

# Guardant®

Система защиты от компьютерного пиратства

## Электронные ключи Guardant Code первого поколения

### **Внимание!**

Вся информация в данном документе относится только к ключам Guardant Code первого поколения (сняты с производства 1 июля 2011 года)!

Для работы с современными ключами Guardant Code пользуйтесь вторым томом Руководства пользователя Guardant, глава 13.



# Содержание

<b>Работа с ключами Guardant Code /Code Time .....</b>	<b>5</b>
Устройство ключей Guardant Code.....	5
Память ключа Guardant Code .....	5
Разработка приложений для Guardant Code.....	7
Выбор кода для размещения в ключе .....	7
Средства разработки.....	8
Guardant Code API. Интерфейс прикладного программирования загружаемого кода.....	9
Компиляция загружаемого кода .....	11
Устройство загружаемого кода.....	18
Отладка загружаемого кода.....	22
Загрузка кода в электронный ключ .....	25
Отладка защищенного приложения .....	26
Дистанционное обновление загружаемого кода .....	26
Примеры использования загружаемого кода .....	27
Структура используемого файла маски.....	27
Краткая характеристика примеров.....	28
Характеристики и вычислительные возможности Guardant Code.....	29



# Работа с ключами Guardant Code /Code Time

Электронные ключи Guardant Code и Guardant Code Time представляют собой новое поколение аппаратных средств защиты программного обеспечения семейства Guardant. Основным защитным механизмом, реализуемым этими ключами, является возможность разработки и размещения в электронном ключе собственных алгоритмов, реализующих полезную для защищаемого приложения функциональность. Эти алгоритмы в дальнейшем будут называться загружаемым кодом. Выполнение загружаемого кода происходит внутри ключей Guardant Code / Code Time, выполняющих роль своего рода доверенной платформы.

## Устройство ключей Guardant Code

### Важная информация

Обратите особое внимание на отличия методов адресации памяти и на то, какие области памяти доступны для использования.

### Память ключа Guardant Code

В контроллере электронных ключей Guardant Code существует два вида памяти: **оперативная** (RAM) и **постоянная** (Flash).

**Оперативная память** (далее – RAM) используется для хранения переменных, стека, буферов ввода-вывода. Часть RAM используется системной микропрограммой (System RAM), а другая часть – загружаемым кодом (20 Кб).

Память, которая используется для хранения загружаемого кода и его констант, в дальнейшем будет называться **Flash-памятью**. Часть Flash-памяти занята системной микропрограммой (System Flash), другая часть предназначена для загружаемого кода (128 Кб или 352 Кб, в зависимости от модификации ключа).

В предыдущих моделях ключей Guardant энергонезависимая память для хранения ключевой информации защищаемого приложения была выполнена в виде отдельной микросхемы EEPROM (электрически перепрограммируемая постоянная память). Поэтому для сохранения преемственности и совместимости EEPROM эмулируется во Flash-памяти контроллера Guardant Code. В ней (точно так же, как в обычных ключах Guardant) хранятся системные поля данных, дескрипторы защищенных ячеек и аппаратных алгоритмов. Для ясности и краткости эта эмулируемая память в дальнейшем будет называться просто **EEPROM**.

### **Организация EEPROM**

Ключи с загружаемым кодом обладают EEPROM объемом 4096 байт. Организация этой области памяти идентично устройству EEPROM ключей Guardant Sign/Time/Sign Net/ Time Net и детально рассматривается в **Главе 10. Организация EEPROM памяти ключей Guardant**.

### **Карта Flash-памяти и RAM**

При работе с RAM-памятью и Flash-памятью для хранения загружаемого кода используется иная, чем в случае с EEPROM, адресация, основанная на карте памяти контроллера ключа.

Карта памяти, доступная для загружаемого кода, выглядит так:

<b>Адреса</b>	<b>Назначение</b>
<b>00020000h-0003FFFFh</b>	Flash-память для размещения загружаемого кода, страницы 1-4 (для варианта с 128 кб Flash-памяти)
<b>00040000h-00077FFFh</b>	Flash-память для размещения загружаемого кода, страницы 5-11 (для варианта с 352 кб Flash-памяти)
<b>40003000h-40007FDFh</b>	RAM (ОЗУ), доступная загружаемому коду. Тут размещаются: стек, буфер ввода-вывода, переменные загружаемого кода

Flash-память разделена на страницы по 32 Кб. При этом страница может быть занята только целиком. Вследствие такой организации памяти в Guardant Code может существовать от 1 до 4 (или до 11, в зависимости от объема доступной Flash-памяти) отдельных сегментов загружаемого кода.

Как правило, необходимости в нескольких независимых сегментах загружаемого кода не возникает, а потому в подавляющем большинстве случаев размещения для загружаемого кода резервируется вся доступная Flash-память.

## Разработка приложений для Guardant Code

### Выбор кода для размещения в ключе

Это самый ответственный и нетривиальный этап разработки загружаемого кода. Суть его заключается в том, что разработчику нужно решить, какой именно код будет выполняться в электронном ключе.

Код должен быть устроен таким образом, чтобы выполнять определенную конечную задачу, которая в общем виде выглядит так:

- Получение буфера с входными данными
- Вычисления над данными из этого буфера
- Возврат буфера с выходными данными

Существует целый ряд требований к этому коду, налагающих достаточно серьезные ограничения на выбор. Требования условно можно разделить на несколько типов:

#### Требования по безопасности

Загружаемый код должен быть достаточно сложным, чтобы брутфорс или иные (более эффективные и продвинутые) методы анализа черных ящиков не сделали возможным создания эмулятора в короткое время.

Этот код должен отсутствовать в более ранних версиях приложения. Несоблюдение этого условия делает возможным сравнение версий приложения и нахождение перенесенного кода для внедрения его в эмулятор.

#### Требования по производительности

Контроллер ключа обладает достаточно большими вычислительными возможностями, однако мощность его все же гораздо ниже, чем у современных процессоров. Поэтому важно, чтобы код не был слишком ресурсоемким, в противном случае время его выполнения может возрасти до неприемлемого уровня.

