сути, так и остался системой записи машинного кода в виде мнемокоманд. В так называемом листинге, специальном текстовом файле с расширением `lst`, можно увидеть таблицу, где в первой колонке располагаются адреса ячеек памяти, во второй машинный код команд, т.е. содержимое этих ячеек, а в третьей символьное обозначение команд – мнемокод – исходный текст на языке ассемблер.

Чем же объясняется такая невероятная живучесть самого древнего языка программирования? Только одним – его непревзойдённой эффективностью по сравнению с другими языками, когда речь идёт о скорости работы программы и компактности её машинного кода. Главный недостаток ассемблера – это по-прежнему достаточно высокая трудоёмкость написания программ и относительно длительное время овладения навыками программирования на этом языке, точнее языках. Набор или система команд для одного ядра могут существенно отличаться от другого набора, например, система команд семейства МК MCS51 очень сильно отличается от системы команд AVR, хотя функционально они решают одинаковые задачи.

Канули в Лету времена, когда все программы для микроконтроллеров целиком писались на ассемблере. Сегодня большая часть исходного кода пишется на Си/Си++ и только ключевые, критичные к скорости, участки кода на ассемблере. Видимо, не лишне заметить, что компиляторы с языков верхнего уровня пишутся, как правило, грамотными специалистами и машинный код может получаться вполне приемлемым. Ассемблер лишь предоставляет возможность написать эффективный код, но не обязательно, что эта возможность будет в полной мере использована. Очень многое зависит от уровня квалификации программиста, от степени его знакомства с архитектурой данного микроконтроллера, от его творческих способностей.

После предварительного знакомства с системой команд начинать изучение программирования на ассемблере лучше всего с изучения опыта (приемов программирования) разработчиков компиляторов языка Си/Си++. Используя дизассемблер, можно проследить общую логику построения программы. Такой подход полезен ещё и для овладения навыками эффективного программирования на самом языке Си. Очень часто, с виду элегантная программа на Си, компилируется в громоздкий и нерациональный машинный код. Зная особенности конкретного компилятора Си, можно избегать этих недостатков и создавать вполне эффективный код, даже не используя ассемблер. Как показывает опыт, время, потраченное на изучение ассемблера именно в таком ключе, потом с

лихвой компенсируется сокращением времени отладки и сопровождения программ.

И последний немаловажный аргумент в пользу изучения ассемблера. Компиляторы пишут хотя и грамотные, но всё-таки люди. А людям свойственно ошибаться. Хотя и редко, но и здесь ошибки иногда случаются. Знание ассемблера даёт инструмент для исправления подобных ошибок, в противном случае программист оказывается в такой ситуации почти безоружным.

Есть и такое высказывание одного из авторов учебника по ассемблеру [6]:

«...ассемблер может понадобиться для оптимизации кода программ, написания драйверов, трансляторов, программирования некоторых внешних устройств и т.д. Для себя я, однако, имею другой ответ: программирование на ассемблере даёт ощущение власти над компьютером, а жажда власти – один из сильнейших инстинктов человека»

Уж не знаем, насколько можно разделить это эмоциональное высказывание, но то, что от степени знания компьютера (микроконтроллера), от умения его программировать будет зависеть здоровье и жизнь людей, целостность спутников и пр. – это совершенно точно, проверено экспериментально [5.].

Диплом выпускника ТУСУРа даёт право заниматься этой чрезвычайно трудной, ответственной и интересной работой. Поэтому внутренне нужно настроиться на упорную учёбу и впоследствии на тщательную проверку полученных знаний. МПСАУ – сложный курс.

Итак, поехали ...

2 Создание и компиляция первого проекта в среде IAR. Написание простейшего модуля на языке Assembler. Лабораторная работа № 1

Целью данной работы является ознакомление с одной из популярнейших сред разработки программ для микроконтроллеров IAR¹⁶ Embedded Workbench IDE.

2.1 Создание нового проекта

Создадим новый проект Project=>Create New Project.

Выбираем шаблон проекта(ProjectTemplates): **C-main**

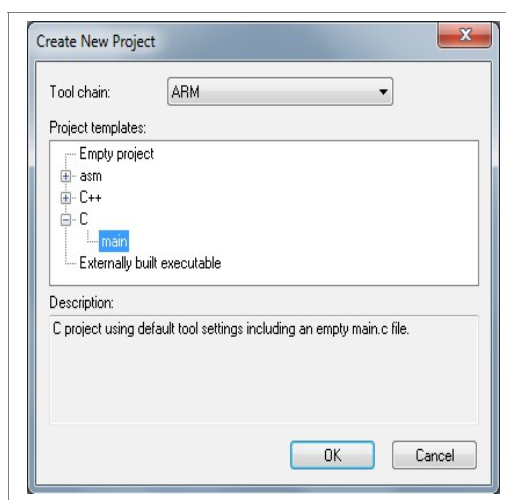


Рисунок 1 Создание нового проекта.

Сохранив проект под каким-либо именем, далее в свойствах проекта выбираем модель микроконтроллера Milandr 1986BE9x (Рис 1.2). Для этого правой кнопкой мыши щелкаем по нашему проекту, выбираем Options... и в GeneralOption

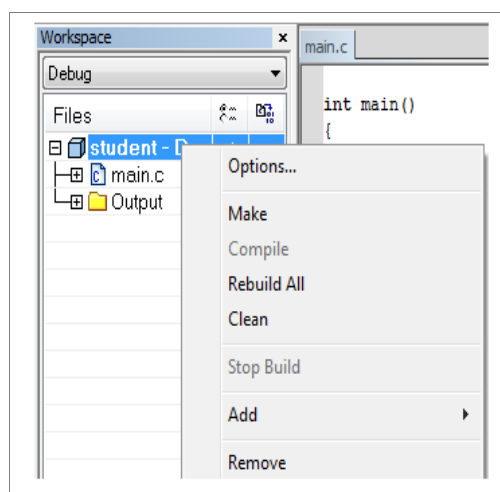


Рисунок 2 Выбор свойств проекта

на закладке Target (Рис 1.3) выберем модель микроконтроллера **Milandr 1986BE9x**

¹⁶ IAR Systems (www.iar.com) - старейшая компания, основанная в 1983 г. занимающаяся средствами разработки ПО для встраиваемых вычислительных систем.

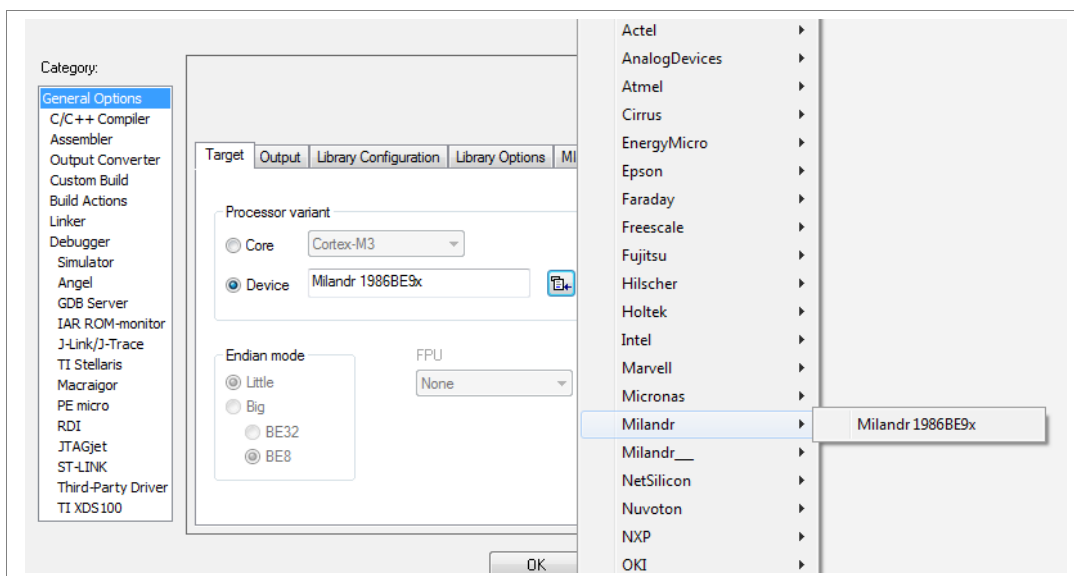


Рисунок 3. Выбор кристалла

В случае, если в списке отсутствует Milandr, попробуйте проверить файлы, которые скопировали в корневую директорию среды разработки IAR. Если, тем не менее, проблему устранить не удалось, т.е. после перезапуска IAR Milandr по-прежнему отсутствует в списке, временно можно воспользоваться готовыми примерами проектов IAR для ARM. Для первого знакомства со средой разработки этого будет вполне достаточно.

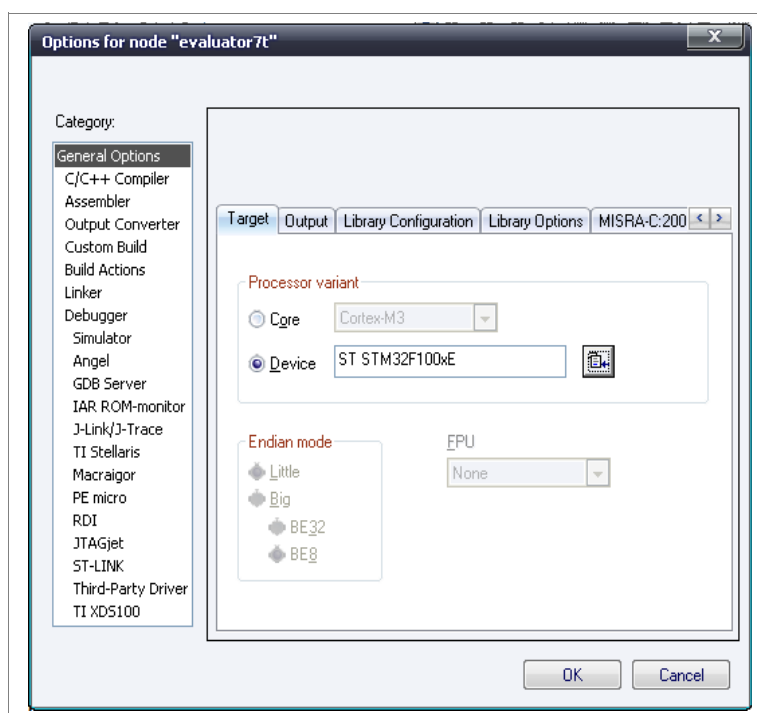


Рисунок 4. Выбор кристалла

Здесь, рис. 4, открыт готовый проект evaluator7t из среды разработки IAR и выбран микроконтроллер ST STM32F100xE. Поскольку у обоих микроконтроллеров одно и то же ядро: Cortex-M3, то наша программа, отлаженная в симуляторе для ST STM32F100xE, будет работать и на микроконтроллере Milandr 1986BE9x.

Чтобы выбрать в качестве средства отладки Simulator нужно щёлкнуть мышью по категории Debugger и в открывшемся окне выбрать Simulator.

Далее, щелкнув мышью по категории C/C++ Compiler, выбрать вкладку Optimization и поставить отметку None.

2.2 Разработка первой программы для микроконтроллера

Исходный текст функции main() на языке C:

В открытом окне main введем следующий код

```
extern int func1_c(void);           // прототип функции на языке C
//extern int func1_asm(void);       // прототип функции на языке ассемблер
int main (void)
{
    int k;
    k = func1_c();
    // k = func1_asm();
return k;
} // main
```

Для того чтобы создать новый файл с именем func1_c.c выбираем File=>New=>File или Ctrl+N

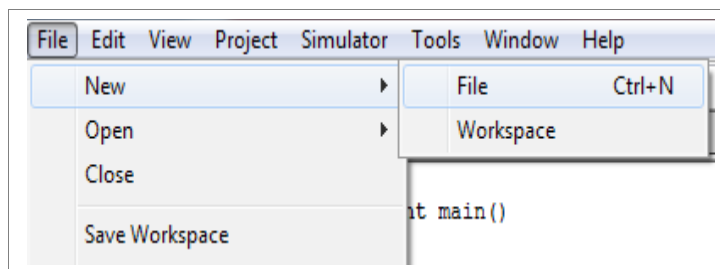


Рисунок 5 Создание нового файла для редактирования

Исходный текст функции func1_c на языке C:

В окне редактирования файла введем код

```
int func1_c(void) {
    int i = 2;
    int k = 5;
    int x = k;
    x = x + i;
return x;
}
```

Сохраним данный файл как *func1_c.c* и добавим его к проекту

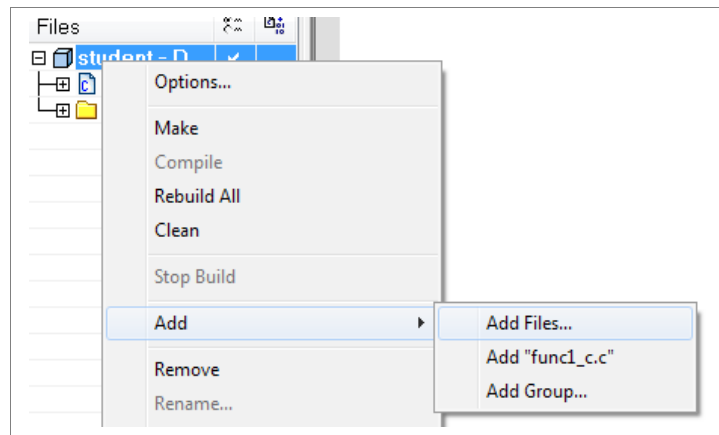



Рисунок 6 Добавление нового файла к проекту

Запустим проект, нажав на кнопку . По умолчанию откроется окно дизассемблера (Disassembly).

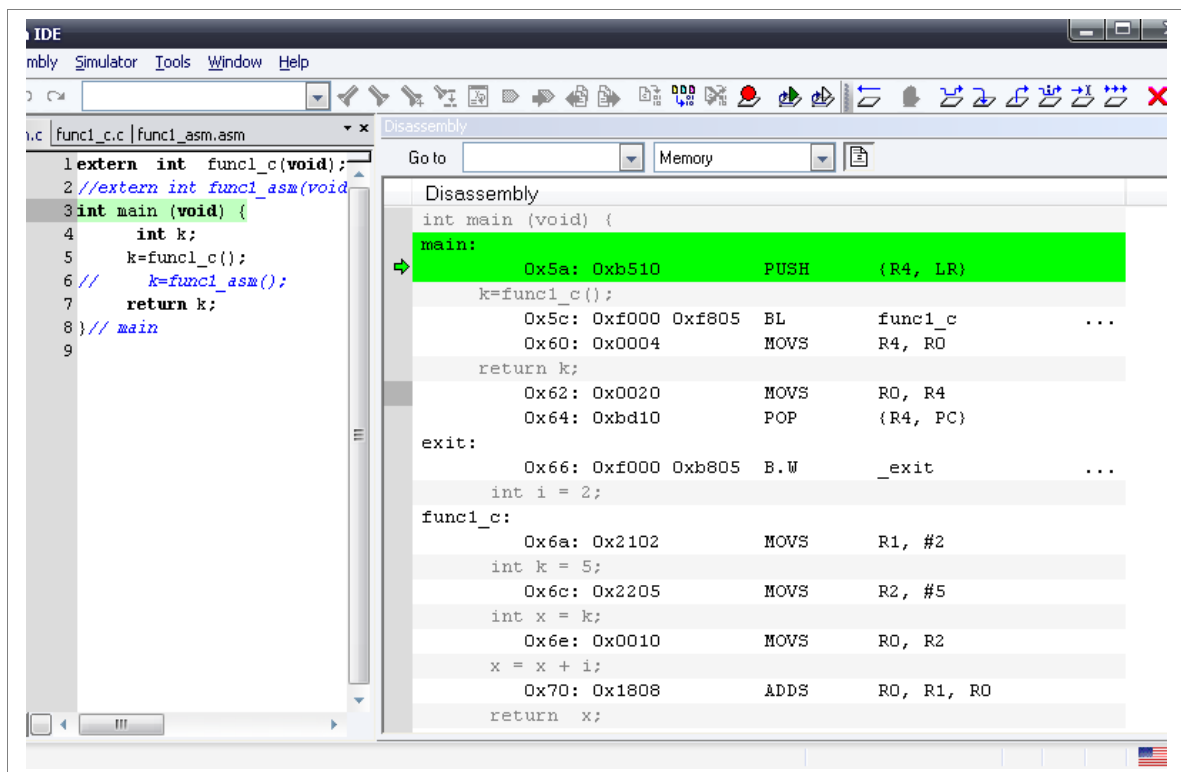


Рисунок 7 Режим отладки IAR, окно Disassembly

Внимательно рассмотрим это окно. В нем мы можем увидеть текст только что набранной нами программы. Сначала идет строка нашего кода на языке Си и далее несколько строк – её эквивалент на языке ассемблер, точнее сказать, фрагмент листинга. Как мы уже говорили, листинг состоит из 3-х колонок: в первой колонке располагается адрес; во второй машинный код команды и в третьей её мнемокод – исходный текст на ассемблере.

Попробуем написать свою функцию на этом замечательном древнем языке программирования, скопируем исходный текст функции func1_c с дизассемблера. Здесь мы,

лишь, поменяем метку `func1_c` на метку `func1_asm`. Откроем новый файл и, глядя на рис.7, наберём в нём следующий текст:

`func1_asm:`



```
MOVS R1, #2

MOVS R2, #5

MOVS R0, R2

ADDS R0, R1, R0

BX LR
```

Сохраним этот файл под именем `func1_asm.s` или `func1_asm.asm` и добавим его к проекту. Прежде чем добавлять, нужно прервать сеанс отладки, нажав на кнопку . Сейчас мы попытаемся откомпилировать наш проект. Снова нажимаем на кнопку .

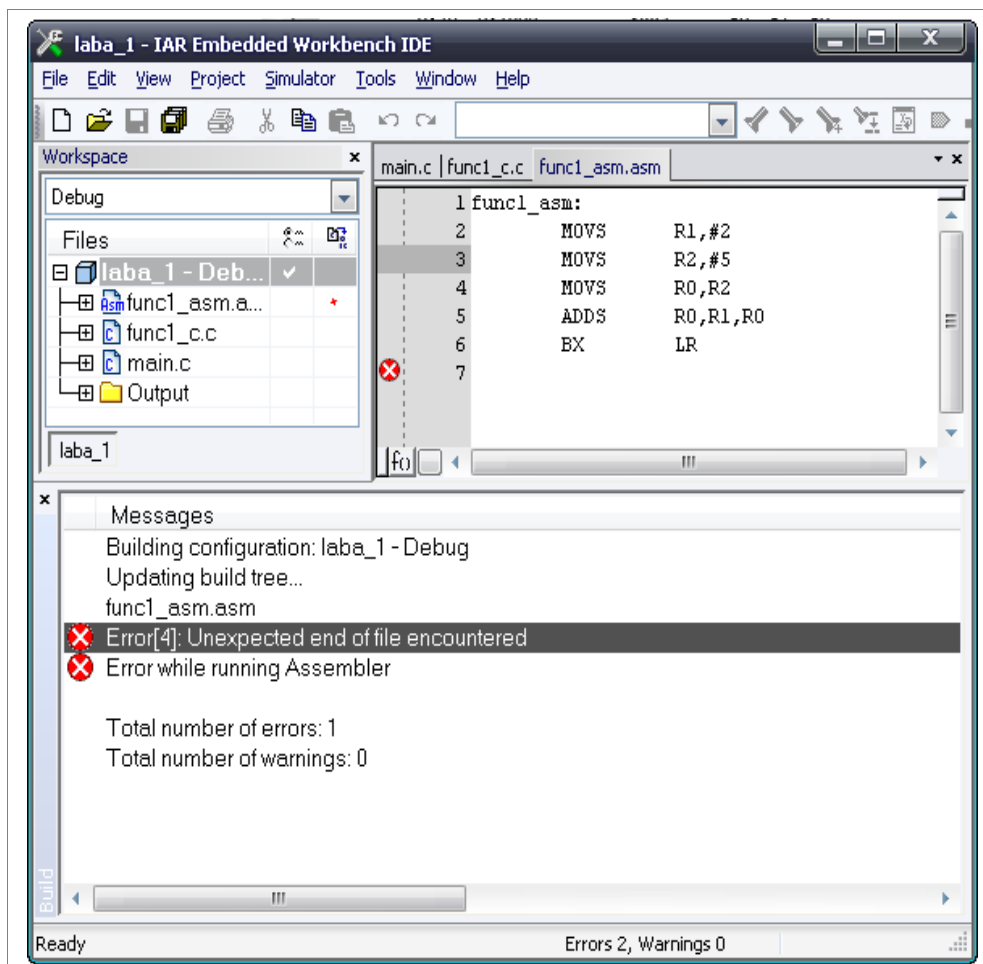


Рисунок 8 Ошибка при компиляции

Как и следовало ожидать, нас постигла неудача. Ассемблерный файл содержит ошибки рис.8.

Error[4] : Unexpected end of file encountered – неожиданный конец файла обнаружен.

Добавление в конец файла слова END тоже не приводит к успеху, количество ошибок только увеличивается.

Делаем предположение, что наш первый исходник на ассемблере как-то не совсем корректно оформлен. За пополнением нашего багажа знаний обращаемся к Help->Assembler Reference Guide. Бегло просматриваем руководство с целью обнаружить в нём примеры оформления исходных текстов на ассемблере. На стр.11,12 мы находим интересующие нас примеры.

Вот пример со стр.12 руководства Assembler Reference Guide:

```
EXTERN third
SECTION MYDATA : DATA (2)
first:      DC32 3
second:    DC32 4
Then in the section MYCODE, the following relocatable expressions are legal:
SECTION MYCODE : CODE (2)
CODE32
; MYDATA must be linked in the range 0-255,
; otherwise the immediate values #first etc.
; will be out of range
MOV   R1,#first
MOV   R2,#second
MOV   R3,#third
LDR   R1,=first+4
LDR   R2,=second
```

Внимательно изучаем его. Мы видим, что в данном примере исходный текст на ассемблере состоит из 2-х секций: секции данных - SECTION MYDATA : DATA (2) и секции кода - SECTION MYCODE : CODE (2). Наша функция func1_asm никаких данных не содержит, поэтому секция данных может отсутствовать, а вот секцию кода мы должны правильно оформить. Добавим из примера две недостающих строки в начало нашего файла

```
SECTION MYCODE : CODE (2)
CODE32
```

и снова запустим наш проект на компиляцию. Получаем результат рис. 9

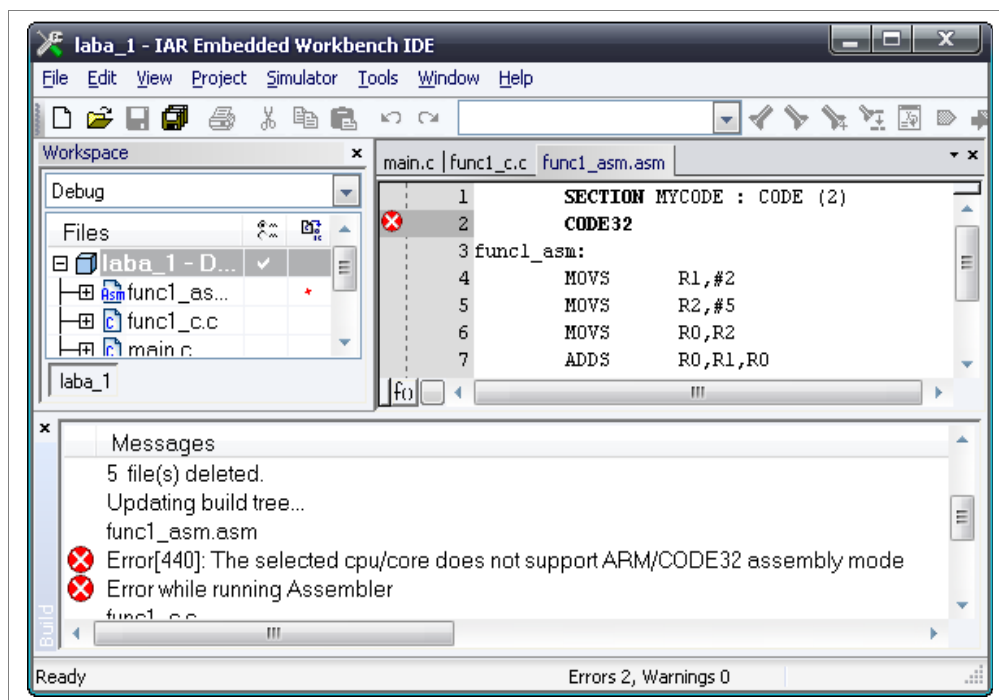


Рисунок 9. Ошибки при компиляции

Закомментируем строку, содержащую ошибку. Признаком комментария на языке ассемблер является точка с запятой. Но в ассемблере от IAR есть приятная особенность – он воспринимает комментарии языка C, т.е. мы можем использовать `//`. Устранив таким образом ошибку мы наконец-то достигаем желаемого результата, наш ассемблерный файл успешно компилируется.

Теперь нам осталось правильно организовать вызов нашей функции. Раскомментируем строки 2 и 6 в файле `main.c`, рис. 7, и снова запустим наш проект.

Сейчас ошибки выдаёт компоновщик или редактор связей (на программистском жаргоне «линкер», «линковщик») рис. 10. Он не видит ссылку `func1_asm`. Как нам известно, чтобы объявить ссылку видимой из внешних модулей, в языке C++ используется служебное слово **public**. Возможно, и в ассемблере есть нечто похожее? Проверим. Снова обращаемся к руководству *Assembler Reference Guide*. В приведенном примере есть строка:

```
EXTERN third
```

Здесь вместо служебного слова **extern** на языке Си используется **EXTERN** на ассемблере, поэтому логично предположить, что вместо **public** нужно искать **PUBLIC**. Используя поисковик, находим в руководстве фразу:

For example, use the PUBLIC directive to make one or more symbols available to other modules.

Что для нас, собственно, и требуется.

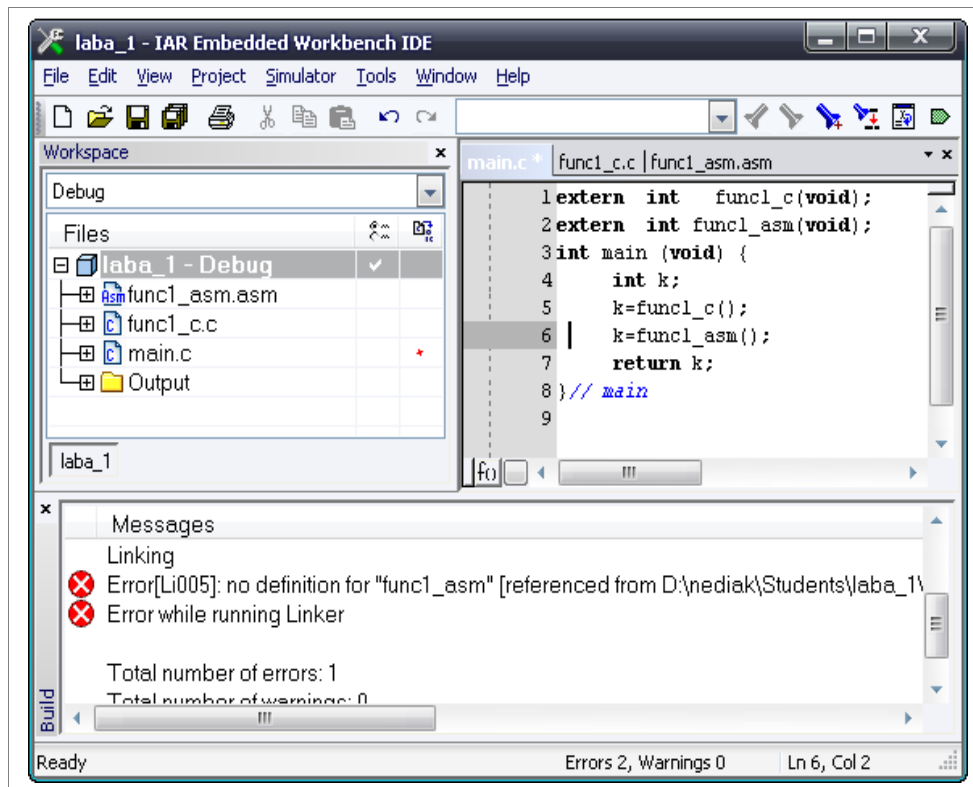


Рисунок 10 Ошибки при компоновке

Дополняем наш файл func1_asm.asm строкой
 PUBLIC func1_asm
 Запускаем проект и ... Ура! Наконец-то у нас всё получилось.

