Исходный текст (английский): Google
APPLICATION NOTE
Предложить лучший вариант перевода

ПРИМЕНЕНИЕ

Atmel AT02333: безопасный и безопасный загрузчик
Реализация для SAM3 / 4

32-разрядный микроконтроллер Atmel

Особенности

- Знакомство с концепцией модернизации в полевых условиях и загрузчиком
- Обсуждение проектных соображений при разработке безопасной и надежной загрузчик
- Понимание того, как загрузчик для SAM3 / 4 работы

Описание

В этом документе представлена модернизация прошивки в полевых условиях и описаны различные аспекты Реализации безопасного и безопасного загрузчика. В нем обсуждается несколько проектов Соображения в разработке такого программного обеспечения, о котором могут обратиться читатели, когда Консультируя этот документ.

Правильная реализация загрузчика создает несколько проблем, например, правильно Переназначение памяти чипа с помощью новой загруженной программы. Эти проблемы будут описано и решена в следующих разделах, с акцентом на Atmel ® ARM ® Cortex ® -M Основанный микроконтроллер.

Также представляется простейшее программное обеспечение загрузчика вместе с этим Приложение приложения как пример.

42141A-SAM-06/2013

Содержание

1. Введение

Микроконтроллеры все чаще используются в различных электронных продуктах. Устройства становятся более гибкими,
Благодаря перепрограммируемой памяти (обычно вспышке), часто используемой для хранения прошивки продукта. Это значит, что
Устройство, которое было продано, все еще может быть обновлено в поле, например, для исправления ошибок или добавления новых функци

Рисунок 1-1 иллюстрирует эту концепцию.

Рисунок 1-1. Обновление в полевых условиях

1. Производитель разрабатывает устройство и первоначальную прошивку
2. Устройства продаются клиентам
3. Производитель разрабатывает новую версию прошивки
4. Новая прошивка распространяется клиентам
5. Клиенты залатать свои устройства с новой прошивкой

Page 5

2. Обновление в полевых условиях

2,1 Загрузочный загрузчик

Многие современные микроконтроллеры используют флэш-память для хранения своего кода приложения.
Основным преимуществом Flash является то, что
Память может быть изменена самим программным обеспечением. Это ключ к программированию в полевых условиях: небольшая часть кода
Добавлено в основное приложение, чтобы обеспечить возможность загрузки обновлений, заменив старую прошивку устройства. Эта
Код часто называют загрузчиком, поскольку его роль заключается в загрузке новой программы при загрузке.

Загрузочный загрузчик всегда находится в памяти, чтобы обеспечить возможность обновления устройства в любое время.
Поэтому он должен
Быть как можно меньше. Поскольку никто не хочет тратить большой объем памяти на кусок кода, который делает
Не добавляйте никаких прямых функций для пользователя.

Рисунок 2-1. Организация памяти с загрузчиком

Чтобы загрузить новую прошивку на устройство, должен быть способ сообщить загрузчику подготовиться к передаче.
Существует два типа условий запуска: аппаратное и программное обеспечение. Условие оборудования может быть нажатой кнопкой
Во время сброса, тогда как условием программного обеспечения может быть отсутствие действительного приложения в системе.
Когда система
Запускается, загрузчик проверяет предопределенные условия. Если одно из них верно, оно попытается подключиться к хосту и подождать
Новая прошивка. Этот хост теоретически может быть любым устройством;
Однако стандартный ПК с соответствующим программным обеспечением
Чаще всего используется.
Передача прошивки может осуществляться с помощью любого носителя, поддерживаемого целевым устройством, то есть RS232,
USB, CAN и т. Д.

Page 6

Рисунок 2-2. Обновление прошивки с помощью загрузчика

1. Производитель выпускает новую версию прошивки
2. Новая прошивка распространяется для пользователей
3. Состояние загрузки инициируется клиентом
4. Загрузчик подключается к принимающей
5. Хост посылает новую прошивку
6. Существующие приложения заменяется
7. Новое приложение запускается

Как только передача завершена, загрузчик заменяет старую прошивку новой версией. Это новое приложение
загружен.

Существует несколько других способов осуществления внутрипрограммного программирования продукта.
Например, основное приложение может
Сделайте свое обновление для устройства, использующего внешнюю память, новая прошивка может быть записана на нем как файл.
Основное преимущество использования загрузчика заключается в том, что вам не нужно разрабатывать приложение по-другому.
Следовательно,
В то время как изменение вашего приложения для включения механизма обновления может быть утомительным, всегда можно использовать з
Без дополнительного программирования (при условии, что загрузчик легко доступен, конечно).

