Научно-технический отчет по внутреннему ОКР “Защищенный режим”.
Внедрение технологии защищенного режима исполнения Эльбрус в задаче разработки и отладки СПО промышленных контроллеров.

Исполнители:
Мустафин Т. Р., АО «МЦСТ» начальник сектора «Защищенный режим»
Алехин А. И., АО «МЦСТ» инженер-программист
Черкашин С. Ю., АО «МЦСТ» инженер-программист
Зубов И. Н., АО «МЦСТ» инженер-программист
Лубинец М. И., АО «МЦСТ» инженер-программист
Кравцунов Е.М. АО «МЦСТ» начальник отдела «АЗК ОССН»

Москва, 15 декабря 2017 г.

Содержание:

1. Введение

1.1 Поддержка защищенного режима в микропроцессоре, компиляторе и ядре ОС

Работа программ в защищенном режиме строится на основе аппаратной реализации поддержки доступа к объектам исключительно через дескрипторы объектов. Дескриптор отличается от обычного указателя тем, что кроме адреса объекта дескриптор хранит размер объекта и смещение относительно начала объекта в памяти. Дескрипторы защищены от произвольного изменения внешними тегами. Аппаратура (микропроцессоры Эльбрус всех моделей) не допускает сборку нового дескриптора из частей существующих дескрипторов либо создание дескриптора вручную.

Компилятор для защищенного режима использует специальные аппаратные команды для доступа в память через дескрипторы. Также компилятор использует специальные аппаратные инструкции для формирования дескриптора на стековые переменные. Кроме того, в загружаемой с жесткого диска программе или библиотеке нет внешних тегов глобальных переменных, компилятор генерирует код инициализации глобальных переменных для каждой программы или библиотеки.

Для выполнения приложений, скомпилированных в защищенном режиме и использующих системные вызовы, имеется поддержка в ядре операционной системы. Ядро операционной системы реализует обработчик системных вызовов, способный принимать дескрипторы в качестве аргументов системных вызовов. Также ядро обеспечивает контроль над дескрипторами в глобальных переменных, указывающими на стековые переменные.

Поддержка защищенного режима реализована во всех моделях микропроцессора Эльбрус, то есть начиная с Эльбрус-3М, появившегося в 2004 году и до микропроцессора Эльбрус-8С, - 2017-й год. Поддержка защищенного режима в ядре ОС, компиляторе и минималистичной библиотеке языка С была также реализована в 2004 году, что позволило на государственных испытаниях микропроцессора Эльбрус-3М продемонстрировать работу простых синтетических тестов в защищенном режиме. Однако синтетические тесты не

представляют какой-либо практической ценности в применении к реальным задачам. Основной трудностью применения защищенного режима в прикладных задачах являлся набор правил защищенного программирования на языке С. Как выяснилось реальные программы практически всегда написаны с использованием опасных конструкций языка С, таких как передача указателя через целое число, адресная арифметика, «законный» выход за границу массива («законный» - в пределах одной страницы, используемый при реализации алгоритмических оптимизации). Этот факт заведомо обуславливал очень большую потенциальную трудоемкость переноса любого универсального дистрибутива Linux для Эльбрус в защищенный режим. Поэтому практическое применение поддержки защищенного режима в Эльбрус и работы по внедрению этого режима в каких либо реальных задачах постоянно откладывалось. В конце 2015-го года в АО «МЦСТ» группой разработчиков ядра ОС, компилятора и библиотек была инициирована работа по созданию минималистичного дистрибутива ОС для встраиваемых систем, работающего в защищенном режиме. Предпосылкой для постановки работы был опыт взаимодействия с программистами «авиационного клуба», которые писали боевые программы СПО для бортового устройства построенного на «Эльбрус-2С+». Эти программы были написаны на языке С и не использовали большого количества зависимостей от каких-либо библиотек дистрибутива кроме libc, libpthread, libm и библиотеки высокопроизводительных примитивов libeml. Такой ограниченный набор библиотек был «обозримым» для модификации и переноса в защищенный режим. Выполнение такого переноса обеспечило бы внедрение защищенного режима в реальных прикладных программах в той сфере (бортовые устройства), где использование защищенного режима является крайне полезным, так как бортовые устройства — это объекты критической инфраструктуры, и ошибки в программах, работающих на таких устройствах могут быть причиной катастроф или серьезного ущерба.

В 2016 году началась проработка последовательности действий по реализации минималистичного дистрибутива в защищенном режиме. В результате предварительного исследования было принято решение портировать на Эльбрус библиотеку uclibc-ng, являющуюся ориентированным на встраиваемые системы минималистичным аналогом glibc. Портирование было предложено делать в два этапа: первый этап — портирование и тестирование uclibc-ng в незащищенном режиме, второй этап — перенос uclibc-ng в защищенный режим. Эксперимент с портированием основной части uclibc-ng на Эльбрус в незащищенном (обычном) режиме занял 6 месяцев, что можно было считать положительным результатом: при существенно меньшем объеме исходных текстов портированная uclibc-ng позволяла исполнять на встраиваемой системе достаточный набор типовых прикладных

пакетов Linux, обеспечивая возможность решения задач графики, видео, сетевого взаимодействия. Для дальнейшего развития необходимо было полностью отладить всю функциональность uclibc-ng в обычном режиме, выполнить перенос uclibc-ng в защищенный режим, определить класс реальных задач, где возможно эффективное применение полученного результата в защищенном режиме, сформировать маленький, но эффективный набор прикладных библиотек в защищенном режиме, позволяющий создавать программы, использующие графику, видео, сетевые интерфейсы, высокопроизводительные вычисления. Такой набор библиотек должен быть написан на языке C.

Для решения этих задач в конце 2016 года был открыт внутренний ОКР «Защищенный режим». ОКР включал в себя несколько работ, описанных в техническом задании. Помимо формализации требований и определения сроков выполнения работ, ОКР позволил финансировать работу подразделения АО «МЦСТ» в ОЭЗ «Иннополис». Это подразделение было создано для привлечения к работе новых талантливых специалистов, выпускников Университета Иннополис. Это позволило решить еще одну задачу проекта — нехватку специалистов по защищенному режиму. На момент написания отчета основные задачи проекта были выполнены.

Данный отчет описывает результаты выполнения внутреннего ОКР «Защищенный режим», технологические инструкции по сборке и использованию результата работы, исходные тексты (прилагаются к отчету на USB-флэш), предложения по дальнейшим шагам внедрения защищенного режима.

1.2 Постановка задачи

Основная задача проекта: внедрение технологии защищенного режима в типовых прикладных задачах встраиваемых систем путем разработки набора системных библиотек, функционирующих в защищенном режиме.

Дополнительная задача 1: подготовить группу специалистов по защищенному программированию, способную сопровождать процессы интеграции и внедрения разработанной технологии на промышленных предприятиях и предприятиях ВПК

Дополнительная задача 2: определить целевую группу потребителей технологии защищенного режима, стратегию работы с этой группой, определить дальнейшие

технические и организационные шаги по привлечению финансирования и внедрению разработанных технологий.

1.3 Потенциальная область применения защищенного режима

Исходя из особенностей технологии защищенного режима, реализованной в аппаратуре, компиляторе и ядре ОС, область применения разрабатываемого ПО определяется следующими требованиями:

1) все системные библиотеки и прикладные программы должны быть написаны на языке С;
2) количество и объем системных библиотек должны быть минимальными, но обеспечивающими всю необходимую функциональность;
3) применение защищенного режима в программах должно быть оправданным например повышенными требованиями к надежности ПО.

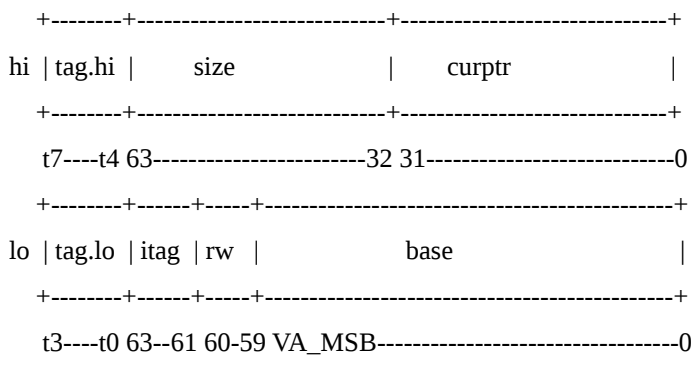
Таким требованиям соответствуют две области применения: ПО бортовые систем военных самолетов, ПО промышленных контроллеров. В обеих областях вероятность успешного внедрения высока: интерес к технологии проявили «Компания Сухой» (бортовые устройства) и ПАО «Татнефть» (промышленные контроллеры). В случае бортовых устройств имеется требование обеспечения работоспособности технологии в операционной системе ОС-4000 (ОС Linux не поддерживает стандарт ARINC-653, поэтому не может применяться на бортовом оборудовании). Поддержка защищенного режима в ОС-4000 является параллельной темой, разрабатываемой параллельно с выполнением данного проекта. В случае с промышленными контроллерами ограничений на применение Linux нет.

Таким образом наиболее вероятная область внедрения результата — промышленные контроллеры. В этой области могут быть использованы наработки ПАО «ИНЭУМ им. И.С. Брука» в части поддержки программирования промышленных контроллеров на языках стандарта IEC 61131-3 с помощью среды разработки «Beremiz». Языки IEC 61131-3 являются промышленным стандартом, поэтому широко используются. Интерфейс языков – элементы графических схем. Графические схемы трансформируются с помощью промежуточного компилятора matIEC (часть среды «Beremiz») в язык С. Отладка производится по методу таргет-хост. На таргете хранится минимальное количество библиотек. Алгоритм отладки и используемые в нем программные компоненты описаны в разделе 2.4.1.