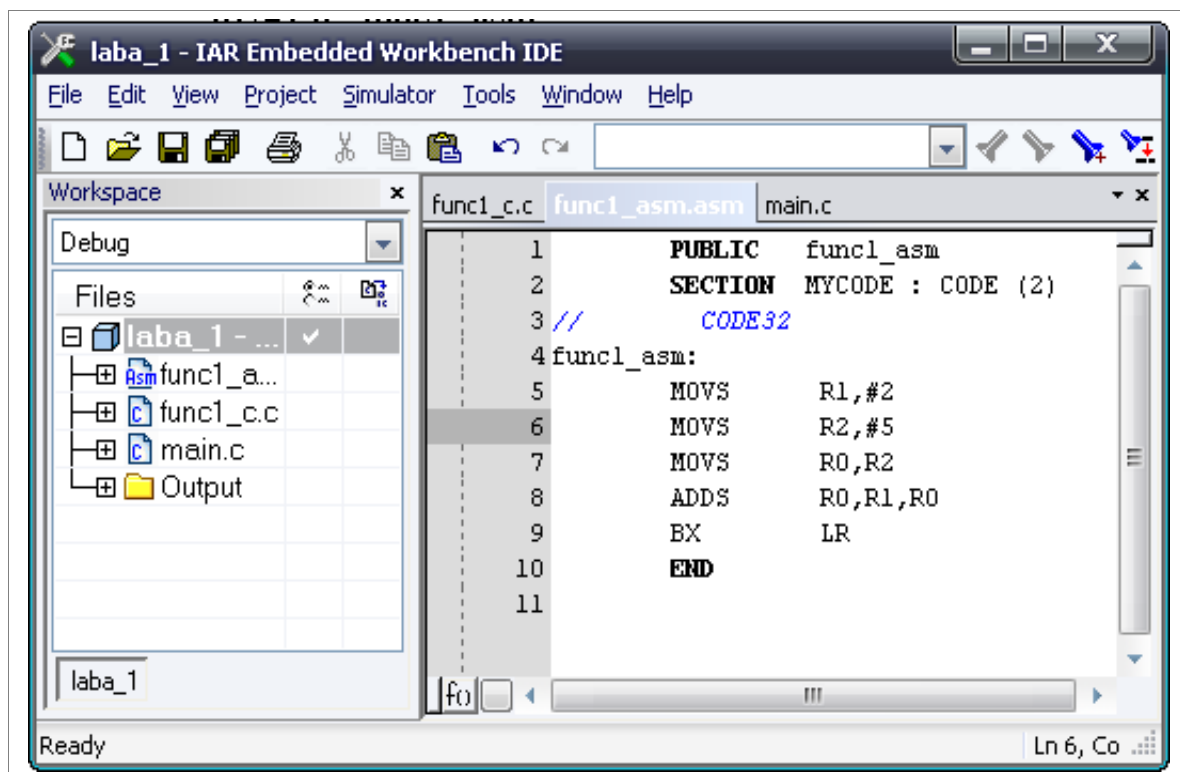


Рисунок 11. Ассемблерный код

Вот наш первый шедевр на ассемблере, рис. 11. В том, что это действительно шедевр, не может быть никакого сомнения! Ведь мы умудрились написать нашу функцию на ассемблере, фактически не зная самого языка. Мы воспользовались дизассемблером и чуть-чуть проявили сообразительность.

Пока мы не знаем команд, нам лишь известно, что эта функция делает – она складывает числа 2 и 5, и возвращает их сумму. Так ли это на самом деле, самое время проверить.

Запустите проект на отладку и на панели инструментов выберите View->Register. Должно открыться окно, где в оперативном режиме отслеживается содержимое регистров. С помощью мыши отрегулируйте размеры окон Disassembly и Register так, чтобы их одновременно было хорошо видно. У вас должна получиться картинка, как показано на рис.12. Если зелёная стрелка отсутствует в окне Disassembly, щёлкните мышью в этом окне, и она появится. Последовательно нажимая функциональную клавишу F11 или соответствующую кнопку на панели инструментов можно пошагово отследить исполнение нашей программы.

Положение курсора и синхронное изменение регистров однозначно информируют нас о назначении той или иной команды. Например, команда

ADDS R0,R1,R0

складывает содержимое регистра R1 с содержимым R0 и сохраняет результат в R0.

Команда

MOVS R1,#2

загружает 2 в регистр R1 и т.д.

Функции func1_c() и func1_asm() вызываются из главной функции main() одна за другой, поэтому результат работы первой функции сохраняется в регистрах перед вызовом второй функции. По этой причине функцию func1_c() лучше закомментировать.

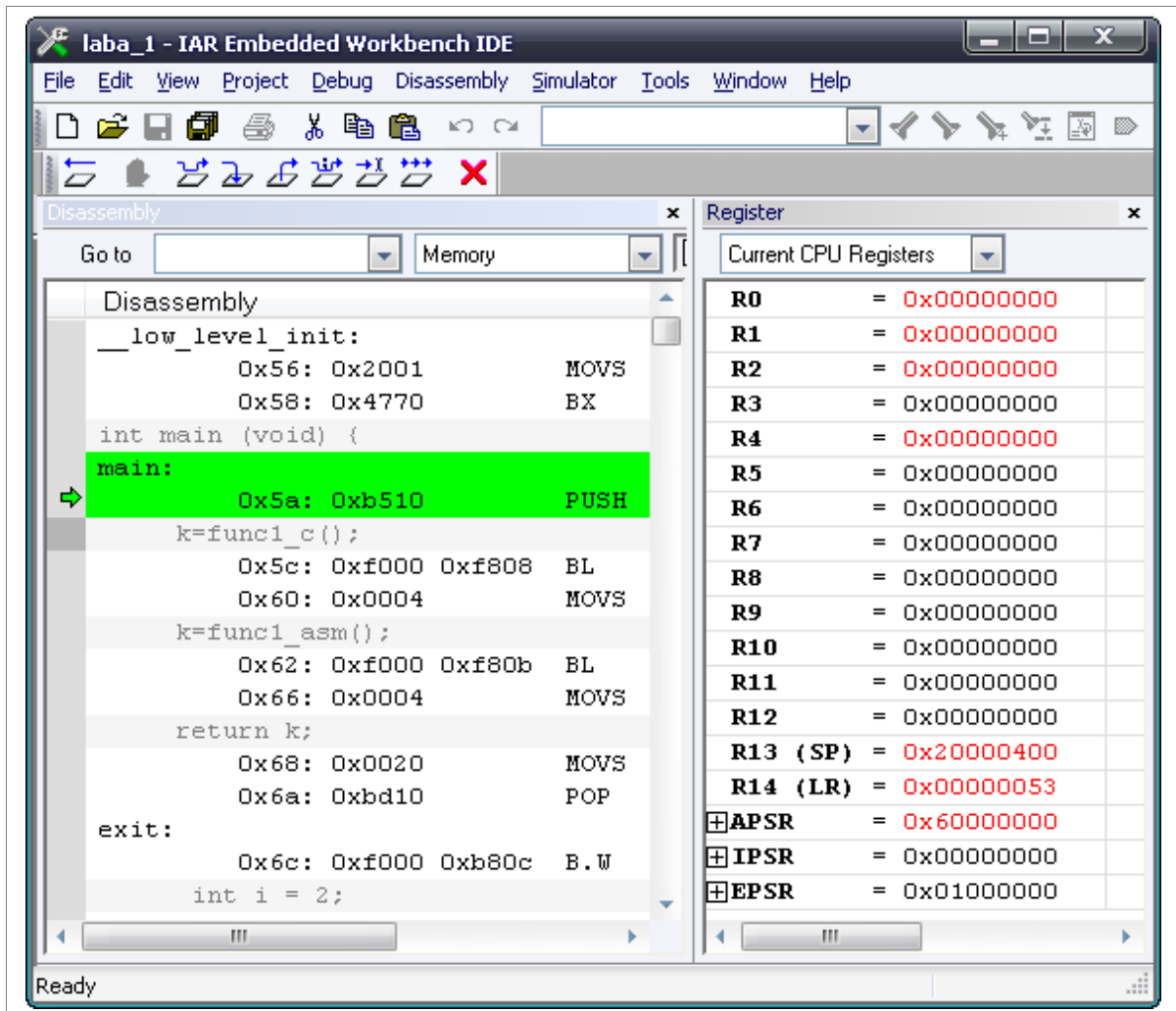


Рисунок 12 Дизассемблер IAR

Заметьте, что мы изучили функциональное назначение ассемблерных команд, не прибегая к чтению какой либо документации. В общем случае так поступать не удастся. Прежде чем вы овладеете профессиональными навыками программирования, придётся затратить много часов на чтение специальной литературы. Причём, почти вся она на английском языке, то, что есть на русском, как правило, уже устаревшее. Есть устоявшееся мнение, что в природе не существует программистов, не знающих английского языка. Чем раньше будет воспринят этот по-настоящему печальный природный факт – тем лучше.

К большому счастью студентов, не владеющих английским языком в должной степени, фирма "Миландр" поставляет документацию к своим контроллерам на русском языке. Поэтому на ближайший семестр эта документация станет основным учебником по ассемблеру, точнее, по системе команд для линейки микроконтроллеров этой фирмы. Синтаксис ассемблера придётся осваивать в основном по справочнику *Assembler Reference Guide*. Здесь без знания английского уже никак не обойтись.

Ещё раз взгляните на рис. 12. На нём вы видите самого хорошего учителя по ассемблеру, его имя Дизассемблер, это любимый друг хакеров. Он очень квалифицирован, терпелив, не раздражителен, никогда не устаёт. Многие поколения программистов трудились над его появлением на свет. Мы надеемся, что вы с ним подружитесь. Но, как и любой учитель, он предъявляет к своим ученикам требования: некоторую сообразительность и регулярность в занятиях. Неуважительного отношения к себе он, увы, не прощает, особенно

во время сессии.

2.3 Указания к выполнению лабораторной работы №1

Итак, подведём некоторый итог. Мы попытались подробно описать алгоритм работы программиста. Рекомендуем пошагово повторить все действия. Появление ошибок при разработке программ это совершенно нормальная (штатная) ситуация. Диагностическое сообщение, как правило, даёт исчерпывающую информацию для исправления ошибки. И, самое главное, имеющихся знаний тоже, как правило, оказывается вполне достаточно. Очень важно в такой ситуации просто не растеряться. Хороший специалист как раз и вырастает из множества маленьких побед ... над собой.

1. Вам необходимо придумать какую-нибудь свою простую функцию и написать её сначала на Си.
2. Затем, используя дизассемблер, написать её аналог на ассемблере.
3. Обратившись к документации по микроконтроллерам "Миландр" снабдить комментариями каждую команду.

На кафедральном сервере по адресу `_For_Students\MPSSAU_Nediak\Arch (TODO переместить в !Labs)` есть образцы лабораторных работ, выполненных по предлагаемому алгоритму. Ознакомиться можно, копировать и потом пытаться сдавать не стоит. Используя творческий подход, попытайтесь самостоятельно разобраться, как происходит обмен информацией между вызывающей и вызываемой функцией, в частности, как передаются параметры в функцию и как затем возвращается результат вычислений.

2.4 Требования к содержанию отчёта

1. Описание последовательности действий для создания проекта, ошибочных действий в том числе.
2. Исходный текст функции main().
3. Исходный текст простейшего модуля (функции) на языке Си.
4. Исходный текст его аналога на Assembler-e.
5. Заключение по лабораторной работе.
6. Отчет вместе с проектом разместить Redmine.

Замечание. Исходный текст на Assembler-e должен быть снабжен подробными комментариями в каждой строке. Также он должен быть уникален, т.е. разработан самостоятельно, а не переписан у одногруппника.

3 Создание и компиляция первого проекта в среде KEIL. Написание простейшего модуля на языке Assembler. Лабораторная работа № 2

3.1 Введение

В лабораторной работе № 1 мы познакомились с широко известной средой разработки IAR. Эта лабораторная работа написана по рекомендации нашего партнера — фирмы "Миландр". Сотрудники компании аргументируют свои пожелания тем, что их потребители в основном пользуются Keil-ом.

3.2 Создание нового проекта в среде Keil

Запустите среду разработки Keil uVision и выберите пункт главного меню Project=>New uVision Project. В появившемся окне создайте новую папку и в ней сохраните новый файл-проекта. Далее появится окно выбора микроконтроллера: «Select Device for Target...» (Рисунок 13).

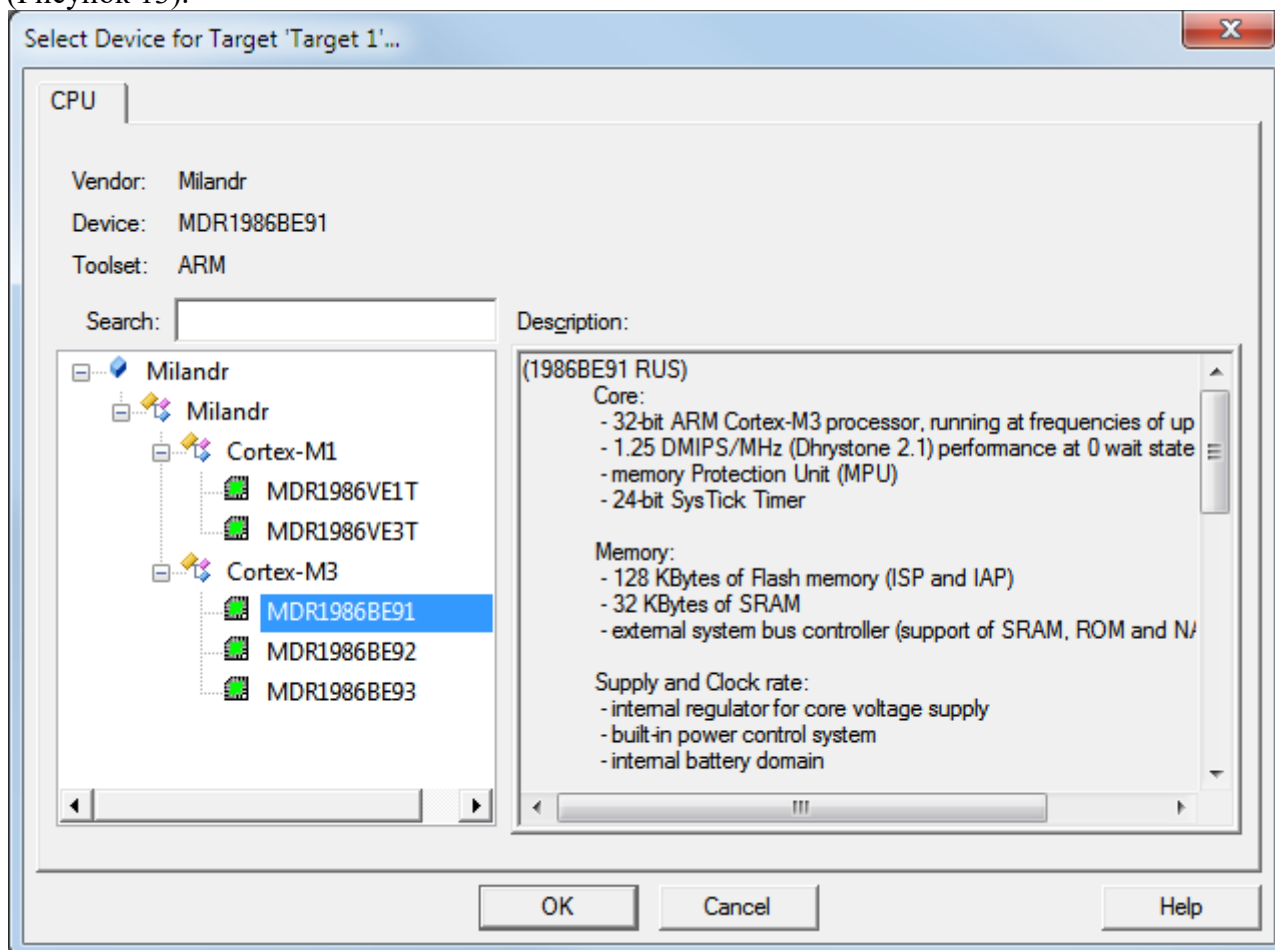


Рисунок 13 - Выбор МК при создании проекта

Далее в появившемся окне «Manage Run-Time Environment» (рис. 14) осуществляется выбор конфигурации МК: элементов библиотеки **CMSIS**, файла начального запуска для выбранного устройства (**Devices**) и драйверов работы с периферийными блоками МК

(Drivers). Последующие пункты конфигурации проекта относятся к программному обеспечению более верхнего уровня, так называемому «middleware» (см раздел. 4.2.2)- мы в этой работе использовать не будем, т. к. они выходят за рамки задачи освоения МК.

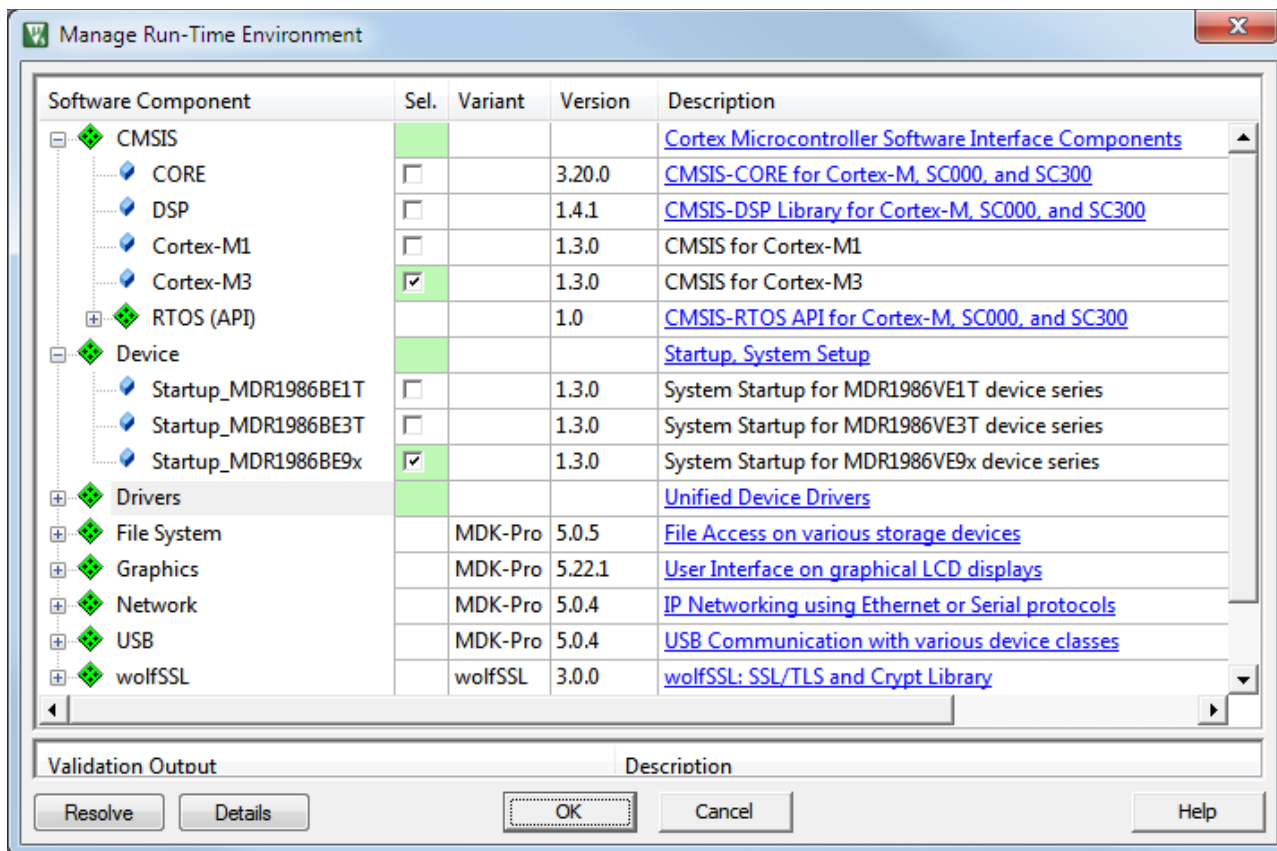


Рисунок 14 - Выбор библиотек при создании проекта

После этого откроется среда разработки с готовым скелетом проекта рис. 15 В группу **Source Group 1** следует разместить файлы с исходными текстами Ваших программ.

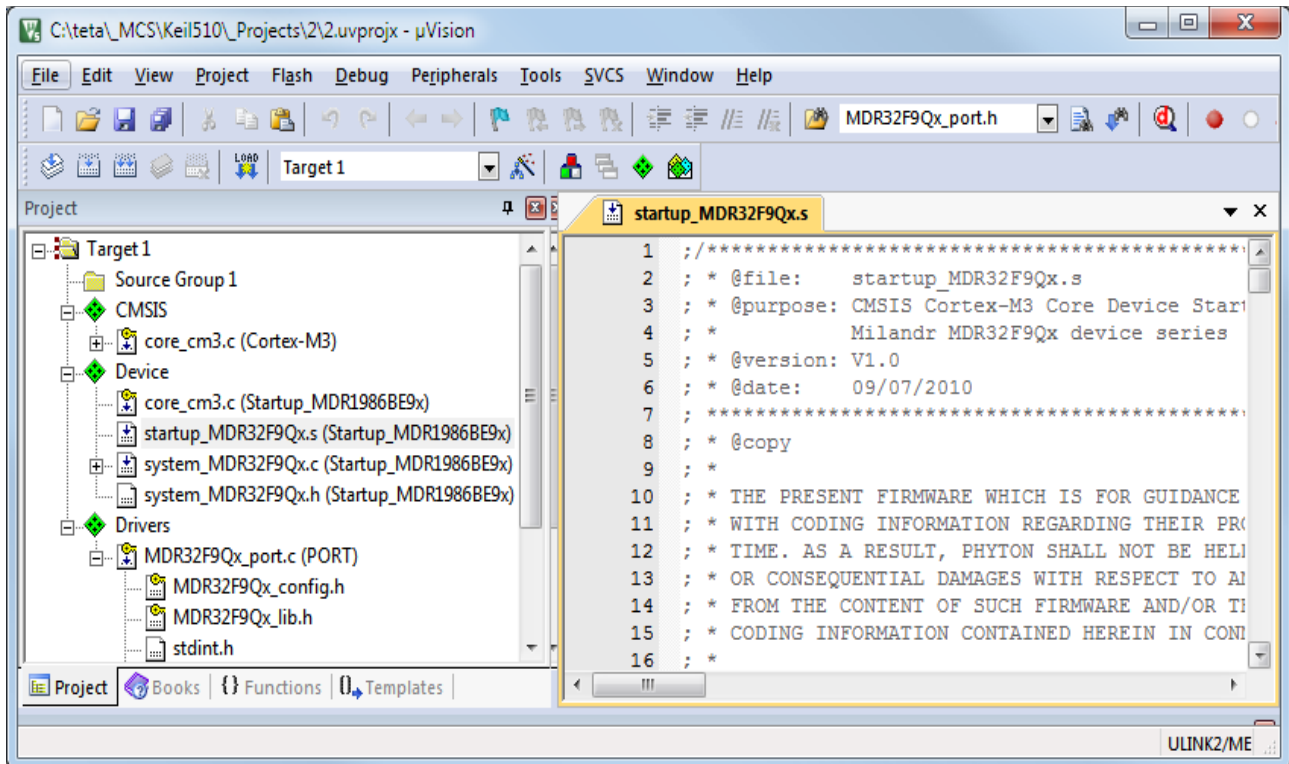


Рисунок 15 Keil со скелетом проекта.

3.3 Разработка простейшей программы для микроконтроллера

Мы продолжаем знакомиться с приёмами программирования микроконтроллеров. Одной из часто решаемых задач является задача управление внешними устройствами. Каждое устройство имеет в адресном пространстве микроконтроллера своё множество адресов. Ячейки памяти в этом пространстве называют регистрами. Образно говоря, каждый разряд регистра является рукояткой (тумблером) для управления этим устройством. Для того, чтобы научиться управлять, например, автомобилем, сначала нужно овладеть правилами управления. Точно также и здесь, для того чтобы научиться управлять, например, АЦП (Аналого-Цифровым Преобразователем), сначала нужно познакомиться с правилами управления этим устройством. Изучению и практическому использованию управления внешними устройствами (ВУ) посвящена вторая часть настоящего пособия. А сейчас мы решим очень скромную задачу — научимся читать и писать данные в область памяти (регистр) с заданным адресом.

Из курса изучения языка Си мы знаем как это сделать. Можно, например, объявить указатель и присвоить ему желаемое значение адреса. Так мы и поступим. И к тому же учтём ещё один важный момент.

Микроконтроллер, как правило, имеет несколько однотипных устройств, которые имеют разные зоны адресов, но одинаковую структуру расположения регистров. Т.е базовые адреса разные, а смещения регистров относительно базового адреса для всех однотипных устройств одинаковы. Поэтому программу пишут так, чтобы переход от одного устройства к другому происходил с минимальными затратами на редактирование исходного кода.

Прежде чем писать какие-то данные по заданному адресу, нужно предварительно узнать, а что же располагается в этой зоне адресов — иначе последствия могут быть мало предсказуемы. Из документации к MDR1901BC1F¹⁷ узнаём, что 0x20000000 — это адрес

¹⁷ Здесь и далее по тексту этой и следующей лабораторной работы вместо микроконтроллера MDR1901...

начала данных, а 0xE000E010 – это адрес регистра управления системным таймером. По работе с этим таймером может быть полезной информация:

<http://easyelectronics.ru/arm-uchebnyj-kurs-systick-sistemnyj-tajmer.html>

С учётом сказанного, у нас может быть такой тестовый код, рисунок 16.

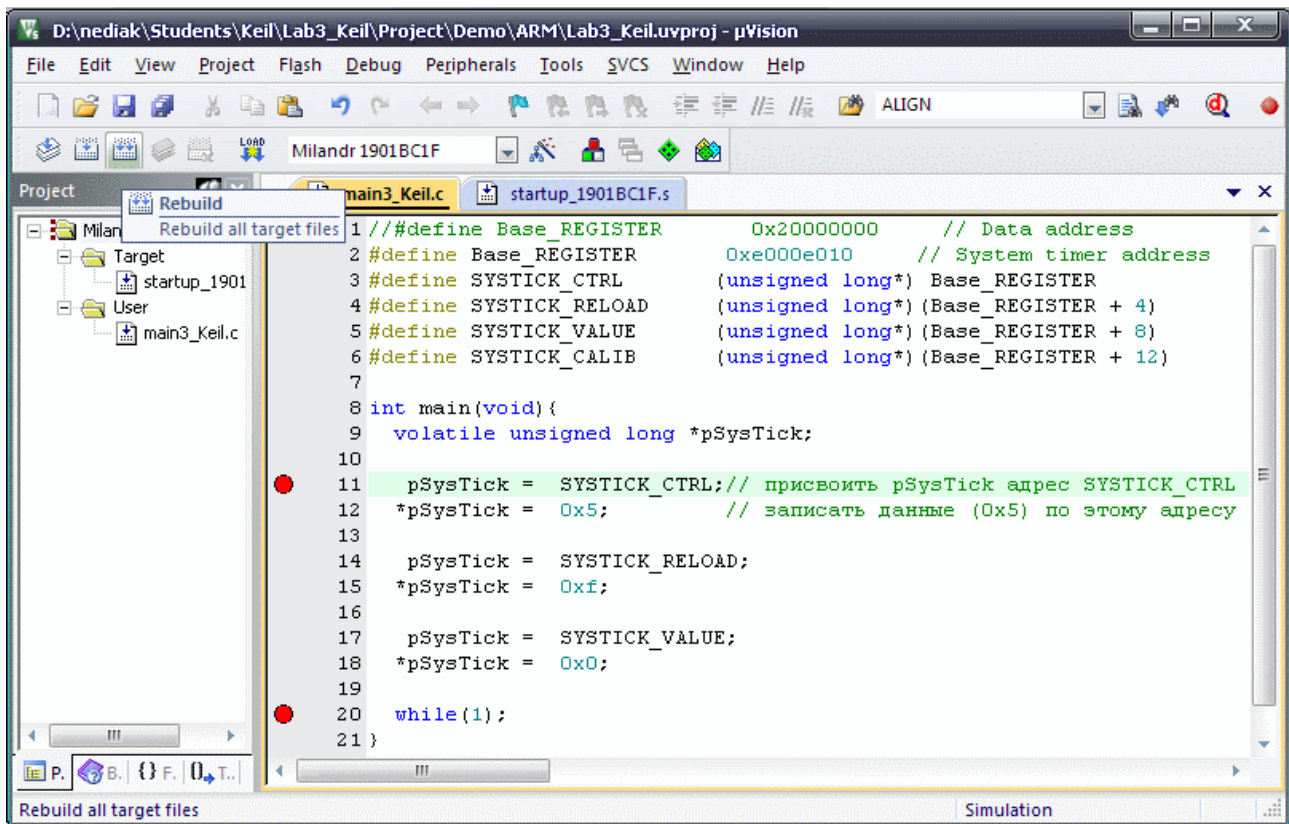


Рисунок 16. Окно среды Keil.