Кроме того, код, загружаемый в ключ, не должен вызываться слишком часто, например, в цикле. Если при единичном вызове задержка при выполнении не будет значительной, то при циклическом вызове она может оказаться очень существенной.

### **Требования по реализуемости**

Код, размещаемый в электронном ключе, не должен:

- Содержать вызовов API, которые нельзя было бы перенести в электронный ключ, или иметь внешние зависимости
- Использовать потоки ввода-вывода
- Использовать вывод на консоль
- Использовать динамическое распределение памяти

Коду, исполняемому внутри ключа, доступны лишь стандартные библиотеки C и функции для работы с электронным ключом.

Ключи Guardant Code позволяют исполнять алгоритмы до 16 тысяч строк кода на C (до 50 тысяч строк в моделях ключей с увеличенным размером памяти). Соответственно, размер кода должен укладываться в эти пределы.

### **Средства разработки**

Несмотря на то, что для процессоров ARM7 существуют компиляторы многих языков программирования, эффективнее всего использовать язык C, а точнее его подмножество, основой для которого является стандарт ANSI C.

Требование соблюдения стандарта связано с тем, что первоначальная разработка и отладка загружаемого кода может выполняться с использованием различных компиляторов. Т. е. предполагается, что код разрабатывается в привычной интегрированной среде, а затем компилируется для использования в Guardant Code. Поэтому для разработки загружаемого кода пригодятся навыки кроссплатформенного программирования.

Код перед загрузкой в ключ должен быть скомпилирован. Поскольку система команд процессора архитектуры ARM7, используемого в ключе, отличается от системы команд x86-совместимых процессоров, для работы потребует компилятор, способный генерировать двоичный код, совместимый с архитектурой ARM7.

Для компиляции загружаемого кода можно применять как коммерческие средства (например, Keil), так и распространяемые под различными «свободными» лицензиями. К последним можно отнести GCC (включая различные решения на его основе, такие как WinARM, и т. д.).

Большинство средств разработки, распространяемых под свободными лицензиями, рассчитаны на работу под Linux. Тем не менее, существуют решения и для Windows. Кроме того, есть возможность использовать средства разработки, предназначенные для Linux под операционными системами семейства Windows, применяя для их запуска Cygwin или MinGW.

Примеры и makefile, входящие в комплект разработчика Guardant, рассчитаны на использование компилятора **GCC** и инструментария **WinARM**, как на самый доступный вариант. При использовании альтернативного инструментария и собственных сборок GCC рекомендуется использование GCC версии 4.1.1. У более новых версий GCC имеются проблемы с использованием библиотеки **newlib** во встроенных системах.

Хотя для большинства высокоуровневых языков программирования перенос кода на C достаточно несложен, но альтернативно, можно использовать и другие языки программирования высокого уровня, для которых есть компилятор ARM7. Однако в данном документе эта возможность не описывается.

## **Guardant Code API. Интерфейс прикладного программирования загружаемого кода**

При разработке загружаемого кода с большой вероятностью может возникнуть необходимость обращаться к ресурсам ключа, находящимся в области **EEPROM** (защищенным ячейкам, алгоритмам), или к таймеру. Поэтому был разработан специальный интерфейс прикладного программирования **Guardant Code API** (см. Справочную систему Guardant API, файл **GrdAPI.chm**) Библиотека этого API содержит большинство функций Guardant API, адаптированных для использования из среды загружаемого кода.

Основной нюанс при работе с **Guardant Code API** состоит в том, что хэндл защищенного контейнера в загружаемом коде теряет смысл, поскольку этот код, во-первых, имеет доступ только к одному ключу, а во-вторых, не существует ситуации конкурентного доступа к ресурсам ключа из разных потоков одного приложения и из разных приложении.

Вместе с тем, функциям внутреннего API загружаемого кода передается параметр типа HANDLE. Это сделано для соблюдения единообразия и удобства отладки загружаемого кода.

**Guardant Code API** поддерживает основные функции Guardant API, связанные с хранением данных и работой с алгоритмами.

Кроме того, в API загружаемого кода существует возможность вызывать криптографические алгоритмы, не используя дескрипторы, а напрямую, подобно тому, как в Guardant API вызываются программно-реализованные алгоритмы. Для этого вместо числового имени ячейки, содержащей дескриптор, указывается специальное зарезервированное имя алгоритма.

Если в загружаемом коде присутствуют функции Guardant API (например, в ключ переносится алгоритм, который раньше защищался ключами Guardant), то для большинства этих функций существуют аналоги в Guardant Code API, и портиро-

вание будет заключаться в смене префикса с GrdXXX на GcaXXX или GccaXXX.

Название функции	Краткое описание
<b>GcaCrash()</b>	Инициировать ошибку среды исполнения Guardant Code
<b>GcaExit()</b>	Выйти из загружаемого кода. В вызывающее приложение передается код возврата
<b>GcaLedOn()</b>	Включить светодиодный индикатор
<b>GcaLedOff()</b>	Выключить светодиодный индикатор
<b>GcaRead()</b>	Прочитать данные из EEPROM, аналогично GrdRead()
<b>GcaWrite()</b>	Записать данные в EEPROM, аналогично GrdWrite()
<b>GcaPI_Read()</b>	Прочитать данные из защищенной ячейки, аналогично GrdPI_Read()
<b>GcaPI_Update()</b>	Изменить данные защищенной ячейки, аналогично GrdPI_Update()
<b>GcaPI_GetTimeLimit()</b>	Получить время жизни защищенной ячейки, аналогично GrdPI_GetTimeLimit(). Для Guardant Code Time
<b>GcaPI_GetCounter()</b>	Получить значение счетчика оставшихся выполнений алгоритма, аналогично GrdPI_GetCounter()
<b>GcaGetTime()</b>	Получить текущее время из RTC ключа, аналогично GrdGetTime(). Для Guardant Code Time
<b>GcaGetRTCQuality()</b>	Проверить валидность значения RTC, аналогично GrdGetRTCQuality(). Для Guardant Code Time
<b>GcaGetLastError()</b>	Получить последний код ошибки, аналогично GrdGetLastError(). Возвращает код ошибки, которая произошла при последнем вызове функции Guardant Code API
<b>GccaCryptEx()</b>	Шифровать данные симметричным алгоритмом, аналогично GrdCryptEx(). Для программного алгоритма AES128 требуется другой размер контекста. Контекст должен быть выровнен по границе в 4 байта (!) применением макроса ALIGNED
<b>GccaSign()</b>	Вычислить ЭЦП, аналогично GrdSign(). Присутствует дополнительный параметр – ключ подписи и вариант вызова, работающий в обход таблицы дескрипторов
<b>GccaVerifySign()</b>	Проверить ЭЦП, аналогично GrdVerifySign()
<b>GccaGenerateKeyPair()</b>	Генерировать ключевую пару для алгоритма ЭЦП ECC160
<b>GccaHash()</b>	Вычислить хэш-функцию, аналогично GrdHash()
<b>GccaGetRandom()</b>	Генерировать случайное однобайтовое число
<b>GcaSetTimeout</b>	Установить разрешенное время работы загружаемого кода
<b>GcaCodeGetInfo</b>	Запросить информацию из дескриптора загружаемого кода
<b>GcaCodeRun</b>	Выполнить код из другого участка загружаемого кода