1.4 Используемые термины

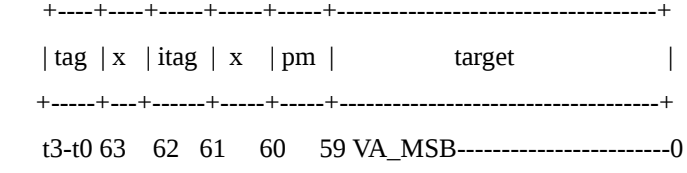
Определим термины, используемые для описания технических аспектов программных реализаций.

Дескриптор области памяти имеет размер 128 бит, включает в себя указатель на начало этой области памяти, размер этой области, указатель на текущую позицию и **внутренние тэги**, описывающие права доступа.



tag.hi, tag.lo - внешние теги
itag - внутренний тег
base - виртуальный адрес базы массива (в терминах байтов)
size - размер массива (в терминах байтов)
rw - права доступа (read/write)
11 - "rw" - чтение/запись (read/write)
01 - "r" - только чтение (read)
10 - "w" - только запись (write)
00 - "ad" - доступ запрещен (access disabled)
curptr - текущее значение индекса (current pointer) (в терминах байтов)

Дескриптор функции имеет размер 64 бита и включает в себя **внутренние теги**, флаги привилегий и адрес вызова.



tag - внешний тег
itag - внутренний тег
pm - флаг привилегированного режима;
target - адрес перехода

Внешние тэги — специальные биты, описывающие тип данных в защищённом режиме. В режиме защищённых вычислений все данные имеют **внешние тэги**. Каждые 32 бита

хранимых данных (одинарное слово) описываются двумя битами **внешних тэгов,** 64 бита (двойное слово) — четырьмя битами, 128 бит (дескриптор) — восемью битами. Каждый тип данных имеет свои значения тэгов (целочисленные данные, дескриптор функции, дескриптор памяти). При выполнении любых операций, будь то сравнение или обращение в память в защищённом режиме производится проверка внешних тэгов для подтверждения корректности операции. Например, в защищённом режиме невозможно использовать целочисленные данные для обращения в память или для вызова функции.

Статическая библиотека - это специальный архив, содержащий несколько объектных файлов, полученных компиляцией исходного кода библиотеки. В Linux статические библиотеки обычно имеют расширение **.a** (Archive). Статическая библиотека связывается с программой на этапе её компоновки. Такая компоновка называется **статическим связыванием**. При использовании **статического связывания** весь объектный код библиотеки, который необходим для работы целевой программы, помещается в исполняемый файл этой программы.

Динамическая библиотека (или совместно используемая библиотека) — файл, содержащий объектный код, предназначенный для совместного использования несколькими программами. Динамические библиотеки подключаются к программе непосредственно в момент её выполнения. Процесс подключения совместно используемых библиотек к программе на этапе исполнения называется **динамическим связыванием**. При использовании **динамического связывания** код совместно используемой библиотеки не включается в исполняемый файл программы, вместо этого туда помещается только ссылка на требуемую библиотеку.

Для поддержки динамического связывания и работы с разделяемыми библиотеками в uclibc-ng как и в других библиотеках языка С существует специальная программа - **динамический загрузчик.** В Linux он обычно находится по пути **/lib/ld.so. Динамический загрузчик** запускается в момент начала исполнения программы и отвечает за подключение к коду совместно используемых библиотек.

Для поддержки **системных вызовов** со стороны библиотеки uclibc-ng необходим механизм обращение прикладной программы к ядру операционной системы. За передачу параметров и обращение к системным вызовам в библиотеке uclibc-ng отвечают макросы INLINE_SYSCALL. Данные макросы архитектурно зависимы, поскольку механизм обращения к системным вызовам из пользовательской программы на разных платформах отличается. Параметры системных вызовов и возвращаемые ими значения передаются через регистровый файл.

В ядре Linux для организации взаимодействия с пользовательской программой через системные вызовы существует специальный **вход для системных вызовов. Вход для системных вызовов** - это функция в ядре, на которую передаётся управление при обращении к системному вызову из пользовательской программы. Данная функция отвечает за приём переданных на регистрах параметров и за вызов соответствующих функций-обработчиков для конкретного системного вызова.

libmcst — небольшая библиотека, работающая в режиме защищённых вычислений и входящая в состав оптимизирующего компилятора «Эльбрус».

1.4.1 Ранее реализованные программные компоненты.

На момент начала работы по внутреннему ОКР «Защищенный режим» в системе программирования для Эльбрус и ядре ОС присутствовали программные компоненты, необходимые для демонстрации работы технологии защищенного режима на синтетических тестах. Эти компоненты в процессе работы использовались как источник знаний о защищенном режиме. К таким компонентам относились:

- поддержка опции -mptr128 в компиляторе;
- поддержка в ядре ОС:
  - вход №10 для системных вызовов,
  - контроль работы над указателями в стеке;
- поддержка библиотеки языка С в системе программирования - библиотека libmcst:
  - без поддержки потоков,
  - без поддержки сети,
  - без поддержки dlopen (динамическая линковка была, но не было поддержки динамических библиотек),
  - без реализации malloc в библиотеке (каждый malloc приводит к системному вызову),
  - без поддержки сигналов и таймеров.

Реализация входа №10 для системных вызовов программ в защищенном режиме исполнения, выполненная в ядре ОС, функционировала на синтетических тестах, но была сложной и низкопроизводительной. Для использования на реальной задаче (описана в 2.4.1) имеющуюся реализацию стоило переделать. Поэтому в ядре ОС был реализован вход №8 (описан в разделе 2.3.3). Реализация входа №8 унифицирует интерфейс системных вызовов за счет использования общего оптимизированного фрагмента исходного текста для множества системных вызовов.

Библиотека libmcst, не имеющая поддержки многопоточности и работы с динамическими библиотеками, также была непригодна для использования в реальной задаче. Все отсутствующие функциональности библиотеки языка С для защищенного режима были реализованы в библиотеке uclibc-ng. Детали реализации описаны в разделах 2.3.2, 2.3.4, 2.3.5, 2.3.6.

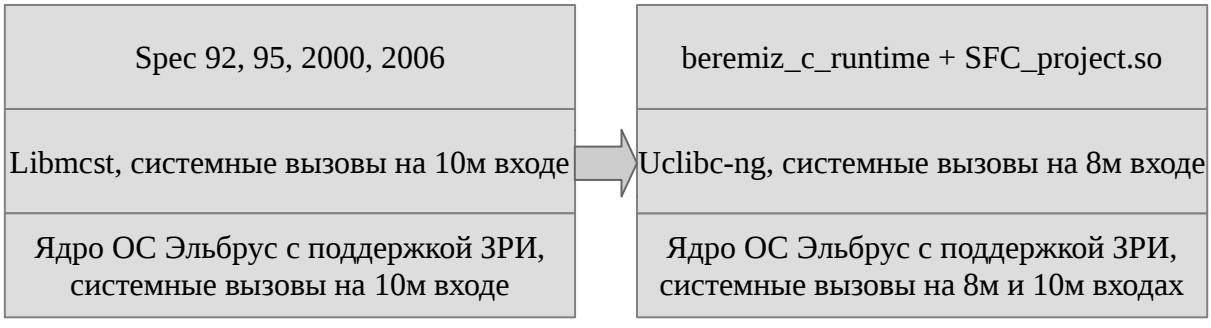
1.4.2 Созданные программные компоненты.

В процессе выполнения внутреннего ОКР «Защищенный режим» были реализованы следующие программные компоненты:

- Вход № 8 в ядро ОС для обслуживания системных вызовов в защищенном режиме;
- Библиотека uclibc-ng в защищенном режиме, включающая в себя поддержку следующих функциональностей:
  - поддержка динамического связывания (dlopen, dlsym) в защищенном режиме;
  - поддержка многопоточности (pthread_create, pthread_exit, pthread_join) в защищенном режиме;
  - поддержка функций синхронизации (pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_unlock) в защищенном режиме;
  - поддержка функций сигналов и таймеров;
  - поддержка простой реализации malloc (без реализации в библиотеке, каждый malloc приводит к системному вызову);
  - поддержка функциями работы с сетью (socket).
- Библиотека libmodbus была перенесена в защищенный режим, библиотеки libmodbusregshelper и libmodbussrv были реализованы на языке Си (переписаны с языка C++ на C) для функционирования задачи, описанной в 2.4.1.

На рисунке 1.4.2.1 приведены две схемы взаимодействия программных компонент: до и после выполнения внутреннего ОКР «Защищенный режим».

Рисунок 1.4.2.1: Стек ПО с libmcst (до проекта) и стек ПО с uclibc-ng (после проекта)



2. Детали реализации: проблемы и их решения

2.1 Обзор проблем: соответствие планов и реальности

Выполнение работ по внутреннему ОКР «Защищенный режим» было разделено на два этапа. По результатам первого этапа, завершившегося в мае 2017 года, предполагалось получить отлаженную версию библиотеки uclibc-ng с поддержкой потоков в незащищенном режиме. Но до мая 2017 года полностью отладить uclibc-ng в незащищенном режиме не удалось. Это сдвинуло срок начала работы над минималистичным дистрибутивом buildroot, основанного на uclibc-ng и срок начала переноса uclibc-ng в защищенный режим. Завершить работу по отладке в незащищенном режиме предстояло на втором этапе. Несмотря на это вся работа по внутреннему ОКР была выполнена в срок. На втором этапе было успешно компенсировано отставание по срокам: к исправлению ошибок в незащищенном режиме были привлечены новые сотрудники МЦСТ в Иннополисе, работу по реализации поддержки динамического связывания и по реализации потоков удалось распараллелить. Создание дистрибутива buildroot на uclibc-ng было также завершено в срок. Далее приведено достаточно детальное описание работ, выполненных на втором этапе.

2.2. Отладка uclibc-ng в незащищенном режиме

Библиотека uclibc-ng - небольшая библиотека C для разработки встроенных систем Linux. Она намного меньше, чем библиотека GNU C, но почти все приложения, поддерживаемые glibc, также отлично работают с uclibc-ng. Перенос приложений из glibc в uclibc-ng обычно включает в себя просто перекомпиляцию исходного текста. uclibc-ng поддерживает разделяемые библиотеки и потоки. В настоящее время он работает на стандартных системах Linux и MMU-less (также известных как uClinux) с поддержкой многочисленных платформ, в том числе теперь и e2k.