Если наш текст не содержит синтаксических ошибок, то при последовательном нажатии иконок «Rebuild» и «Start/Stop session» откроется окно сессии отладки программы, рисунок 17.

следует выбирать MDR1986... Ядро Cortex-M3 у этих микроконтроллеров одинаково. Отличие заключается в том, что MDR1901... двух-ядерный, у него ещё есть ядро DSP (Digit Signal Processor, аналог TMS320C54). То есть фразу «Из документации к MDR1901BC1F...» следует читать как «Из документации к MDR1986BE9x, а фразу «...заглянем в стандартный файл нашего проекта startup_1901BC1F.s» следует читать как «...заглянем в стандартный файл нашего проекта startup_MDR32FQx.s». Адаптировать текст этой методички под MDR1986BE9x — это будет очередное задание для студентов. Славные традиции 539-й группы мы намерены продолжить. Смотри стр.12. О том, как писалась одна из лабораторных работ по Keil-у есть почти протокол: <http://esau.tusur.ru:8085/issues/808>. Это в качестве примера. От вас, господа студенты, требуется примерно то же самое — протоколирование своих действий, **ошибочных в том числе.**

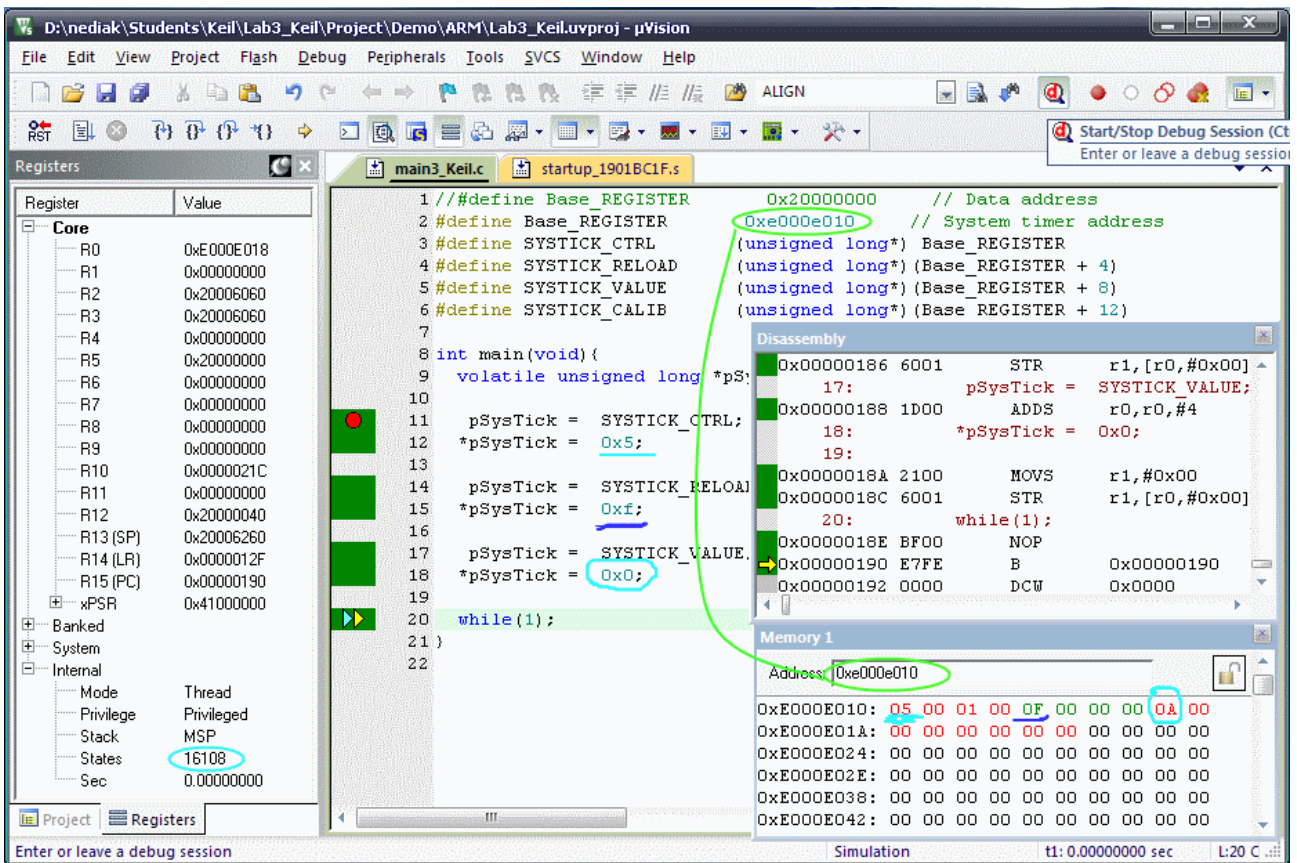


Рисунок 17. Сеанс отладки.

При первом запуске вид окна будет другим. Мы пока закрыли не интересующие нас окна и оставили только те, что показаны на рис. 17 – это Registers, Disassembly и Memory. Наблюдая происходящее в этих окнах, можно с точностью до отдельного бита сказать, что происходит в микроконтроллере, точнее, в математической модели микроконтроллера, поскольку отладку мы ведём в симуляторе.

При включении окна Memory(память) View->Memory Window курсор остановится на строке Address. Keil вам предложит ввести адрес интересующего участка памяти. В нашем случае это будет либо адрес 0x20000000, либо 0xE000E010, в зависимости от того, какая из строк main3_Keil.c активна, 1-я или 2-я. Двойной щелчок мышкой в начале строки либо на исходном тексте, либо в окне Disassembly зафиксирует/уберёт точку останова. Точками останова можно управлять во время сеанса отладки.

Нажимая кнопку F11 или F10 на клавиатуре или соответствующую иконку на панели инструментов, проанализируем работу нашей программы. В окне Registers есть строка States. Она показывает какое число тактов прошло с момента запуска до момента останова микроконтроллера. Наблюдая, на сколько тактов увеличивается этот показатель при очередном шаге, можно увидеть время выполнения машинных (ассемблерных) команд микроконтроллера.

Например, команда инициализации (Move Signed) регистра r1 значением 0

MOVS r1,#0 выполняется за один такт;

команда записи (Store) значения из регистра r1 в память, а в данном случае в регистр внешнего устройства с адресом (SYSTICK_VALUE = 0xE000E018), это значение адреса находится в регистре r0,

STR r1,[r0, #0] выполняется за два такта;

а команда ветвления (Branch)

B 0x00000192 выполняется за три такта.

Если поставить две точки останова в разных местах исследуемого кода, то взяв разность показателя States в этих точках, можно узнать время работы в «тиках» синхросигнала для участка кода программы между этими точками останова. Чтобы узнать время в секундах нужно эту разность разделить на тактовую частоту процессора Options for Target → Xtal(MHz). В нашем примере это Xtal = 12.0 MHz.

Узнаем за какое время происходит запись трёх значений в регистры системного таймера

$$dt = (15894 - 15881) / 12e6 = 1.083e-6 \text{ с,}$$

примерно за одну микросекунду.

После предварительного знакомства с машинным кодом, сгенерированным Keil-ом, сразу возникает вопрос — а нельзя ли его оптимизировать. Иногда это возможно. Например, в нашем случае смещение относительно базового регистра вычисляется отдельной командой

```
ADDS r0,r0,#4.
```

С точки зрения логики она лишняя, поскольку код

```
ADDS r0,r0,#4  
MOVS r1, #0x0  
STR r1,[r0, #0x00]
```

эквивалентен такому

```
MOVS r1, #0x0  
STR r1,[r0, #0x04]
```

и мы аж на 25% можем увеличить скорость работы!

А вот с точки зрения практики вопрос об оптимальности остаётся открытым. Не всегда самый скоростной код оказывается рабочим для внешнего устройства. Если речь идёт о записи/чтении памяти данных, то наш код будет однозначно лучше, а вот если речь идёт о записи/чтении регистров внешних устройств, то более медленный код может иногда оказаться более надёжным. Можете объяснить почему?

Мы рассмотрели этот пример в качестве учебного. На практике для общения с внешними устройствами сейчас чаще пользуются стандартными библиотеками. Это выгодно не только с точки зрения переносимости программ, но и с точки зрения надёжности работы, поскольку библиотечный код, как правило, многократно протестирован. Но и из этого правила бывают исключения.

Итак, компилятор Keil-a оставляет нам обширное поле для творчества, осталось научиться кодировать на ассемблере. Чтобы узнать как оформляются функции на ассемблере, заглянем в стандартный файл нашего проекта startup_1901BC1F.s¹⁸ и в справочную систему Keil-a.

Мы не будем подробно описывать процесс добывания информации, это мы сделали в предыдущей лабораторной работе, и здесь он примерно такой же. Сразу приведём готовый ответ. Это файл скелетов функций на ассемблере skeleton.s и аналог нашего main() кода для работы с таймером setTimer.s.

В файле skeleton.s, рисунок 17, приведены два варианта оформления процедур (функций) на языке ассемблера. Первый вариант Skeleton1 для случая, когда из ассемблерной процедуры никакие другие функции не вызываются, а второй, Skeleton2, для случая, когда из данной процедуры вызывается ещё одна или несколько процедур(функций). Для примера мы вызвали Skeleton1.

¹⁸ Смотри сноску на стр. 45.

```

1      AREA    |.text|, CODE, READONLY
2      ;-----
3  skeleton1\
4      PROC
5      EXPORT  skeleton1          [WEAK]
6      ;
7      code
8      ;
9      BX      lr                ; return
10     ENDP     ; of procedure
11     ;-----
12  skeleton2\
13     PROC
14     EXPORT  skeleton2          [WEAK]
15     push   {lr}                ; save return address
16     ;
17     push   {r0-r12,lr}         ; save registers
18     ;
19     code
20     ;
21     bl     skeleton1          ; call skeleton1
22     ;
23     pop    {r0-r12,pc}         ; load registers, return
24     pop    {pc}                ; return
25     ENDP     ; of procedure
26     ;-----
27     END      ; of asm-file

```

Рисунок 18. Скелеты для *asm*-процедур

```

1      AREA    |.text|, CODE, READONLY
2      ;-----
3  setTimer PROC
4      EXPORT  setTimer          [WEAK]
5      ;
6      MOVS   r1,#0x05           ; r1 <- 5
7      STR    r1,[r0,#0x00]      ; r0 = timer address
8      MOVS   r1,#0x0f           ; r1 <- f
9      STR    r1,[r0,#0x04]      ; (r0 = timer address) + 4
10     MOVS   r1,#0x00           ; r1 <- 0
11     STR    r1,[r0,#0x08]      ; (r0 = timer address) + 8
12     ;
13     BX     lr                ; return
14     ENDP     ; of setTimer procedure
15     ;-----
16     END      ; of asm-file
17

```

Рисунок 19. Установка системного таймера

В файле **setTimer.s** приведён исходный текст на ассемблере, выполняющий те же самые действия, что и наш прежний Си-код, работавший с таймером.

Включаем процедуру `setTimer()` в проект и проверяем насколько она получилась скоростной. Вывод неутешителен — `setTimer()` работает медленнее чем код, сформированный Keil-ом. Наш код, действительно, экономит 4 такта, но мы не учли

накладные расходы на вызов самой функции `setTimer()` — 8 лишних тактов. Причём, это самый быстрый вариант вызова по `skeleton1`, а для варианта `skeleton2` затраты на вызов функции будут ещё больше, поскольку в нём присутствуют ещё и затраты на сохранение регистров в стеке и затем извлечение сохранённых значений из стека.

Таким образом, если речь идёт о получении наиболее скоростного кода, то в нём желательно избегать вызовов функций. Язык ассемблера позволяет это делать при сохранении читаемости кода. Как именно, мы познакомимся в следующих лабораторных работах. В языках высокого уровня для этой цели существует очень эффективный приём — ассемблерные вставки. Пока в Keil-е не удалось этим воспользоваться.

3.4 Заключение

Здесь мы привели пример выполнения лабораторной работы. Установили среду разработки Keil, создали проект и написали простейший текст на ассемблере. Попутно продолжили знакомиться с системой команд для ядра Cortex-M3. Заметим, что для того, чтобы обнаружить возможность оптимизации кода, никаких особых программистских изысков мы не использовали. Достаточно было просто внимательно читать документацию по микроконтроллеру. В нашем случае это знакомство с командами обращения к памяти.

Каждый студент получает или придумывает свой объект для исследования, точнее, свой фрагмент Си-кода, который нужно будет подробно описать в терминах системы команд микроконтроллера.

3.5 Требования к содержанию отчёта

Те же самые, что и в предыдущей лабораторной работе (см. пункт 2.4).

3.6 Контрольные вопросы

1. Как установить на компьютер Keil?
2. Как адаптировать эту среду для работы с определённой моделью микроконтроллера и определённым изготовителем, например, ф. "Миландр"?
3. Как создать новый проект в среде Keil?
4. Как выбираются основные опции проекта?
5. Что такое симулятор и для чего он служит?
6. Что такое система команд микроконтроллера и для чего её нужно знать разработчику систем управления? Почему нельзя ограничиться только изучением языка верхнего уровня?
7. Расскажите о командах, которые вы изучили в ходе выполнения этой лабораторной работы.
8. Можно ли из процедуры на ассемблере вызвать функцию на Си?
9. Что такое JTAG?
10. По какой причине может не работать внутрисхемная отладка?
11. Что в себя включает понятие «оптимизация кода»?
12. Для чего служит директива `WEAK`? См. рисунок 19.

4 Интерфейс Си и ассемблера. Лабораторная работа № 3

Цель. Изучить способы обмена данными между модулями на языке Си и языке ассемблера.

4.1 Введение

Интерфейс (interface(*англ*) - стык, взаимосвязь) в данном контексте означает исследование обмена данными между функциями на языке Си и ассемблера. Существует два способа обмена — через общую (видимую) область памяти и через параметры функций. Какой из них эффективнее (быстрее), нужно решать для каждого отдельного случая. При этом необходимо помнить, что наиболее быстро осуществляется обмен данными между регистрами.

Как показала учебная практика, выполнить самостоятельно анализ обмена данными между Си и ассемблером для подавляющей массы студентов оказалось затруднительным. Поэтому мы выделяем эту тему в отдельную лабораторную работу.

С программной точки зрения существует два способа обмена данными между функциями — через параметры функций и через общие (видимые) области памяти.

Для архитектуры ARM есть свои особенности при обмене данными между функциями. Они даже регламентируются специальным документом IHI0042E_aapcs.pdf¹⁹. Мы вам рекомендуем ознакомиться с ним на досуге. Сами же будем исследовать обмен данными, используя дизассемблер, поскольку эта методика работает на любых архитектурах. А Cortex мы воспринимаем здесь только как один из примеров.

В архитектуре INTEL при передаче данных через параметры используется только стек, а в архитектуре Cortex всё несколько сложнее — здесь используется и стек, и регистры. Если число параметров не превышает 4-х, то стек не используется вообще.

4.2 Содержание работы

В данной лабораторной работе каждый студент получает свой объект (тип данных) для исследования. Требуется эти данные передать в функцию на ассемблере, произвести их обработку и вернуть результат в вызывающую функцию на языке Си.

Используя дизассемблер, мы сначала анализируем, как осуществляется обмен данными между функциями, написанными на языке Си, и на основании этого исследования пишем функцию на ассемблере. В ассемблере, кстати, принято использовать вместо термина «функция» термин «процедура».

4.3 Обмен данными через параметры функций

В качестве примера сделаем лабораторную работу, в своё время не выполненную одной из студенток 530-й группы. Кратко задание формулировалось следующим образом:

Передать 3-х мерный массив как параметр, инвертировать, вернуть результат.

Поскольку тема этой лабораторной работы «Интерфейс Си и ассемблера», то в развёрнутом виде задание будет формулироваться так:

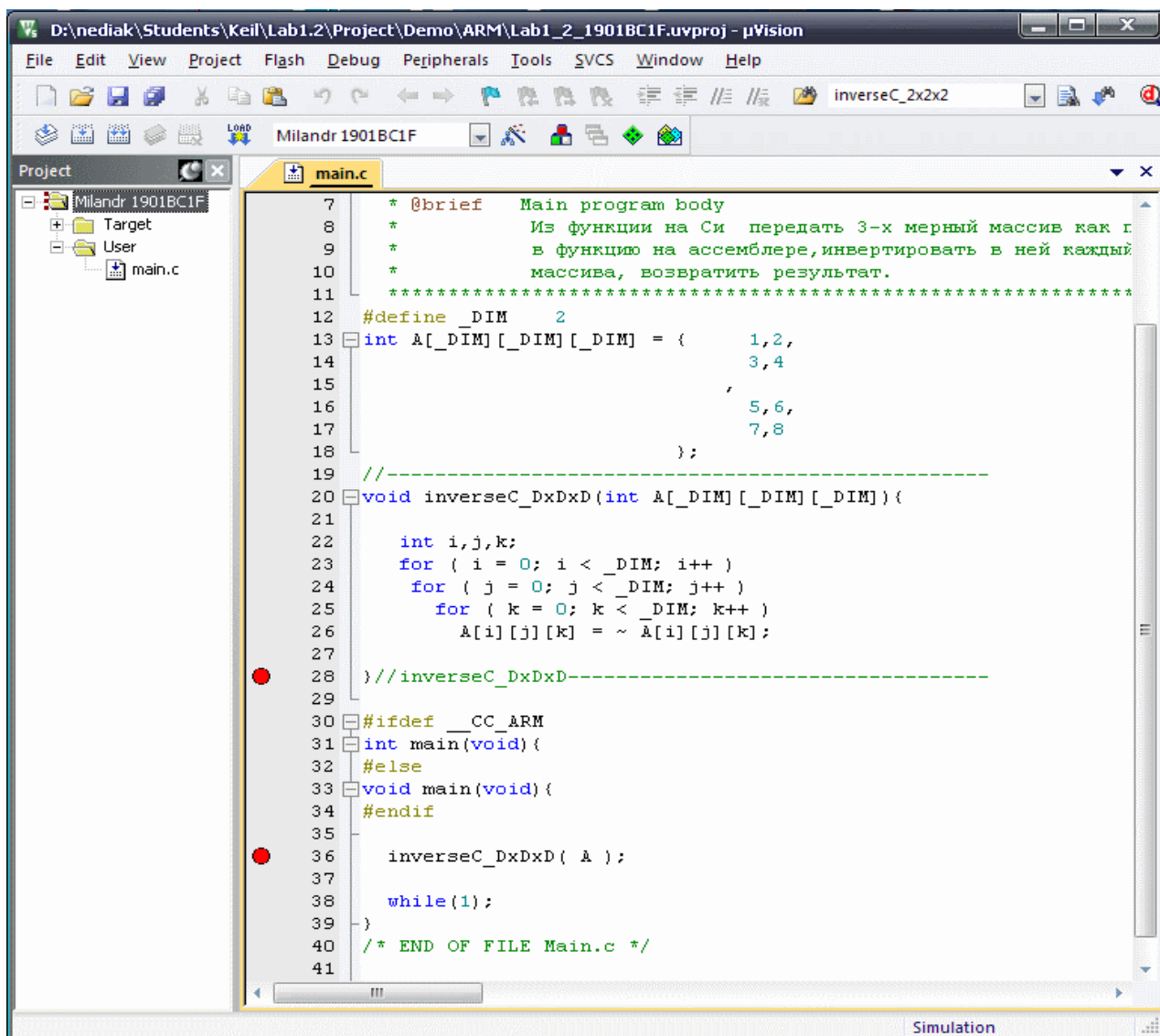
Из функции на Си передать 3-х мерный массив как параметр в функцию на ассемблере, инвертировать в ней каждый элемент массива, вернуть результат.

19 Procedure Call Standard for the ARM® Architecture. ARM IHI 0042E, current through ABI release 2.09.
http://infocenter.arm.com/help/topic/com.arm.doc.ihl0042e/IHI0042E_aapcs.pdf

4.3.1 Выполнение работы

Для решения поставленной задачи мы поступим стандартным способом — напишем функцию на Си, а затем переведём её на ассемблер. Поскольку задача простейшая, то весь код на Си разместим в единственном файле `main.c`.

На рисунке 20 показано окно проекта и окно исходного кода программы.



```
7  * @brief   Main program body
8  *   Из функции на Си передать 3-х мерный массив как г
9  *   в функцию на ассемблере, инвертировать в ней каждый
10 *   массива, вернуть результат.
11 *****
12 #define _DIM    2
13 int A[_DIM][_DIM][_DIM] = {
14     1,2,
15     3,4
16     ,
17     5,6,
18     7,8
19 };
20 void inverseC_DxDxD(int A[_DIM][_DIM][_DIM]) {
21
22     int i,j,k;
23     for ( i = 0; i < _DIM; i++ )
24         for ( j = 0; j < _DIM; j++ )
25             for ( k = 0; k < _DIM; k++ )
26                 A[i][j][k] = ~ A[i][j][k];
27
28 }//inverseC_DxDxD-----
29
30 #ifdef __CC_ARM
31 int main(void) {
32 #else
33 void main(void) {
34 #endif
35
36     inverseC_DxDxD ( A );
37
38     while(1);
39 }
40 /* END OF FILE Main.c */
41
```

Рисунок 20. Программа в среде Keil MDK_ARM

Если запустить теперь программу на отладку (как это сделать, мы узнали в прошлой лабораторной) и открыть окно дизассемблера, то увидим следующую картинку, рис.21.

```

Disassembly
20: void inverseC_DxDxD(int A[_DIM][_DIM][_DIM]){
21:
22:     int i,j,k;
0x00000198 B530     PUSH    {r4-r5,lr}
0x0000019A 4603     MOV     r3,r0
23:     for ( i = 0; i < _DIM; i++ )
0x0000019C 2000     MOVS   r0,#0x00
0x0000019E E017     B      0x000001D0
24:         for ( j = 0; j < _DIM; j++ )
0x000001A0 2100     MOVS   r1,#0x00
0x000001A2 E012     B      0x000001CA
25:             for ( k = 0; k < _DIM; k++
0x000001A4 2200     MOVS   r2,#0x00
0x000001A6 E00D     B      0x000001C4
26:                 A[i][j][k] = ~ A[i
27:
0x000001A8 E8031400 ADD     r4,r3,r0,LSL #4
0x000001AC E80404C1 ADD     r4,r4,r1,LSL #3
0x000001B0 F8544022 LDR     r4,[r4,r2,LSL #2]
0x000001B4 43E4     MVNS   r4,r4
0x000001B6 E8031500 ADD     r5,r3,r0,LSL #4
0x000001BA E80505C1 ADD     r5,r5,r1,LSL #3
0x000001BE F8454022 STR     r4,[r5,r2,LSL #2]
0x000001C2 1C52     ADDS   r2,r2,#1
0x000001C4 2A02     CMP    r2,#0x02
0x000001C6 DBEF     BLT    0x000001A8
0x000001C8 1C49     ADDS   r1,r1,#1
0x000001CA 2902     CMP    r1,#0x02
0x000001CC DBEA     BLT    0x000001A4
23:     for ( i = 0; i < _DIM; i++ )
24:         for ( j = 0; j < _DIM; j++ )
25:             for ( k = 0; k < _DIM; k++
26:                 A[i][j][k] = ~ A[i
27:
0x000001CE 1C40     ADDS   r0,r0,#1
0x000001D0 2802     CMP    r0,#0x02
0x000001D2 DBE5     BLT    0x000001A0
28: }//inverseC_DxDxD-----
29:
30: #ifdef __CC_ARM
31: int main(void){
32: #else
33: void main(void){
34: #endif
35:
0x000001D4 BD30     POP    {r4-r5,pc}

```

Рисунок 21. Дизассемблирование *inverseC_DxDxD()*

Мы ещё вернёмся к более подробному анализу этого кода, а пока лишь отметим, что он получился достаточно громоздким. Из курса изучения языка Си нам известно, что любой многомерный массив представляется в памяти компьютера (микроконтроллера) как одномерный с таким же числом элементов. Поэтому мы несколько модернизируем нашу функцию с целью сокращения размера машинного кода.

Чтобы получить зачёт по данной лабораторной, достаточно перевести на ассемблер и

этот громоздкий код. Но ещё раз напомним, в каких случаях используют ассемблер. Его используют тогда, когда хотят получить максимально эффективный код. Мы попытаемся это сделать.

Вместо функции `inverseC_DxDxD()`, инвертирующей 3-х мерный массив, мы напишем функцию `inverseC_1()`, инвертирующую одномерный.

```

4      * @author  nediak.serg@yandex.ru
5      * @version V1.0.0
6      * @date    18.07.2014
7      * @brief   Main program body
8      *          Из функции на Си передать 3-х мерный массив как параметр
9      *          в функцию на ассемблере, инвертировать в ней каждый элемент
10     *          массива, вернуть результат.
11     *          *****
12     #define _DIM    2
13     int A[_DIM][_DIM][_DIM] = {
14         1,2,
15         3,4
16         ,
17         5,6,
18         7,8
19     };
20     //-----
21     void inverseC_1(int A[_DIM * _DIM * _DIM]){
22         int i;
23         for ( i = 0; i < ( _DIM * _DIM * _DIM ); i++ )
24             A[i] = ~ A[i];
25     } //inverseC_1-----
26     #ifdef __CC_ARM
27     int main(void) {
28     #else
29     void main(void) {
30     #endif
31
32     inverseC_1( (int*) A );
33
34     while(1);
35 }
36 /* END OF FILE Main.c */

```

Рисунок 22. `inverseC_1()`

Такой исходный код на Си, рис. 22, даёт следующий дизассемблированный код, рис. 23.

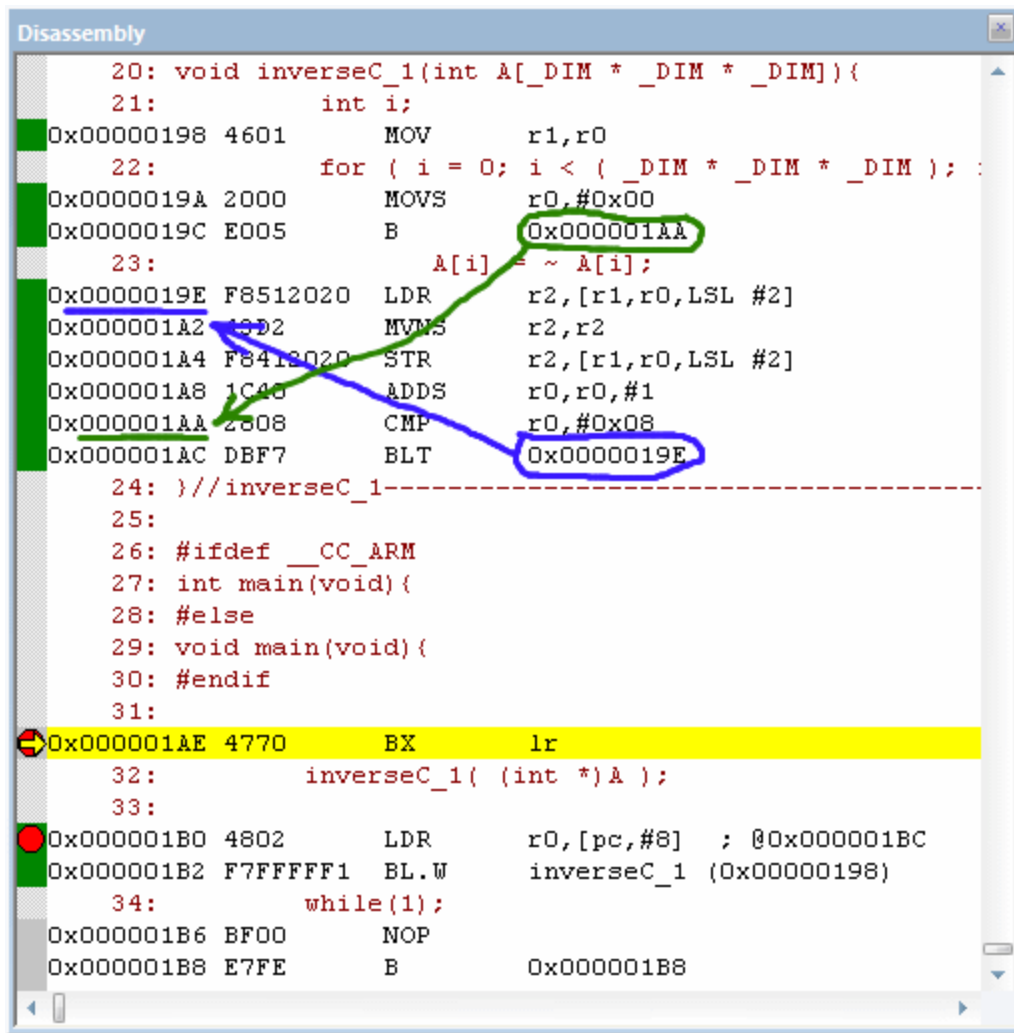


Рисунок 23. Дизассемблирование *inverseC_1()*

Размер кода на Си сократился несущественно, всего на 3 строчки, а вот ассемблерный код и соответственно машинный код сократились вполне ощутимо и, следовательно, этот вариант Си-кода более достоин перевода на язык ассемблера.