### Важная информация

1. Подробную информацию по функциям внутреннего Guardant Code API см. в Справочной системе по Guardant API (файл **GrdAPI.chm**).
2. Поскольку в Guardant Code не реализован алгоритм GSII64 и производные от него (HASH64, RAND64 и т. д.), возможно придется немного переработать существующую схему защиты на использование алгоритмов AES128 для шифрования и SHA256 для хэширования. Все остальные возможности предыдущих поколений ключей присутствуют и в Guardant Code.

## Сервисные функции GrdAPI для работы с загружаемым кодом

Загружаемый код, в отличие от ячеек с данными и дескрипторов аппаратных алгоритмов, хранится в другой области памяти ключа, для которой используются иные принципы адресации. Поэтому для загрузки кода в память ключа, его выполнения и других операций существуют специальные функции Guardant API.

Для записи в ключ загружаемый код преобразуется в файл специального формата **GCEXE** (Guardant Code Executable), обеспечивающего защиту от подделки, подмены и анализа. Эта защита обеспечивается шифрованием криптостойкими алгоритмами и ЭЦП. Такая защита делает возможным безопасное обновление загружаемого кода в ключе, в том числе при пересылке файлов по открытым каналам и через сети общего пользования.

Файл формата GCEXE генерируется на основе данных дескриптора загружаемого кода, скомпилированного кода и map-файла утилитой программирования ключей **GrdUtil.exe**. Однажды сгенерированный файл может быть использован для записи в тиражируемые ключи той же утилитой. Если предпродажное программирование осуществляется при помощи собственных специально разработанных инструментов, файл с загружаемым кодом может быть записан в ключ при помощи функции GrdCodeLoad().

Список сервисных функций Guardant API:

Название функции	Код доступа	Краткое описание
<b>GrdCodeGetInfo</b>	Private Read	Получить информацию из дескриптора загружаемого кода
<b>GrdCodeLoad</b>	Private Read	Записать GCEXE-файл, содержащий загружаемый код, во Flash-память ключа
<b>GrdCodeRun</b>	Private Read	Выполнить загружаемый код
<b>GrdSetDriverMode</b>	Private Read	Задать USB-режим работы ключа

Подробнее см. в Справочной системе по Guardant API (файл **GrdAPI.chm**).

## Компиляция загружаемого кода

### Инсталляция и настройка компилятора GCC

Для компиляции загружаемого кода используются компилятор и линкер **GCC**, стандартная библиотека C для встраиваемых систем **newlib**, утилита **make** и несколько сервисных утилит.

Существует два варианта использования GCC:

1. Загрузить исходный код GCC, make, newlib из соответствующих репозиторийев и самостоятельно скомпилировать.
2. Воспользоваться готовым комплектом инструментов для разработки под ARM.

Для работы с ключами Guardant Code рекомендуется использовать свободно распространяемый инструментарий **WinARM**. Все примеры из комплекта разработчика, тестировались именно на нем.

Для того, чтобы приступить к работе с WinARM требуется выполнить несколько простых шагов:

- Загрузить WinARM 20060606 с сайта разработчика:  
[http://www.siwawi.arubi.uni-kl.de/avr\\_projects/arm\\_projects/#winarm](http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/#winarm)
- Ссылка для загрузки zip-архива:  
[http://www.siwawi.arubi.uni-kl.de/avr\\_projects/arm\\_projects/WinARM-20060606.zip](http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/WinARM-20060606.zip)
- Распаковать загруженный архив в директорию C:\WinARM.
- Добавить путь к компилятору и утилитам в переменную PATH.

Необходимо добавить пути:

```
"C:\WinARM\bin;C:\WinARM\utils\bin;"
```

После выполнения этих шагов компилятор полностью готов к работе. Дальнейшая работа с ним будет выполняться посредством вызова утилиты **make** на заранее созданных и настроенных **makefile**.

### Важная информация

При работа в ОС WinVista и выше при выполнении **make** будут выводиться ошибки:

Перед исполнением:

```
0 [main] sh 3048 sync_with_child: child 1608(0x144)
died before initialization with status code 0x0
601 [main] sh 3048 sync_with_child: *** child state
waiting for longjmp
/usr/bin/sh: fork: Resource temporarily unavailable
```

После выполнения:

```
Size after:
0 [main] sh 2992 sync_with_child: child 5760(0x138)
died before initialization with status code 0x0
580 [main] sh 2992 sync_with_child: *** child state
waiting for longjmp
/usr/bin/sh: fork: Resource temporarily unavailable
make: *** [sizeafter] Error 128
```

Это штатная ситуация.

Об успешности выполнения сборки можно судить по наличию файлов \*.bin/\*.bmap в директории .out.