Основная цель портирования uclibc-ng - необходимость стандартной библиотеки libc для работы на промышленных контроллерах, построенных на микропроцессорах Эльбрус -1C+, в том числе создание минималистичного дистрибутива buildroot на ее основе.

Основной причиной, почему для портирования была выбрана именно uclibc-ng - меньший размер стандартной библиотеки libc, что даёт преимущество в виде простоты

портирования и меньшего размера исполняемых файлов и библиотек, скомпилированных с помощью uclibc-ng.

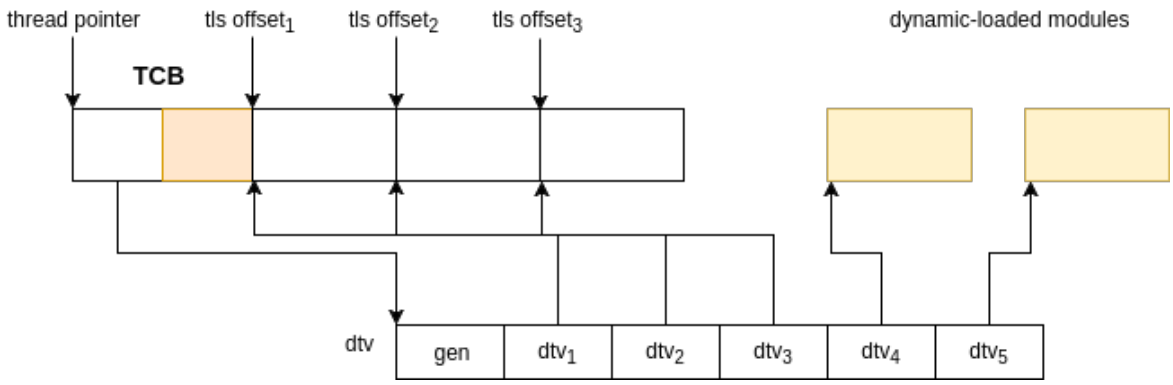
Главная сложность портирования uclibc-ng для Эльбрус связана с многопоточностью. Наиболее распространенными причинами падения тестов являлись:

- отсутствие libgcc_s.so, вспомогательной библиотеки для компилятора gcc, в нашем дистрибутиве
- неправильная аллокация памяти в главном потоке для TLS при использовании DTV и в других потоках при клонировании
- отсутствие реализации примитивов синхронизации
- лишний вызов функций обработки исключений, блокирующий работу cleanup

Исправление ошибок из этого списка исправляло сразу несколько тестов.

TLS (Thread Local Storage) - блок памяти с глобальными или статическими переменными, локальными для каждого потока программы. Рисунок 2.2.1 иллюстрирует внутреннюю структуру описания потока в памяти.

Рисунок 2.2.1: Структура описания потока в памяти



где структура описания потока содержит указатель на **DTV (Dynamic Thread Vector)** - массив, который содержит в себе указатели на блоки локальных переменных потока. В изначальной версии uclibc-ng для Эльбрус при инициализации главного потока и его локальных переменных, первый блок (tls $offset_1$) локального хранилища аллоцировался в блоке описания потока, из-за чего терялся указатель на DTV.

Также сложность при работе с потоками создавали трудно-отлаживаемые состояния гонки, вызывающие дедлок или рандомные падения тестов. Большинство этих проблем решились реализацией атомарных целочисленных типов и спинлока.

После внесения исправлений в часть библиотеки, отвечающей за работу потоков, ещё 76 тестов стали проходить без ошибок. Часть uclibc-ng, отвечающая за математику (libm), также не проходила часть тестов. В большинстве случаев, причиной было отсутствие правильной реализации в изначальной версии библиотеки. То есть большая часть тестов libm не проходила на всех платформах (не только на e2k). Данные тесты проверяли соответствие библиотеки стандартам. Были выявлены и исправлены следующие проблемы:

- Использование глобальной переменной signgam при вызове функций lgamma, lgammaf, lgammal. Ошибка была вызвана тем, что некоторые спецификации стандартной библиотеки языка C не предусматривают объявление и использование данной глобальной переменной. При компиляции программы с указанием определённой версии стандарта библиотека должна использовать модификации функций, соответствующих данному стандарту.
- Функции nan, nanf, nanl, strtof_nan, strtod_nan возвращали значение nan с неверной мантиссой. По стандарту мантисса должна быть такой же, как и у числа, переданного данным функциям в качестве аргумента.
- Была добавлена возможность устанавливать флаг исключения FE_DENORMAL с помощью функции feraiseexcept.
- Был добавлен изначально отсутствующий функционал обработки исключений математических функций с помощью функции matherr. Данная функция может быть определена пользователем для перехвата исключений (аналог try/catch в C++).

Также была обнаружена проблема с изначальным состоянием флагов исключений FE_INEXACT, FE_DIVBYZERO, FE_UNDERFLOW, FE_OVERFLOW, FE_DENORMAL, FE_INVALID (объявлены в fenv.h). При запуске программы на e2k данные флаги уже установлены, хотя должны устанавливаться в процессе выполнения программы при возникновении исключений. Данная ошибка не исправлена, т.к. вероятнее всего является аппаратной. В общей сумме было починено 7 тестов для libm. Работа над этой частью библиотеки uclibc-ng для незащищённого режима была полностью завершена.

Кроме этого были исправлены ошибки на тестах с regex, обусловленные использованием устаревшей функции regex_old и futimens, где переполнялись поля структуры timespec.

Таким образом, всего было починено 83 теста в uclibc-ng.

2.3. Сборка и отладка uclibc-ng в защищенном режиме

2.3.1. Сборка uclibc-ng в минимальной конфигурации в защищённом режиме с использованием существующего входа в ядро.

Первым шагом переноса uclibc-ng в защищенный режим было создание минимальной конфигурации uclibc-ng. Такая конфигурация предусматривает поддержку только статического связывания программ без многопоточности. Для запуска минимальной конфигурации было реализовано:

- Функция **_start**, - точка входа в библиотеку. Данная функция отвечает за приём параметров, передаваемых программе на стеке (argc, argv, envp) и дальнейшую передачу их вспомогательной функции **__uClibc_main()**, которая затем вызывает пользовательскую функцию **main**(). Функция **_start** архитектурно зависима и написана на языке C с ассемблерными вставками. Для режима защищённых вычислений данная функция уже реализована в библиотеке **libmcst** в составе компилятора, поэтому она была перенесена в uclibc-ng с внесением минимально необходимых изменений;

- Механизм обращения к системным вызовам из библиотеки. Основная сложность реализации этого механизма в защищённом режиме связана с передачей параметров. Параметры системных вызовов и возвращаемые значения передаются через регистровый файл. Параметры системных вызовов делятся на два основных типа: целочисленные данные и указатели. Так как реализация нового механизма обращения к системным вызовам для защищённого режима требует существенное время, то было принято решение сначала произвести сборку и отладку библиотеки uclibc-ng с использованием уже существующего **входа в ядро** с номером 10. Для этого в библиотеке uclibc-ng использовались макросы INLINE_SYSCALL, перенесённые из **libmcst**;

- Были исправлены операции, которые работают нормально в обычном режиме, но в защищённом режиме недопустимы. Примерами таких операций, исправленных в ходе переноса uclibc-ng в защищённый режим, являются:

- Исправление некоторых оптимизированных алгоритмов работы с памятью, которые ведут себя некорректно с точки зрения защищённого режима. Например в обычном режиме чтение массива производится не поэлементно, а блоками. Если размер массива не кратен размеру блока, то при чтении последнего происходит выход за границу массива. Так как размер блока подбирается таким образом, чтобы не происходило выхода за границу страницы памяти, то в обычном режиме такая оптимизация работает корректно. В защищённом же режиме любой выход за границу массива является нарушением, поэтому приходится добавлять в такие алгоритмы дополнительные проверки на выход за пределы массива.

- Замена сравнения адреса функции с нулём на специальный макрос сравнения для защищённого режима. В режиме защищённых вычислений загрузка программ в виртуальное адресное пространство начинается не с нулевого адреса а с адреса с фиксированным значением 0x5000000. Поэтому «нулевым» значением для адреса функции будет не 0, как в обычном значении, а 0x5000000. Следовательно при проверке адреса функции на нулевое значение надо не с 0, а со значением 0x5000000.
- Замена целочисленных типов данных, используемых для хранения указателей на void*. В защищённом режиме преобразование указателя в целочисленное значение недопустимо.
- Выравнивание всех указателей на 16 байт в защищённом режиме. Обращение по не выровненным указателям в защищённым режиме запрещено. Также следует отметить, что все дескрипторы в защищённом режиме должны быть размещены только на регистрах с чётным номером.
- Замена некоторых ассемблерных инструкций, недопустимых в защищённом режиме на аналогичные для защищённого режима (ldw на ldapw , stw на stapw).

2.3.2. Реализация в uclibc-ng с поддержкой динамического связывания в защищенном режиме

Для обеспечения поддержки **динамического связывания** в защищённом режиме был реализован **динамический загрузчик**. Детали реализации динамического загрузчика описаны в п. 2.3.5. Для работы динамического загрузчика была необходима реализация обращения к системным вызовам. Так как реализация обращения к системным вызовам — это длительный процесс, то было решено производить отладку динамического загрузчика с использованием уже существующего **входа в ядро** для системных вызовов до тех пор пока не будет готов новый. После реализации нового **входа в ядро** с номером 8 для защищённого режима в динамическом загрузчике использовался только он.