2,2 вопросы

Существует несколько проблем, связанных с использованием такого простого загрузчика. Проблемы могут возникать в двух точках
Модернизация потока: либо во время транспортировки прошивки от производителя до клиента, либо во время
Загрузить на целевое устройство. Рисунок 2-3 представляет собой диаграмму , показывающую несколько вопросов , которые будут
обсуждаться в следующем
разделы.

Рисунок 2-3. Обновление проблем с потоком

2.2.1 Безопасность

Для большинства устройств очень важно иметь встроенную в них встроенную прошивку, поскольку они, вероятно, не могут Нормально функционирует или даже не может функционировать вообще без него.
Однако использование загрузчика может привести к возникновению проблемных Ситуация, когда новая прошивка не была установлена должным образом, что ухудшает поведение системы.
Следующие проблемы (в Рисунок 2-3) может случиться:

• Ошибка № 1, Ошибка передачи. В результате часть кода повреждена.

• Проблема №2, Ошибка передачи: В результате область приложения будет повреждена и непригодна для использования. Эта проблема возникает:

- Если устройство внезапно теряет питание во время процесса обновления.

- Если соединение с хостом потеряно во время передачи.

• Проблема № 3: Потеря информации: некоторые данные могут быть потеряны при передаче прошивки, которая полностью повреждена Код после недостающей части.

2.2.2 Безопасность

Обеспечение безопасности системы означает соблюдение нескольких функций: **конфиденциальность, целостность** и **подлинность.**

Конфиденциальность означает , что часть данных не может быть прочитана неавторизованных пользователей или устройств. Основная проблема прошивки
Разработчики должны убедиться, что приложение, которое они разработали, не может быть пропущено конкурентами.
Таким образом, они хотят
Код должен оставаться закрытым, причем целевыми устройствами являются только авторизованные «пользователи».

Микроконтроллеры обычно предоставляют механизм, позволяющий злоумышленникам читать программный код Записанных на устройстве. Однако для обновления встроенного ПО в полевых условиях производитель должен предоставить новый код Чтобы они могли сами исправлять свои устройства.

Это означает, что квалифицированный специалист может потенциально декомпилировать его для извлечения исходного кода (номер 7 в Рисунок 2-3) .

Подлинность позволяет убедиться , что прошивка от самого производителя, а не кто - либо другой. В самом деле, Другая проблема перепрограммирования заключается в том, что устройству может быть предоставлена прошивка, которая не разработана ори Производителя, но третьей стороной (выпуск № 5 в Рисунок 2-3) . Это может быть особенно проблематично, если эта прошивка Разработанный для вредоносного использования, т. Е. Для обхода защиты безопасности, для незаконного использования критических функци вперед.

Подлинная прошивка может также использоваться на другом устройстве, отличном от того, на котором оно предназначено (номер № 4 в Рисунок 2-3) . Это может быть несанкционированная аппаратная копия продукта или устройство, предназначенное для взлома. Это Опять же проблема подлинности, на этот раз относительно целевого устройства.

И, наконец, **целостность** требуется , чтобы обнаружить модификацию данных. Например, авторизованная прошивка может быть немного Изменен (выпуск № 6 в Рисунок 2-3) . Это казалось бы подлинным, но атаки, подобные тем, которые описаны в Предыдущий пункт мог быть достигнут таким образом.

Возможный список проблем безопасности:

• Проблема № 4, Использование прошивки на неавторизованном устройстве.

• Проблема № 5, Использование несанкционированной прошивки.

• Проблема №6, модификация встроенного ПО.

• Проблема № 7, обратная инженерия прошивки.

Без какой-либо функции безопасности прошивка будет подвержена всем атакам, касающимся конфиденциальности, целостности и подлинно Поэтому для обеспечения соблюдения этих трех аспектов необходимы некоторые методы.

2,3 Возможные решения

В следующих двух разделах предлагается практическое решение проблем, указанных в предыдущих разделах. Однако большая часть Методы обхода этих проблем представляют собой компромисс между уровнем безопасности и безопасности, а также Скорость системы. Таким образом, самое безопасное и безопасное решение также, вероятно, самое большое и самое медленное. Эта Означает, что сначала необходимо тщательно проанализировать, что необходимо с точки зрения безопасности и безопасности в системе, реал Только требуемые функции.