## Общие сведения о компиляции и сборке

Все инструкции для компилятора и линкера, а также команды для обработки скомпилированного кода, содержатся в конфигурационном файле утилиты **make** (имя этого файла по умолчанию - **makefile**).

Соответственно, для компиляции приложения необходимо:

1. Отредактировать настройки в секции **Main Configuration** внутри **makefile**:
  - Задать желаемое имя точки входа, адреса RAM, ROM
  - Задать требуемый размер стека
  - Задать имена и размеры буферов ввода-вывода
  - Задать путь к системным утилитам **hex2bin.exe** и **map\_parse.exe**
  - Задать путь к папке с заголовочными файлами **GrdAPI.h** и **GcaAPI.h**
2. Сгенерировать шаблон проекта: **make template** (для использования компиляторов, отличных от GCC, требуется ручная настройка проекта). Проект должен иметь определенный набор файлов и быть настроен для работы в среде Guardant Code
3. Скопировать в папку проекта файлы с исходными текстами с модулями на C/C++ и добавить имена этих файлов в переменные **makefile SRC** и **CPPSRC** соответственно
4. В скопированных модулях должна присутствовать функция с именем, совпадающим с именем заданной точки входа (и соответствующим прототипом)
5. После этого можно выполнить команду: **make** и приложение будет собрано
6. В папке проекта будет создана папка **.out** с двумя файлами, имеющими расширения **bin** и **bmap**. Эти файлы требуются для генерации GCEXE-файла и загрузки его в электронный ключ при помощи утилиты GrdUtil.exe

## Команды утилиты make

Приложение собирается при помощи GNU-утилиты **make**, которая использует конфигурационный файл с именем **makefile**.

Для утилиты **make** доступны следующие команды:

### 1. Сборка проекта

Если конфигурационный файл имеет имя по умолчанию (makefile):

```
make
```

Если конфигурационный файл имеет имя, отличное от имени по умолчанию:

```
make -f confname
```

### 2. Удаление всех файлов, создаваемых при сборке (файлы, создаваемые при `make template` не удаляются)

```
make clean
```

или

```
make -f confname clean
```

### 3. Создание шаблона приложения

```
make template
```

или

```
make -f confname template
```

#### **Важная информация**

Если в сгенерированные при создании шаблона проекта файлы **Startup.S** и **rom.ld** были внесены изменения, они будут потеряны при повторном создании шаблона!

### Полная пересборка приложения

```
make clean  
make all
```

или

```
make -f confname clean  
make -f confname all
```

Пересборка может требоваться при изменении уровня оптимизации и при добавлении новых файлов (см. следующий раздел).

#### **Важная информация**

Если в системе одновременно с GCC установлены другие компиляторы, использующие собственные утилиты `make` (например, Borland C), при вызове `make` следует указывать полный путь, поскольку пути к другим утилитам `make` могут быть прописаны в переменной среды `PATH`. Можно придумать и иные способы дифференциации.

## Настройка универсального makefile

Универсальный **makefile** содержит секции настроек с параметрами:

- Генерации шаблона проекта командой **make template**
- Сборки приложения по команде **make/make all**

При внесении изменений в первую секцию требуется регенерация шаблона проекта (см. соответствующий раздел). При внесении изменений во вторую секцию требуется пересборка проекта путем подачи команды **make clean** и затем **make all**.

Настройки секции генерации шаблона:

Имя параметра	Значение
CFG_ENTRYPOINT_NAME	Имя точки входа (по умолчанию функция main)
CFG_PROGRAM_ADDR *)	Адрес Flash-памяти, по которому располагается приложение
CFG_PROGRAM_SIZE *)	Размер приложения во Flash-памяти
CFG_RAM_ADDR *)	Адрес начала RAM, зарезервированной для загружаемого кода
CFG_RAM_SIZE *)	Размер RAM, зарезервированной для загружаемого кода
CFG_INPUT_BUFFER_NAME	Имя буфера ввода, через который данные передаются в загружаемый код
CFG_INPUT_BUFFER_SIZE	Размер буфера ввода
CFG_OUTPUT_BUFFER_NAME	Имя буфера вывода, данные из которого возвращаются вызывающему приложению
CFG_OUTPUT_BUFFER_SIZE	Размер буфера вывода
CFG_STACK_SIZE	Размер программного стека
CFG_INCLUDE_DIR	Путь до директории, содержащей заголовочные файлы GcaAPI.h и GrdAPI.h
CFG_SYS_DIR	Путь до директории, содержащей служебные утилиты
CFG_TARGET_NAME	Имя двоичного bin-файла, получаемого при компиляции

\*) Поскольку в ключе под загружаемый код по умолчанию резервируется вся Flash-память и вся RAM, значения в этих пунктах изменять не нужно.

В **makefile** доступны следующие настройки сборки проекта:

Имя параметра	Значение
<b>OPT</b>	Уровень оптимизации. Рекомендуемые значения 2 или s (так же допустимые значения 0 и 1, значение 3 крайне не рекомендуется)
<b>SRC</b>	Набор C-файлов, используемых в проекте
<b>ASRC</b>	Набор ASM-файлов, используемых в проекте

Важно заметить, что задаваемые имена файлов зависят от регистра. К примеру, при несовпадении регистра в имени файла **main.c**, при сборке может возникнуть следующая ошибка:

```
----- begin -----
make: *** No rule to make target `main.o', needed by
`elf'. Stop.
```

Несовпадение имен как таковых проверяется отдельно и вызывает более внятную ошибку:

```
----- begin -----
File main.c not found! Please check makefile (SRC, ASRC
and CPPSRC values).
```

### Точка входа в приложение

При старте приложения, в самом начале начинает исполняться код, находящийся в файле **Startup.S**. Он инициализирует стек и С-окружение (предварительно инициализированные переменные) и обнуляет неинициализированные переменные и область стека, при необходимости вызывает конструкторы глобальных объектов С. После этого он передает управление в приложение на С. Этот файл генерируется автоматически из универсального **makefile**.