Сохраним этот фрагмент дизассемблированного кода в отдельном файле с именем *inverseASM_1.s* и удалим из него лишнее. К тому же добавим необходимые строки из файла-скелета ассемблерной функции. Результатом нашей работы будет исходный код функции *inverseASM_1()*; на ассемблере, рисунок 24.

```

2 ;----- Три следующие строки добавлены. -----
3             AREA x, CODE
4 inverseASM_1 PROC
5             EXPORT inverseASM_1                [WEAK]
6 ;----- См. skeletonASM.s -----
7
8             MOV     r1,r0
9 ;   21:         int i;
10            MOVS   r0,#0x00
11            B      CMP_
12 ;   22:         for ( i = 0; i < ( _DIM * _DIM * _DIM ); i++ )
13 for__       LDR    r2,[r1,r0,LSL #2]          ; чтение из памяти
14 ;   23:         A[i] = ~ A[i];                //инверсия на Си
15            MVNS  r2,r2                        ; инверсия на ASM
16            STR   r2,[r1,r0,LSL #2]          ; запись в память
17            ADDS  r0,r0,#1                    ; i++
18 CMP_        CMP   r0,#0x08                  ; ( i < 8 ) ?
19            BLT   for__                       ; Переход, если истина.
20
21            BX    lr
22
23            ENDP                               ; inverseASM_1 Эта строка добавлена.
24 ;
25 ;   24: }//inverseC_1
26            END                               ; Эта строка тоже добавлена

```

Рисунок 24. Функция (процедура) *inverseASM_1*

Включаем нашу функцию в проект, компилируем и тестируем её. Всё — лабораторная готова. Пишем отчёт и можно предьявлять работу для сдачи.

Но на всякий случай ещё раз читаем задание «...задан трёхмерный массив...». А трёхмерный массив чего? В качестве элементов массива могут быть целые, символьные, числа с плавающей точкой, структуры, наконец — т.е. любые элементы, допустимые в языке Си. У нас есть два пути: либо уточнить у преподавателя задание, либо написать функцию, которая будет работать в любом случае.

Попробуем написать такую универсальную функцию. Заметим, что для того, чтобы решить поставленную задачу, совершенно не обязательно знать, какого типа данные располагаются в массиве. Достаточно знать только адрес начала массива и его размер в байтах. Отсюда следует, что функция инвертирования будет универсальной, если она будет выполнять это действие побайтно. Реализовываем наши идеи сначала на языке Си, рисунок 25.

```

5      * @version V1.0.0
6      * @date    18.07.2014
7      * @brief   Main program body
8      *           Из функции на Си передать 3-х мерный массив как п
9      *           в функцию на ассемблере, инвертировать в ней каждый
10     *           массива, вернуть результат.
11     *****
12     #include "stdint.h"
13     #define _DIM    2
14     int32_t   A[_DIM][_DIM][_DIM] = { 1,2,3,4,5,6,7,8 };
15     float    F[_DIM][_DIM][_DIM] = { 1,2,3,4,5,6,7,8 };
16     char     C[_DIM][_DIM][_DIM] = { 1,2,3,4,5,6,7,8 };
17
18     //-----
19     void inverseCh( uint8_t *A, int32_t N){
20
21         int32_t i;
22         for ( i = 0; i < N; i++ )
23             A[i] = ~ A[i];
24     }//inverseCh-----
25
26     #ifdef __CC_ARM
27     int main(void){
28     #else
29     void main(void){
30     #endif
31
32     inverseCh( (uint8_t *) A, sizeof(A) );
33     inverseCh( (uint8_t *) F, sizeof(F) );
34     inverseCh( (uint8_t *) C, sizeof(C) );
35     while(1);
36     }
37     /* END OF FILE Main.c */

```

Рисунок 25. Исходный код на Си

а затем и на ассемблере. Наша универсальная функция будет называться `inverseASM` и мы её добавим в тот же исходный файл `inverseASM_1.s`, рисунок 26.

```

1  ;----- Три следующие строки добавлены. -----
2      AREA x, CODE
3  inverseASM_1  PROC
4      EXPORT  inverseASM_1                      [WEAK]
5  ;----- См. skeletonASM.s -----
6      MOV     r1,r0                            ; r0 - адрес начала массива.
7  ;    21:      int i;
8      MOVS   r0,#0x00
9      B      CMP_
10 ;    22:      for ( i = 0; i < ( _DIM * _DIM * _DIM ); i++ )
11 for__      LDR     r2,[r1,r0,LSL #2]          ; чтение из памяти слова
12 ;    23:      A[i] = ~ A[i];                //инверсия на Си
13      MVNS  r2,r2                            ; инверсия на ASM
14      STR   r2,[r1,r0,LSL #2]                ; запись в память слова
15      ADDS  r0,r0,#1                          ; i++
16 CMP_      CMP    r0,#0x08                    ; ( i < 8 ) ?
17      BLT  for__                              ; Переход, если истина.
18
19      BX   lr                                ; Возврат из inverseASM_1
20      ENDP                               ; inverseASM_1 Эта строка добавлена.
21 ;    24: }//inverseC_1
22 ;-----
23 inverseASM  PROC
24      EXPORT  inverseASM                      [WEAK]
25 ;----- См. skeletonASM.s -----
26      MOV     r2,r0                            ; r0 - адрес начала массива,
27      MOVS   r0,#0x00                          ; r1 - число байт N.
28      B      CMP2_
29 for2_      LDRSB  r3,[r2,r0]                  ; чтение из памяти байта
30      MVNS  r3,r3                            ; инверсия на ASM
31      STRB  r3,[r2,r0]                        ; запись в память байта
32      ADDS  r0,r0,#1                          ; i++
33 CMP2_      CMP    r0,r1                      ; ( i < N ) ?
34      BLT  for2_                              ; Переход, если истина.
35
36      BX   lr                                ; Возврат из inverseASM
37      ENDP                               ; Конец inverseASM
38 ;-----
39      END                                  ; Конец asm-файла

```

Рисунок 26. Исходник на ассемблере

Универсальная функция `inverseASM()` получилась такого же размера как и первая `inverseASM_1()`, а вот скорость её работы в несколько раз меньше; вычислять скорость мы научились в предыдущей лабораторной работе. Это и понятно, поскольку в первом случае за одинаковое количество тактов обрабатывается (читается, инвертируется, записывается) сразу по четыре байта, а во втором только один. Исходный код `main()`, откуда вызываются наши функции, находится в файле `main2.c`.

Понятие «эффективный код» требует уточнения. Желательно, чтобы этот код был максимально коротким и скоростным. В подавляющем большинстве случаев эти требования находятся в противоречии — их нельзя обеспечить одновременно. На вопросе оптимизации кода мы остановимся более подробно во второй части этой лабораторной работы. А сейчас в качестве упражнения предлагаем читателю доработать последнюю функцию `inverseASM()` так, чтобы она была не только универсальной, но достаточно скоростной.

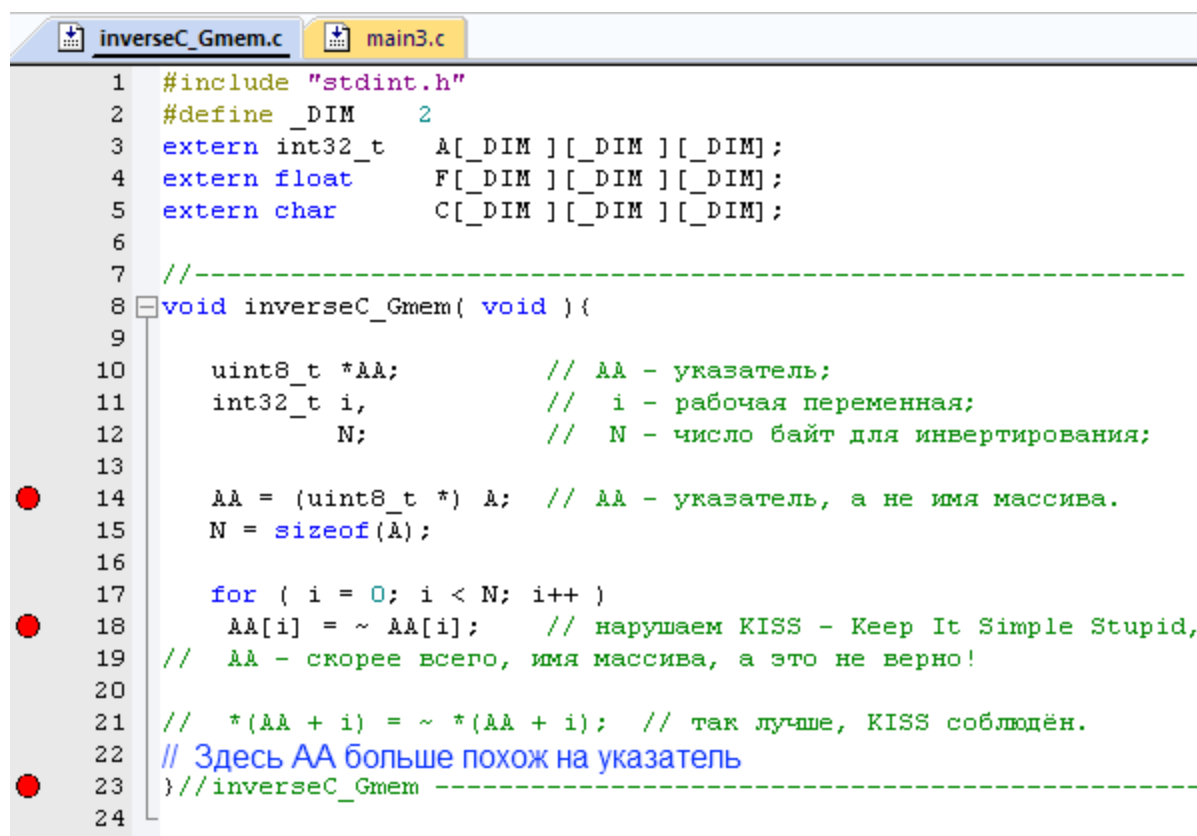
Заключение 4.3. При передаче параметров из функции на языке Си в функцию(процедуру) на ассемблере в нашем случае в регистре r0 передаётся первый параметр — адрес массива, а в регистре r1 передаётся второй параметр — число элементов массива.

4.4 Обмен данными через общую область памяти. Глобальные переменные в Си-модуле.

В первой части этой лабораторной работы, п. 4.3, мы осуществили обмен данными между функциями на Си и ассемблере через параметры функции. Сейчас решим следующую задачу.

В общей (видимой) области памяти объявить трёхмерный массив. С помощью ассемблерной функции инвертировать в нем каждый элемент.

Для достижения поставленной цели мы поступаем прежним способом — сначала пишем функцию на Си, а затем переводим её на ассемблер. Работу в ассемблере с трёхмерным массивом мы даже не рассматриваем, поскольку код в этом случае получается неэффективным — большим и медленным.



```

1  #include "stdint.h"
2  #define _DIM      2
3  extern int32_t   A[_DIM][_DIM][_DIM];
4  extern float     F[_DIM][_DIM][_DIM];
5  extern char      C[_DIM][_DIM][_DIM];
6
7  //-----
8  void inverseC_Gmem( void ){
9
10     uint8_t *AA;      // AA - указатель;
11     int32_t i,        // i - рабочая переменная;
12     N;               // N - число байт для инвертирования;
13
14     AA = (uint8_t *) A; // AA - указатель, а не имя массива.
15     N = sizeof(A);
16
17     for ( i = 0; i < N; i++ )
18         AA[i] = ~ AA[i]; // нарушаем KISS - Keep It Simple Stupid,
19 // AA - скорее всего, имя массива, а это не верно!
20
21 // *(AA + i) = ~ *(AA + i); // так лучше, KISS соблюден.
22 // Здесь AA больше похож на указатель
23 }//inverseC_Gmem -----
24

```

Рисунок 27. Функция *inverseC_Gmem*.

Получилась вот такая короткая функция `inverseC_Gmem()`, которая работает с трёхмерным массивом `A` как с одномерным.

Её дизассемблирование выглядит так.

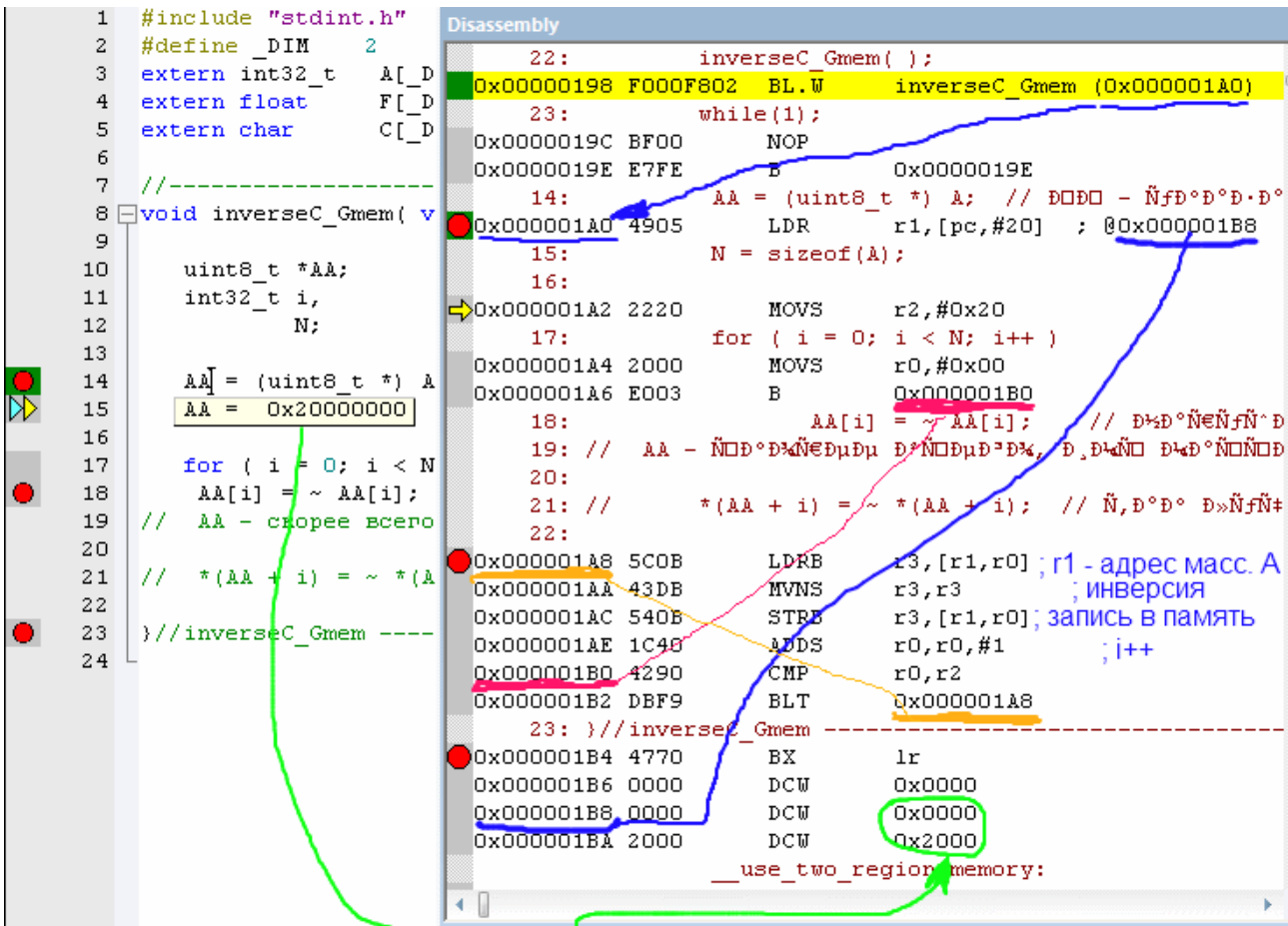


Рисунок 28. Дизассемблирование `inverseC_Gmem`

Сразу напомним о выгоде писать комментарии на английском языке, поскольку вместо русского текста мы видим в окне дизассемблирования абракадабру. Хотя нас это пока мало беспокоит, поскольку наша функция очень короткая и запутаться в нескольких операторах просто невозможно. Но когда анализируется большой проект, то эта «мелочь» уже может доставлять ощутимое неудобство.

Глядя на рисунок 28, мы видим уже хорошо знакомый нам цикл побайтного инвертирования массива `A`, это код по адресам: `0x000001A8 ... 0x000001B2`. Он почти совпадает с прежним. Если `r0` – индекс массива, `r3` – регистр для инвертирования, тогда `r1` будет адресом начала массива, поскольку `[r1, r0]` — это ничто иное как ссылка на адрес очередного инвертируемого байта. Этот адрес равен `[r1] + [r0]` - сумме значений в указанных регистрах.

Как же инициализируется `r1`? Этот регистр инициализируется такой инструкцией

```
LDR    r1, [pc, #0x20]    ; @0x000001B8
```

Она предписывает загрузить в регистр `r1` содержимое слова, находящегося по адресу на `0x20` больше чем текущий. А текущий, как известно, записан в счётчике команд `PC`. Конвейер??? В комментарии к инструкции `LDR` Keil любезно сообщает нам этот адрес — `0x000001B8`. Остаётся только посмотреть, какое значение находится по этому адресу. Там у нас лежит константа `0x20000000`. Именно по этому адресу располагается начало трёхмерного массива `A`. Этим значением инициализируется указатель `AA` (см. исходник `inverseC_Gmem.c`). Если на несколько секунд задержать курсор на переменной `AA`, то Keil

покажет её текущее значение. Естественно, это возможно, когда в сеансе отладки программа остановлена.

LDR — это так называемая **псевдокоманда**, т. е. она преобразуется из какого-то исходного текста на ассемблере в другую настоящую ассемблерную команду. Для того, чтобы определить, какой именно исходный текст ей соответствует, снова обратимся за консультацией к нашему виртуальному учителю Disassembly. Запустим программу на отладку и затем нажмём кнопку «Reset» - самая левая кнопка на панели инструментов. Мы увидим примерно следующее.

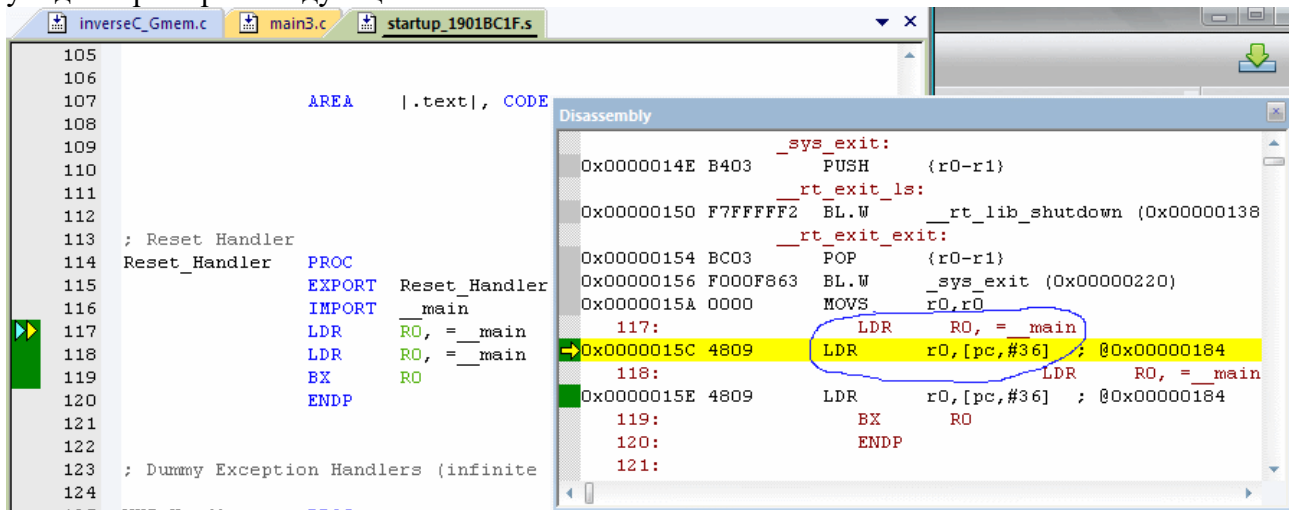


Рисунок 29 Псевдокоманда LDR

Здесь команде

```
LDR          LDR r0, [pc, #36]
```

соответствует оригинал (псевдокоманда)

```
LDR      R0, =__main
```

Ссылка `__main` объявлена как внешняя с помощью директивы `IMPORT __main`. В нашем случае вместо ссылки `__main` это будет, видимо, имя массива `A`.

Полученных знаний достаточно, чтобы написать функцию на ассемблере. Попробуем! Используя результат дизассемблирования и скелет для функций на ассемблере, создаём следующий код.

```

1 ;
2 ;----- Four next strings added. -----
3             AREA x, CODE
4 inverseASM_Gmem  PROC
5             EXPORT  inverseASM_Gmem          [WEAK]
6             IMPORT  A           ;;;;;;;;; insert ;;;;;;;;;
7 ;----- See skeletonASM.s -----
8 ;   14:         AA = (uint8_t *) A; //
9 ;             LDR    r1,[pc,#20] ; @0x000001B8
10 ;            LDR    r1, =A ;;;;;;;;; insert ;;;;;;;;;
11 ;   15:         N = sizeof(A);
12 ;             MOVS   r2,#0x20 ; r2 <- N - number of bytes
13 ;   17:         for ( i = 0; i < N; i++ )
14 ;             MOVS   r0,#0x00 ; r0 <- ( i = 0;)
15 ;             B      CMP__
16 ;   18:         AA[i] = ~ AA[i];
17 ;             inversion loop
18 for__        LDRB   r3,[r1,r0] ; r3 <- AA[i]
19             MVNS   r3,r3 ; r3 inversion
20             STRB   r3,[r1,r0] ; r3 -> AA[i]
21             ADDS   r0,r0,#1 ; i++
22 CMP__        CMP    r0,r2 ; i < N ?
23             BLT    for__ ; goto for__ if true
24 ;   23: }//inverseC_Gmem -----
25             BX     lr ; return;
26             ENDP   ; inverseASM_Gmem PROC END.
27             END    ; inverseASM_Gmem.asm END.
28

```

Рисунок 30. *inverseASM_Gmem*

Функцию *inverseASM_Gmem* включаем в проект и тестируем. Она успешно работает. Одно смущает — предупреждение при компиляции проекта:

..\..\src\ASM\inverseASM_Gmem.asm(27): warning: A1581W: Added 2 bytes of padding at address 0x16 — предупреждение: A1581W: Добавлено 2 прокладочных байта к адресу 0x16.

Что за байты, где этот адрес? Справочная система Keil-а и просмотр машинного кода, 0x16 — это один из адресов таблицы векторов прерывания, мало что дают. Хотя машинный код *inverseASM_Gmem* в точности совпадает с оригиналом *inverseC_Gmem*, в этом легко убедиться, сравнивая их в дизассемблере.

Впадаем в очередную программистскую прострацию — **не знаем, что делать дальше**. Одно из требований MISRA гласит, что проект должен компилироваться без предупреждающих сообщений, следовательно, работа в таком виде не может быть принята.

Самое эффективное средство в таких случаях — выспаться. Это совет не от авторов данного пособия, а от самого Норберта Винера, отца кибернетики. Мы лишь многократно испытали его реальное действие :)). Справедливости ради отметим, что Винер работал в отсутствие Интернета и Википедии.

Отдохнув, продолжаем. Ещё раз читаем предупреждение компилятора. Он нам выдал его, когда обрабатывал строку 27 исходника *inverseASM_Gmem.asm*. А в этой строке находится директива `END`. Уж она-то никак ошибкой (источником предупреждения) быть не может. **Вспоминаем...** У нас похожий код компилировался без предупреждений в функции *inverseASM*. Он отличается от теперяшнего всего лишь одной строкой, а именно

LDR r1, =A

Исключим (закомментируем) её и попытаемся откомпилировать `inverseASM_Gmem.asm`. Ура! Зловредное, потому что непонятное, предупреждение исчезло. Итак, отдохнув, мы сразу же достигли важнейшего результата — **локализовали** ошибку. Теперь с точностью до оператора мы знаем источник проблемы. В большинстве случаев как раз локализовать ошибку бывает значительно труднее, чем её устранить, особенно, если речь идёт о больших проектах.

Мы описали процесс локализации ошибки не самого трудного типа. Наиболее трудно устранимый тип ошибки — это нерегулярные (случайные) зависания программы. Как правило, такой тип ошибок появляется на завершающей стадии проекта, когда размер кода уже достаточно внушительный. Существенно облегчить поиск ошибок этого типа может подробное и регулярное протоколирование своих действий, фактов зависания программы, а также правильное проектирование программы. При наличии хорошего протокола **вспоминать**, когда и при каких условиях начал возникать неприятный эффект, будет значительно легче.

Обращаемся к первоисточнику, а именно к стандартному ассемблерному файлу `startup_1901BC1F.s`²⁰. Ещё недавно мы из него черпали свои знания по программированию на ассемблере. В обработчике прерывания по Reset стоят подряд две команды LDR — строки 117,118. Мы попробуем поступить аналогично — тоже запишем подряд две команды LDR. Собираем проект, проверяем корректность работы. Вот теперь всё успешно работает. Можно писать отчёт и предъявлять лабораторную работу для сдачи.

Перед отправкой работы на сдачу ещё раз, тоже на всякий случай, читаем задание.

В общей (видимой) области памяти объявить трёхмерный массив. ...

Здесь не определено в каком именно модуле этот массив объявить, в Си-шном или ассемблерном. Этап уточнения задания является неотъемлемой его частью. При условии его игнорирования не исключён вариант, что вам придется во время предсессионного цейтнота фактически делать ещё одно задание. В практике часто бывают случаи, когда на завершающей стадии большого проекта выясняется, что Заказчик совсем не то имел в виду. Умение работать с Заказчиком (пока с преподавателем) также важно, как и знание, например, архитектуры микроконтроллера. Это во-первых.

А во-вторых, машинный код `inverseASM_Gmem.asm` получился чуть длиннее и, следовательно, медленнее чем машинный код `inverseC_Gmem.c`. А разве может будущий российский разработчик высоконадёжных систем, да ещё ТУСУРовец, допустить, чтобы его ассемблерный код оказался хуже чем код компилятора Keil ... да ещё в условиях усложняющейся международной обстановки :)).

Поэтому продолжаем совершенствовать нашу функцию `inverseASM_Gmem()`. Первая идея продублировать команду LDR оказалась рабочей, но, к сожалению, замедляющей работу функции. А нам нужно добиться, чтобы ассемблерный код компилировался без предупреждающих сообщений и чтобы не было замедления работы.

Поскольку мы проблему локализовали, то понять, что это за 2 прокладочных (`padding`) байта, теперь будет значительно легче. Снова смотрим на окно дизассемблирования `inverseC_Gmem()`. Вычисляем величину смещения константы, где хранится адрес массива A, относительно адреса команды LDR.

```
0x000001A0  4905      LDR r1, [pc, #20] ; @ 0x000001B8
```

$$\#offset = 0x000001B8 - 0x000001A0 = 0x18 = 24 \neq \#20$$

`#20 = #0x16`. Именно это шестнадцатеричное число присутствовало в сообщении. И, скорее всего, это вовсе и не адрес, как писалось в предупреждении, а смещение (`offset`). Последовательно добавляя после команды LDR несколько команд NOP, т. е. меняя размер

²⁰ Смотри сноску на стр. 45.