2.3.3 Реализация входа № 8 для защищенного режима

Прикладная программа взаимодействует с устройствами через интерфейс системных вызовов ядра ОС. Существуют системные вызовы с указателями в качестве параметров. Во-первых, для использования таких системных вызовов защищенными программами необходимо проводить преобразование дескриптора в обычный указатель. С точки зрения защиты преобразование и анализ дескрипторов необходимо делать в ядре. Во-вторых,

системные вызовы ядра linux принимают на вход до 6 параметров. Стандартный размер окна для передачи параметров составляет 8 двойных регистров (по 64 бита). В защищенном режиме дескриптор занимает два двойных регистра — один квадрорегистр (128 бит). Поэтому не для всех системных вызовов получается разместить параметры внутри регистрового окна стандартного размера. В-третьих, некоторые системные вызовы принимают дескрипторы внутри структур, эти дескрипторы также необходимо преобразовывать в указатели. В-четвертых, некоторые системные вызовы возвращают указатели. Для передачи пользователю указатели необходимо преобразовать в дескрипторы. Перечисленная работа ложится на реализацию входа в ядро.

При реализации входов №10 разработчики решили проблему с помощью написания уникальной обёртки для каждого системного вызова, принимающего на вход хотя бы один указатель. Обёртка системного вызова анализирует дескрипторы-аргументы, преобразует их в указатели и вызывает системные функции ядра linux. В случае, если аргументы не умещаются в стандартном окне передачи параметров, часть аргументов передаётся через память, а в обёртке считываются параметры из памяти. Стоит отметить, что из-за необходимости плотно размещать параметры, номер регистра на котором окажется параметр зависит от количества дескрипторов среди предыдущих параметров системного вызова. Такой подход создает необходимость в библиотеке записывать параметры системных вызовов на регистры и в память разными способами, в зависимости от номера системного вызова. Это вытекает в необходимость руками разработчиков в библиотеке изменять каждый системный вызов (в ucling-ng ~ 500 мест, в libmcst ~ 160 мест). В случае, если системный вызов принимает дескрипторы внутри структур без потоков

- без динамической линковки

- статически, дескрипторы преобразуются обёрткой системного вызова. Также, если системный вызов возвращает указатель, обёртка производит преобразование указателя в дескриптор. Уникальных оберток на 10-м входе насчитывается порядка 130.

Решение проблемы во входе №8 строится на расширении регистрового окна передачи параметров: вместо 8 двойных регистров окно состоит из 14 двойных регистров (спасибо Александру Фёдорову sanekf@mcst.ru за идею и консультации). Увеличенный размер окна передачи параметров позволяет размещать каждый параметр на отдельном квадрорегистре, и проводить преобразование дескрипторов в указатели с помощью одного, общего для всех системных вызовов, участка кода. Для этого в ядре заведена таблица с масками для системных вызовов. Маска системного вызова показывает на каких местах в окне передачи параметров лежат дескрипторы. При таком подходе все аргументы всех системных вызовов

умещаются в окне передачи параметров, не требуя передачи через память. Уникальные обёртки системных вызовов необходимы только системным вызовам, принимающим дескрипторы внутри структур, а также системным вызовам, возвращающим указатели. Уникальных оберток на 8-м входе сейчас насчитывается порядка 10.

Всего системных вызовов около 400. Реализация простых системных вызовов требует лишь указания правильной маски в таблице. Реализация сложных системных вызовов, в которых ядру через память передаются дескрипторы или которые возвращают указатели, требует написания уникальной обёртки, либо использования обертки из 10-го входа, если такая имеется.

Существуют системные вызовы, реализация которых в защищенном режиме требует очень больших усилий. Например, пара системных вызовов timer_create и rt_sigtimedwait. В режимах 32/64 системный вызов timer_create передает ядру указатель на пользовательскую память. Ядро сохраняет этот указатель в описывающую процесс структуру ядра. Во время системного вызова rt_sigtimedwait ядро возвращает указатель пользователю. Для защищенных программ требуется сохранить в структуре ядра дескриптор, а также его внешние теги. Но в структуре ядра, описывающей процесс нет места на вторую половину дескриптора и на его внешние теги. Для работы этих системных вызовов необходимо переделывать описывающую процесс структуру ядра, и переделывать функции, которые работают с этой структурой. В составе стенда системный вызов rt_sigtimedwait работает с уязвимостью в защите: всегда возвращает пользовательской программе дескриптор ожидаемого размера.

Первыми во входе №8 были реализованы системные вызовы ioctl, write и exit. Этих системных вызовов хватает для исполнения программы hello_world. Далее реализован более сложный системный вызов mmap, необходимый для исполнения функции malloc. Дальше работа распараллелилась: реализация pthread и реализация динамического лоадера на входах № 8.

2.3.4 Функция malloc в защищенном режиме: три варианта реализации

В бибилотеке uclibc-ng реализованно 3 варианта функции malloc: malloc-simple, malloc и malloc-standard.

Реализация malloc-simple является самой простой, медленной и легковесной. Malloc-simple использует только системный вызов mmap для выделения и munmap для освобождения памяти. Malloc-simple не использует системный вызов brk. В malloc-simple

каждое выделение памяти (malloc, calloc, realloc) приводит к системному вызову mmap, а каждое освобождение памяти (realloc, free) приводит к системному вызову munmap.

В реализации malloc для выделения памяти используется системный вызов mmap, либо системный вызов brk в зависимости от определения макроса MALLOC_USE_SBRK. В malloc реализовано переиспользование освобожденной ранее памяти, что позволяет снизить количество системных вызовов и время исполнения функций выделения памяти. Free вызывает munmap/brk если есть свободная непрерывная область памяти больше восьми страниц.

Malloc-standard — адаптированная реализация dlmalloc для использования в составе uclibc-ng. В malloc-standard также реализовано переиспользование освобожденной ранее памяти. Для выделения небольших участков памяти используется системный вызов brk, а для больших используется системный вызов mmap. По умолчанию небольшими участками памяти считаются участки размером менее 64 страниц, но это значение можно изменить с помощью функции mallopt. Malloc-standsrd используется в качестве реализации malloc по умолчанию в библиотеке uclibc-ng.

С точки зрения защищенного режима системный вызов brk невозможно использовать для выделения памяти, поскольку brk возвращает указатель на конец выделенной памяти. Кроме того, в освобожденной памяти могут храниться дескрипторы, поэтому перед переиспользованием необходимо делать чистку. Также при переиспользовании памяти в malloc и malloc-standard не производится перерасчитывание поля size возвращаемого дескриптора. Ввиду приведенных причин наиболее подходящим к использованию в защищенном режиме вариантом реализация malloc-simple, а в дальнейшем написание собственной реализации malloc-protected со свойствами из внутренней ошибки Bug 93263.

2.3.5 Реализация динамического лоадера в защищенном режиме

Для поддержки динамического связывания и работы с разделяемыми библиотеками в uclibc-ng как и в других библиотеках языка C существует специальная программа - динамический загрузчик. Для динамического загрузчика характерно особенно интенсивное использование адресной арифметики. Основной же проблемой для адаптации динамического загрузчика для режима защищённых вычислений является применение целочисленных типов данных для хранения адресов в структурах, используемых самим динамическим загрузчиком для представления информации о загруженных модулях. Из-за данной особенности требуются значительные модификации исходного текста динамического загрузчика при его переносе в защищённый режим. В силу этого было принято решение перенести в

защищённый режим усечённую реализацию динамического загрузчика уже существующую в библиотеке libmcst, входящей в состав компилятора. Впоследствии данная реализация будет дополняться необходимой функциональностью.

Обычно операционная система загружает в адресное пространство процесса непосредственно сам исполняемый файл и динамический компоновщик, путь к которому указан в специальном поле в любой динамически связанной программе. После этого управление передаётся динамическому загрузчику. Задачей динамического загрузчика является загрузка в адресное пространство процесса необходимых разделяемых библиотек и разрешение зависимостей между ними (этот процесс называется релокацией). Динамический загрузчик хранит информацию об используемых библиотеках в виде специальной структуры **- цепочки загруженных модулей**. Стоит отметить что сам **динамический компоновщик** тоже является разделяемой библиотекой и включается в **цепочку загруженных модулей**, чтобы обеспечить возможность дальнейшего использования его функций другими библиотеками. После того как процесс подгрузки библиотек и разрешения зависимостей завершён, компоновщик передаёт управление на точку входа запускаемой программы.

Работа **динамического компоновщика**, а также функций ядра, отвечающих за загрузку исполняемого файла в память в режиме защищённых вычислений имеют ряд своих особенностей. Самой главной особенностью является то, что в защищённом режиме ядро загружает в память только **динамический компоновщик** необходимый для запускаемого исполняемого файла и передаёт ему управление.

После получения управления **динамический компоновщик** загружает в адресное пространство процесса саму программу, все необходимые для неё разделяемые библиотеки, а также разрешает все зависимости. При разрешении зависимостей в каждом модуле, в том числе и в самой программе, необходимо модифицировать специальную секцию исполняемого файла — глобальную таблицу смещений (got), содержащую в себе адреса функций и глобальных переменных. В режиме защищённых вычислений доступ к области памяти осуществляется через дескрипторы размера 128 бит, которые формируются в момент выделения данной области памяти. Поэтому для хранения информации о загруженных модулях, содержащей в себе указатели, используется специальная дополнительная структура mdd.

Структура mdd получается специальным системным вызовом uselib для каждого модуля и содержит в себе указатели на функцию инициализации глобальной таблицы смещений, функции init и fini, на точку входа в программу и на глобальную таблицу смещений:

typedef struct

```
{
    void (* mdd_init_got) ();
    void (* mdd_init) ();
    void (* mdd_fini) ();
    void (* mdd_start) (int, ...);
    gen_ptr *mdd_got;
} mdd_t;
typedef union
{
  void *ptr;
  void (*fptr) ();
  char dummy[16];
} gen_ptr;
```

Использование специального объединения gen_ptr для хранения вхождений глобальной таблицы смещений связано с тем, что для указателей на переменные и на функции в режиме защищённых вычислений используются разные типы дескрипторов [1].

При портировании библиотеки uclibc-ng в защищенный режим исполнения программ была использована существующая реализация динамического связывания из библиотеки **libmcst** с внесением необходимых изменений. Были внесены изменения в структуры данных, используемые **динамическим загрузчиком**, а также добавлены дополнительные функции и структуры для режима защищённых вычислений.