Несколько методов обеспечения безопасности и охраны представлены в следующих разделах. Обратите внимание, что нет Программное решение может обеспечить отличную безопасность.
Действительно, существует множество аппаратных атак (таких как микрозондирование, мощность Анализ, анализ времени и т. Д.), Которые позволяют злоумышленнику нарушать защиту программного обеспечения. Эти атаки лучше всего Решаются с помощью выделенного защищенного чипа.
Однако использование мягкой защиты не является неуместным, поскольку они делают его более (Как во времени, так и в деньгах), чтобы атаковать вашу систему.

2,4 Решения безопасности

Следующие методы - способы предотвращения ошибок, связанных с безопасностью. Однако интересно отметить Что, поскольку загрузчик никогда не должен подвергаться риску (поскольку он не может быть обновлен), пользователь может просто попробс Его устройство снова, если есть сбой. Естественно, это не всегда может быть желательной альтернативой, поэтому следующие Решения.

2.4.1. Протокол протокола связи

Стек протоколов используется в большинстве стандартов связи, чтобы предложить среди других функций надежные передачи. Эта Надежность важна для загрузчика, поскольку прошивка не должна быть повреждена во время загрузки (см. Вопрос № 2 и № 3 в Рисунок 2-3).

В стандартной модели OSI система связи делится на семь слоев, которые образуют стек протоколов. каждый Слой отвечает за предоставление набора функций. Например, физический уровень отвечает за физическое Взаимосвязь устройств. Надежность обычно реализуется на транспортном уровне.

Надежность транспорта обычно достигается с использованием нескольких методов: кода обнаружения ошибок / коррекции, нумерации блоко И подтверждение приема пакетов. Они представлены ниже. Существующие протоколы для средств массовой информации , доступной на Atmel ARM Thumb ® 2 основе ATMSAM микроконтроллеры затем описаны в Раздел 2.4.1.4 ,

2.4.1.1 Обнаружение / исправление ошибок

Оно является общим для использования **обнаружения ошибок** и коды **коррекции ошибок** для передачи данных. Действительно, многие Поэтому было разработано несколько кодов, позволяющих обнаруживать и даже исправлять ошибки передачи.

Обнаружение кодов используют простые математические свойства для вычисления значения над данными , которое должно быть Сравнивает его с тем, который он дал. Если оба они равны, передача прошла успешно; В противном случае есть один или Более недействительные биты.

Корректирующие коды работают точно так же, за исключением того, что они могут обнаруживать ошибки, а также восстановить Это полезно, чтобы не потребовать от отправителя повторной передачи ошибочных данных.

Чтобы быть полезным, обнаружение / исправление ошибок не может выполняться на всей прошивке. Поскольку он записан в память Поскольку он передается, было бы бессмысленно обнаруживать ошибку только тогда, когда файл был полностью получен. Вместо этого Прошивка передается небольшими частями, называемыми кадрами. Код рассчитывается и проверяется для каждого кадра; Если ошибка Обнаружено, оно либо исправляется с помощью кода (если возможно), либо кадр повторно передается.

Рисунок 2-4. Обнаружение ошибок при передаче прошивки

Тем не менее, существуют ограничения на коды обнаружения / исправления ошибок. В зависимости от того, как они математически Они будут иметь максимальное количество обнаруживаемых / исправляемых ошибок. Таким образом, тщательный анализ Система должна быть выполнена до выбора метода для использования, чтобы избежать выбора несоответствующего кода.

Наконец, исправление ошибок в этом конкретном случае действительно не требуется. Это наиболее подходит, когда нецелесообразно повторно отправлять Поскольку коды исправления ошибок обычно требуют больших накладных расходов, чем Обнаружение ошибок, они должны быть реализованы с уважительными причинами.

2.4.1.2 Нумерация блоков

Цель нумерации блоков состоит в том, чтобы избежать потери блока данных или наличия двух блоков в неправильном порядке. Это Критически важная для переноса файлов, например, при загрузке прошивки: эти ошибки могут сделать полученный код непригодным.

Как следует из названия, нумерация блоков - это просто добавление порядкового номера в каждый переданный блок. Эта Число увеличивается на единицу для каждого блока. Поэтому приемник может легко обнаружить, что два блока были заменены, если Он получает блок № 3 перед блоком №2. Аналогично, если последовательность идет прямо от # 3 до # 5, то блок № 4 будет потерян.

Рисунок 2-5. Нумерация блоков

2.4.1.3 Подтверждение пакета

Подтверждение пакета работает следующим образом. Каждый раз, когда отправитель передает блок данных, он ожидает Приемник, чтобы подтвердить его (т. Е. Ответить, что он был правильно принят).
Время, отправитель предполагает, что пакет потерян, и повторно передает его.