кода нашей процедуры и одновременно наблюдая за изменениями смещений и за адресом хранения указателя на массив A (адресом метки A), приходим к выводу, что

$$\text{Адрес_метки_A} = [\text{pc}] + \#\text{offset} + 2.$$

Если этот адрес получился не выровненным по границе слова, т. е. этот адрес не кратен 4, то компилятор его выравнивает, прибавляя к адресу метки A двойку, и выдаёт нам упомянутое предупреждение: *Added 2 bytes of padding at address 0xNN*. Кстати, более корректным будет сообщение: *Added 2 bytes of padding at offset #NN*.

Итак, осталось получить функцию, работающую не хуже автоматически генерируемой компилятором Keil. Сделать это легко, разместив, например, команду NOP после команды

```
BX    lr
```

Здесь в качестве 2-х прокладочных байтов мы использовали двухбайтовую команду NOP.

Заключение 4.4. Чтобы обработать ассемблерной функцией массив `arrayName`, объявленный в модуле на языке Си глобальным, нужно в ассемблерном модуле директивой **IMPORT** `arrayName` объявить на него ссылку как внешнюю, а затем командой **LDR** `rX, =arrayName`, где `rX` – один из рабочих регистров `r0...r12`, установить на этот массив указатель.

4.5 Обмен данными через общую область памяти. Глобальные переменные в ассемблерном модуле

Решаем ту же самую задачу.

В общей (видимой) области памяти объявить трёхмерный массив. С помощью ассемблерной функции инвертировать в нем каждый элемент.

Перейдём к размещению 3-х мерного массива в ассемблерном модуле. За пополнением наших знаний по ассемблеру снова обращаемся к первоисточнику: к файлу `startup_1901BC1F.s`²¹.

```

50 ; Vector Table Mapped to Address 0 at Reset
51
52     AREA    RESET, DATA, READONLY
53     EXPORT  __Vectors
54
55     __Vectors    DCD    __initial_sp           ; Top of Stack
56                 DCD    Reset_Handler        ; Reset Handler
57                 DCD    NMI_Handler          ; NMI Handler
58                 DCD    HardFault_Handler   ; Hard Fault Hand
59                 DCD    MemManage_Handler   ; MPU Fault Handl
60                 DCD    BusFault_Handler    ; Bus Fault Handl
61                 DCD    UsageFault_Handler  ; Usage Fault Han
62                 DCD    0                    ; Reserved
63                 DCD    0                    ; Reserved
64                 DCD    0                    ; Reserved
65                 DCD    0                    ; Reserved
66                 DCD    SVC_Handler         ; SVC Call Handler

```

Рисунок 31. Секция данных

На рисунке 31 мы видим фрагмент оформления секции данных. Пробегая взглядом по этому очень полезному файлу, можно обнаружить ещё одну любопытную для нас строчку.

²¹ Смотри сноску на стр. 45.

```

260         EXPORT    __user_initial_stackheap
261     __user_initial_stackheap
262
263         LDR      R0, = Heap_Mem
264         LDR      R1, =(Stack_Mem + Stack_Size)
265         LDR      R2, =(Heap_Mem + Heap_Size)
266         LDR      R3, = Stack_Mem
267         BX      LR
268
269     ALIGN
270

```

Рисунок 32. Добавление прокладочных (padding) байтов

Мы только что для выравнивания размера машинного кода функции `inverseASM_Gmem` использовали команду `NOP`, а профессионалы в этом месте просто пишут директиву `ALIGN`.

Ассемблер — это низкоуровневый язык программирования и абстрактное понятие «многомерный массив» в нём отсутствует. На низком, очень близком к «железу», уровне мы можем только выделить и проинициализировать некоторую область памяти, которая впоследствии будет интерпретироваться языком высокого уровня как многомерный массив, например, трёхмерный.

Выделяя память, мы поступаем также, как это делается в стандартном ассемблерном файле `startup_1901BC1F.s` при определении таблицы векторов прерывания, а именно.

```

1         AREA     xxx, DATA
2         EXPORT  A
3     ;extern int32_t A[_DIM][_DIM][_DIM] ; See main4.c, inverseC_Gme:
4     A      DCD    1, 2, 3, 4, 5, 6, 7, 8
5

```

Рисунок 33. Секция данных

Мы зарезервировали 8 слов памяти, проинициализировали их, обозначили эту область памяти меткой `A` и объявили её видимой из других модулей директивой `EXPORT A`. Как говаривали в старину, делов на пять копеек, ... и целый рабочий вечер программиста, чтобы понять, что система отладки Keil-а эти странности не совсем адекватно воспринимает. Она упрямо рисует окно `Watch1`, рис. 34. В нём область памяти, отмеченная меткой `A`, интерпретируется как скалярное беззнаковое целое `uint`, а не как массив целых.

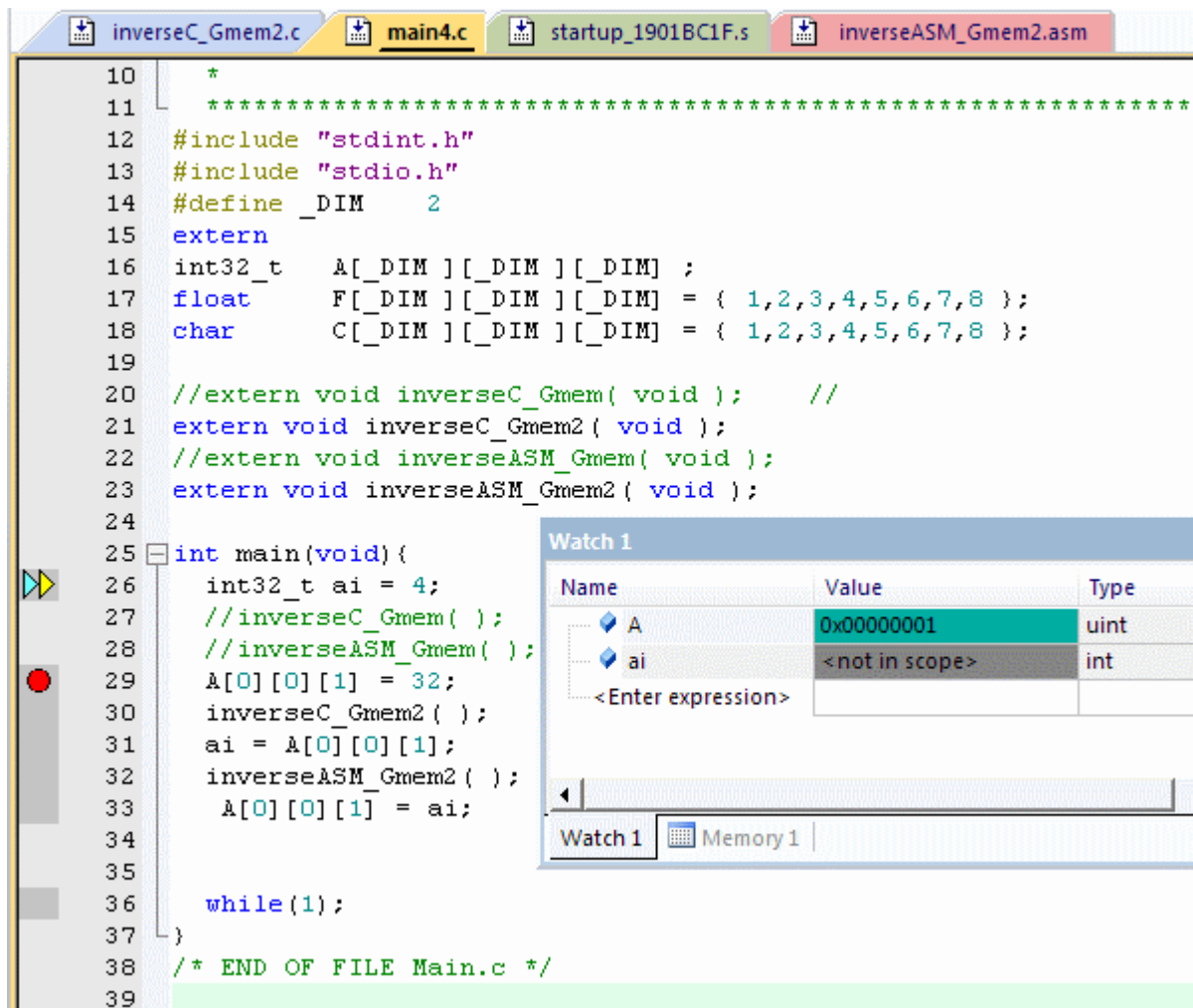


Рисунок 34. Странности восприятия массива A

Заключение 4.5. При выделении общей (видимой) области памяти в ассемблерном модуле под многомерный массив, он **не** воспринимается системой отладки Keil как многомерный. Хотя программа работает корректно, в этом можно убедиться, наблюдая за памятью, где располагается этот массив.


4.6 Заключение

Итак, вместо одной мы фактически запротоколировали выполнение трёх лабораторных работ, взяв в качестве объекта для исследования трёхмерный массив. Заключение по каждому случаю мы описали выше. Надеемся, что методика исследования обмена данными между функциями теперь не вызовет затруднений.

Сейчас поделимся первым впечатлением по поводу среды разработки Keil. Это вполне пригодный продукт для профессиональной работы. По интерфейсу тяготеет к Visual Studio да и вообще к изделиям MicroSoft со всеми вытекающими отсюда достоинствами и недостатками. Более подробно скажем об ассемблере и системе отладки.

Все познаётся в сравнении. Интерфейс по сравнению с IAR-ом лучше — много удобных фишек, облегчающих отладку, например, активная подсветка, обозначающая уже отработавшие участки кода. А вот логика, строгость — хуже.

В первом дизассемблерном окне для функции inverseC_DxDxD(), рис. 21, часть исходного кода функции main(), строки 28: ... 35:, ни с того, ни с сего прилеплена в теле функции inverseC_DxDxD(). IAR с этим же исходником обращается более аккуратно и

подобные глупости не выдаёт. С терминологией неточности — вместо offset в Keil почему то пишут address, а вместо compile пишут translate — подпись к кнопке .

Вначале был план сравнить Keil и IAR по эффективности генерируемого машинного кода — одной из важнейших характеристик среды разработки, но потом от первоначального плана пришлось отказаться. Похоже, это тема для отдельной лабораторной работы, а эта и так уже велика.

В предыдущей и в этой лабораторной работе мы вскользь затронули тему оптимизации машинного кода и достигли «серьёзного» результата — увеличили скорость работы на несколько десятков процентов. При грамотном использовании возможностей компилятора Си этот порядок цифр сохраняется и в более сложных случаях. Улучшение в разы и даже десятки раз при использовании ассемблера обусловлено, как правило, недостаточной опытностью программиста, писавшего код на Си. Напомним, что язык Си был изобретён Николасом Виртом для облегчения труда системных программистов, писавших свои первые операционные системы на ассемблере. Вот уже десятки лет этот язык (инструмент) исправно выполняет свои функции. Но, как и любым инструментом, им нужно уметь пользоваться. Чтобы овладеть навыком эффективного программирования на Си нужно знать ассемблер, как это ни парадоксально звучит. Изображение лупы на иконке Keil выбрал очень удачно.

Здесь без комментариев мы приведём исходные тексты, где показана возможность существенно увеличить скорость /_For_Students/MPSSAU/!Labs/labs-3/Lab3_src.zip. Методика оптимизации программ с использованием ассемблера хорошо изложена у Магды Ю.С [6.].

4.7 Контрольные вопросы

1. Что означает слово «интерфейс» в контексте темы этой лабораторной работы?
2. Какие способы передачи данных из одной функции в другую Вам известны? В чём достоинства и недостатки каждого способа?
3. В чём отличие передачи данных через параметры для архитектур Cortex и Intel? Какое решение с вашей точки зрения наиболее выгодно?
4. Можно ли из ассемблерной функции (процедуры) вызвать Си-функцию и если можно, то как осуществить передачу данных?
5. Что такое псевдокоманда? Как вычисляется смещение относительно текущего адреса? Есть ли различия в формуле, приведённой в этом исследовании, и в формуле, приведённой в справочной системе Keil-a? Ваше мнение по этому поводу?
6. Можно ли сократить код `inverseASM_1()` ? А `inverseASM()`? Смотри рисунок 26.
7. В чём суть метода разворачивания циклов при оптимизации программ с целью увеличения их скорости работы? Смотри [6.].

5 Исследование битовых полей машинного кода с помощью дизассемблера.

Лабораторная работа № 4

Цель. Изучить структуру и алгоритм формирования hex-файлов, получить навыки редактирования таких файлов.

5.1 Введение

В настоящей лабораторной работе мы продолжим знакомиться с ассемблером. Как мы уже говорили ассемблер – это специфический язык. Программирование на нём осуществляется почти на уровне железа, на уровне архитектуры ядра микроконтроллера. Следовательно, сколько архитектур, столько и ассемблеров. К ассемблеру мало подходит слово «изучить», к нему больше подходит слово «выучить» как, например, таблицу умножения. Если в таблице умножения требуется запомнить значения сомножителей и результат перемножения, то в случае ассемблера требуется запомнить мнемонику команды и результат её элементарного воздействия на состояние микроконтроллера. В руководствах по семейству микроконтроллеров система команд записывается в виде таблицы, где кратко поясняется назначение каждой команды. Если чуть прищуриться;-) то она будет очень похожа на таблицу умножения.

Несмотря на достаточно большое разнообразие систем команд, есть в них общая черта – это структура команды, она у всех одинакова. Каждая ассемблерная команда состоит из двух основных частей: символьного кода операции (КОП, англ. COP) и символьного кода операндов. А машинный код, соответственно, состоит из двух битовых полей: поля кода операции и поля кода операндов, рисунок 35.

func1_asm:			
0x7c:	0x2102	MOVS	R1, #2
0x7e:	0x2205	MOVS	R2, #5
0x80:	0x0010	MOVS	R0, R2
0x82:	0x1808	ADDS	R0, R1, R0
0x84:	0x4770	BX	LR
		машинный код	код
		код	операции
			операндов
		адрес	

Рисунок 35. Формат листинга

Если в символьном коде мы можем легко различить две составные части команды, то в машинном коде всё несколько сложнее. Тема этой лабораторной работы в основном и посвящена исследованию структуры машинного кода.

В профессиональной жизни встречаются случаи, когда приходится модифицировать уже работающую программу сторонних разработчиков с целью некоторого изменения её параметров. Сейчас мы научимся это делать, образно говоря, мы научимся разговаривать с микропроцессором на его родном машинном языке.

Ниже на стр. 72 (Рис. 39) изображена машинная программа для микроконтроллера. Она записана в формате машинного кода. Это почти «фотография» памяти контроллера после его программирования. Такие программы на профессиональном жаргоне называют «прошивкой», а процесс программирования «заливкой» программы.

Программирование происходит путём передачи в контроллер с помощью специальных

устройств пакетов данных. Каждый пакет данных имеет свою структуру. Строка в hex-файле как раз и представляет из себя такой пакет данных. Чтобы её отредактировать, нужно обязательно соблюдать правила формирования этой строки. А ещё нужно знать смысл отдельных фрагментов машинного кода. Выполняя эту лабораторную работу, можно почувствовать, насколько был труден хлеб первых программистов. Страшно представить, как можно было написать и отладить большую программу в машинных кодах.

5.2 Содержание работы

1. Получить у преподавателя ассемблерную команду для исследования.
2. Создать новый проект.
3. Создать файл исходного кода функции (скелета) на языке Assembler и включить его в проект.
4. Изменить настройки проекта (options) таким образом, чтобы сформировался загрузочный код вашей программы в Intel-HEX формате.
5. Написать исходный ассемблерный файл так, чтобы его машинный код легко находился в сформированном hex-файле в обычном текстовом редакторе.
6. Изучить формат hex-файла. Отметить различия в представлении машинного кода в дизассемблере и hex-файле.
7. Последовательно меняя операнды заданной команды, исследовать назначение битовых полей машинного кода этой команды.

Техника анализа битовых полей машинного кода хорошо описана в отчёте студентов 539гр. Бушуевой Анастасии Олеговны и Климова Владимира Сергеевича. В конце отчёта есть шутивная фраза: «Получили навыки в редактировании HEX файлов. Мы молодцы =)» Как преподаватели, мы можем совершенно ответственно засвидетельствовать, что это сущая правда – они, действительно, молодцы. Однажды пришли на консультацию с вопросом, что они, де, ничего не знают и ничего не понимают. Если коротко, то смысл ответа был примерно таков: у вас достаточно знаний и умений, идите, думайте и делайте лабораторную работу. Эти студенты блестяще справились с поставленной задачей. Именно поэтому мы посчитали возможным в качестве примера полностью включить их отчёт в настоящее пособие. Далее идёт отчёт студентов.

5.3 Выполнение работы

Ход работы:

1. **Создание нового проекта.**

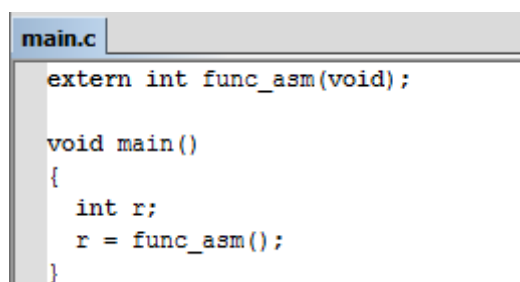
Запускаем IAR и создаем новый проект:

1) Опция Project -> Create New Project. Выбираем C -> main. Сохраняем файл проекта, указывая имя.

- 2) В открывшемся окне main.c объявляем внешнюю функцию на языке Assembler:

```
extern int func_asm(void);
```

- 3) В саму функцию main добавляем код для исполнения внешней объявленной функции:



```
main.c
extern int func_asm(void);

void main()
{
    int r;
    r = func_asm();
}
```

Рисунок 36 Функция main

2. Создание файла исходного кода функции (скелета) на языке Assembler и включение его в проект.

Создадим объявленную функцию и добавим ее в проект:

1) Создадим скелет функции на языке Assembler: Опция File -> New -> File. Откроется пустое окно. Записываем в него скелет функции:

```
main.c func_asm.asm *
PUBLIC func_asm
SECTION SSS : CODE (2)

func_asm:

end
```

Рисунок 37 - Скелет функции на языке Assembler

Сохраняем ее: Правая Кнопка Мыши (ПКМ) по имени окна -> Save ...-> ->Имя_Функции.asm.

2) Добавляем созданную функцию к нашему проекту:

ПКМ по проекту -> Add -> Add Files -> Имя созданной функции

3) В опциях проекта меняем архитектуру:

ПКМ по проекту -> Options -> General Options -> Target -> Device выбираем Milandr 1986BE9x.

4) Запускаем проект в режиме Отладки(Debugging) – Кнопка Download and Debug, - и убеждаемся, что компилятор не выдал ошибок, и открылось окно Dissassembly.

3. Изменение настроек проекта для формирования загрузочного кода программы в Intel-HEX формате.

1) Закрываем режим отладки – Кнопка Stop Debugging. Меняем опции проекта для создания Hex файла:

ПКМ по проекту -> Options -> C/C++Compiler -> Optimizations -> Level ставим None. ...Options -> Output Converter -> Output -> ставим галочку Generate additional output, Output format выбираем Intel Extended и ставим галочку Override default.

2) Изменяем Debug на Release. Запускаем проект в режиме отладки. HEX файл создается в папке проекта: ...\\Debug\\Exe\\Имя_проекта.hex
Файл открывается блокнотом.

4. Написание исходного ассемблерного файла.

В данной работе в качестве объекта исследования выступали две ассемблерные команды PUSH и POP. В ходе работы нам необходимо изучить битовые поля этих инструкций.

Синтаксис

PUSH{cond} reglist

POP{cond} reglist

где:

cond - необязательный код условия.

reglist - заключенный в фигурные скобки список из одного или нескольких регистров, которые должны быть записаны или считаны. В списке можно указывать диапазон номеров регистров. Начальный и конечный регистр диапазона разделены знаком "-". Элементы списка

(отдельные регистры или диапазоны) разделяются запятыми.

Команды PUSH и POP являются синонимами команд STMDB и LDM (LDMIA) в которых базовый адрес памяти содержится в регистре указателя стека SP, а режим записи обратной записи значения базового регистра включен.

Мнемокоды PUSH и POP являются предпочтительными вариантами записи.

Описание

PUSH - сохраняет регистры в стеке в порядке уменьшения номеров регистров, при этом регистр с большим номером сохраняется в память с большим значением адреса.

POP - восстанавливает значения регистров из стека в порядке увеличения номеров регистров, при этом регистр с меньшим номером считывается из памяти с меньшим значением адресом.

Ограничения

В данных инструкциях:

- список регистров reglist не должен содержать указатель стека SP;
- в инструкции PUSH список регистров не должен содержать счетчик команд PC;
- в инструкции POP список регистров не должен одновременно содержать регистры PC и LR.

В случае, если инструкция POP содержит в списке reglist счетчик команд PC:

- бит [0] загружаемого значения должен быть равен 1, передача управления при этом осуществляется по выровненному по границе полуслова адресу;
- если инструкция является условной, то она должна быть последней инструкцией в IT-блоке.

Инструкции PUSH и POP могут быть выражены через инструкции LDM и STM.

Напишем код исходной функции на языке ассемблер, для изучения заданных инструкций:

```
main.c func_asm.asm *
PUBLIC func_asm
SECTION SSS : CODE (2)

func_asm:
    PUSH {R0};
    PUSH {R1};
    PUSH {R2};
    PUSH {R3};
    PUSH {R4};
    PUSH {R5};
    PUSH {R6};
    PUSH {R7};
    PUSH {R8};
    PUSH {R9};
    PUSH {R10};
    PUSH {R11};
    PUSH {R12};
    POP {R0};
    POP {R1};
    POP {R2};
    POP {R3};
    POP {R4};
    POP {R5};
    POP {R6};
    POP {R7};
    POP {R8};
    POP {R9};
    POP {R10};
    POP {R11};
    POP {R12};
    BX LR;           // возврат из функции
end
```

Рисунок 38 Исходник функции на языке ассемблер

В процессе работы мы будем изменять код этой функции, меняя операнды команд для изучения битовых полей.

5. Изучение формата *hex-файла*.

Файл HEX – это текстовый файл, содержащий в символьном виде машинный код программы.

Файл состоит из текстовых ASCII строк. Каждая строка представляет собой одну запись. Каждая запись начинается с двоеточия (:), после которого идет набор шестнадцатеричных цифр кратных байту:

- Начало записи (:).
- Количество байт данных, содержащихся в этой записи. Занимает один байт (две шестнадцатеричных цифры), что соответствует 0...255 в десятичной системе.
- Начальный адрес блока записываемых данных — 2 байта. Этот адрес определяет абсолютное местоположение данных этой записи в двоичном файле.
- Один байт, обозначающий тип записи. Определены следующие типы записей:

0 — запись содержит данные двоичного файла.

1 — запись обозначает конец файла, данных не содержит. Имеет характерный вид «:00000001FF».

2 — запись содержит начальный адрес сегмента памяти.

4 — запись расширенного адреса.

- Байты данных, которые требуется сохранить в EPROM (их число указывается в начале записи, от 0 до 255 байт).
- Последний байт в записи является контрольной суммой. Рассчитывается так чтобы сумма всех байтов в записи была равна 0.
- Строка заканчивается стандартной парой CR/LF (0Dh 0Ah).

После успешной сборки проекта был получен текстовый файл (HEX-фа) LR2_BIT.hex рисунок 39:

```
:1000000000040020C10000008700000087000000FD
:1000100087000000870000008700000000000004B
:10002000000000000000000000000000000008700000049
:100030008700000000000000000000008700000002B
:1000400003B405B409B411B421B441B481B42DE9A9
:1000500001012DE901022DE901042DE901082DE935
:10006000011003BC05BC09BC11BC21BC41BC81BC56
:10007000BDE80101BDE80102BDE80104BDE80108D9
:10008000BDE801107047FEE70120C046002801D0FE
:10009000C046C046002000F002F800F002F8FFF76A
:1000A000CFBF00F001B800000746384600F002F864
:1000B000FBE7000001491820ABBEFBE72600020069
:1000C000C046C046C046C046FFF7DEFF0120801C88
:0400D0007047000075
:04000005000000C136
:00000001FF
```

Рисунок 39 Пример содержимого hex-файла

Рассмотрим структуру одной из записей получившегося hex-файла:

:1000400003B405B409B411B421B441B481B42DE9A9

: - начало записи;

10 - количество байт данных (16 байт);

0040 - адрес памяти, куда будет помещена запись;

00 - тип записи — данные;

03B405B409B411B421B441B481B42DE9 – данные;

A9 - Контрольная сумма записи.

Контрольная сумма вычисляется как дополнение по модулю 256 до нуля суммы всех байт. Т.е. нужно последовательно сложить все байты, так, чтобы результат каждого сложения занимал также один байт, при этом переполнение не учитывается и просто отбрасывается, - это и будет операция сложения по модулю 256. Далее нужно от 256 отнять полученный байт или можно сделать инверсию полученного байта и увеличить результат на 1 – это будет вычисление дополнения по модулю 256 до 0. В результате получается, что сумма по модулю 256 всех байт вместе с контрольной суммой дает ноль. Так проверяется целостность и безошибочность записи при считывании.

Посчитаем контрольную сумму для разобранный ранее записи hex-файла и сравним правильность наших расчетов.

Складываем все байты по модулю 256:

$10+40+03+B4+05+B4+09+B4+11+B4+21+B4+41+B4+81+B4+2D+E9 = 57;$

Инвертируем полученный результат: $\text{not}(57) = A8;$

Прибавляем 1: $A8+1=A9.$

Мы видим, что полученная контрольная сумма совпадает с той, которая была

сформирована в hex-файле, значит можно сделать вывод, что мы посчитали верно.

Если сравнить hex-файл с кодом сформированным дизассемблером, то мы видим, что представление порядка байтов немного разное. В hex-файле сначала идет младший байт, затем старший, а в дизассемблере наоборот, сначала старший, потом младший. Например, запись команды PUSH {R0,R1} в дизассемблере будет выглядеть следующим образом b403, а в hex-файле 03B4.

6. Исследование назначения битовых полей машинного кода заданной команды.

Заданы команды Push и Pop. Необходимо исследовать битовые поля этих команд, меняя их операнды.

Разберемся в логике построения команды в битовом поле. Возьмем для примера команду Push, продублируем ее около 10 раз с разными операндами регистра, заданными по порядку, начиная с нуля. Поставим точку останова рядом с первой командой и запустим Disassembly (рис. 40).

Address	Disassembly
0x40: 0xb401	PUSH {R0}
0x42: 0xb402	PUSH {R1}
0x44: 0xb404	PUSH {R2}
0x46: 0xb408	PUSH {R3}
0x48: 0xb410	PUSH {R4}
0x4a: 0xb420	PUSH {R5}
0x4c: 0xb440	PUSH {R6}
0x4e: 0xb480	PUSH {R7}
0x50: 0xf84d 0x8d04	STR.W R8, [SP, #-0x4]!
0x54: 0xf84d 0x9d04	STR.W R9, [SP, #-0x4]!

Рисунок 40 Отображение команды в Disassembly

Обратив внимание на изменение байтов 0xb401, 0xb402, 0xb404, 0xb408 и т.д., где b4 – байт самой команды, проследим логику изменения младшего байта, т.е. последних двух цифр. Ей соответствует следующее пояснение:

01 есть 0000 0001 в двоичной системе для регистра R0

02 есть 0000 0010 в двоичной системе для регистра R1

04 есть 0000 0100 в двоичной системе для регистра R2

08 есть 0000 1000 в двоичной системе для регистра R3

10 есть 0001 0000 в двоичной системе для регистра R4 и т.д.

Т.е. повышение номера регистра на 1 соответствует изменению порядка единицы в двоичном коде на 1 шаг влево. И переводя двоичный код в Шестнадцатеричный(HEX), мы получаем подобный порядок чисел. Но что будет, если произойдет переполнение байта?