Однако у существующей в библиотеке **libmcst** реализации динамического связывания есть существенный недостаток — в ней отсутствует поддержка стандартных функций для ручной работы с динамическими библиотеками, таких как dlopen, dlsym, dlclose. Поддержка данных функций необходима для выполнения целевых задач, в которых будет использоваться библиотека uclibc-ng. Поэтому были произведены соответствующие доработки как со стороны ядра Linux, так и со стороны самого **динамического загрузчика** в библиотеке uclibc-ng.

Для того чтобы обеспечить работу таких функций как dlopen, dlclose, dlsym, необходимо включить **динамический загрузчик** в **цепочку модулей** сделать его функции доступными другим библиотекам. В существующей же реализации **динамический компоновщик** загружается в память как обычная программа и не может быть включён в цепочку загруженных модулей как это делается в обычном режиме. Для того, чтобы включить **динамический компоновщик** в **цепочку загруженных модулей** из ядра на стеке

пользовательской программы передаются **дескрипторы** на области его исполняемого кода и данных. Использую переданные дескрипторы загрузчик сам включает себя в **цепочку загруженных библиотек** и производит динамическое связывание.

2.3.6 Многопоточность: реализация библиотеки pthread в защищенном режиме

В библиотеку uclibc-ng входит две реализации библиотеки pthread: inuxthreads и nptl. Linuxthread считается устаревшей реализацией, и при портировании uclibc-ng на архитектуру e2k была портирована только реализация nptl. Поэтому и в контексте защищенного режима рассматривалась реализация nptl. Реализация nptl состоит из ~ 120 функций. Тестирование и отладка всех функций потребует времени, поэтому работа ограничилась отладкой функций для реальной задачи из пункта 2.4.1. Задача из пункта 2.4.1 требует работы следующих функций: **pthread_create**, **pthread_join**, **pthread_mutex_init**, **pthread_mutex_lock**, **pthread_mutex_trylock**, **pthread_mutex_unlock**, **pthread_mutex_destroy**.

Работа функции pthread_create строится на использовании функции clone, которая основана на системном вызове sys_clone. В библиотеке libmcst существует готовая реализация функции clone на языке assembler для защищенного режима. Также во входе №10 в ядро есть готовая реализация обёртки системного вызова sys_clone. Во входе №10 часть параметров clone передаются через память, во входе №8 все параметры передаются на регистрах. Поэтому реализация системного вызова clone и функции clone состояла в изменении алгоритма передачи и приёма параметров. В ходе отладки функции pthread_create была обнаружена ошибка, приводящая к kernel_panic. Ошибка заключалась в неправильном обращении к структуре подвала при замене дескриптора стека на дескриптор массива.

Работа функций синхронизации pthread_mutex_* и pthread_join опираются на системный вызов futex. Для работы системного вызова futex достаточно было указать правильную маску в таблице системных вызовов.

Кроме того, во время создания и завершения нитей исполнения требуются функции setjmp и longjmp. Их реализация была взята из библиотеки libmcst и входа №10. В процессе отладки системного вызова longjmp была обнаружена и исправлена ошибка в на выходе из системных вызовов №8 и №10 в ядре 4.9.

2.3.7 Тестирование работоспособности библиотеки uclibc-ng в режиме защищённых вычислений.

Тестирование работоспособности библиотеки uclibc-ng в защищённом режиме использовался набор из 100 встроенных тестов. Тесты поставляются в приложении к отчету, вместе со всеми исходными текстами (см. раздел 4, таблица 4.1).

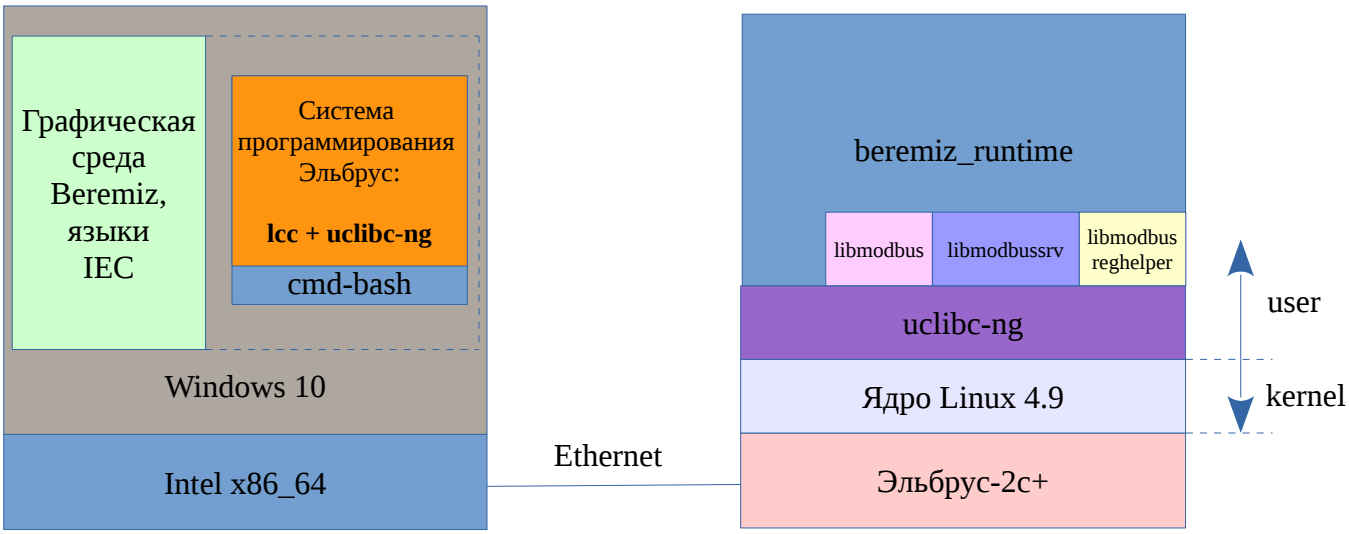
2.4. Сборка и отладка реальной конфигурации в защищенном режиме

“Деритесь настоящими мечами” К. Мацусита

2.4.1. Реальная задача: ПО для промышленного контроллера

Реальная задача, моделирующая работу промышленного контроллера была предоставлена разработчиками ПАО «ИНЭУМ им. И.С. Брука», занимающихся развитием графической среды программирования Beremiz и внедрением ее на объектах атомной промышленности. Задача основана на использовании прикладной программы beremiz_runtime и схемы программирования промышленных контроллеров по методу таргет-хост. Схема стенда, использованного для отладки реальной программы приведена на рисунке 2.4.1.1.

Рисунок 2.4.1.1: система таргет-хост



beremiz_runtime – специальное программное обеспечение для управления промышленными контролерами. В нашем случае beremiz_runtime запускается на контроллере с процессором «Эльбрус» и взаимодействует по сети с управляющей машиной с Windows 10, на которой работает среда для автоматизации Beremiz.

beremiz_runtime — многопоточное приложение. При запуске **beremiz_runtime** создаются следующие основные потоки, у каждого из которых есть своё предназначение:

* status_thread: служит для периодической отправки текущего статуса целевого устройства на управляющую машину.

* control_thread: служит для приёма и обработки управляющих команд от среды Beremiz.

(например, это запуск или остановка отладки или запрос debug-трейса — текущего состояния переменных в запущенном отладочном коде)

* set_trace_thread/get_trace_thread: обеспечивают установку/получения состояния переменных в отладочном коде.

* retain_thread: служит для работы с сохраняемыми (retain) переменными, которые записываются в энергонезависимую память (на носитель информации) целевого устройства. Данный поток, например, отвечает за приём и сохранение библиотеки с отладочным кодом от среды Beremiz.

Суть работы **beremiz_runtime** состоит в том, чтобы запустить на целевом устройстве отлаживаемый код из динамической библиотеки, полученной из программы, созданной в Beremiz, а также обеспечить интерфейс для взаимодействия со средой Beremiz и управления процессом отладки. При этом логика работы программы написанной в Beremiz будет содержаться в динамической библиотеке, передаваемой на целевое устройство.

Рисунок 2.4.1.2: процесс отладки в режиме таргет-хост. Шаг 1: Хост: компиляция программы на языке SFC в графической среде «Beremiz»