Рисунок 2-6. Пакетные подтверждения

Никакие другие данные не отправляются эмиттером, пока он ожидает подтверждения. Поэтому пакеты не могут быть получены Не по порядку, так как только один отправляется одновременно.

2.4.1.4 Существующие протоколы

Средство связи (например, RS-232 или Ethernet) редко используется, как есть. Стек протокола чаще всего требуется для В полной мере использовать среду связи.

TCP / IP - наиболее широко используемый стек протоколов поверх Ethernet. Протокол управления транспортом (TCP) реализует Надежность с использованием номера последовательности пакетов, а также контрольной суммы (простой код обнаружения ошибок).
Вариант подтверждения пакета. Но так как сразу несколько пакетов могут быть отправлены, они могут выйти из строя (таким образом Нумерация блоков по-прежнему необходима).

Рисунок 2-7. Структура кадра TCP

В протоколе USB используется проверка циклического избыточности (CRC) для обнаружения ошибок. Номер последовательности на Пакетов, но приемник подтверждает каждый блок данных. Это то же самое для шины CAN.

Рисунок 2-8. Структура каркаса CAN

Наконец, для интерфейса RS-232 существует несколько протоколов связи, ориентированных на файловую систему.
Один из них - X-MODEM, который был разработан в 1970-х годах. Он содержит простую однобайтную контрольную сумму, нумерацию блоков и пакет подтверждение.

Рисунок 2-9. Структура кадра X-Modem

2.4.2 Разделение памяти

Основная идея раздела памяти - всегда иметь копию рабочей прошивки где-то в памяти.
Достижение этого означает, что даже если что-то пойдет не так во время обновления, все же можно вернуться к этому прошивка.

2.4.2.1 Память и память

Память, встроенная в микроконтроллер, обычно организуется в одном или нескольких банках (или самолетах), которые являются аппаратным зависимый. На практике часто используются один (единственный) банк или два (двойных) банка. Функция двойного банка позволяет Программирование одного банка, в то время как другое считывается (обычно при запуске кода приложения). Он также позволяет загружать Выбор кода путем сопоставления различных физических банков для загрузки области программы.

В следующих разделах описываются решения, основанные на однобайтовой или двойной банковской памяти.

2.4.2.2 Единое банковское разделение

Решение, представленное здесь, немного искажает эту технику. Память разделяется всегда в двух разных Регионов (исключая область загрузчика):

• Код приложения (регион A)

• Буфер для новой прошивки (область B)

Рисунок 2-10. Организация единой банковской памяти с разделением

Область B используется в качестве буфера. Новая прошивка загружается в этот регион и проверяется. Если прошивка загружена на Эта область проверена «ОК», она запрограммирована на область A, так что загрузка ошибок не повлияет на рабочую прошивку В регионе A. Этот метод гарантирует, что после обновления всегда будет работать прошивка на устройстве, независимо от того, было ли это Успешным или нет.

Для получения более подробной информации о рассмотрении и внедрении отдельных банковских разделов см. Раздел 3.1.3.1 ,

2.4.2.3 Двойное банковское разделение

Поскольку память уже выделена в банки, и они могут быть отображены в загрузочную область для загрузки в этом случае, Разделение просто использует преимущество организации: каждый банк имеет ту же самую копию загрузчика в начале А затем прошивка. При загрузке из одного банка другой банк используется как буфер обновления, чтобы принимать новые прошивка. После получения и проверки новой прошивки загрузочные банки переключаются.
Таким образом, могут быть две работоспособные Версии прошивки в памяти.
Чтобы загрузить другую версию прошивки, просто измените отображение загрузочной памяти, чтобы отобразить Банк, который содержит эту версию прошивки в качестве банка запуска, следующим образом:

• Сначала банк памяти 0 является загрузочным банком и прошивкой (v1) на нем.

• Область прошивки в банке 1 используется в качестве зоны буфера обновления, которая принимает недавно загруженную прошивку (v2).

• После загрузки прошивки (v2) в банк 1 и проверки ее «ОК», банк загрузки изменяется на банк 1, то есть банк 1
Сопоставляется с загрузочной областью для следующей загрузки системы.

• Теперь банк памяти 1 является загрузочным банком, а прошивка (v2) на нем будет запускаться при запуске.

• Когда система перезагружается, она запускает прошивку (v2) на bank1.

В этом случае область прошивки на банке 0 используется как обновление
Буферную зону, чтобы получить новую прошивку (старая прошивка не удаляется, а затем новая загруженная прошивка
Перезапишите его в этой зоне).