Исходя из рисунка 41, мы видим, что вместо команды Push подключается встроенная команда сохранения регистров STR с окончанием “.W”, что значит “расширенный”. Данная команда (помимо выполнения той же цели, что и PUSH) добавляет следующий байт, чтобы старшим регистрам было, куда продолжать движение единицы в двоичном коде. Но так как команда PUSH заменилась на STR, то само построение немного отличается - изменение

будет происходить уже не с двумя последними цифрами первого байта, а уже в старшем байте (рис. 41):

```

PUSH {R6};
    0x4c: 0xb440      PUSH      {R6}
PUSH {R7};
    0x4e: 0xb480      PUSH      {R7}
PUSH {R8};
    0x50: 0xf84d 0x8d04  STR.W    R8, [SP, #-0x4]!
PUSH {R9};
    0x54: 0xf84d 0x9d04  STR.W    R9, [SP, #-0x4]!
PUSH {R10};
    0x58: 0xf84d 0xad04  STR.W    R10, [SP, #-0x4]!
PUSH {R11};
    0x5c: 0xf84d 0xbd04  STR.W    R11, [SP, #-0x4]!
    
```

Рисунок 41 Изменение в старшем байте

Здесь, в отличие от предыдущих изменений, число в байте уже соответствует числу самого регистра или числу порядка единички в двоичном коде.

Рассмотрим пример посложнее, где в команде будет уже не по одному операнду, а несколько (рис.42).

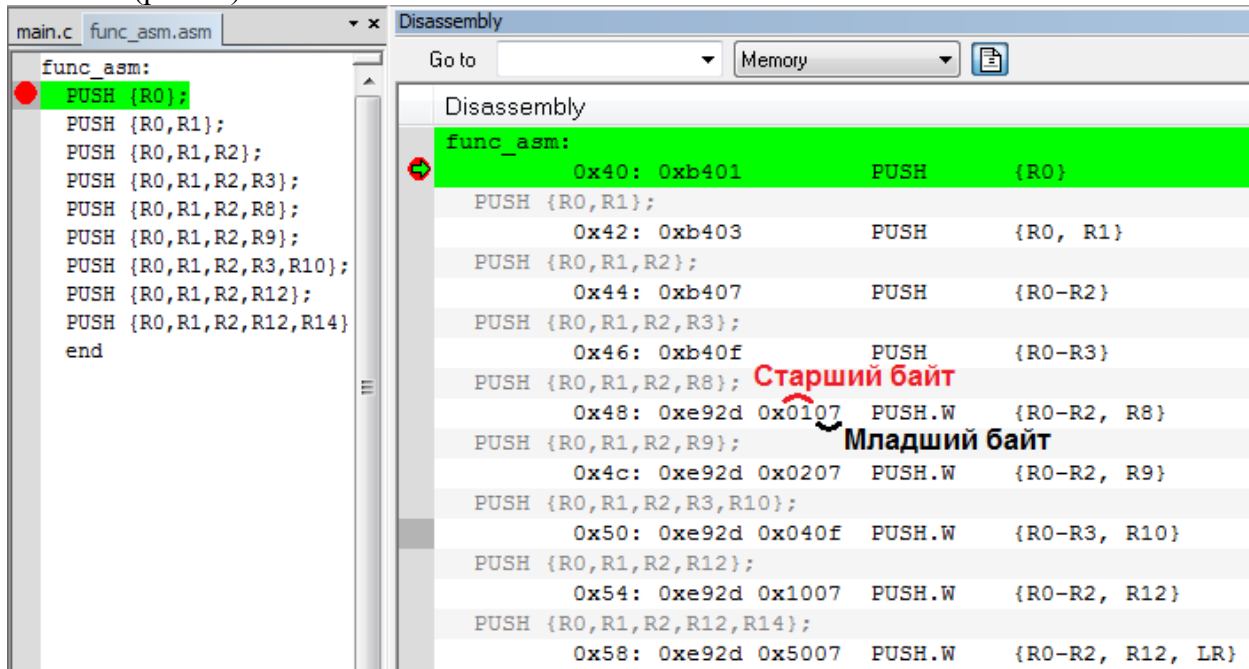


Рисунок 42 - Усложненный пример

01 есть 0000 0001 в двоичной системе для регистра R0

03 есть $\begin{matrix} 0000 & 0001 \\ + & 0000 & 0010 \\ \hline 0000 & 0011 \end{matrix}$ в двоичной системе для регистров R0,R1

07 есть $\begin{matrix} 0000 & 0001 \\ + & 0000 & 0010 \\ + & 0000 & 0100 \\ \hline 0000 & 0111 \end{matrix}$ в двоичной системе для регистров R0,R1,R2

0f есть $\begin{matrix} 0000 & 0001 \\ + & 0000 & 0010 \\ + & 0000 & 0100 \\ + & 0000 & 1000 \\ \hline 0000 & 1111 \end{matrix}$ в двоичной системе для регистров R0-R3

Т.е. происходит обыкновенное сложение двоичных чисел и перевод его в шестнадцатеричное. После расширения “.w” (добавления еще одного байта) в команде Push, порядок начинается заново, но уже в старшем байте. При этом, числа которые складывались для младших регистров записываются в младший байт (показано на рис. 6.3):

01 07 есть $0000\ 0001\ 0000\ 0111$ в двоичной системе для регистров R0-R2,R8

...

50 07 есть $+ \frac{0001\ 0000}{0100\ 0000}$ и $0000\ 0111 = 0101\ 0000\ 0000\ 0111$ в двоичной системе для регистров R0-R2, R12, R14

Точно также формируются последовательности в байтах и для команды POP(рис. 43).

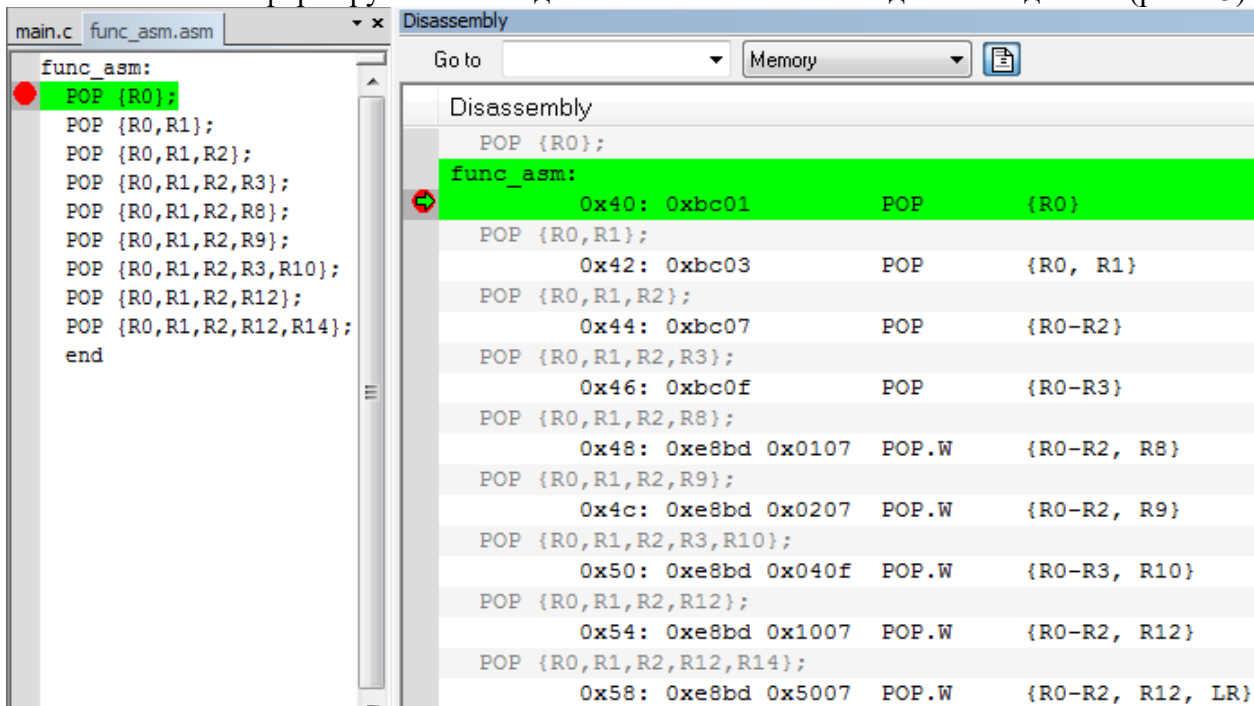


Рисунок 43. Усложненный пример для команды POP

Как видно из рисунка 43, младшие и старшие байты команды POP выстраиваются абсолютно таким же способом, что и в команде PUSH. Отличие состоит только в байтах самих команд.

1. Редактирование одной из строк hex-файла.

Попробуем отредактировать одну из строк hex-файле.

Исходная строка: :1000400001B402B404B408B410B420B440B480B411.

01B4 соответствует регистру R0, поменяем это значение на 03B4, что соответствует R0,R1. Пересчитаем контрольную сумму: 0F.

Сохраним изменения в hex-файле.

5.4 Заключение

В ходе лабораторной работы, мы изучили структуру и алгоритм построения битовых полей в HEX файле и в самом Disassembly. Разобрались в построении битовых полей для заданных команд. Получили навыки в редактировании HEX файлов. Мы молодцы =)

5.5 Требования к содержанию отчёта

1. Подробное описание (протоколирование) работы с использованием копий экрана (screenshot-ов), ошибочных (неудачных) попыток в том числе.
2. Описание назначения битовых полей исследованной команды.
3. Побайтное описание строки из hex-файла, содержащей исследованную команду.
4. Заключение по проделанной работе.
5. Папка с проектом данной лабораторной работы должна быть полностью сохранена в redmine.

Замечание. Допускается представлять только электронную копию отчета, но она должна быть немедленно распечатана по первому требованию преподавателя.

6 Исследование условного исполнения группы команд. Лабораторная работа № 5

Цель. Освоить реализацию на языке Assembler условного оператора «if – then – else» (одна из особенностей ядра Cortex-M3) в традиционном стиле и с использованием возможностей процессора Cortex-M3, т. е. с помощью IT-блока.

6.1 Введение

В этой лабораторной работе мы впервые реально испытаем преимущества написания программ на языке ассемблер, мы оценим скорость работы условного оператора с использованием компилятора Си и с использованием аппаратно реализованного условного исполнения команд. Компилятор Си среды разработки IAR, к сожалению, этого свойства ядра Cortex-M3 не учитывает.

Для начала разберёмся, за счёт чего достигается ускорение работы ЦПУ. Тактовая частота остаётся прежней, а скорость работы, тем не менее, увеличивается.

В микропроцессорах уже давно используется специальное устройство, оно называется конвейер. В разных архитектурах оно реализовано по-разному, но идеология у всех одна – в каждый момент времени выполняется не одна, а несколько команд, процесс очень напоминает работу на конвейере. Так, например, в микропроцессорах Intel одновременно могут выполняться следующие операции:

- выборка команды из памяти;
- декодирование команды;
- генерация адреса операндов;
- выполнение операции с помощью АЛУ;
- запись результата.

В ядре ARM Cortex-M3 одновременно исполняются только три операции:

- выборка;
- декодирование;
- исполнение команды.

Конвейер замечательно работает до тех пор, пока в программе не встретится ветвление, т.е. не встретится нарушение последовательности исполнения команд. В этом случае происходит полный сброс (очистка) конвейера и программа начинает исполняться уже с нового адреса. Задержка сказывается тем ощутимее, чем длиннее конвейер.

В Cortex-M3 есть возможность организовать непрерывную работу конвейера и в случае, когда по алгоритму необходимо ветвление. Здесь реализовано так называемое условное исполнение команд.

В IT-блоке (в *условном блоке*) могут участвовать до 4-х команд. При чтении IT-команды (первой инструкции блока) информация о выполнении или невыполнении каждой команды блока, а также о количестве команд, принадлежащих данному блоку, записывается в регистр состояния исполнения ESPR. И далее на основании этой информации очередная команда либо выполняется, либо заменяется инструкцией NOP (no operation). Таким образом, мы видим, что ветвления, как такового, нет. Тем не менее, алгоритм работы условного оператора сохраняется и, самое главное, сохраняется конвейерный режим обработки команд. Именно этим и достигается увеличение скорости работы контроллера.

При выполнении лабораторной работы обратите внимание на машинный код команд, участвующих в IT-блоке. Значение суффикса условного исполнения никак не меняет машинный код.

6.2 Содержание работы

1. Изучить разделы документации к контроллеру "Миландр" серии 1986BE9x *spec_seriya_1986BE9x.pdf*: [стр.59](#) - программный регистр состояния исполнения EPSR, [стр. 75-76](#) – суффиксы условного исполнения, [стр.126](#) – IT-блок²². Пункт выполнить до занятий в аудитории.
2. Создать новый проект.
3. Написать функцию на Assembler-е, содержащую условный оператор. Си-компилятор среды IAR использует традиционный стиль, без использования расширенных возможностей Cortex-M3.
4. Написать функцию на Assembler-е, реализующую тот же алгоритм, но с использованием возможностей условного исполнения инструкций.
5. Сравнить время исполнения условного оператора в том и в другом случае, зажигая светодиод на отладочной плате микроконтроллера.
6. Пример функций с условным оператором должен быть разработан самостоятельно.

6.3 Выполнение работы

1. Запускаем готовый пример. Сейчас мы воспользуемся исходным файлом, поставляемым с документацией фирмы "Миландр" для работы с портами ввода-вывода. Для этого нужно либо с сайта фирмы, либо с кафедрального сервера скачать следующий пример исходного файла на языке Си: архив `_BKN\MCU\Milandr\Library.rar` и откройте файл `Library\Examples\1986BE91_EVAL\PORT\Joystick_LEDs\main.c`
Исходный текст `main.c` – это разработка *российской* фирмы Phytion. Все комментарии *на английском*, на студентов разработчики явно не рассчитывали...
2. Запустите проект сначала в симуляторе. Он должен успешно откомпилироваться. Если содержатся ошибки – устраните.
3. Внимательно ознакомьтесь с правилами обращения с отладочной платой, детально вспомните содержание лабораторной работы № 0. Помните, что некачественным обращением можно легко вывести отладочный комплект из строя.
4. Подключите отладочный комплект к персональному компьютеру или вашему ноутбуку. Если всё сделано правильно, то светодиод на плате J-Link должен гореть непрерывно. Мигание светодиода свидетельствует об отсутствии связи с компьютером или неверных настройках отладочного комплекта, либо об отсутствии драйвера на J-Link.
5. Измените настройки проекта так, чтобы включился внутрисхемный режим отладки, рис. 44.
 1. В категории Linker следует отметить галочкой `Override default` и указать следующий файл: `$PROJ_DIR_Config\MDR32F1.icf`. Следует убедиться, что этот файл действительно находится по указанному адресу.
 2. В категории Debugger =>Download отметить галочкой `Use flash loader(s)` и отметить галочкой `Override default board file`
`$TOOLKIT_DIR\config\flashloader\Milandr\FlashMDR32F1x.board`.
И в этом случае нужный файл должен находиться по указанному адресу.

22 - В разных версиях документации номера страниц могут быть отличны от здесь указанных, "Миландр" регулярно обновляет документацию. Здесь использовалась Версия 3.2.0 от 20.09.2012.

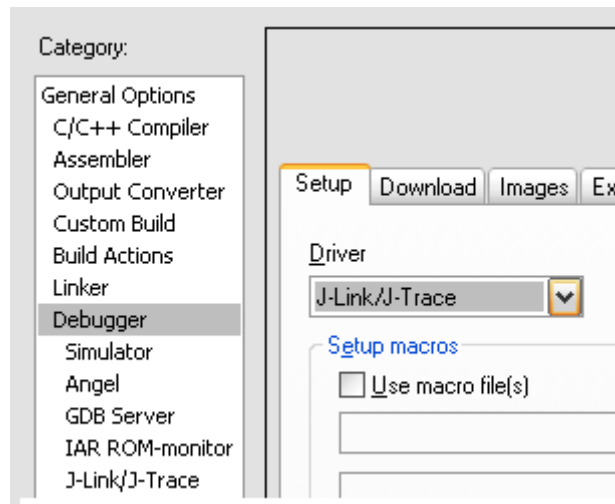


Рисунок 44 Выбор устройства JTAG

3. Запустите проект на отладку. Если все предыдущие действия были выполнены правильно, то светодиоды на отладочной плате должны начать поочерёдно периодически мигать. Итак, вы только что «залили» программу в контроллер. С чем мы вас и поздравляем!
4. Программирование контроллера. Сегодня, возможно, первый раз в жизни вы начнёте (уже начали) программировать контроллеры. Первое, что вы должны сделать, меняя (редактируя) программу в контроллере, это подумать, каким способом вы можете вывести электронные приборы из строя. В некоторых случаях сделать это на удивление легко. Поэтому, если такое событие возможно, нужно предпринять все меры, чтобы оно не произошло. Доскональное изучение документации и текста программ может избавить от многих неприятных проблем.
5. Переключите отладчик (Debugger) для работы в режиме симулятора. Воспользовавшись методикой из лабораторной работы №1, напишите функцию с эквивалентом условного оператора на ассемблере. Эта функция должна работать по алгоритму компилятора с языка Си. Как пример приведём исходный текст на Си:

```
int Func(int r){
    if( r<20 ) r = 5;
    else      r = 6;
return r;
} //Func
```

Эквивалент Func() на ассемблере по алгоритму Си-компилятора:

```

PUBLIC Func
SECTION SSS : CODE (2)
Func:
    CMP     R0, #20      ; if( r<20 ) ~ if( r - 20 <=> 0)
    BGE.N  Func_0       ; if( r - 20 >= 0) goto Func_0;
    MOVS   R1, #5        ; r = 5;
    MOVS   R0, R1
    B.N    Func_1       ; goto Func_1;
Func_0:

```

```

Func_1:
    MOVS    R1, #6      ; r = 6;
    MOVS    R0, R1
    BX      LR          ; return r;

```

И тот же алгоритм с использованием IT-блока:

```

FuncA:
    PUBLIC  FuncA
    SECTION SSS : CODE (2)
    CMP     R0, #20     ; if( r<20 ) ~ if( r - 20 <=>0)
    ITE     LT          ; ----- IT-block begin
    MOVLT   R0, #5     ; r = 5;
                                ; else
    MOVGE   R0, #6     ; r = 6; ----- IT-block end
    BX      LR          ; return r;

```

Обратите внимание, что в теле функции FuncA отсутствуют метки, код адаптирован для безостановочной работы на конвейере.

В пошаговом режиме протестируйте корректность работы данных функций. Для сдачи отчета по лабораторной работе вам нужно придумать какую-нибудь свою функцию с условным оператором.

- Теперь наша задача сравнить скорость работы функции без IT-блока и с таковым. Для оценки времени работы мы будем использовать светодиоды на отладочной плате. Длительность горения светодиода будет характеризовать время работы функции. От студентов 539гр. поступало предложение организовать вызов наших функций в модуле main.c из функции delay(). Т.е. эту функцию предлагалось отредактировать следующим образом:

```

extern int Func(int);           // Func.c | Func.asm
extern int FuncA(int);         // asm - file
void delay( uint32_t time )
{
    for (;time; time--) //;
        Func(22);
        //FuncA(22);
}

```

- Окончательно соберите проект и ещё раз прогоните его в симуляторе. Если ошибки отсутствуют, то переключите отладчик для внутрисхемной отладки и вновь запустите проект.
- Подбирая должным образом времена задержки при зажигании светодиодов, попытайтесь определить длительность работы функций Func() и FuncA().
- Измените настройки проекта Options->C/C++Compiler->Optimizations с None на High и повторите пп. 2.1 – 2.4. Убедились, что компиляторы пишут грамотные люди? Преимущества IT-блока почти сошли на нет. К сожалению, в практике встречаются случаи, когда программа без оптимизации работает устойчиво, а с оптимизацией начинает зависать. Так что с этой опцией обращаться следует осторожно. Если речь идёт о большой программе, то менять степень оптимизации на завершающей стадии проекта нежелательно, поскольку, избавившись от одной

проблемы, можно запросто приобрести другую, ещё большую. На тестирование тоже сильно полагаться не стоит. Здесь нельзя не согласиться с Юровым В.И.[7]: «*надёжность программы достигается, в первую очередь, благодаря её правильному проектированию, а не бесконечному тестированию*». Т.е. в программировании лучше всего, надёжнее всего, полагаться на свою собственную голову. Именно таким программированием мы немедленно и займёмся.

10. Разработайте (придумайте) функцию с условным оператором, чтобы преимущества использования ИТ-блока проявлялись в полной мере. Каким требованиям эта функция должна отвечать? Снова повторите пп. 2.1 – 2.4.

6.4 О побочных возможностях внутрисхемной отладки

Внутрисхемная отладка может обеспечивать перепрограммирование микроконтроллера, когда он уже находится в полностью собранной машине. Это очень удобно.

Но у каждой медали есть обратная сторона. На практике это часто выливается в потребность заниматься программированием (отладкой) в очень шумных производственных условиях.

Теперь контроллер управляет уже не отладочным комплектом, а сложным реальным комплексом, когда очередное неверное изменение программы может иметь катастрофические последствия. Положение усугубляется ещё и тем, что часто оказывается психологическое давление – пусконаладка обычно происходит в напряжённой обстановке.

Здесь нужно придерживаться правила: чем больше вокруг шума, тем *медленнее* и внимательнее вы работаете, а если чувствуете усталость, то от работы нужно сразу же отказаться, как бы сильно всякие источники вокруг не шумели. Работать быстро в такой ситуации – значит работать точно, без ошибок. В состоянии усталости это физически невозможно.

Собой, своим состоянием, тоже нужно уметь управлять. Излишняя самонадеянность здесь абсолютно противопоказана. История техники богата печальными примерами по этой части [5.].

6.5 Вопросы для самопроверки

1. Как работает конвейер в микропроцессорах?
2. Как в ARM Cortex-M3 работает условное исполнение команд? Для чего оно служит?
3. Как передаются параметры в функцию на ассемблере? Как возвращается результат вычислений?

6.6 Требования к содержанию отчёта

1. Краткое изложение теории. Назначение EPSR. С какой целью в Cortex-M3 сделана возможность условного исполнения команд?
2. Описание последовательности работы, ошибочных (неудачных) действий в том числе.
3. Каким способом и с какой точностью было измерено время работы условных операторов.
4. Заключение по проделанной работе.
5. Папка с проектом данной лабораторной работы должна быть полностью сохранена в redmine.

7 Макросредства языка Assembler. Лабораторная работа № 6

Цель: научиться разрабатывать эффективный (быстро работающий) и компактный (читаемый) исходный код на языке Assembler.

7.1 Введение

Макросредства языка ассемблер являются, пожалуй, самым мощным инструментальным средством этого языка. Некоторые из ассемблеров даже и называются в честь этого «макроассемблерами», хотя макро в них не на много больше чем в остальных. Что же это за такое замечательное средство?

Если коротко, то это возможность обозначать группу команд некоторой последовательностью символов и в дальнейшем работать с ней как с новой, лично вашей командой. Называться эта конструкция будет макрокомандой или макро, макросом. Слово произошло от греческого *μάκρος* (*макрос*) — большой, длинный. Синтаксис построен настолько рационально и просто, что новая команда в применении почти ничем не будет отличаться от настоящих команд – у неё также будет присутствовать и мнемокод операции и операнды.

Другими словами, ассемблер можно при желании адаптировать под себя, под решаемую вами задачу. И ещё одно важное свойство. Если вы однажды продумали и оптимизировали вашу группу команд, входящую в макро(с), то машинный код при этом будет всегда получаться наилучшим из всех возможных. Когда используется *макро*, нет накладных расходов на дополнительное использование вычислительных ресурсов как в случае, если бы та же группа команд была оформлена в виде отдельной функции. Но размер машинного кода при использовании *макро* будет больше – за всё хорошее всегда приходится чем-то платить.

Инструмент *макро* может быть также использован при декларировании и инициализации данных, при образовании структур.

Транслятор ассемблера работает в два прохода. Сначала раскрываются все макроопределения – работает программа-макроассемблер, а на втором проходе работает собственно сам транслятор, мнемокод преобразуется в объектный код. Следовательно, *макро* должно быть определено до того, как будет использовано.

7.1.1 Определение *макро*

Вы можете определить (декларировать) макро следующими образом:

```
name MACRO [argument] [,argument] ...  
...  
ENDM
```

name - имя вашей новой макрокоманды, *MACRO* - служебное слово (директива), *argument* – значения аргументов, которые могут передаваться в макрокоманду. А могут вообще и не передаваться. `[]` – скобки означают необязательность присутствия этих параметров. Напомним, что в документации к микроконтроллеру "Миландр" 1986BE9х то же самое назначение у фигурных скобок `{}`, поскольку квадратные скобки используются там для описания операндов. Ну и, наконец, *ENDM* – служебное слово окончания макро.

В данной лабораторной работе могут пригодиться также следующие директивы (служебные слова):

EXITM - выйти из макро;

LOCAL - создать локальный символ (метку) в макро.

Из руководства по ассемблеру Help -> Assembler Reference Guide приведём пример использования макро. Следующий макрос определяет две различные реализации в зависимости от количества используемых аргументов:

```
SECTION MYCODE : CODE (2)
?add MACRO    a,b,c
    IF _args <3
        ADD    a,a, #b
    ELSE
        ADD    a,b, #c
    ENDIF
ENDM
```

Если присутствуют два аргумента, то первый и второй аргументы складываются и эти команды воспринимаются как одинаковые:

main:

```
MOV    R1, #0xF0
?add   R1, 0xFF    ; это,
?add   R1, R1, 0xFF ; и это,
add    R1, R1, #0xFF ; и это то же самое
```

Для более углублённого знакомства с макросредствами языка ассемблер нужно обратиться к упомянутому руководству. Здесь лишь заметим, что в нём нет «воды». Всё изложено очень кратко и доступно, примеры хорошо продуманы, IAR не зря так популярна. Недостаток всего один и то условный – руководство на английском языке. На крайний случай можно временно обратиться к русскому классику Юрову В.И. [7].

7.2 Содержание работы

1. Изучите тему MACRO в руководстве по ассемблеру: Help -> Assembler Reference Guide, стр. 60 – 66. Освежите в памяти материал прошлых лабораторных работ. Пункт выполнить до занятий в аудитории.
2. Создайте новый проект. Используйте тот же пример:
3. SV2\exchange_BNK\MCU\Milandr\Library.rar:Library\Examples\1986BE91_EVAL\PORT\Joystick_LEDs\main.c
4. Напишите код на Assembler-e, выполняющий сложение двух однозначных чисел, представленных в двоично-десятичном коде BCD.
5. Данный код оформите в виде макрокоманды и в виде функции.
6. Сравните время работы макрокоманды и функции, зажигая светодиод на отладочной плате. Для оценки времени вам потребуется написать ещё одну функцию на ассемблере, которая в цикле вызывает функцию сложения двух чисел, либо в том же цикле использует для этой цели макрокоманду. Время горения светодиода измеряйте с помощью ручных часов.
7. Решите ту же задачу определения времени работы макрокоманды и функции с помощью стандартных функций из библиотеки языка C. Сравните полученный результат. Если есть большое расхождение, попытайтесь его объяснить.
8. Запустите проект в симуляторе и снова измерьте время. Оцените, насколько точно моделируется работа микроконтроллера.

7.3 Выполнение работы

Сначала вспомним, что такое двоично-десятичный код ([англ. binary-coded decimal](#)) - BCD. Но прежде всего мы твёрдо усвоим разницу между понятиями «бит» и «разряд».

«Однако в один бит много информации не запишешь. Поэтому разработчики вычислительных машин организуют память так, чтобы её основная компонента, или слово, представляла собой группу битов, достаточную для солидной порции информации.»

Сингер М. Мини-ЭВМ PDP-11: программы на языке ассемблера и организация машины. / Перевод с англ. Каргашина А.Ю. и Миркотан А.С. под редакцией Баяковского Ю.М. – М: «Мир», 1984

Если немного перефразировать приведённую здесь цитату, то получится такая абракадабра: «Однако в один килограмм много веса не запихаешь, поэтому производители весов придумали центнер...»

Издание книги относится ко времени, когда мы усердно перенимали (по меткому замечанию Лебедева Сергея Алексеевича «передирали») зарубежный опыт в области создания вычислительной техники.

Бит – это единица измерения информации, т.е. количество информации как раз и измеряется в битах, количество бит может быть дробной величиной, как, например, килограмм – это единица измерения веса, а литр – это единица измерения объема, вес и объём могут измеряться дробными величинами.

Разряд (числовой) – это позиция или место цифры в позиционной системе счисления. Номер позиции всегда целая величина. В цифровой технике в основном используется двоичная система счисления. Это удобно. Разряд в данном случае ассоциируется с реальным логическим устройством, имеющим два устойчивых состояния. Мы можем сказать: « В 32-х разрядный регистр записано 8.3 бита информации». Максимальное количество информации, которое можно записать 32-х разрядный регистр, равно 32-м битам.