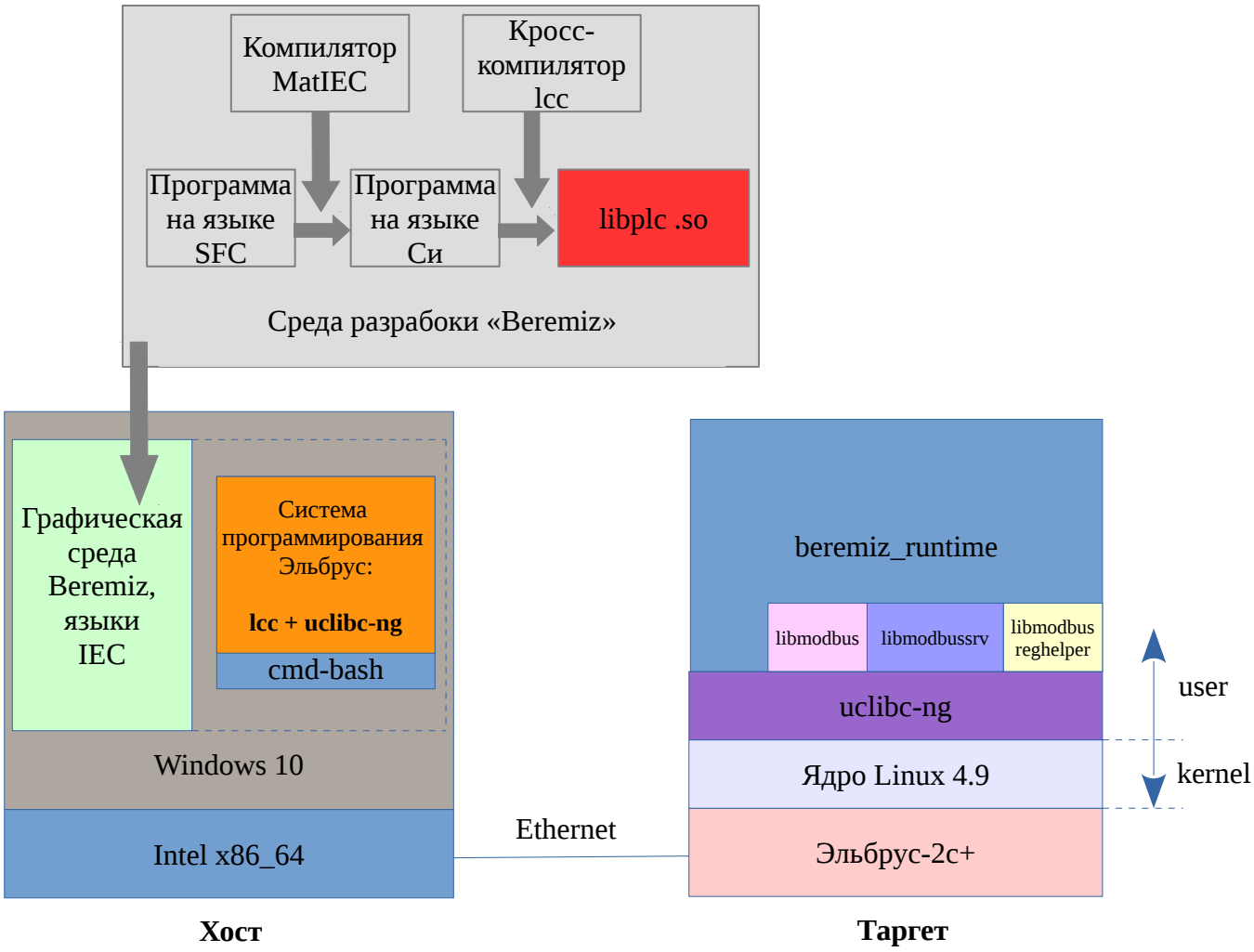
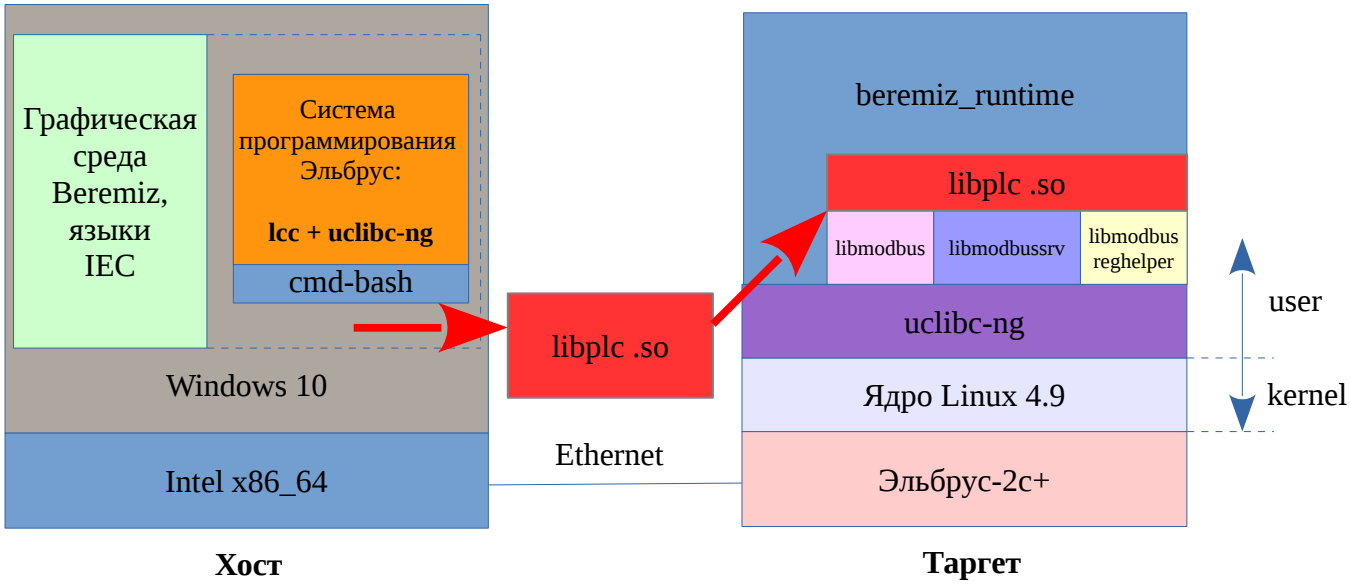


Рисунок 2.4.1.3: процесс отладки в режиме таргет-хост. Шаг 2: Хост: установка соединения с target и передача libplc.so на таргет;Таргет: получение libplc.so, открытие его с помощью dlopen и запуск на исполнение



В ходе работы по переносу программы **beremiz_runtime** в защищённый режим в uclibc-ng были внесены существенные доработки.

- Для работы **beremiz_runtime** необходимы системные вызовы из сетевого стека (socket, bind, send и др.). Для обеспечения возможности использования этих системных вызовов была переработана передача параметров для них из библиотеки uclibc-ng. Также были внесены соответствующие изменения в новый **вход для системных вызовов в ядро** Linux для приёма и обработки параметров этих системных вызовов.

- Для приёма и обработки библиотеки с отладочным кодом с управляющей машины необходимы функции для работы с файлами (fopen, fclose, и т.д). В некоторых из этих функций используется преобразование указателей в целочисленные значения, что в защищённом режиме недопустимо. Были внесены правки для хранения указателей в переменных типа void*.

- Различные потоки исполнения, созданные **beremiz_runtime** используют системные таймеры для отсчёта промежутков времени при отправке и приёме сигналов с управляющей машины. Поэтому была реализована поддержка системных вызовов для работы с таймерами (timer_create, timer_settime) как со стороны библиотеки uclibc-ng так и со стороны нового входа в ядро Linux для защищённого режима.

На каждом шаге реализации алгоритма отладки возникала минимум одна ошибка исполнения. Ниже приведено описание возникших проблем и их решений.

2.4.2. Потеря тегов дескриптора при записи дескриптора в структуру

Описание ошибки: теги дескриптора пропадали при записи дескриптора в структуру, если память под структуру выделена через malloc. Malloc-simple возвращал не выровненный на 16 байт адрес, а запись дескриптора по не выровненному адресу приводила к потере внешних тегов дескриптора. Решение проблемы заключается в выравнивании возвращаемого malloc-simple адреса на 16 байт.

2.4.3. Запись за границу описателя потока

Описание ошибки: запись за границу структуры pthread_attr. Размер структуры pthread_attr определяется макросом __SIZEOF_PTHREAD_ATTR_T, а структура pthread_attr определяется как массив байт длиной __SIZEOF_PTHREAD_ATTR_T. Структура pthread_attr содержит указатели, поэтому её размер в защищенном режиме больше. Решение проблемы заключается в изменении значения макроса __SIZEOF_PTHREAD_ATTR_T для защищенного режима.

2.4.4. Неинициализированные данные в longjmp

Описание ошибки: функция longjmp использовала неинициализированне значение поля __mask_was_saved. Функция setjmp не во всех случаях инициализировала поле __mask_was_saved, но функция longjmp использовала это значение во всех случаях. Исправление ошибки заключается в доработке кода setjmp так, чтобы поле __mask_was_saved инициализировалось во всех случаях.

2.4.5. Ошибки в системных вызовах access, futex

Описание ошибки: системные вызовы access, futex не работали. Исправление ошибки заключается в указании правильной маски в таблицу масок системных вызовов.

2.4.6. Ошибки в системном вызове rt_sigtimedwait

Описание ошибки: системный вызов rt_sigtimedwait не работал. Системного вызова rt_sigtimedwait должен возвращать в пользовательское пространство указатель. Этот указатель сохраняется ядром во время системного вызова timer_create в структуре ядра, описывающей текущий процесс. Для защищенного режима в структуре ядра необходимо

сохранять дескриптор, который в два раза больше 64-битного указателя по размеру, и не помещается в описывающую текущий процесс структуру ядра. Решение проблемы в составе стенда: обработчик rt_sigtimedwait всегда возвращает дескриптор одного и того же размера.

2.4.7 Библиотека libmodbus в защищенном режиме, ошибки в системном вызове setsockopt

Modbus - протокол, основанный на архитектуре ведущий-ведомый, широко применяющийся для организации связи между электронными устройствами. Используется для передачи данных через последовательные линии связи и TCP/IP.

В протоколе MODBUS существует 4 типа данных:

- **Discrete Inputs** — однобитовый тип, доступен только для чтения.
- **Coils** — однобитовый тип, доступен для чтения и записи.
- **Input Registers** — 16-битовый знаковый или беззнаковый тип, доступен только для чтения.
- **Holding Registers** — 16-битовый знаковый или беззнаковый тип, доступен для чтения и записи.