Преимущество этого метода заключается в том, что в памяти имеется максимум две рабочие версии прошивки, и нет
Необходимо выполнить дополнительное программирование (просто изменив загрузочное сопоставление вместо копирования прошивки из ре
Регион), но в каждом банке должен быть загрузчик, так что доступная память приложения немного меньше
Однобанговое решение.

Для получения более подробной информации о рассмотрении и внедрении разделов с двумя банками см. Раздел 3.1.3.2 ,

Рисунок 2-11. Организация с двойной банковской памятью с разделением

2.4.3 Резюме

Обнаружение и исправление ошибок (Раздел 2.4.1.1)

• Плюсы

- Обнаруживает ошибки передачи

• Против

- Кодекс должен выбираться с умом
- Немного увеличенный размер кода
- Немного уменьшенная скорость (только при обновлении)

Разделение памяти (Раздел 2.4.2)

• Плюсы

- Решает все проблемы, связанные с безопасностью

• Против

- Требуемый объем памяти удваивается
- Немного увеличенный размер кода
- Уменьшенная скорость выполнения (только при обновлении, для двойной банковской памяти, это не проблема, так как требуется код
Не копироваться)

2.5 Решения безопасности

Несколько связанных с безопасностью методов для решения вышеупомянутых проблем (см. Раздел 2.2.2) представлены в этом раздел. Видеть Раздел 2.6.2 для углубленной информации о вопросах безопасности.

2.5.1 Целостность

Проверка целостности означает проверку следующих параметров:

• Целенаправленное изменение прошивки

• Случайная модификация прошивки

Случайное изменение - проблема безопасности. Обычно он решается с помощью кодов обнаружения ошибок (см. Раздел 2.4.1.1).

Существует несколько способов проверить, что прошивка не была добровольно изменена озорным пользователем. Они есть
Представленные в следующих параграфах, и позволяют решить проблему № 6, описанную в Рисунок 2-3 ,

2.5.1.1 Функция хеширования

Концептуально цель хеш-функции - создать цифровой «отпечаток» части данных. Это значит, что,
Наоборот, к коду обнаружения ошибок, каждая часть данных должна иметь свой собственный уникальный отпечаток.

Чтобы проверить целостность прошивки, ее отпечаток рассчитывается и прикрепляется к файлу. Когда загрузчик получает
Как прошивку, так и ее отпечаток пальца, он повторно вычисляет отпечаток пальца и сравнивает его с оригинальными. Если они
Идентично, тогда прошивка не была изменена.

На практике хеш-функция принимает строку любой длины в качестве входных данных и производит вывод фиксированного размера, называе
дайджест. Он также имеет несколько важных свойств, например, хорошую диффузию (способность производить совершенно другой выход
Даже если перевернут только один бит ввода).

Рисунок 2-12. Прошивка

Поскольку выходная длина фиксирована (независимо от ввода), невозможно создать другой дайджест для каждой части
Данных. Однако хэш-функции гарантируют, что практически невозможно найти два разных сообщения, которые будут
Имеют тот же дайджест. Это достигает почти того же результата, что и уникальность, по крайней мере на практике.

Недостатком простого хэширования прошивки является то, что любой может это сделать. Это означает, что злоумышленник может
Файл и перечислить хэш. Таким образом, загрузчик не сможет сказать, что было сделано изменение.

Тем не менее, только хеш-функция все еще может использоваться для проверки целостности встроенного программного обеспечения во врем
заявление.

2.5.1.2 Цифровая подпись

Поскольку хэш можно легко пересчитать, решение должно зашифровать его. Это основа цифровой подписи: дайджест
Микропрограмма вычисляется (с использованием хэш-функции), а затем зашифровывается с использованием криптографии с открытым ключ
Это создает
Цифровая подпись, сродни подписям, используемым в повседневной жизни.

Шифрование с открытым ключом (или асимметричным) основывается на использовании двух ключей.
Производитель хранит этот свой секретный ключ (секрет) для
Зашифровать подпись, в то время как устройство использует соответствующий открытый ключ для его расшифровки.

Рисунок 2-13. Создание и проверка цифровой подписи

Поскольку только закрытый ключ может шифровать данные, никто, кроме производителя, не может произвести подпись. Таким образом,
Злоумышленник не сможет выполнить атаку, описанную в предыдущем разделе. Но кто-то может проверить
С использованием открытого ключа производителя.