По умолчанию точка входа в С-приложении на GCC имеет стандартное имя **main**. Прототип, однако, отличается от стандартного ANSI C и имеет следующий вид:

```
int main(
DWORD dwInDataLng, DWORD dwOutDataLng, DWORD dwP1);
```

Где:

**dwInDataLng** – размер данных поступивших из PC,

**dwOutDataLng** – размер данных, который PC запрашивает назад,

**dwP1** – параметр dwP1, переданный функции GrdCodeRun().

Если требуется изменить адрес точки входа, то в файле Startup.S требуется исправить строчки:

```
.global main
...
LDR    R4, =main
```

### Адресное пространство

В микроконтроллерах на основе ядра ARM7, на которых построен ключ Guardant Code, имеется единое адресное пространство в 4Гб.

Для загружаемого кода доступны следующие диапазоны адресов:

Адреса	Назначение
<b>00020000h-0003FFFFh</b>	Flash-память для размещения загружаемого кода и ROM-секции микропрограммы (для варианта с 128 кб Flash-памяти)
<b>00020000h-00077FFFh</b>	Flash-память для размещения загружаемого кода и ROM-секции микропрограммы (для варианта с 352 кб Flash-памяти)
<b>40003000h-40007FDFh</b>	RAM (ОЗУ), доступная загружаемому коду. Тут размещаются: стек, буфер ввода-вывода, переменные загружаемого кода

Диапазон используемых адресов указывается в **makefile** (параметры **CFG\_PROGRAM\_ADDR**, **CFG\_PROGRAM\_SIZE**, **CFG\_RAM**, **CFG\_RAM\_SIZE**). Задаваемые адреса должны быть кратны 0x8000 байт, и быть выровнены по границе 32768 байт.

Диапазон адресов, доступных загружаемому коду, описывается в соответствующем дескрипторе аппаратного алгоритма. GrdUtil автоматически заполняет соответствующие поля дескриптора информацией из файла **\*.bmap**.

Поскольку по умолчанию под загружаемый код резервируется вся Flash-память и вся RAM, значения этих настроек без насущной необходимости изменять не нужно.

### Буферы ввода-вывода

Имена буферов ввода и вывода в **makefile** могут быть разными, а могут и совпадать.

В случае, когда они совпадают, выделяется один буфер, который работает одновременно и на ввод, и на вывод. В C-коде буфер ввода-вывода может быть объявлен так:

```
extern BYTE iodata[];
```

При этом в параметрах **CFG\_INPUT\_BUFFER\_NAME** и **CFG\_OUTPUT\_BUFFER\_NAME** указывается значение **iodata**.

Если же используются отдельные буферы, то каждый из них объявляется в C-коде отдельно, и имеет собственное имя и размер.

При объявлении буферов допустимо использование любых типов данных, однако в случае структур рекомендуется добавлять в определение макрос **ALIGNED**, например:

```
extern struct
{
    double x;
    ...
} iodata ALIGNED;
```

Это указывает компилятору, что структура выровнена в памяти, и позволяет генерировать более эффективный код для доступа к полям структуры.

По умолчанию размер буфера ввода-вывода установлен равным 1024 байта. Для ввода-вывода в примерах используется единый буфер. Перед запуском загружаемого приложения данные в него помещаются, а после окончания работы возвращаются обратно в PC.

Максимальный суммарный размер буферов для ввода-вывода составляет 0x3F00 байт (16128 байт). Так же в объявлении переменной желательно указание макроса **ALIGNED**, который говорит компилятору, что буфер выровнен в памяти, и, в некоторых случаях, оптимизировать доступ к данной переменной.

### Стек

Размер программного стека для GCC указывается в **makefile**. За это отвечает параметр:

```
CFG_STACK_SIZE = 0x800;
```

Т. е., размер стека по умолчанию равен 2кБ (0x800 байтам).

Поскольку размер RAM достаточно сильно ограничен, рекомендуется небольшие и простые, но часто используемые функции оформлять как **inline**. Можно использовать макрос **INLINE** из **syscalls\_public.h**. Например:

```
INLINE void add(int a, int b)
{
return a+b;
}
```

За счет этого происходит экономия памяти стека и увеличивается быстродействие кода.

## Устройство загружаемого кода

Прямой перенос кода из исходного приложения может быть сопряжен с определенными трудностями. В общем случае, код, перенесенный в том же виде, как он существует в приложении, будет неработоспособен в электронном ключе. Поэтому код должен быть модифицирован и оптимизирован для выполнения на платформе ARM7. Желательно, чтобы этот код был написан заново и реализовывал функции, которых в ранних версиях приложения не было, либо эти функции должны быть видоизменены.

### Параметры функции main()

Прототип функции **main()** объявляется следующим образом:

```
DWORD main (
    DWORD dwInDataLng, DWORD dwOutDataLng, DWORD dwP1)
```

Параметры **dwInDataLng** и **dwOutDataLng** устанавливают количество данных, считываемых из буфера ввода и возвращаемых в буфер вывода. Параметр **dwP1** используется для передачи кода подфункции загружаемого кода.

Параметр **dwP1** передается функции **GrdCodeRun()** и его можно получить в загруженном коде в виде третьего параметра функции **main** (функции, которая первой получает управление в С-коде):

```
DWORD func1(dwInDataLng, dwOutDataLng)
{
    // Логика работы 1:
    return 101;
}

DWORD func2(dwInDataLng, dwOutDataLng)
{
    // Логика работы 2:
    return 102;
}

DWORD func3(dwInDataLng, dwOutDataLng)
{
    // Логика работы 3:
    return 103;
}

DWORD main(DWORD dwInDataLng, DWORD dwOutDataLng, DWORD
            dwP1) {
    switch (dwP1)
    {
        case 0x01:
            return func1(dwInDataLng, dwOutDataLng);
        case 0x02:
            return func2(dwInDataLng, dwOutDataLng);
        case 0x03:
            return func3(dwInDataLng, dwOutDataLng);
        case 0x04:
            // ...
        default:
            return -1;
    }
}
```

### Статические и глобальные переменные

По возможности, переменные лучше объявлять глобально (хотя это и противоречит принципам функционального программирования), а не в теле функции, чтобы не передавать данные через стек. Этим экономится память стека и увеличивается быстродействие.