В силу этого замечательного факта понятия «бит» и «разряд» часто отождествляются.²³ Для англоязычной публики такое простительно, а для русскоязычных студентов ТУСУРа рекомендуем прочесть книгу замечательных авторов [8].

Сразу же оговоримся. Мы тоже, следуя традиции, вместо слова «разряд» часто будем использовать слово «бит», но так, чтобы из контекста было понятно, когда речь идёт о реальном физическом носителе информации емкостью столько-то бит, а когда о количестве информации.

Вот теперь мы готовы к изучению BCD кодировки. Существует два формата этой кодировки: упакованный формат и неупакованный формат. Носитель информации емкостью в один байт или восемь бит состоит из двух *тетрад*, младшей и старшей. *Тетрада* (четвёрка) состоит из четырёх бит. Упакованный формат кодирует в одном байте две десятичные цифры, по одной цифре на тетраду, а неупакованный одну. Старшая (левая) тетрада в неупакованном формате всегда равна 0 и это поле из 4-х бит называется *зоной*.

Таким образом, упакованный формат кодирует в одном байте десятичные числа в диапазоне от 0 до 99, а неупакованный в диапазоне от 0 до 9. Приведём примеры в Таблице 1

23 Бит - англ. *binary digit* – двоичная цифр

Таблица 1 - Двоично-десятичное представление чисел

Число	Шестнадцатеричн. BCD-код		Двоичный BCD-код	
	Упаков.	Неупак.	Упакованный	Неупакованный
0	00h	00h	0000 0000b	0000 0000b
3	03h	03h	0000 0011b	0000 0011b
63	63h	06 03h	0110 0011b	0000 0110 0000 0011b
863	08 63h	08 06 03h	0000 1000 0110 0011b	0000 1000 0000 0110 0000 0011b

В микропроцессорах INTEL есть специальные машинные команды для работы BCD-кодом. У ядра ARM Cortex-M3 такой возможности нет. В этой лабораторной работе мы напишем и испытаем на скорость работы функцию сложения двух однозначных чисел в BCD формате.

Cortex-M3 умеет складывать только числа, представленные в бинарном коде. Следовательно, выход у нас один – мы должны воспользоваться существующими командами и затем, если необходимо, скорректировать полученный результат.

Если сумма двух однозначных слагаемых не превышает 9, то ничего корректировать не нужно, результат в бинарном коде в точности совпадает с результатом в BCD-коде.

Если сумма превышает 9, то от результата нужно вычесть 10 и в старший байт записать 1, а в младшем оставить разность между полученной суммой и десятью. Так мы получим представление нашего результата в неупакованном BCD-коде.

Всё, что мы только что сказали, запишем на языке C.

Листинг

```
char * sum2BCD( char a, char b )
{
    static char sum_ab[2];
    sum_ab[0] = a + b;
    sum_ab[1] = 0;
    if( 9 < sum_ab[0] )
    {
        sum_ab[0] -= 10;
        sum_ab[1] = 1;
    } //if
    return &sum_ab[0];
} // sum2BCD
```

При обычной записи для выполнения арифметических вычислений мы размещаем старший разряд числа слева, а в памяти контроллера он будет располагаться справа, т.е. число 08 06 03 h в памяти будет располагаться как 03 06 08 h. Здесь действует правило: младший байт по младшему адресу. Так же будет располагаться и наш результат в массиве sum_ab[], младший байт (единицы) в 0-ом элементе массива, а старший (десятки) в 1-м.

Как перевести, приведённый здесь алгоритм, на язык ассемблера, мы уже знаем.

Обсудим метод оценки скорости работы программы с помощью ручных часов. Этот метод, видимо, такой же древний, как и сам ассемблер. Он работает всегда и на любых самых малоразмерных контроллерах. Суть его заключается в закликивании интересующего участка кода на достаточный промежуток времени.

В 5-й лабораторной работе мы уже однажды воспользовались этим методом. Но там мы из модуля (файла) на языке Си вызывали ассемблерную функцию. Сейчас мы должны сначала оценить скорость работы функции, а затем макрокоманды. Это можно сделать только в модуле на языке ассемблера, поскольку нельзя часть файла написать на языке C, а

часть на ассемблере.

Сейчас мы должны научиться из одной ассемблерной функции вызывать другую. Скелет функции, решающей нашу задачу, будет выглядеть следующим образом.

Листинг

```

PUBLIC    Funny
SECTION  SSS : CODE (2)
Funny:
  push   {r9, LR}      ;
  mov    r9, #0x0493e0; 0x0493e0 = 300000
  ;mov   r9, #300000   ; егг ; инициализация параметра цикла
LOOP_:
  ;
  ; код для исследования
  ;
  ;BL   sum2BCDf      ; вызываем функцию sum2BCDf
  ;
  ; конец кода для исследования
  ;
  subs   r9, r9, #1    ; конец
  BGT   LOOP         ; цикла
  pop    { r9, PC }   ; return
end

```

Обратите внимание на команды `push` и `pop`. В первой команде мы запоминаем адрес возврата в стеке, беря его из регистра связи `LR`, а во второй мы извлекаем его и загружаем в счётчик команд `PC`, т.е. осуществляем возврат. Регистр `LR` мы уже использовали при вызове функции `sum2BCDf` и этим действием затёрли адрес возврата. Поэтому осуществить возврат командой

`BX LR`

как мы это делали раньше, в данном случае уже нельзя.

Чтобы измерить время, мы должны проинициализировать параметр цикла так, чтобы время его работы было достаточным для измерения по обычным часам с секундной стрелкой. Затем нужно произвести два измерения: одно с работающим кодом, а второе с закомментированным. Разница времён, делённая на число циклов, и будет равна времени однократной работы интересующего нас участка кода.

На кафедральном сервере по адресу:

\\SV2\exchange\For_Students\MPSSAU\Nediak\Arch

есть готовые примеры исходных кодов рассматриваемых здесь функций. Но вслед за Питером Нортонем и Джоном Соухэ [7.] мы вам советуем попытаться написать их самостоятельно, только так можно чему-нибудь научиться.

Очень ВАЖНОЕ Замечание!!

Студенты иногда возмущаются – почему вы задаёте вопросы, о которых вы нам ничего не рассказывали? Мы вам попытались рассказать и показать, как можно узнать ответы на эти вопросы, а попутно и на многие другие. Профессиональная работа постоянно задаёт задачи, «которые мы ещё не проходили». Это абсолютно нормально. Владение техникой узнавания куда важнее, чем знание готовых ответов.

7.4 Контрольные вопросы

1. Что такое макрос в ассемблере?
2. Возможны ли вложенные макросы?
3. Как в 32-х разрядный регистр записать 8.3 бита информации?
4. В каких случаях выгодно использовать макрокоманды, а в каких функции?
5. Как в BCD кодируются отрицательные числа?
6. Напишите программу на ассемблере, преобразующую неупакованный BCD-код в ASCII-код.
7. Функция `sum2BCD()` написана неверно (непрофессионально). Она содержит очень распространённую ошибку. Исправьте её.
8. По какой причине на ваш взгляд мы сдали свои позиции в разработке вычислительной техники и продолжаем откатываться назад? Что нужно сделать, чтобы выправить ситуацию и нужно ли её выправлять?

7.5 Требования к содержанию отчёта

1. Краткое изложение теории. Для каких целей служат макросредства языка ассемблера?
2. Описание последовательности работы, ошибочных (неудачных) действий в том числе. Возьмите за правило всегда исполнять этот скучный пункт. Иногда протокол помогает в поиске ошибок в большом серьёзном проекте.
3. Каким способом и с какой точностью было измерено время работы функции и макрокоманды.
4. Заключение по проделанной работе.
5. Папка с проектом данной лабораторной работы должна быть полностью сохранена в redmine.

8 Задачи для любителей поупражнять свои мозги

1. Опровержение Второго закона Вейнберга (из прикладной Мерфологии).
Если бы строители строили здание так же, как программисты пишут программы, первый же залетевший дятел разрушил бы цивилизацию.
Можно ли написать программу для микроконтроллера так, чтобы после изменения произвольных k бит в её машинном коде, она, тем не менее, сохраняла свою работоспособность? На вас как на программиста распространяется закон Вейнберга? :))
2. По мнению Пирогова В.Ю. [8.], для людей, знающих ассемблер, все компьютерные запреты, что зайцу стоп-сигнал. Можно ли **законным** способом снять ограничения кик-старт версии IAR?

9 Литература

Основная

1. Джозеф Ю. **Ядро Cortex-M3 компании ARM**: полное руководство : [перевод]. - Додэка-XXI, 2012 — с.535.
2. ARM® IAR Assembler Reference Guide for Advanced RISC Machines Ltd's ARM Cores. [Электронный ресурс] - Eighth edition: June 2007 — 137 p. Режим доступа: Документ доступен из «хелпа» среды разработки IAR Embedded Workbench.
3. **Серия 1986BE9x, K1986BE9x, MDR32F9Qx, K1986BE91H4**, высокопроизводительных 32-х разрядных микроконтроллеров на базе процессорного ядра ARM Cortex-M3. Спецификация микроконтроллеров серии 1986BE9x, K1986BE9x и MDR32F9Qx - © ЗАО «ПКК Миландр» - Версия 3.2.0 от 20.09.2012 URL: "ftp://student:@esau.tusur.ru/_For_Students/MPSSAU/Milandr/Микроконтроллеры и микропроцессоры/1986/spec_seriya_1986BE9x.pdf" - Дата обращения: 01.04.13.
4. **ASM в STM32. Начало**. URL: <http://we.easyelectronics.ru/STM32/asm-v-stm32-nachalo.html>.

Дополнительная

5. Аджиев В. Мифы о безопасном ПО - уроки знаменитых катастроф. URL: "ftp://student:@esau.tusur.ru/_For_Students/MPSSAU/Аджиев Валерий. Мифы о безопасном ПО - уроки знаменитых катастроф — modernlib.ru.doc"
6. Магда Ю.С. Программирование и отладка C/C++ приложений для микроконтроллеров ARM. — М:2012
7. Нортон Питер, Соухэ Джон Язык ассемблера для IBM PC - М:1992
8. Пирогов В. Ассемблер для Windows - СПб:2003
9. Юров В. Assembler. – СПб:2001
10. Яглом А.М., Яглом И.М. Вероятность и информация. – М:1973
11. Cortex™-M3 Technical Reference Manual - Revision: r1p1 - ARM© 2005, 2006 - 384 с.

Часть II. Ввод-вывод в МК «Миландр»

Вторая часть лабораторного практикума связана с задачами практической реализации электронных устройств с использованием МК "Миландр", где МК будет главным управляющим элементом, связанным интерфейсами с датчиками или/и другими микросхемами.

Целью лабораторного практикума является реализация и отладка типовых подпрограмм, реализующих получение данных и обмен данными по различным интерфейсам, имеющихся в МК.

Освоение этой части лабораторного практикума поможет студентам отладить и реализовать устройства по курсовому проектированию и групповому проектному обучению.

1 Общие теоретические замечания

1.1 Порядок работы с блоками ввода-вывода МК

Любая работа с периферийными блоками МК семейства Cortex-M (и других семейств МК), заключается в строго определенной последовательности действий:

1. Задать и подключить нужную периферийную частоту к выбранному блоку. Это достигается функциями модуля *Сброса и тактовых частот* и драйвера *MDR32F9Qx_rst_clk.c*. Для лучшего понимания распределения тактовых частот смотрите рис. 28 на стр.161 технической документации на МК [1].
2. Настроить порты ввода-вывода, соответствующие выбранному периферийному модулю. см. таблицу 2 (Описание выводов микроконтроллеров серии 1986BE9x стр. 10)
3. Сконфигурировать модуль в соответствии с желаемым режимом работы, используя регистры нужного модуля.
4. Сконфигурировать режимы прерывания, если будет использован данный вид обмена информацией.
5. Сконфигурировать режим прямого доступа к памяти, если будет использован данный вид обмена информацией.
6. Если по логике работы программы используемый блок ввода-вывода больше не нужен, его можно отключить, сняв генерацию тактовой частоты этого блока. При этом потребление электроэнергии МК сократиться. Кроме этого сократиться «цифровой шум», что может быть важным для работы аналоговой периферии.

1.2 Стандартная библиотека ввода-вывода и Стандартный программный интерфейс микроконтроллеров ARM® Cortex™

MDR32F9Qx StdPeriph_Driver — стандартная библиотека ввода-вывода, созданная компанией **Фитон**²⁴ на языке Си для микроконтроллеров семейства Cortex-M производства "Миландр". Она содержит функции, структуры и макросы для облегчения работы с периферийными блоками микроконтроллеров. Библиотека документирована, включает примеры по каждому периферийному устройству, полностью поддерживает **CMSIS** (Cortex Microcontroller Software Interface Standard) и бесплатно предоставляется компанией "Миландр".

CMSIS — программный интерфейс микроконтроллеров фирмы ARM семейства Cortex-M. Это программы с открытым исходным кодом обеспечивающие уровень аппаратных

24 Фирма "Фитон" специализируется на разработке, изготовлении и поставке инструментальных средств для микроконтроллеров семейств: 8051, Intel MCS-96, PICmicro фирмы Microchip, Atmel AVR, Хемис XE8000, Sensory RSC-4x, MSP-430 фирмы Texas Instruments, ARM7, MAXQ фирмы MAXIM.

абстракций (Hardware Abstraction Layer - HAL). HAL - программное обеспечение, которое позволяет программисту абстрагироваться от конкретных особенностей аппаратного обеспечения, это обуславливает переносимость кода на другие вычислительные платформы. CMSIS одинакова для всех производителей МК данного семейства, т.е. это стандартный программный модуль для всех МК семейства Cortex-M.

Необходимо заметить, что использование CMSIS и стандартной библиотеки ввода-вывода не является обязательным при разработке программ для МК Cortex-M. Так в книге [6.] примеры программ написаны без использования оных, в них используется традиционный подход **системного программирования** прямого обращения к нужным регистрам МК. Создание CMSIS разработчиком процессора и создание библиотеки ввода-вывода, производителем МК призвано лишь сократить трудозатраты на разработку нового ПО и наличие этих библиотек является ничем иным как важным конкурентным преимуществом по сравнению со многими другими микроконтроллерными разработками. Во многом этот подход и определил на многие годы вперед рыночное преимущество разработки Cortex-M.

Библиотека ввода-вывода и CMSIS находится по адресу «\For_Students\MPSSAU\Milandr\Программное обеспечение\MDR_Library». В этой папке находятся примеры «Examples» для разных отладочных плат «MDR32F9Q1_EVAL», «MDR32F9Q2_EVAL», «MDR32F9Q3_EVAL» и библиотеки CMSIS и MDR32F9Qx_StdPeriph_Driver.

1.2.1 Структура CMSIS

Общий состав, назначение файлов и их взаимосвязь описана в документации на CMSIS [CMSIS-SP-00300-r3p1-00rel0.zip]. Кратко она представлена на рисунке 1.

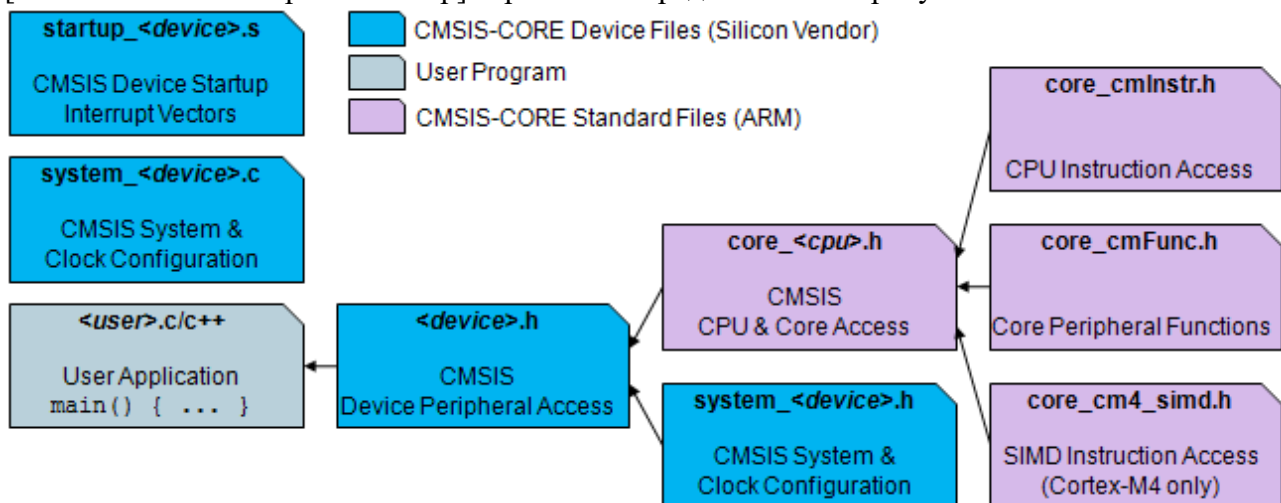


Рисунок 1. - Структура файлов CMSIS²⁵

В папке CMSIS\CM3\CoreSupport находятся программный модуль `core_cm3.c` (*.h). В нем определены функции доступа к регистрам процессора. Такие как, доступ к указателям стека, обращение порядка байт и бит в переменной...см. раздел **Встроенные функции** на стр. 67 [1]. В `core_cm3.h` определены основные структуры для работы с модулями и регистрами центрального процессора: системным блоком управления **System Control Block**, с системным таймером **SysTick**, с блоком защиты памяти **MPU**, с модулем отладки **Core Debug**, с контроллером прерываний **NVIC**.

Вместо файла `<device>.h` как на рисунке, мы имеем файл `MDR32Fx.h`. В нем определена вся информация, которая зависит от конкретного производителя МК — это

²⁵ Рисунок взят из оригинальной документации CMSIS-SP-00300-r3p1-00rel0 с сайта arm.com.

таблица векторов прерываний, структуры и адреса регистров специальных функций (SFR), битовые маски для доступа к отдельным битам SFR.

В папке DeviceSupport\MDR32F9Qx\startup\iar\ находятся два файла **startup_MDR32F9Qx.s** и **system_MDR32F9Qx.c**, как нетрудно догадаться содержимое этих файлов зависит от среды разработки, в данном случае рассматриваем среду разработки фирмы **IAR**. В ассемблерном модуле начального запуска определено, какие функции будут вызываться первыми после системного сброса (SystemInit и __iar_program_start), а также определена таблица адресов вызова обработчиков прерываний и исключительных ситуаций, в нем же инициализируется указатель главного стека (MSP). Как вы думаете, что находится в файлах **core_cmInstr.h** и **core_cmFunc.h** ?

1.2.2 Описание библиотеки MDR32F9Qx_StdPeriph_Driver

Библиотека по своей структуре достаточно проста и разбита на модули, соответствующие модулям МК. Например, работа с модулем аналого-цифрового преобразования (ADC) реализована в файлах MDR32F9Qx_adc.c(*.h) и т.д. для каждого модуля МК. Заголовочные файлы находятся в папке **inc**, исходные тексты в папке **src**.

1.2.3 Описание примеров работы с блоками МК

Примеры конфигурирования и использования всех блоков МК находятся в папке: **_For_Students\MPSSAU\Milandr\Программное обеспечение\MDR_Library\Library\Examples\MDR32F9Q1_EVAL**.

В файлах MDR32F9Qx_board.h, MDR32F9Qx_config.h содержится выбор отладочной платы и определение частот работы МК, некоторых констант и определений.

Далее в папках ADC, ВКР, CAN... находятся примеры проектов работы с соответствующими блоками МК. В каждом примере есть файл **main.c** и может быть MDR32F9Qx_it.c (h) в котором реализован обработчик прерывания.

1.3 Описание демонстрационного проекта MDR32F9Qx_Demo

Кроме описанного программного обеспечения мы также будем использовать демонстрационный проект, который, как правило, предоставляется вместе с отладочной (оценочной) платой, в нашем случае - это **Evolution Board for MCU 1986VE91T rev.4** (см. Таблицу 1).

Проект находится в репозитории по адресу svn://esau.tusur.ru/labs/MDR32F9Qx_Demo/Trunk, который нужно «извлечь» себе в рабочую папку с помощью клиентской программы TortoiseSVN. Или по адресу на сервере sv2 «_For_Students\MPSSAU\Milandr\Программное обеспечение\MDR_Library\MDR32F9Qx_Demo».

Демонстрационный проект московской фирмы «Фитон» реализован достаточно профессионально и неплохо документирован (см. раздел Оформление и документирование), поэтому его можно взять за основу для подражания. Разберем состав проекта и правила разработки многомодульных проектов, когда весь программный код разбивается на отдельные, как правило, функционально различные и в тоже время взаимосвязанные участки кода.

Принцип разбивки на модули достаточно прост и интуитивно понятен и основан на (*правильном!*) представлении иерархичности всей проектируемой программы. Иерархичность заключается в «близости» программного кода к «железу». Для того чтобы это понять, нужно вспомнить (узнать) устройство любой современной операционной системы.

Исследуем состав демонстрационного проекта. Первое, что мы видим в рабочем пространстве - это дерево, привычное по структуре каталогов жесткого диска (ЖД), но эта структура может не соответствовать структуре файлов и папок на ЖД, где хранятся файлы проекта. В корне имя проекта, далее расположены группы файлов (желтая иконка папки). Программный модуль, как правило, состоит из двух файлов с расширениями *name.c* и *name.h*. Программный модуль может использовать функции, переменные, и т.д... другого программного модуля, что приводит к иерархичности и взаимосвязи модулей всего проекта.

1.3.1 Иерархичность проекта MDR32F9Qx_Demo

Самым близким к железу, являются модули, реализующие драйвера отдельных модулей МК. Они находятся в группе **MDR32F9Qx_FWLib**. Развернув группу, мы увидим файлы реализующие функции для работы различных устройств МК. Названия файлов говорят сами за себя. Достаточно лишь знать структурный состав МК. Например, *MDR32F9Qx_adc.c* и *MDR32F9Qx_adc.h* это драйвер модуля АЦП, в котором реализованы все необходимые функции для работы АЦП данного МК.

В Следующих группах **MDR32F9Qx_FatFsLib**, **MDR32F9Qx_USBLib** располагаются модули для реализации файловой системы на флеш-носителях типа «Micro-SD» и модули для обеспечения работы контроллера USB-шины.

В группе **Target** находятся файлы инициализации конкретного МК и конфигурация линковщика.

В группе **Users** находятся «прикладные» или «пользовательские» функции, реализующие уже непосредственно те задачи, которые закладываются в техзадании на разработку ПО.

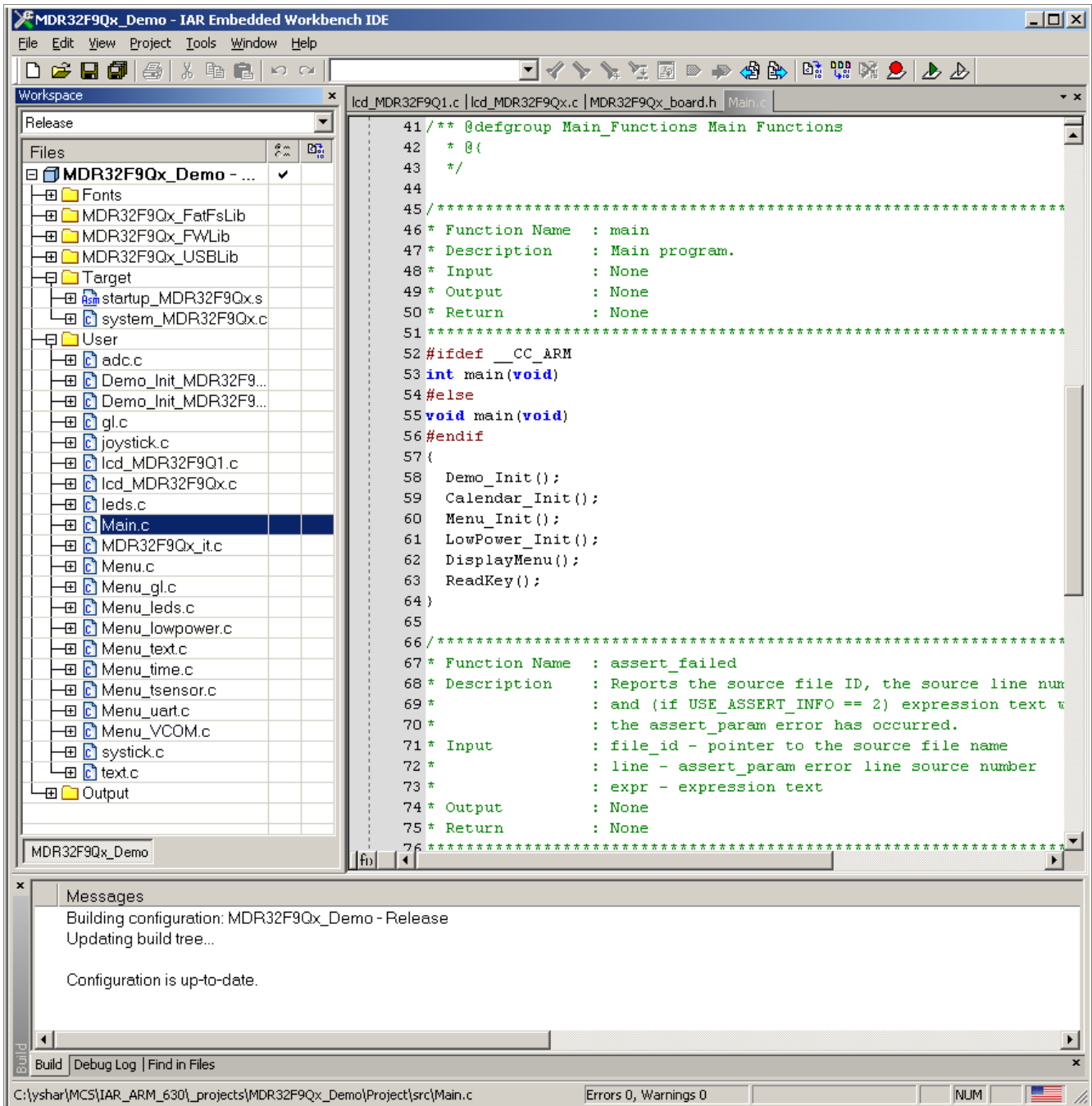


Рисунок 2 — Внешний вид окна IDE с открытым проектом MDR32F9Qx_Demo

Описанная структура групп и файлов выбирается на усмотрение программиста или группы программистов. Мы описанную организацию не считаем достаточно логичной и удобной и навяжем вам свою точку зрения. Вот список рекомендуемых групп:

- 1 **HL_Appl** — прикладные функции верхнего уровня проекта, соответствует группе Users.
- 2 **MCU_drv** — драйвера устройств микроконтроллера, содержащие подгруппы:
 - 2.1 **Config** — где находятся файлы конфигурации проекта : *.icf — конфигурация ликовщика, platform_config.h — привязка к конкретному железу, допустим содержащая строки #define LCD_PORT PORTA.
 - 2.2 **Interrupts** – логично собрать все обработчики прерывания в одном месте. Подумайте почему ?

- 2.3 **Std_Lib** – стандартные библиотеки ввода-вывода, соответствует **MDR32F9Qx_FWLib**
- 2.4 И далее файлы, в которых содержится прикладной функционал ввода-вывода (АЦП, ЦАП, ШИМ, SPI, ...)
- 3 **Protocols** – протокол(ы) обмена данными с другими ЭВМ, если есть обмен с другими ЭВМ.
- 4 **Doc** или **txt** — текстовые файлы описания проекта. Данную группу нужно исключить из обработки компилятором в свойствах группы.
- 5 **Unit_drv** – МК как, правило, всегда работает в связке с другими микросхемами или электронными модулями, здесь и располагаются драйвера внешних по отношению к МК устройств (например дисплей, клавиатура, цифровой датчик...).
- 6 Файл **main.c**, реализующий бесконечный основной цикл работы МК.
- 7 Файл **includes.h** – общий для всех прикладных модулей, в котором подключаются через директиву `#include` заголовочные файлы других модулей проекта, тем самым реализуется сквозная взаимосвязь модулей и группируются общие для всего проекта определения, макросы, псевдофункции.

Для изучения проекта наиболее эффективно его выполнить в пошаговом режиме отладки и проследить, как работают отдельные функции и операторы.

Документация по демопроекту находится в файле `MDR32F9Qx_Demo.chm`, который генерируется автоматически при помощи ПО **DoxyGen** см. раздел 2.2.2 .