2.5.1.3 Коды аутентификации сообщения

Коды аутентификации сообщений (MAC) предоставляют те же функции, что и цифровые подписи, за исключением того, что они используют
Секретный ключ криптография. Современные ключевые алгоритмы шифрования частных (также называемые шифры) являются в
основном **блочными шифрами** (то есть они работают на блоках данных фиксированного размера), в отличие от **потоковых шифров** (которые работают на потоке данных).

Рисунок 2-14. Проверка кода аутентификации сообщения

Шифрование Private Key использует только один секретный ключ, который совместно используется производителем и устройствами. Эта Имеет несколько последствий, по сравнению с цифровой подписью:

• Любой, кто может проверить MAC, может также произвести его.

• Если закрытый ключ внутри устройства открывается, система безопасности полностью скомпрометирована.

Первый момент не является проблемой на практике;
Поскольку устройство не будет использовать свой закрытый ключ для создания MAC-адресов, только для их проверки.
Вторая инструкция больше, что если злоумышленнику удается извлечь ключ из загрузчика (который должен
Быть заблокированным с использованием битов безопасности), то он сможет изменить прошивку и все еще принять ее как немодифицирован
цель. В зависимости от ваших требований это может быть или не быть проблемой.

Следует отметить, что поскольку криптография с секретным ключом намного быстрее, чем открытый ключ, MAC будет вычисляться и verified faster than a digital signature. But since only one MAC/signature is required for the firmware, it would probably not make a big difference in practice.

2.5.2 Authentication

Authentication is about verifying the identity of the sender and the receiver of a message. In the case of the bootloader, this means verifying that the firmware has been issued by the manufacturer, and that the target is a genuine one. Это solves issue #4 and #5 described in Figure 2-3 ,

It happens that the methods which provide authentication also provide integrity: **digital signatures** and **MACs** . поскольку they are described from the integrity point of view (see Section 2.5.1) , this section only discusses their authentication properties.

This section only discusses firmware authentication; authentication of the target device will be treated further.

2.5.2.1 Digital Signature

Only the manufacturer is supposed to possess the private key used to produce the signature attached to a firmware. This means that any valid signature (once decrypted using the corresponding public key) will certify that the signed data comes from the manufacturer and not from anyone else.

However, since the signature is freely decipherable by anyone possessing the public key (which is not supposed to be secret), the computed hash of the firmware can be obtained by anyone. This means that an attacker could find a collision in the hash function which is used, eg, two different texts giving the same hash. The signature would also authenticate this data as produced by the original sender.

This may not be a problem in practice however, as a collision is extremely hard to find and it is unlikely that it would result in a valid program. It would only enable a malicious user to create a fake firmware which would render the device unusable.

2.5.2.2 Message Authentication Code

Converse to a signature, a MAC cannot be used to certify that it is the sender who created the message. Indeed, since the receiver also has the private key used to compute the MAC, he may have generated it. The advantage is that only the two parties can decrypt the MAC, preventing anyone else from verifying that the message is indeed valid.

In practice, this does not create an issue as only the firmware would be MAC'ed. The bootloader would not use its private key to generate any other MAC, thus achieving the same authenticity verification as a digital signature.

The additional concern of a MAC compared to a digital signature is that an attacker should never be able to retrieve the private key inside the bootloader. If he manages to do that, he would be able to create or modify a firmware, issuing the associated MAC needed to authenticate it as a genuine one to any target.

2.5.3 Privacy

Data privacy is enforced using encryption: the data is processed using a cryptographic algorithm along with an encryption key, generating a cipher text which is different from the (plain) original one. Without the required decryption key, the data will look like complete nonsense, preventing anyone unauthorized from reading it. This takes care of issue #7, as described in Figure 2-3 ,

In practice, a private-key algorithm is used to generate the encrypted firmware. It is obvious that a public-key system cannot be used, as the firmware would then be decipherable by anyone. The encryption and decryption keys are thus identical and shared between the bootloader and the manufacturer.

Figure 2-15. Firmware Encryption

Note that code encryption does not solve every security issue all by itself. For example, the firmware might still be modified, even if it is quite difficult. An attacker could manage to pinpoint the location in the code of an important variable and tweak it until he gets the desired result.

Code encryption also combines itself well with a message authentication code. Since they both use a symmetric encryption algorithm, they can use the same one to save code size. There are also secure modes to combine both a block cipher and a MAC while using the same key (see Section 2.6.2.1) .

2.5.4 Target Device Authentication

There are two ways of verifying that a device is genuine. The first one is passive, ie, no special functionality is added to perform the verification. Instead, the authenticity of the device is implicitly checked by other security mechanisms.