В загружаемом коде можно создать глобальные переменные, содержимое которых не будет обнуляться между вызовами. Такие переменные требуется объявлять с макросом **NO\_INIT**:

```
DWORD buffer[100] NO_INIT
```

Эти переменные являются аналогами статических переменных.

Польза от них может заключаться в возможности запоминания некоторых состояний загружаемого кода. Это делает анализ «черного ящика» гораздо более сложным.

### Возврат из загружаемого кода

Выход из приложения можно осуществлять следующим образом:

- Возврат из `main` при помощи **return**. Код возврата будет помещен в параметр `dwRetCode` функции **GrdCodeRun()**
- Вызов функции **GcaExit()**. Ей также передается код возврата

Кроме того, принудительное завершение приложения происходит в следующих случаях:

- Наступление таймаута времени выполнения загружаемого кода (3 секунды)
- Попытка выполнения приложением недопустимого действия (обращение к недопустимым адресам памяти и т.д.)

В примерах в качестве кода возврата с ошибкой используется значение `-1`. Для упрощения отладки можно возвращать значения макроса `__LINE__` или пользоваться вызовом **GcaExit(0, \_\_LINE\_\_)**. Этот способ поможет определить строку, на которой произошел выход из приложения. Оставлять в конечных версиях возвраты в данном виде нежелательно, так как это может дать дополнительную информацию для злоумышленника.

Для отладки можно, к примеру, использовать следующий макрос:

```
#define ASSERT(cond) {if (cond) GcaExit(0, __LINE__);} ;
```

`ASSERT(x != 0);` // Если `x!=0`, осуществит возврат из программы с указанием номера строки, в которой вставлен `ASSERT`.

**Особенности выравнивания в ARM7**

Ядро ARM7, лежащее в основе ключа Guardant Code, предъявляет следующие требования к выравниванию данных:

- Слова (2-байтные переменные), должны быть выровнены по границе в 2 байта
- Двойные слова (4-байтные переменные) – в 4 байта

Поэтому при приведении типов могут возникать ошибки.

Компилятор учитывает эти ограничения, и организует верный доступ к невыровненным полям структур, а также размещает массивы типов WORD и DWORD выровненными в памяти. Однако при работе с указателями и явными приведениями типов могут возникнуть проблемы.

Например, если задать массив типа BYTE и обращаться к нему следующим образом:

```
BYTE abData[10];
*(DWORD*)(abData+4) = 1;
```

.. то данное обращение может работать неверно, в случае если адрес **abData** не будет кратен 4. Так как этот массив типа BYTE, компилятор не видит причин выравнивать его в памяти, и после приведения к DWORD происходит обращение к невыровненным данным. Чтобы в данном случае код работал верно, при объявлении массива abData требуется применить дополнительный макрос ALIGNED, который явно укажет компилятору расположить данный массив по адресу, кратному 4:

```
BYTE abData[10] ALIGNED;
*(DWORD*)(abData+4) = 1;
```

Для того чтобы понять поведение ARM ядра при обращении к невыровненным данным, можно привести следующий код:

```
DWORD adwData[2] = { 0, 0 };
BYTE *pbP = (BYTE*)adwData;
DWORD *pdwP = (DWORD*)(&pbP[1]);
dwP[0] = 0x12345678;
```

После его работы в переменной adwData будет лежать не ожидаемое значение { 0x345678, 0x12 }, а { 0x34567812, 0 }.

Во избежание проблем с выравниванием следует использовать переменные типа DWORD, INT (32 бита). Также можно использовать BYTE, CHAR, но они обрабатываются не столь эффективно.

Использование HALF, WORD не рекомендуется, так как вызывает наибольшее замедление скорости работы.

Подробнее о данном поведении ARM7-ядра можно прочитать в официальной документации на ядро ARM7TDMI-S (к примеру, ARM7TDMI-S Data Sheet (ARM DDI 0029E) на странице 72) в разделе с описанием работы инструкций LDR/STR.

### Использование арифметики с плавающей точкой

За вычисления с плавающей точкой отвечает библиотека **libm** из комплекта GCC. Полное описание математических функций, доступных в ней, можно найти в документации к данной библиотеке. Для каждой функции имеется 2 варианта: обычный, для вычислений с двойной точностью (тип double), а также с приставкой «f», для вычислений с половинной точностью (тип float).

Имеются особенности, связанные с обратным хранением слов в double-значении, которые продемонстрированы в соответствующем примере (см. [Краткая характеристика примеров](#), пункт 16).

## Отладка загружаемого кода

Разработка и первоначальная отладка загружаемого кода производится на компьютере. Для этого можно использовать любую IDE и отладчик языка C.

Основная проблема состоит в том, что отлаживать уже загруженный код затруднительно, поскольку нет возможности «залезть» отладчиком в контроллер ключа. Поэтому первоначально отлаживают сам алгоритм загружаемого кода.

Загруженный в ключ код имеет ограниченные возможности для трассировки. Например, нельзя вывести трассу на консоль или записать в файл. Однако некоторые средства все же есть. Для этой цели можно использовать **функции управления светодиодом**. Сигналы, подаваемые с его помощью, можно применять в качестве признаков прохождения тех или иных веток кода.

Как вариант, можно использовать принудительный возврат из загружаемого кода с соответствующим кодом возврата и передачей необходимых для отладки данных через буфер вывода.

Методы отладки кода такие же, как и при разработке на PC. Если в загружаемом коде используются вызовы Guardant Code API, то для отладки не нужно загружать этот код в ключ: можно использовать входящую в комплект разработчика отладочную библиотеку.