Основа структуры запросов и ответов - элеменатрный пакет протокола PDU:

код функции	**данные**
1 байт	N ≤ 252 байта

Пакет PDU помещается в пакет ADU, который содержит дополнительные поля, для передачи по физическим линиям связи:

ADU TCP

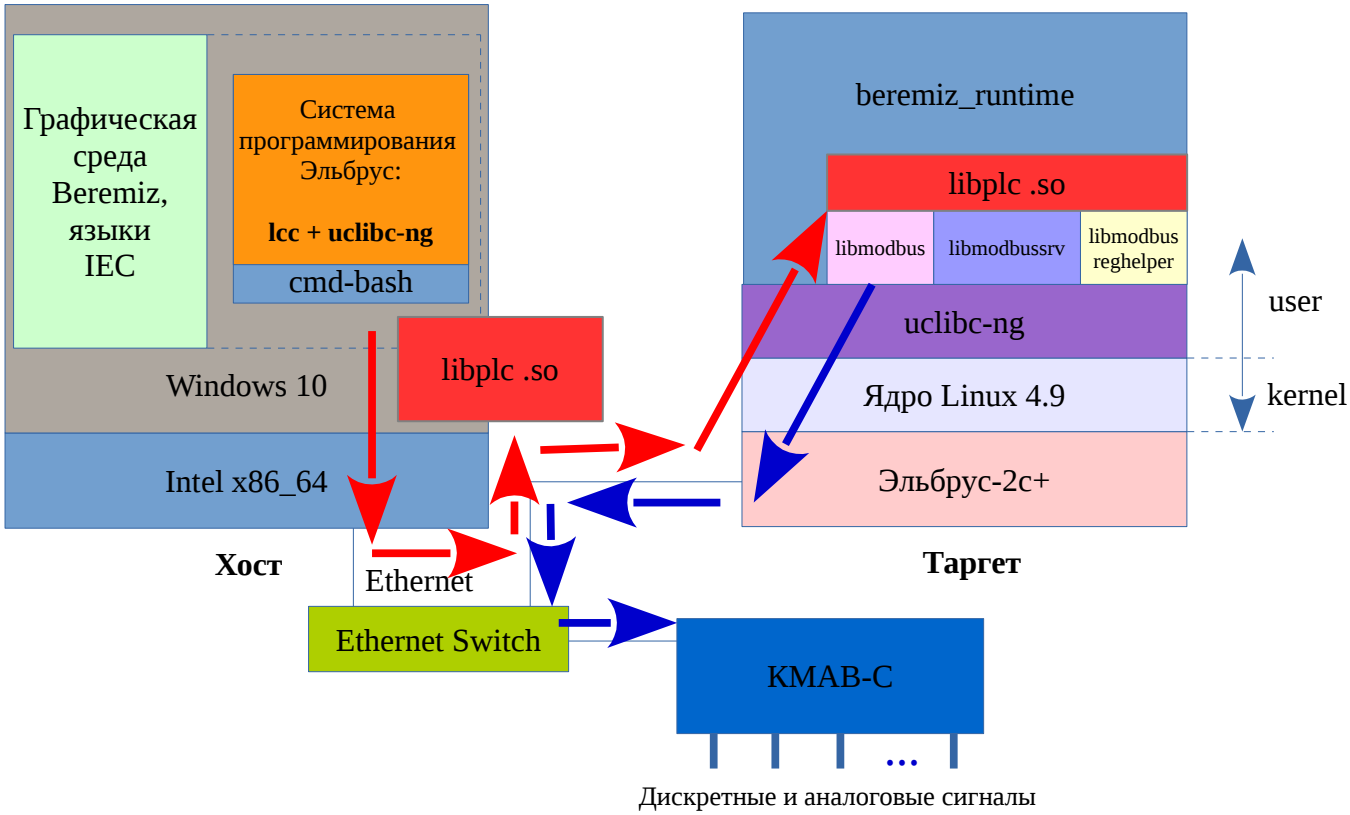
ID транзакции	ID протокола	длина пакета	адрес ведомого устройства	код функции	данные

В сети присутствует одно ведущее устройство (master) и несколько ведомых (slave). Ведущее устройство передает запросы (транзакции) и может адресовать их любому ведомому как по отдельности, так и с помощью широковещательного сообщения для всех устройств. Ведомое устройство, опознав свой адрес, отвечает на запрос.

На рисунке 2.4.7.1 контроллер КМАВ-С является сервером в libmodbus, то есть ведущим устройством, а “Эльбрус-2с+” (таргет) - клиентом. Контроллер имеет каналы

дискретного и аналогового ввода для связи с другими контроллерами и дискретного вывода для связи с клиентом.

Рисунок 2.4.7.1: Схема стенда отладки с контроллером KMAB-C



При запуске libmodbus возникла ошибка соединения клиента с сервером из-за системного вызова setsockopt, при котором происходила неправильная передача аргументов из uclibc-ng в ядро. Аналогично была исправлена ошибка с вызовом getsockopt, после чего клиенту удалось установить соединение с сервером.

2.5 Отладка программ в защищённом режиме

Отладка **статически связанных** программ в защищённом режиме производится при помощи gdb также как и в обычном режиме. В случае **статического связывания** вся отладочная информация, также как и код, включается в исполняемый файл, поэтому она становится полностью доступна для gdb.

Основной проблемой отладки **динамически связанных** программ при помощи gdb является то, что на данный момент отладчик не способен распознавать структуру данных (**цепочка загруженных модулей**), в которой хранится информация из динамически загружаемых модулей. На данный момент отладка при помощи gdb возможна только для

динамического загрузчика. Для того, чтобы обеспечить возможность полноценной отладки динамически связанной программы в защищённом режиме при помощи gdb в будущем необходима доработка самого отладчика.

3. Демонстрация работы технологии

3.1. Состав и схема стенда

Схема демонстрационного стенда приведена на рисунке 2.4.1.1. Стенд состоит из соединенных по ethernet 2-х физических машин:

- рабочая станция x86-64 с установленной ОС Windows 10;
- целевое устройство «Монокуб».

На целевом устройстве «Монокуб» работает ядро ОС «Эльбрус» и дистрибутив на основе buildroot и библиотеки uclibc-ng собранные в 64-х разрядном режиме. Дистрибутив используется для запуска среды исполнения beremiz_runtime. Среда исполнения beremiz_runtime нужна для того, чтобы обеспечить функционал удаленного перезапуска обновленной программы, а также предоставить интерфейс управления и отладки целевой программы в среду разработки «Beremiz».

На Windows 10 работает среда разработки «Beremiz». Среда разработки «Beremiz» предназначена для создания и отладки прикладных программ на языках стандарта IEC 61131-3 для целевых устройств. В среде разработки «Beremiz» реализована двухэтапная компиляция. Сначала отрабатывает MatIEC компилятор, который преобразует написанную на языке SFC стандарта 61131-3 программу в код на языке Си. Далее используется кросс-компилятор lcc для сборки программы в виде динамической библиотеки .so под целевую платформу. После сборки программа в виде динамической библиотеки передается на целевое устройство «Монокуб» в среду исполнения beremiz_runtime.

В составе стенда используется операционная система Windows 10, поскольку в Windows 10 реализована технология «Windows subsystem for Linux». Технология «Windows subsystem for Linux» позволяет запускать 64х разрядные приложения Linux на Windows 10. Технология «Windows subsystem for Linux» используется для запуска кросс-компилятора lcc. Вместо Windows 10 имеется возможность использовать 32-х и 64-х разрядные дистрибутивы Linux.

3.2 Технологическая инструкция по настройке стенда

Стек программного обеспечения, предназначенного для промышленной автоматизации, включает в себя:

- Среду Beremiz, работающую на управляющей машине с Windows 10.

- Программу beremiz_runtime, работающую на целевом устройстве на Эльбрусе

- Библиотеку языка с uclibc-ng, работающую на целевом устройстве на Эльбрусе

Программа beremiz_runtime и библиотека uclibc-ng работают в режиме защищённых вычислений. Отлаживаемая программа открывается в среде Bremiz, транслируется в код на языке C, а затем компилируется в динамическую библиотеку с помощью оптимизирующего компилятора «Эльбрус». Полученная библиотека передаётся на целевое устройство по сети и обрабатывается программой beremiz_runtime с помощью функции dlopen. Управление отладкой на целевом устройстве также осуществляется из среды Beremiz.

Стенд для демонстрации отладки простейшего проекта состоит из: управляющей машины на Windows 10 со средой Beremiz и целевого устройства на Эльбрусе с программой beremiz_runtime и библиотекой uclibc-ng, работающими в режиме защищённых вычислений.

Инструменты для сборки программного обеспечения.

Сборка программ в режиме защищённых вычислений с библиотекой uclibc-ng для архитектуры «Эльбрус» на данный момент осуществляется при помощи **специального bash-скрипта e2k-linux-gcc**, который производит вызов оптимизирующего компилятора «Эльбрус» с необходимыми опциями. Опции компилятору подаются для того, чтобы настроить собранную программу на использование библиотеки uclibc-ng во время исполнения, указав пути к её основным компонентам: **разделяемым библиотекам** и **динамическому загрузчику**.

Программу beremiz_runtime необходимо собрать при помощи **скрипта e2k-linux-gcc** использую его в качестве компилятора и указав в скрипте пути, по которым будут размещаться основные компоненты библиотеки uclinbc-ng на целевой машине.

Установка программного обеспечения и настройка стенда.

1. На целевом устройстве на Эльбрусе необходимо:

- Загрузить дистрибутив Buildroot с uclibc-ng и ядро Linux 4.9 на любой свободный раздел.

- Выполнить загрузку машины с этого раздела.

- Разместить uclibc-ng, собранную в защищённом режиме по тому пути, на который было настроено программное обеспечение, а в частности beremiz_runtime при сборке.

- Разместить программное обеспечение beremiz_runtime по любому удобному пути.

2. Соединить управляющую машину с Windows 10 и целевое устройство кабелями Ethernet через коммутатор. После этого присвоить обоим 2 разных ip-адреса из одной подсети. Например: Windows 10: 192.168.0.110, целевое устройство: 192.168.0.90.

3. Запустить среду Beremiz на управляющей машине с Windows 10. Открыть проект который требуется отлаживать. В конфигурации проекта нужно внести следующие изменения:

- Указать путь к **сборочному скрипту e2k-linux-gcc** в качестве компилятора.

- Указать ip-адрес целевого устройства на Эльбрус, на котором будет отлаживаться программа и номер порта для подключения (по умолчанию используется порт 3000).

3.3. Запуск отладки программы из Beremiz на целевом устройстве.

Среда разработки «Beremiz» предназначена для создания и отладки прикладных программ на языках стандарта IEC 61131-3 для целевых устройств (программируемых логических контроллеров) на базе СМ1820М. В качестве языков описания алгоритмов и логики работы данных программ, могут выступать как текстовые Structured Text (ST) и Instruction List (IL), так и графические Function Block Diagram (FBD), Ladder Diagram (LD), Sequential Function Chart (SFC).