In this particular case, encrypting the firmware will also authenticate the device. Indeed, it will need the private key to decrypt the firmware. As only genuine device shave them embedded in their bootloader, an unauthorized target will not be able to recover the original code and run the application.

This is especially applicable as target authentication cannot be achieved without encrypting the code anyway; otherwise, it could be simply downloaded to the device.
An active authentication method would involve adding an authentication technique for the target. Since the device identity would be verified during the upgrade process by the host, a message authentication code cannot be used. Indeed, since it would require the host to have the private key, an attacker could easily retrieve it.

Adding such a mechanism would also incur a significant overhead, both in terms of bootloader size (for storing the additional key and the digital signature encryption algorithm) and upgrade speed (because of the transactions needed to identify the device). In addition, the host upgrading program could be modified to get rid of that additional mechanism anyway.

2.5.5 Summary

Hash function (Section 2.5.1.1)

- Pros
  - Detects accidental and voluntary changes
  - Can be used to check firmware integrity at runtime
- Cons
  - Can be recomputed by a malicious user
  - Slightly increased code size
  - Slightly reduced execution speed

Digital signature (Section 2.5.2.1)

- Pros
  - Detects third-party and modified firmware
  - If the key inside the bootloader is compromised, the system remains safe
- Cons
  - Slower than a MAC
  - Requires a large key length
  - Increased code size
  - Reduced execution speed (during upgrade only)

Message authentication code (Section 2.5.1.3 and Section 2.5.2.2)

- Pros
  - Detects third-party and modified firmware
  - Faster than a digital signature
- Cons
  - If the key inside the bootloader is compromised, the system is broken
  - Increased code size
  - Slightly reduced execution speed (during upgrade only)

Code encryption (Section 2.5.3)

• Pros

– Prevents reverse-engineering

– Authenticates the target

• Cons

– If the key inside the bootloader is compromised, the system is broken

– Increased code size

– Reduced execution speed (during upgrade only)

2,6 Design Considerations

There are several choices and problems which arise when designing a bootloader such as the one described in this application note. This section gives an overview of the major topics.

2.6.1 Transmission Media

SAM microcontrollers provide a wide variety of peripherals to communicate with an external host, such as:

• USB

• CAN

• RS232

• Ethernet

• External memory (eg, DataFlash ®)

Choices should be made on the implementation priority (and relevance) of each method. Given the simplest one to implement is probably RS232; it could be used to get the system ready. Other interfaces could then be added in an easier way.

It should be noted that there is a USB device class geared toward firmware upgrading. This class, referred to as Device Firmware Upgrade (DFU), may be used to implement the bootloader functionality. However, please note that since it is not supported by Microsoft ® Windows ® , it may not be easy to do so.

Finally, media such as CAN or Ethernet have the potential to allow for batch programming, ie, programming several devices at once. The host could broadcast all the messages it sends, enabling every connected device to receive them and upgrade their firmware.

2.6.2 Cryptographic Algorithms

The secure part of the bootloader relies on different types of cryptographic primitives (hash functions, MACs, digital signature algorithms, block ciphers, etc.). But for every type of primitive, there are many different algorithms to choose from.

This section tries to give a brief overview of the choices available for the following: symmetric block ciphers, hash functions, message authentication codes, digital signature algorithms and pseudorandom number generators (PRNG). While the latter has not been introduced before because it is not a security method itself, it is critical to the design of a secure system.

For further recommendations, you may also look at those made by committees such as CRYP-TREC or NESSIE, which carefully analyze existing and new algorithms.

2.6.2.1 Symmetric Ciphers

Symmetric (or private key) ciphers are used both for computing MACs and encrypting the program code. Thus, they are an important part of the bootloader security and must be chosen wisely.

A symmetric encryption algorithm can be defined and chosen by several characteristics:

• **Key length in bits** : The larger it is, the more difficult it is to perform a brute force attack. A reasonable length seems to be128 bits at the moment, as it is the one key length selected for the Advanced Encryption Standard (AES) cipher.

• **Block length in bits** : is needed to avoid a “codebook attack”. Most block ciphers will use at least 64 bits, with modern ciphers using at least 128 bits.

• **Security** : If the algorithm has flaws, then it may be easily breakable.

• Size & speed: depend on the techniques they use. Since embedded system has limits in resources, more suitable encryption should be chosen.

Table 2-1 gives an overview of several popular ciphers. Note that if the target platform has hardware acceleration available, the resulting code size will be much smaller and the system will be much faster.