## Описание отладочной библиотеки

Отладочная библиотека представлена двумя частями:

- Модуль для загрузки в электронный ключ,
- Динамическая библиотека, содержащая функции, прототипы которых аналогичны тем, что доступны для загружаемого кода внутри электронного ключа (GcaXXX и GccaXXX)

Порядок работы с отладочной библиотекой таков:

1. При помощи GrdUtil.exe создается файл маски, в котором один из алгоритмов представляет собой отладочный модуль загружаемого кода – **DebugModule.bin**. При этом нужно убедиться, что соответствующий модулю **bmap**-файл находится в той же директории. Можно взять готовый файл маски **DebugMask.nsd** из примера и изменить его для использования в собственном приложении. От файла маски зависит, какой номер будет у алгоритма, содержащего загружаемый код.
2. Полученный файл маски с отладочным модулем прошивается в электронный ключ при помощи GrdUtil.
3. К проекту загружаемого кода на PC подключается отладочная библиотека **gcaapidll.dll**. Для этого используется библиотека экспортов **gcaapidll.lib**. Файл **GcaAPI.dll.h** содержит описание прототипов функций GcaXXX/GccaXXX.
4. Перед вызовами функций GcaXXX/GccaXXX из **gcaapidll.dll** в исходном коде следует разместить вызов макроса **DEBUGDLL\_INIT(hHandle, dwAlgoNum)**, который настраивает библиотеку для работы с текущим контекстом Guardant API. Макрос осуществляет привязку библиотеки к используемому контексту Guardant API и открытому ключу, также ему передается номер аппаратного алгоритма, в который был загружен отладочный модуль.
5. Если код, предполагаемый для размещения в электронном ключе, использует вызовы внешнего Guardant API, то их требуется заменить на соответствующие вызовы Guardant Code API. В данном случае функции импортируются из отладочной библиотеки. Если же код содержит функции Guardant API, не имеющие прямых аналогов в Guardant Code API, требуется создать эквивалентные им конструкции из доступных функций.

Следует принимать во внимание, что ни сама отладочная библиотека, ни отладочный модуль не содержат логики работы функций. Они являются всего лишь своеобразным «туннелем», через который параметры вызова функций передаются в электронный ключ и возвращаются обратно.

Пример использования макроса **DEBUGDLL\_INIT**:

```
main()
{
// Хэндл ключа, в котором запускается загруженный
//      пользователем код:
HADNLE hGrd;

// Инициализация API и подключение к электронному ключу
...

#ifdef DEBUG
// Инициализация DLL
DEBUGDLL_INIT(hGrd, 1);
// hGrd - хэндл открытого электронного ключа.
// 1 - номер аппаратного алгоритма, в котором находится
//      загруженный отладочный модуль
#endif

// Передача параметра hGrd необязательна.
GcaGetRandom(0, &iodata[i]);

return 0;
}
```

Также стоит отметить, что функции **GcaExit()** и **GcaLedOn()/GcaLedOff()** не могут работать в отладочном режиме. Первая – из-за того, что результат ее работы просто нельзя зафиксировать, а функции управления светодиодом – из-за того, что сразу после их вызова работа кода будет завершаться, при этом индикатор просто загорается вновь.

Использование отладочной библиотеки демонстрируется в примере №9 (см. [Краткая характеристика примеров](#)).

## Загрузка кода в электронный ключ

Для загрузки кода в электронный ключ первоначально используется GrdUtil. При помощи этой утилиты создается дескриптор аппаратного алгоритма типа **Загружаемый код**.

В свойствах алгоритма указывается бинарный файл, который содержит скомпилированный загружаемый код. Этому файлу должен сопутствовать файл `map`, содержащий настройки адресов памяти.

Бинарный файл перед загрузкой должен быть преобразован в файл типа GCEXE (Guardant Code executable). Преобразование осуществляется в автоматическом режиме утилитой программирования ключей GrdUtil.

При выполнении преобразования GrdUtil генерирует ключевые пары:

- **Для зашифрования и расшифрования загружаемого кода**  
Зашифрование производится на открытом ключе, который хранится в маске и не записывается в электронный ключ.  
Расшифрование — на закрытом ключе, который хранится и в файле маски, и в дескрипторе алгоритма, записанного в электронный ключ
- **Для электронной цифровой подписи загружаемого кода**  
Подписывание производится — на закрытом ключе, который хранится только в маске и не записывается в сам ключ. Проверка — на открытом, который будет храниться и в маске, и в дескрипторе алгоритма, который будет записан в ключ

Перед загрузкой бинарный файл зашифровывается на сеансовом ключе и подписывается ЭЦП. Это гарантирует возможность загрузки кода только разработчиком. При необходимости файл GCEXE можно сгенерировать таким образом, чтобы он мог быть загружен только в ключ с указанным ID. Эта возможность полезна для создания адресных обновлений, например — платных.

При записи данных в ключ первоначально записывается дескриптор алгоритма, а уже затем — файл GCEXE.

Однажды сгенерированный файл GCEXE может быть в дальнейшем записан и в другие ключи, содержащие соответствующие ключи шифрования и подписи. Для этого используется функция **GrdCodeLoad()**.

## **Отладка защищенного приложения**

Отладке приложений, использующих загружаемый код, следует уделить особое внимание, поскольку поиск ошибок при работе с «черным ящиком» является непростым делом.

Очень важным является итоговое быстродействие загруженного кода. Если оно получается неудовлетворительным, требуется принять меры по приведению кода к обозначенным в начале этой главы требованиям.

## **Дистанционное обновление загружаемого кода**

Проблема обновления информации в ключах, уже находящихся у пользователей приложения, актуальна и для загружаемого кода. Рано или поздно в этот код может потребоваться внести изменения или исправления.

Для успешного обновления загружаемого кода необходимо выполнение следующих условий:

- У разработчика должна храниться прошивка (файл маски), содержащая ключевые пары для шифрования и подписи загружаемого кода,
- У конечного пользователя должен находиться электронный ключ Guardant Code, содержащий дескриптор алгоритма с загружаемым кодом, а также закрытый ключ для расшифрования кода и открытый ключ для проверки ЭЦП
- Ключевые пары в маске и ключе должны быть идентичны.

Для обновления загружаемого кода необходимо сгенерировать новый GCEXE-файл с обновленным кодом, зашифрованным и подписанным на соответствующих ключах.

Само обновление может производиться как при помощи технологии TRU, так и прямой загрузкой GCEXE-файла из защищаемого приложения функцией GrdCodeLoad().