Для запуска отладки программы из Beremiz на целевом устройстве необходимо:

- Нажать кнопку «сборка проекта» в среде Beremiz. В случае успешного завершения сборки сгенерируется динамическая библиотека с исполняемым кодом, содержащим отлаживаемую программу.

- Запустить программу beremiz_runtime на целевом устройстве с указанием номера прослушиваемого порта с помощью опции -p, по умолчанию используется порт 3000.

- Нажать на кнопку «передать исполняемый файл на целевое устройство» в Beremiz. В случае успешной передачи будет выдано соответствующее сообщение.

- Нажать на кнопку «запуск исполняемого файла на целевом устройстве» в Beremiz. В случае успешного выполнения на целевом устройстве будет запущена отладка программы и будет выдано соответствующее сообщение.

3.4 Инструкция по компиляции и развертыванию встраиваемого дистрибутива

Для сборки и развертывания встраиваемого дистрибутива buildroot требуются 4 компоненты: кросс-компилятор для Эльбрус, ядро Linux 4.9, библиотека uclibc-ng и buildroot.

Необходимый набор утилит и исходных кодов поставляется в виде приложения к отчету на usb-flash (см. раздел 4).

В процессе портирования был разработан набор патчей и файл конфигурации дистрибутива buildroot для сборки с помощью кросс-компилятора lcc для архитектуры e2k.

Подготовка окружения

- Установить необходимые пакеты:

  # apt install sudo python unzip rsync bc python3 python3-pip

- Установить необходимые python3 пакеты:

  # pip3 install docopt

- Убедиться, что установлена хотя бы одна локаль:

  # dpkg-reconfigure locales

  Если команда запершается с ошибкой, установить пакет locales и повторить

  # apt install locales

  #dpkg-reconfigure locales

Сборка

- Создать директорию /opt/mcst и разрешить запись в нее непривилегированному пользователю:

  # mkdir -p /opt/mcst

  # chown username /opt/mcst
- Распаковать загруженный архив в директорию *./e2k_buildroot*
- Перейти в директорию *./e2k_buildroot*
- Запустить предварительный этап сборки: *make prepare*

  Если этап завершился ошибкой, проверить права на запись в /opt/mcst у активного пользователя.
- Запустить сборку: *make build*

  Если этап завершился ошибкой, проверить интернет-соединение.

Развертывание

- Подключить жесткий диск к рабочей станции
- Проверить какой идентификатор был присвоен этому диску (sda/sdb итд.).

  Далее по инструкции используется этот идентификатор вместо sdX.
- С помощью вызова утилиты *cfdisk sdX* создать 2 раздела:

  500M primary bootable

  20G+ primary

- Создать файловые системы:

  mkfs.ext2 sdX1

  mkfs.ext4 sdX2

- Вызвать скрипт развертывания дистрибутива: *install.py sdX1 sdX2*

  Дождаться завершения, отмонтировать диски:

  umount /dev/sdX1

  umount /dev/sdX2

Запуск на Эльбрусе

- Подключить диск, запустить машину
- Нажимать клавишу пробел до появления на экране **boot menu**
- Нажать клавишу **b**, посмотреть какой номер диска присвоен подключенному диске с buildroot, далее DX
- Нажать клавишу **c**, далее заполнить поля меню:

  drive_number: 7
  partition_number: 0
  command_string: console=tty0 consoleblank=0 hardreset transparent_hugepage=madvise root=/dev/sdb2
  filename: /image.boot
  initrdfilename: /initrd.img
  autoboot in: 3

- Нажать клавишу **s** .

Запуск демо

- После окончания загрузки система предложит ввести имя пользователя и пароль:

  login: root

  passwd: 1q2w3e

- После вывода bash promt запустить демо:

  fb_example

Для сборки и развертывания встраиваемого дистрибутива buildroot требуются 4 компоненты: toolchain, ядро Linux, uclibc-ng и buildroot.

4. Результаты

В процессе выполнения внутреннего ОКР «Защищенный режим» были достигнуты следующие результаты:

- Получена полнофункциональная библиотека uclibc-ng для Эльбрус в незащищенном режиме;

- Разработан минималистичный дистрибутив buildroot для Эльбрус, ориентированный на использование во встраиваемыъ системах, использующий библиотеку uclibc-ng. Этот дистрибутив может использоваться в качестве вспомогательного варианта при реализации сложных внедрений (как в случае с проектом Татнефть);
- В защищенном режиме отлажена библиотека uclibc-ng с поддержкой многопоточности и динамического связывания;
- В защищенном режиме отлажена библиотека libmodbus и сопутствующие библиотеки libmodbussrv, libmodbusregshelper;
- С помощью полученного набора библиотек выполнена интеграционная работа по внедрению результата в задачу программирования промышленных контроллеров. Результат пригоден для использования в промышленных контроллерах, построенных на «Эльбрус-1с+», разрабатываемых в ИНЭУМ (образцы – середина 2018 года);
- Получен новый набор тестов для защищенного режима;
- Создан задел для развития минималистичного дистрибутива в защищенном режиме части поддержки видео режима и графического режима.

Исходные тексты всех реализаций, полученных в процессе выполнения проекта поставляются на usb-flash в качестве приложения к данному отчету. Таблица 4.1 описывает структуру каталогов и содержимое usb-flash.

Таблица 4.1: описание каталогов на usb-flash (приложение к отчету)

№	Каталог	Краткое описание
1	kernel	Ядро linux-4.9 для Эльбрус с реализацией входа №8. Исходные тексты, бинарный образ ядра.
2	uclibc-ng	Библиотека uclibc-ng в защищенном режиме. Внутри библиотеки: libpthread (поддержка потоков), реализация динамического загрузчика. Исходные тексты, бинарные файлы библиотек (в обоих режимах: защищенном и незащищенном).
3	uclibc-tests	Исходные тексты тестов для защищенного режима
4	libmodbus	Исходные тексты библиотек libmodbus, libmodbussrv, libmodbusregshelper, бинарные файлы библиотек (в обоих режимах: защищенном и незащищенном).
5	buildroot_np	Исходные тексты и образ минимального дистрибутива buildroot в незащищенном

		режиме
6	plc	Исходные тексты демонстратора: проект Beremiz, бинарный образ ОС таргета в защищенном режиме
7	doc	Научно-технический отчет, содержащий технологические инструкции.

5. Люди, дальнейшее развитие специалистов и технологий защищенного режима

Дополнительным результатом выполнения ОКР «Защищенный режим» является успешное формирование группы разработчиков, способных мыслить в терминах защищенного режима, вести совместную разработку и общаться на языке защищенного режима: Мустафин Тимур, Алехин Андрей, Черкашин Сергей, Зубов Игорь, Лубинец Михаил. Сформированный коллектив способен решать задачи в направлении дальнейшего развития технологии защищенного режима. Дальнейшее развитие возможно в двух направлениях.

Первое направление: создание автоматизированных средств подготовки исходных текстов универсального дистрибутива к компиляции в защищенном режиме.

Работа показала, что в минимальной конфигурации полученный стек ПО можно применять в программировании промышленных контроллеров и демонстрировать уникальное свойство микропроцессора Эльбрус в этой области. Но сборка универсального дистрибутива типа Debian с пакетным менеджером, большим количеством программ на C++ потребует очень большого (неоправданно большого) времени. Это связано с тем, что в исходных текстах прикладных пакетов дистрибутива существует огромное количество мест, использующих опасные конструкции языка C, заведомо не работающие в защищенном режиме. В процессе работы были выявлены типичные случаи применения опасных конструкций:

1) использование выравнивания указателей с помощью приведения указателей к типу int при оптимизации работы с массивами (примером является реализация стандартной функции strlen в uclibc-ng);

2) использование чтения из-за границы массива с целью оптимизации (такие оптимиизации считаются нормальными если разработчик уверен, что чтение не пересекает границу страницы);

3) использование глобальной переменной для хранения указателя на стек.

Выявление таких конструкций – это наиболее затратное по времени действие при переносе ПО в защищенный режим. Возможным решением является создание встроенного в фронтенд компилятора анализатора перечисленных шаблонов программирования. Такой анализатор сможет при компиляции программы в защищенном режиме выдавать предупрежденния с указанием заведомо неработающих фрагментов.

Второе направление: дополнение минималистичного дистрибутива библиотеками отображения видео, набором математических библиотек. В параллельной работе, связанной с портированием авиационной ОС4000 на Эльбрус, где разработчики отдела А3К ОССН принимают участие, был получен интересный результат — реализация минимального набора библиотек, написанных на языке С для поддержки видео-библиотеки OpenGL. Полученный набор библиотек при переносе их в зашщищенный режим, позволит обеспечить решение всех типовых задач бортового ПО в защищенном режиме.

6. Сколько потрачено средств и откуда взять деньги для дальнейшего развития.

На выполнение внутреннего ОКР «Защищенный режим» было потрачено 3,9 миллиона рублей. Поиск финансирования для продолжния работ по развитию технологии защищенного режима производился в следующих направлениях:

1) Поиск средств среди целевой аудитории «авиационный клуб». Результат — договор ЦНПО «Ленинец» - ПАО «ИНЭУМ им. И.С. Брука», в стадии подготовки — договоры ПАО «Компания Сухой» - ПАО «ИНЭУМ им. И.С. Брука», АО «РПКБ» - ПАО «ИНЭУМ им. И.С. Брука». Работы предусмотренные договорами относятся к смежной теме, особенно в части формирования набора библиотек на языке С, необходимых для задач бортовых приложений.

2) Подготовка проекта «Процессор», финансируемого из Фонда Перспективных Исследований. В проекте предусмотрено финансирование темы «Защищенный режим» в размере 40 миллионов рублей в течение 2 лет. Проект готов к защите на НТС ФПИ. В случае положительного решения по финансированию средства ФПИ поступят в апреле 2018 года.

Для продолжения работы с января 2018 необходимо открытие нового внутреннего ОКР «Защищенный режим-2» с бюджетом 4 миллиона рублей.