1.4 Требования к содержанию отчета

1. Титульный лист.
2. Цель работы.
3. Теоретические основы по работе.
4. Структурная(ые) схема(ы) участка устройства, изучаемого в работе.
5. Исходный текст модифицируемых Вами участков кода и сопутствующего контекста.
6. Список допущенных ошибок их анализ!
7. Выводы по лабораторной работе.
8. Отчет разместить Redmine, проект зафиксировать в репозитории Subversion с заполненным комментарием. Комментарий обязательно должен содержать: Номер ЛР, ФИО, группа и описание изменений в коде.

1.5 Литература

1. Джозеф Ю. Ядро Cortex-M3 компании ARM: полное руководство : [перевод]. - Додэка-XXI, 2012 — 535 с.
2. Мартин Т., Микроконтроллеры фирмы STMicroelectronics на базе ядра Cortex-M3. Серия STM32. Москва: Техносфера, 2009 — 168 с.
3. Документация на библиотеку CMSIS – CMSIS-SP-00300-r3p1-00rel0. URL: www.arm.com (требуется регистрация).

2 Пользовательский ввод-вывод информации в малых вычислительных системах.

Лабораторная работа № 7

В некоторых устройствах на основе МК используются клавиатуры и индикаторы или дисплеи, т.е. эти устройства предназначены для работы с человеком. Конечно же есть и устройства где не предполагается никакого непосредственного интерфейса с человеком. Подумайте и назовите примеры разных устройств на основе МК.

Целью лабораторной работы, является изучение основ ввода-вывода пользовательской информации и программного управления этим процессом в «малых» или встраиваемых (embedded) вычислительных системах.

2.1 Ввод-вывод двоичной информации

Самый простой и необходимый ввод-вывод в микроконтроллерных системах – это обмен двоичной информацией через порты общего назначения, которые подключены к «ножкам» МК. Обычно программист знакомящейся с новым МК пишет первую программу мигания светодиодом, при этом он должен изучить систему тактирования МК, организацию вывода двоичной информации и оценить нагрузочную способность порта вывода. С проектом мигания светодиодом («привет мир») вы уже знакомы, он находится по адресу <svn://esau.tusur.ru/labs/laba0/Trunk/test1>.

В статье «*ARM Cortex-M3 на пальцах: порты ввода-вывода (GPIO)*» находящейся по адресу <http://freehabr.ru/blog/technology/765.html> достаточно хорошо описана работа с отдельными портами ввода-вывода для МК STM32F1xxx.

Для «отличного» понимания работы порта ввода-вывода, поясним схему отдельного порта, приведенную в технической документации на стр. 188, рисунок 32.

Начнем справа налево или снаружи внутрь. PAD – одна «ножка» (вывод) МК. Два защитных диода включенных последовательно между самым низким потенциалом и самым высоким потенциалом схемы, и соединенных с выводом МК. Защитный резистор ESD_R от которого идут два сигнала на аналоговые блоки. Два коммутируемых резистора, программно подключаемых к низкому потенциалу и к высокому, в документации они называются резисторами «подтяжки» (PULL_UP и PULL_DOWN). Два треугольника TX и RX программно управляемые усилители-формирователи выходного и входного сигналов. MUX мультиплексоры, обеспечивающие подключение различных функций и переключение режимов порта. К входам мультиплексоров подключены различные управляющие триггеры, описание которых нужно смотреть в тексте документации.

2.1.1 Работа с отдельными портами. Вывод информации.

С этой задачей большинство студентов должно быть уже знакомо из первого проекта «привет мир» <svn://esau.tusur.ru/labs/laba0/Trunk/test1>, с которого начинается практический курс освоения МК фирмы «Миландр». На «отлично» стоит освоить особый режим доступа к битам порта: «bit-banding». Для этого нужно прочитать раздел «Bit-band регионы» в спецификации на стр. 30., а также соответствующий раздел в [1,2] в предыдущей части пособия 1.5.

2.1.2 Опрос двоичного датчика. Ввод информации.

Под двоичным датчиком мы будем понимать устройство на выходе, которого формируется «логическая единица» или «ноль». Это могут быть различные «концевики»,

контакты, переключатели... и просто кнопки. На первый взгляд может показаться, что опрос двоичного датчика дело крайне простое, конечно, так оно и есть! Но не для новичка... Дело в том, что любая неидеальная система, обладает некоторыми нежелательными свойствами. В случае с механическими коммутаторами – это механические, резонансные колебания и возможно искровые и тепловые явления приводящие к многократному переходу сигнала с коммутатора из одного состояния в другое. Это явление получило название «дребезг контактов». Как с ним бороться придумает, найдет или вспомнит любой студент четвертого курса. Укажем лишь, что борьба с этим явлением ведется, как на аппаратном уровне, так и на программном. Также обратите свое внимание на схему отладочной платы!!

Приведите или придумайте пример двоичного датчика, не обладающего эффектом дребезга контактов?

2.2 Вывод символьной информации

Прежде всего, следует вспомнить (из предшествующих тем электроники и вычислительной техники) виды представления информации в ЭВМ, а также узнать и вспомнить какие виды электронных приборов предназначены для этого.

Таблица 1 — Устройства вывода информации

Тип информации	Устройство
Битовая (логическая) информация	Светодиод, лампа..
Цифровая двоичная	Линейка светодиодов, ламп
Цифры десятичные, шестнадцатеричные...	Семисегментный индикатор
Символы алфавита (ASCII – символы).	Алфавитный-цифровой индикатор
Графическая информация	Графический индикатор (дисплей)

Основные физические принципы лежащие в основе построения индикаторов:

1. Лампа накаливания
2. Газоразрядная лампа
3. Электронно-лучевая трубка (люминесценция при бомбардировки электронами)
4. Излучение светодиода (LED, СИД)
5. Поляризация света жидкими кристаллами (LCD, ЖК)
6. Излучение органического светодиода (OLED)
7. Добавить самим.

Таблица 2 — Таблица подключения дисплея к МК на отладочной плате

Номера выводов дисплея	Обозначение выводов дисплея	Назначение вывода дисплея	Наименование цепи (дописать самим !)	Выводы МК (дописать самим !)
1	UCC	Питание модуля (цифровая часть)		
2	GND	Общий вывод (0В)		
3	U ₀	Вход питания ЖК панели		
4–11	DB0–DB7	Шина данных		
12	E1	Выбор кристалла 1		
13	E2	Выбор кристалла 2		
14	RES	Сброс (начальная установка)		
15	R/W	Выбор: Чтение/Запись		
16	A0	Выбор: Команды/Данные		
17	E	Стробирование данных		
18	UEE	Выход DC\$DC преобразователя		
19	A	+ питания подсветки		
20	K	\$ питания подсветки		

2.3 Ввод информации

Перечислим типы клавиатур:

1. Одна механическая кнопка.
2. Несколько механических кнопок.
3. Матрица кнопок.
4. Матрица кнопок с отдельным контроллером, например клавиатура персонального компьютера.
5. Аналоговые клавиатуры.
6. Сенсорные клавиатуры.
7. Оптические клавиатуры.
8. Добавить самим.

Для ознакомления с принципами устройства клавиатур обратитесь к материалам предыдущих учебных курсов и дополнительной литературе [6-8, 10 - 12].

2.4 Задания

1. Ознакомиться с физическими принципами работы ЖК индикаторов.
2. Ознакомиться с номенклатурой электронных компонентов ЖК индикаторов и дисплеев. Предоставить вариант классификации ЖК индикаторов и дисплеев.
3. Изучить принцип работы графического дисплея МТ-12864j, используемого в отладочных платах фирмы "Миландр".
4. Изучить программный код работы с дисплеем демонстрационного проекта MDR32F9Qx_Demo . Изучить программный код работы с клавиатурой.
5. Добавить свой пункт главного меню в демонстрационном проекте.
6. Из проекта MDR32F9Qx_Demo сформировать свой проект, в котором будет только драйвер дисплея.
7. Добавить драйвер клавиатуры. Продемонстрировать работу клавиатуры и дисплея.

8. Вывести на экран «Привет мир!».
9. Вывести на экран графические примитивы круга, прямоугольника, треугольника.
10. Написать программу вращения прямой линии.

2.5 Контрольные вопросы

1. Назовите примеры различных устройств с интерфейсом «машина-человек» и «машина-машина» на основе МК, МП или ПЛИС.
2. Объясните физические принципы работы известных вам видов индикаторов и дисплеев.
3. Объясните физические принципы работы устройств ввода информации.
4. Объяснить устройство и принцип работы модуля МТ-12864j.
5. В какой функции демопроекта реализуется бесконечный основной цикл программы ?

2.6 Литература

1. Самарин А. В. Жидкокристаллические дисплеи. Схемотехника, конструкция и применение., - СОЛОН-Р - 2002. - 304 с.
2. Томилин М.Г., Невская Г.Е. Дисплеи на жидких кристаллах – СПб: СПбГУ ИТМО, 2010. – 108 с.
3. Жидкокристаллический модуль МТ–12864J. [Электронный ресурс] — Техническая документация. - URL: http://www.melt.com.ru/docs/MT-12864J_en.pdf
4. **Работа с графическим дисплеем WG12864 на базе контроллера KS0107.** - Опубликовано 27 Июль 2011 автором DI HALT <http://easyelectronics.ru/rabota-s-graficheskim-displeem-wg12864-na-baze-kontrollera-ks0107.html>
5. Николайчук О.И., Системы малой автоматизации. — М.: СОЛОН-Пресс, 2003. 256 с. — (Серия «Библиотека инженера»).
6. Электроника и схемотехника : учебное пособие: В 2 ч. / Н. П. Денисов, А. В. Шарапов, А. А. Шибаев; Министерство образования Российской Федерации, Томский государственный университет систем управления и радиоэлектроники. - Томск : ТМЦДО, 2002 - . Ч. 1 : Компоненты электронных устройств. Схемотехника цифровых электронных устройств. - Томск : ТМЦДО, 2002. - 234 с.
7. Николайчук О.И. Схемотехника универсальных технологических контроллеров (цикл статей) // Схемотехника — [ftp://esau.tusur.ru/_BKN/Magazine/Scheme_Tech/shems].
8. Саварин А. Интерфейсы с клавиатурой // Схемотехника - Режим доступа: [[_BKN/Magazine/Scheme_Tech/shems/Digit/st44-50.pdf](ftp://_BKN/Magazine/Scheme_Tech/shems/Digit/st44-50.pdf)].
9. Сташин В.В., Урусова А.В., Мологонцева О.Ф. Проектирование цифровых устройств на однокристалльных микроконтроллерах, М.: Энергоатомиздат, 1990. –224 с. Глава Ввод информации с клавиатуры.
10. Кнопки и клавиатуры. Режим доступа: [ftp://_BKN/Books/_Electrical_Engineering/Electronics/smart card/HTML/Кнопки и клавиатуры.htm](ftp://_BKN/Books/_Electrical_Engineering/Electronics/smart_card/HTML/Кнопки_и_клавиатуры.htm).
11. Пикунوف Владимир Васильевич, Глава 4.Ввод-вывод в микро-ЭВМ. Режим доступа: <http://drive.ispu.ru/elib/pikunov/4.html>, Ивановский государственный энергетический университет, Кафедра электропривода и автоматизации промышленных установок , Электронный конспект лекций - ЭЛЕМЕНТЫ СИСТЕМ АВТОМАТИКИ.
12. KS0108B. Руководство по управлению 64-сегментным драйвером для растровых. ЖКИ. URL: <http://www.gaw.ru/html.cgi/txt/lcd/chips/ks0108b/index.htm>

3 Таймеры-счетчики. Лабораторная работа № 8

Любой микроконтроллер содержит несколько встроенных таймеров-счетчиков (ТС). Причем по своему назначению их можно разделить на две категории. К первой категории относятся **таймеры общего назначения**. Другую категорию составляют **сторожевые таймеры (WDT)**. Сторожевой таймер предназначен для автоматического перезапуска микроконтроллера в случае «зависания» его программы. Более подробно о WDT в дополнительной литературе [<http://www.pic24.ru/doku.php/osa/articles/wdt>]. Здесь мы не говорим о системном таймере **SysTick**, который не относится к периферийным блокам МК, а является частью процессорного элемента.

Таймеры-счетчики цифровые устройства, предназначенные для формирования различных интервалов времени и прямоугольных импульсов заданной частоты. Кроме того, они могут работать в режиме счетчика и подсчитывать тактовые импульсы заданной частоты, измеряя, длительность внешних сигналов, а также при необходимости подсчитывать количество любых внешних импульсов.

Производители МК стараются сделать ТС как можно более функциональными и универсальными. В зависимости от производителя и предполагаемого целевого назначения МК ТС могут иметь различный набор дополнительных функций и режимов работы. Далее для изучения работы ТС читайте описание блока таймеров-счетчиков МК «Миландр» на странице 256 в спецификации на МК.

Цель данной работы заключается в изучении таймеров-счетчиков и их основных режимов работы.

3.1 Ход работы

1. Изучить исходные коды примера **5PWM_Output**.
2. Подключить пример к своему проекту. Запустить пример в режиме отладки.
3. Снять осциллограмму с выходов ШИМ. Измерить параметры ШИМ.
4. Изменить параметры и измерить параметры ШИМ.
5. Изучить исходные коды примера **TIMER_DMA**.
6. Подключить пример к своему проекту. Запустить пример в режиме отладки.
7. Снять осциллограмму с выхода таймера и записать сформированный таймером массив.

3.2 Контрольные вопросы

1. Что такое время ? (Философский).
2. Какие режимы работы поддерживает ТС в МК 1986ВЕ9х?
3. Для чего нужен режим ШИМ ?
4. Для чего предназначен режим захвата?
5. В чем заключается режим «расширенный таймер»?
6. Как организовать каскадное включение таймеров ?
7. С какой минимальной частотой может работать ТС при тактировании от максимальной частоты процессора.
8. Как преобразовать ШИМ-сигнал, формируемый таймером МК в аналоговый сигнал?

4 Аналоговый ввод-вывод. Лабораторная работа № 9

Комбинирование аналоговых измерительных цепей (АЦП-ЦАП) и быстродействующих цифровых цепей (CPU) на одном кристалле задача очень не простая! Микроконтроллеры со встроенными АЦП и ЦАП появились далеко не сразу и относительно недавно. В начале сего века такие МК были еще редкостью, теперь это стандарт де-факто. Но и сейчас, когда нужно получить большую точность измерений или большую скорость, иногда приходится отказываться от встроенных в МК АЦП и ЦАП.

Теоретическая часть аналого-цифрового и обратного преобразования изложена студентам в предыдущем курсе по электронике, для возобновления и пополнения знаний рекомендуем следующие источники [1 - 4].

Целью лабораторной работы, является изучение принципов организации ввода-вывода аналоговой информации в микроконтроллерах. В работе изучаются АЦП, ЦАП и компаратор.

4.1 Работа с АЦП

Все, что касается работы АЦП, встроенного в МК, описано в технической документации (spec_seriu_1986BE9x.pdf) в соответствующем разделе (стр. 303). Здесь поясним лишь то, что окажется не понятным для большинства студентов.

4.1.1 Описание структурной схемы

Входы АЦП образованы 16-ю внешними каналами (ADC_0 .. ADC_15) и двумя внутренними (встроенный датчик температуры **Temp** и встроенный источник опорного напряжения **Vop**). Эти входы подключены к аналоговому мультиплексу **Analog Matrix**. Входы с обозначением **_REF-** **_REF+** предназначены для подключения внешнего источника опорного напряжения. Сами аналого-цифровые преобразователи обозначены **ADC1** и **ADC2** соответственно. Управление работой аналого-цифрового преобразования осуществляется блоком **ADC Control**, где собственно и расположены все регистры управления и задания режимов работы АЦП.

Надо заметить, что реализация блока АЦП у фирмы "Миландр" несколько проще и уступает по ряду технических характеристик, чем у его ближайшего аналога ST32F103x. Попробуйте сравнить сами! Пожелаем разработчикам "Миландр" превзойти зарубежный аналог по всем характеристикам.

4.1.2 Ход работы

1. Ознакомиться с физическими принципами работы АЦП, ЦАП и компаратора.
2. Изучить структурную схему аналоговых блоков МК.
3. Изучить состав и назначение регистров аналоговых блоков МК.
4. Изучить программный код измерения температуры. Файл **adc.c**. Создать программный код измерения нулевого канала первого АЦП. Вывести на экран результат измерения.

4.2 Работа с ЦАП

Работа блока ЦАП достаточно проста и описана технической документации на МК. В демопроекте нет модуля работы с ЦАП. Пример работы с этим блоком можно посмотреть в папке «**Examples\DAC**».

4.2.1 Ход работы

1. Изучить программный код в «**Examples\DAC**» и запустить его. Создать программный код работы с блоком ЦАП.
2. Добавить в проект программный код работы с блоком ЦАП. Изменить программу и получить на выходе напряжение 2.0 вольта. Изменить программу и получить на выходе пилообразный сигнал. Изменить программу и получить на выходе синусоидальный сигнал число цифровых точек не менее 100. Полученные сигналы проверить осциллографом.
3. Выставить на аналоговом выходе напряжение измеряемом с переменного резистора **TRIM**.

4.3 Работа с компаратором

4.3.1 Ход работы

1. Ознакомиться с физическими принципами компаратора.
2. Добавить в проект программный код работы компаратора *Internal_Scale*. Продемонстрировать работу компаратора.
3. При срабатывании компаратора зажгите светодиод.
4. Сконфигурируйте работу по прерыванию при срабатывании компаратора. Обработчик прерывания должен изменить состояние светодиода.

4.3.2 Контрольные вопросы

1. Расскажите, к какому виду принадлежит АЦП используемое в МК, как оно работает.
2. Сравните технические характеристики аналоговых блоков МК фирмы «Миландр» и STMicroelectronics.
3. Объясните работу блоков АЦП и ЦАП в режиме обмена данными без участия процессора.
4. Какая максимальная частота дискретизации АЦП и ЦАП?
5. Какой максимальной частотой сигнал можно оцифровать с помощью АЦП и «восстановить» с помощью ЦАП ?
6. Каким быстродействием обладает компаратор?

5 **Последовательный обмен данными. Лабораторная работа №10**

5.1 **Краткий обзор последовательных «стандартных» интерфейсов МК.**

Все МК имеют набор стандартных интерфейсов последовательного обмена данными. Исторически первым интерфейсом является универсальный приёмо-передатчик (Universal Synchronous Asynchronous Receiver Transmitter) [1], который реализует базовую логику обмена данными по стандартам **EIA RS-232-C, EIA-422-B, EIA RS-485**.

Интерфейс **UART** часто используется для реализации связи по COM-порту, также может использоваться для связи нескольких МК и организации сетевого обмена данными.

В начале 80-х компания Philips разработала интерфейс **I2C** (он же ИС или TWI в документации фирмы Atmel). Сокращение расшифровывается как «**Inter-Integrated Circuit**», т.е. интерфейс для передачи данных между микросхемами (а не модулями или блоками).

Интерфейс **SPI** (Serial Peripheral Interface Bus) также предназначен для обмена данными между микросхемами. Изначально он был придуман компанией Motorola, а в настоящее время используется в продукции многих производителей.

Интерфейсы SPI и I2C нашли широкое применение в различных микросхемах, таких как ЦАП, АЦП, память (Flash, EEPROM, FRAM, MRAM, ШИМ-контроллеры...), таймеры реального времени (RTC) и др. и применяются для расширения функциональных возможностей микропроцессорной техники или для возможности программного управления током, напряжением, потребляемой мощностью....

Любой интерфейс в МК реализуется посредством отдельного устройства — контролера. Хотя любой из этих интерфейсов можно, при большом желании, реализовать программно. Так же необходимо отметить тенденцию унификации, когда в одном блоке реализуется сразу несколько интерфейсов, например USI в МК AVR фирмы Atmel.

В изучаемом Вами МК контроллер SSP (Synchronous Serial Port) реализует несколько сходных протоколов:

- интерфейс SPI фирмы Motorola;
- интерфейс SSI фирмы Texas Instruments;
- интерфейс Microwire фирмы National Semiconductor.

5.1.1 **Базовые понятия последовательной передачи данных**

Интерфейс - совокупность возможностей взаимодействия двух систем, устройств или программ, определённая их характеристиками, характеристиками соединения, сигналов обмена и т. п. Совокупность унифицированных технических и программных средств и правил (описаний, соглашений, протоколов), обеспечивающих взаимодействие устройств и/или программ в вычислительной системе или сопряжение между системами [1].

Протокол (передачи данных) - набор соглашений интерфейса логического уровня, которые определяют обмен данными между различными программами. Эти соглашения задают единообразный способ передачи сообщений и обработки ошибок при взаимодействии программного обеспечения разнесённой в пространстве аппаратуры, соединённой тем или иным интерфейсом.

Сеть (вычислительная) - система связи компьютеров или компьютерного оборудования (серверы, маршрутизаторы и другое оборудование). Для передачи информации могут быть использованы различные физические явления, как правило — различные виды электрических сигналов, световых сигналов или электромагнитного излучения.

Семиуровневая модель OSI - Для единого представления данных в сетях с неоднородными устройствами и программным обеспечением международная организация по

стандартам ISO (International Standardization Organization) разработала базовую модель связи открытых систем OSI (Open System Interconnection). Эта модель описывает правила и процедуры передачи данных в различных сетевых средах при организации сеанса связи. Основными элементами модели являются уровни, прикладные процессы и физические средства соединения.

Последовательная передача данных - передача бит за битом. Может осуществляться, как младшими битами вперед (LSB) так и старшими битами вперед (MSB).

Асинхронная передача - передача данных, при которой интервалы времени между направляемыми блоками данных не являются постоянными. Для выделения в потоке данных блоков в начале и конце каждого из них записываются старт/стопные биты. При асинхронной передаче передатчик и приемник данных работают не зависимо друг от друга. Сигнал синхронизации отсутствует.

Синхронная передача – подразумевается передача сигнала синхронизации.

Дуплексная передача – прием информации и передача одновременно.

Полудуплексная – прием и передача по очереди.

Симплексная – однонаправленная передача данных

Ведущий (Master) – устройство задающие режим передачи информации в сети или между устройствами.

Ведомый, подчиненный (Slave) – устройство

Адрес устройства (узла) — уникальный двоичный код в сети, идентифицирующий узел сети.

5.2 Контроллер UART

Контроллер UART является многофункциональным устройством, может обеспечивать разные режимы передачи данных (? очень похоже что в синхронном режиме он не работает ?) обеспечивает работу с модемами и работу по стандарту IrDA SIR.

В демопроекте реализована простая задача «эхо»: принять то что сам передал. При этом выход приёмника (RXD) подключают ко входу передатчика (TXD) внутри самого UART, но обычно это делается переключкой (проволочкой, пинцетом) на разъеме.

5.2.1 Ход работы

1. Изучите с работой контроллера UART (стр. 363 spec_seriya_1986BE9x.pdf). Изучить программный код Menu_uart.c демонстрационного проекта и ознакомиться с драйвером MDR32F9Qx_uart.c.
2. Изучите функционирование микросхемы ADM3232.
3. Запустите проект ревизию номер 5, изучите работу программы. Найдите каким способом приёмник подключен к передатчику. Разомкните невидимую цепь соединяющую приёмник и передатчик. Пересоберите программу, запустите в МК, оцените результат. Установите видимую цепь, соединяющую приемник и передатчик. Запустите программу оцените результат.
4. Добейтесь приема информации ПК и приема информации МК от ПК.

5.3 Контроллер I2C

I2C является двухпроводным (SDA SCL), двунаправленным последовательным каналом связи с простым и эффективным методом обмена данными между устройствами. Стандарт интерфейса I2C является многомастерным с обнаружением коллизий и арбитражем, исключая потерю данных при обмене, когда два или более мастера пытаются осуществить передачу одновременно. Дальнейшую информацию смотрите в спецификации

на стр. 327.

5.4 Контроллер SSP (SPI)

Данный контроллер реализует несколько схожих интерфейсов передачи данных. Мы с вами познакомимся с самым широко распространенным – **SPI**. Интерфейс дуплексный, синхронный, трех проводной MISO, MOSI, SCLK (не считая линии выбора кристалла CS). Очень простой! В технической документации на МК смотри дальнейшее описание в спецификации на МК на стр. 333.

5.4.1 Ход работы

1. Изучите принципы организации интерфейса SPI. Найдите в демопроекте, где используется контроллер SSP?.
2. Изучите принципы организации интерфейса I2C. Найдите в демопроекте где используется контроллер I2C?
3. Приведите конкретные примеры микросхем с интерфейсами I2C и SPI. Укажите наименование, производителя и стоимость (воспользуйтесь efind.ru).
4. Изобразите схему подключения к МК двух микросхем AD5326
5. Изобразите схему подключения к МК двух микросхем DAC8581
6. Изучите примеры (Examples/SSP). Подключите их себе в проект. Организуйте передачу данных по SPI.
7. Изучите примеры (Examples/I2C). Подключите их себе в проект. Организуйте передачу данных по I2C.
8. Сфотографируйте осциллограммы работы интерфейсов UART, SPI и I2C.
9. Укажите на измеренных осциллограммах битовые поля фреймов передачи данных на измеренных осциллограммах.

5.5 Контроллер CAN

В отличие от предыдущих интерфейсов, CAN (*Controller Area Network*) разрабатывался в 1980-х годах, как сетевой высоконадежный интерфейс и протокол для автомобильного применения, предназначенный для объединения в одну сеть нескольких контроллеров, исполнительных устройств или датчиков. Режим передачи — последовательный, широкополосный, пакетный. Общую ознакомительную информацию вы легко найдете в Интернете на gaw.ru или в Википедии.

CAN получил широкое распространение не только в автомобильной электронике, но и во многих других отраслях: станкостроение, АСУ ТП, авионика, медицинские приборы...и др. Такое широкое распространение обусловлено в первую очередь надежностью передачи данных и универсальностью разработки. Наличие контроллера CAN в современном МК - это стандарт.

В примерах (Examples) есть три проекта: два из них (*LoopBack_Interrupt*, *LoopBack_Polling*) реализуют «эхо» в режиме прерывания и в режиме программного ввода-вывода. Третий (*LoopBack_RTR*) реализует режим запроса на удаленную передачу.

5.5.1 Ход работы

1. Изучить основы организации сети CAN.
2. Запустить и изучить два первых примера передачи данных по CAN.
3. Добиться связи по CAN между двумя отладочными платами.

5.5.2 Контрольные вопросы

1. Поддерживает ли контроллер I2C режим обмена ПДП ?
2. Чего не хватает на демоплате для организации связи по I2C?
3. Что такое арбитраж и как он выполняется в I2C?
4. Изобразите структуру кадра передачи данных по I2C.
5. С каким устройством на демоплате реализована связь по SPI ?
6. Чем отличаются интерфейсы Microware, SPI и SSI.
7. Изобразите структуру кадра по SPI.
8. Какие уровни OSI реализованы аппаратно в контроллере CAN?
9. Изобразите структуру кадра сети CAN.
10. Каких интерфейсов не хватает в лабораторной работе? Какие интерфейсы вы знаете, кроме приведенных в работе?

5.6 Литература

1. Интерфейсы RS232, RS485, I2C, IrDA ... URL:
<http://www.gaw.ru/html.cgi/txt/interface/index.htm>.
2. Краткий обзор протокола CAN. Часть I URL:
http://www.micromax.ru/articles/article.shtml/rewrite/Can_kvazer_1.

3. Для заметок, найденных ошибок, пожеланий

Уважаемый читатель, если Вы нашли в методическом пособии какие-либо ошибки, неточности или у Вас есть пожелания по улучшению содержания, просим вас написать по одному из адресов: mpsau@iit.tusur.ru или nediak.serg@yandex.ru.

Пожелания себе ;)

1. Добавить раздел «часто встречающиеся ошибки» - уже в работе...
2. Исследовать и сравнить оптимизацию компиляторов IAR vs ARM.
3. В части II расписать подробнее и воткнуть поясняющие осциллограммы и схемы.
4. По CAN-подробнее и в отдельную ЛР (или 2-3...) — она того заслуживает...
5. Уточнить! Следит ли кейл за стеком?
6. Пронумеровать листинги.
7. Навести порядок на кафедральном сервере.
8. Переписать текст в TeX.

Недяк С.П., Шаропин Ю.Б.

**Лабораторный практикум по микроконтроллерам
семейства Cortex-M**

**Методическое пособие по проведению работ на отладочных
платах фирмы "Миландр"**

Формат 60x84 1/16. Усл. печ. л. 1,2

Тираж 60 экз. Заказ

Отпечатано в Томском государственном университете
систем управления и радиоэлектроники.
634050, Томск, пр. Ленина, 40. Тел. (3822) 533018.