При желании можно сделать процедуру обновления загружаемого кода «прозрачной» для пользователя. Тогда от него потребуется только получить обновление, поместить его рядом с исполняемым файлом приложения (или в специально для этого предназначенную директорию) и запустить приложение.

## Примеры использования загружаемого кода

Каждый пример находится в директории установки комплекта разработчика Guardant (по умолчанию **%Program Files%\ Guardant\Guardant 5\%Общий код%\Samples\ARM\**), в отдельной папке вида **XX - SampleName**, и состоит из двух частей:

- *Проект Visual Studio 2005 .NET* в папке **Win32**, демонстрирующей использование Guardant API (т. е. необходимую инициализацию и запуск загруженного кода)
- *Пример загружаемого кода* в папке **Loadable Code**, содержащий проект GCC-ARM в виде исходного кода на C и **makefile**. При сборке примера получаются \*.bin и \*.bmap файлы, которые необходимо указывать GrdUtil для генерации GCEXE-файла соответствующего алгоритма.

## Структура используемого файла маски

Файл маски (**Mask.nsd**), используемый в примерах для ключей Guardant Code / Code Time, отличается от файла маски GrdUtil по умолчанию (**default.nsd**) и имеет следующую структуру, одинаковую для всех примеров:

1. Алгоритм #0 AES128
2. Алгоритм #01 SHA256
3. Алгоритм #02 AES128, зависит от ID ключа
4. Защищенная ячейка #03 – только для чтения
5. Защищенная ячейка #04 – для чтения и записи
6. Алгоритм #05 AES128 Demo
7. Алгоритм #06 SHA256 Demo
8. Защищенная ячейка #07 – Таблица лицензий
9. Алгоритм #08 – ECC160
10. Алгоритм #09 Загружаемый код № Занимает 4 сектора. В него по очереди загружаются все примеры

По умолчанию файл маски **Mask.nsd** находится в директории **%Program Files%\ Guardant\Guardant 5\%Общий код%\ Samples\ARM**.

### Важная информация

Для корректной работы примеров Guardant Code необходимо записать **Mask.nsd** в ключ при помощи GrdUtil.

## Краткая характеристика примеров

### Важная информация

Для корректной работы примеров Guardant Code необходимо записать **Mask.nsd** в ключ при помощи GrdUtil.

0. Шаблон проекта загружаемого кода. На его основе можно разрабатывать собственные модули.
1. Функция `main()`, возвращающая определенный код (демонстрация передачи параметров `dwRet` и `dwP1`).
2. Демонстрация использования буферов ввода-вывода.
3. Демонстрация работы необнуляемого буфера (внутренняя память). В одном вызове можно записать данные, а в следующем – считать обратно.
4. Разделение загружаемого приложения на несколько логических частей (обработка команд).
5. Управление индикатором. Демонстрация работы функций `GcaLedOn()/GcaLedOff()`.
6. Создание интерфейса к EEPROM. Обеспечивает чтение/запись участка в 512 байт EEPROM блоками по 16 байт. Блоки шифруются на AES-128 в режиме ECB, на ключе, хранящемся внутри загружаемого приложения.
7. Демонстрация работы с защищенными ячейками. Проход по аппаратным алгоритмам и получение информации о них (сервисы времени, чтения, модификации, счетчик запусков).
8. Модификация содержимого заданной защищенной ячейки.
9. Демонстрация функций работы с таймером (только для Guardant Code Time). Получение текущего времени и валидности RTC. Функция установки времени отсутствует. Единственный способ установить время в электронном ключе – при помощи GrdUtil.
10. Получение времени жизни защищенной ячейки.
11. Функции криптографии. Демонстрация генерации ключевой пары, подписи и проверки этой подписи по ECC160 с возвратом всех буферов в PC.
12. Использование аппаратного алгоритма для подписи.
13. Подсчет хэша.
14. Возврат блока случайных данных.
15. Отличие форматов хранения чисел с плавающей точкой в GCC ARM и PC.
16. Демонстрация использования встроенных математических функций.

17. Пример реализации вычислительной задачи (вычисление заданной точки на кривой Безье порядка N, параметры кривой хранятся внутри загружаемой микропрограммы).
18. Реализация целочисленной задачи. Шифрование на алгоритме Blowfish для блока данных на заданном ключе.
19. Демонстрация использования отладочной библиотеки.  
Для запуска примера необходимо запустить проект **Demo.sln**  
Пример демонстрирует, как можно использовать отладочную DLL в проектах. В подкаталоге **LoadableCode** содержится проект загружаемого модуля, который собирается в виде самостоятельного модуля, так и в составе проекта **Demo.sln**.  
По умолчанию в проекте задано использование отладочной библиотеки, при этом код из файла **lc.c** собирается в составе VS-проекта. Если убрать опцию **DEBUG\_DLL** и прошить в ключ маску из подпапки **LoadableCode**, код из файла **lc.c** будет выполняться уже внутри ключа. Как видно, в **lc.c** допустимо использование вызовов Guardant Code API.  
Однако в случае исполнения в составе VS-проекта налагается ограничение на функцию GsaExit(), и она не вызовет выход из кода в **lc.c**.

## Характеристики и вычислительные возможности Guardant Code

1. Быстродействие 60 MIPS 32-битных операций
2. 20 кб ОЗУ для программы пользователя
3. Около 128 кб (с возможностью расширения до 352 кб) для программы пользователя (32768 команд или около 15000 строк на Си для варианта 128кб и 98304 команды или около 50000 строк на Си для варианта 384 кб)
4. Приблизительная скорость операций с плавающей точкой:

Операция	Библиотека RealView		Библиотека GCC
	Float	Double	Double
Время выполнения			
+	0.8 мкс	1.3 мкс	2 мкс
*	0.9 мкс	2.2 мкс	2 мкс
/	1.7 мкс	5.7 мкс	11 мкс
Sin	89 мкс	95 мкс	30 мкс
Sqrt	4 мкс	3.2 мкс	2.7 мкс