Table 2-1. Symmetric Encryption Algorithms

Алгоритм	Key Length	Block Length	Безопасность	Speed & Size
AES	128 to 256 bits	128 bits	Безопасный	Fast, small code, small RAM footprint
Blowfish	32 to 448 bits	64 bits	Безопасный	Fast, large RAM footprint
DES	56 bits	64 bits	Broken	Медленный
Triple-DES	168 bits	64 bits	Безопасный	Very Slow
RC6	128 to 256 bits	128 bits	Безопасный	Small code ,large RAM footprint
Serpent	128 to 256 bits	128 bits	Безопасный	Slow, small code, small RAM footprint
Twofish	128 to 256 bits	128 bits	Безопасный	Small RAM function

Note that block ciphers can also be used (with modifications) as hash functions and message authentication codes. This can be useful to save code size when several primitives are needed (by reusing the same algorithm more than once).

Figure 2-16. Block Cipher as Hash Function

Several modes of operations are possible when using a symmetric cipher:

- **Electronic codebook** (ECB): The basic mode of operation, each block of plain text is encrypted using the key and the chosen algorithm, resulting in a block of cipher text. However, this mode is very insecure, as it does not hide узоры.
- **Cipher block chaining** (CBC, CFB, OFB, CTR ...): Encryption is not only done with the current block of plain text, but also with the last encrypted block. Make everything interdependent.

- **Authenticated encryptions** (EAX, CCM, OCB ...): are used to provide privacy, integrity and authentication at once. They are basically the combination of a MAC algorithm and a symmetric block cipher. They are useful when the three components are needed, as using a mode such as EAX will be faster and has less overhead than applying a MAC and a symmetric cipher separately.

The first block is encrypted using a random **Initialization Vector** (IV). While this vector can be transmitted in clear text, the same vector shall never be used more than once with the same key. It is likely that a manufacturer will produce more than one firmware upgrade for a product in its lifetime. This means that the IV cannot be stored in the chip in the same way the key is. Therefore, it will have to be transmitted by the host.

Figure 2-17. CBC Mode of Operation

Figure 2-18. EAX Mode of Operation

2.6.2.2 Hash Functions

A hash function has three defining characteristics:

- **Output length** : needs to be large enough to make it almost impossible to find collisions.
- **Security** : is much more critical than the length of its output. Indeed, MD5 (which only has a 128-bit output) would still be secure if it did not have serious design flaws in it.
- **Size & speed** : the stronger algorithms are often the slowest ones (which are not true for block ciphers), so there will be a security/speed trade-off

Table 2-2. Hash Algorithms

Алгоритм	Block Length	Безопасность	Speed & Size
MD5	128 bits	Broken	Быстро
RIPEMD-160	160 bits	Безопасный	Медленный
SHA-1	160 bits	Broken	Медленный
SHA-256	256 bits	Безопасный	Медленный
WHIRLPOOL	512 bits	Безопасный	Very Slow
Tiger	192 bits	Безопасный	Быстро
HAVAL	128 to 256 bits	Broken	Moderately fast

2.6.2.3 Message Authentication Codes

MACs are constructed by using other cryptographic primitives. Therefore, the choice of which type of MAC to use is mostly dictated by which algorithms are used by other functionalities. Of course, some MAC algorithms have been found faulty; care should be taken to avoid them.

Here are the different types of (secure) MACs available:

- HMAC: a hash function along with a private key
- UMAC: many hash functions and a block cipher
- OMAC/CMAC: block cipher in CBC mode
- PMAC: block cipher in CBC mode

Note that UMAC might not be usable in practice, as it requires many different hash algorithms. The incurred size overhead would thus be far too important for a bootloader

Figure 2-19. CMAC Message Authentication Code

2.6.2.4 Digital Signature Algorithms

There are basically two main systems for generating and verifying digital signatures:

- The Digital Signature Standard (DSS): specifically designed for digital signatures. It is based on a public-key algorithm known as the ElGamal scheme. The key length required to have a strong enough security is at least 1024 bits.
- A system based on the Rivest-Shamir-Adleman (RSA) public-key algorithm: the most popular method, used with a padding scheme (to append data to the message to encrypt). There are three commonly used schemes:
  – Full-domain hashing: involves using a hash function that has an output size equal to the RSA key length.

– Optimal Asymmetric Encryption Padding (OEAP): adding a quantity of random data to convert RSA into a probabilistic encryption scheme.

– Probabilistic Signature Scheme (PSS): adding a quantity of random data to convert RSA into a probabilistic encryption scheme.

2.6.2.5 Pseudorandom Number Generators (PRNG)

There are many things in a secure system which are “random” or need some kind of randomized value. Secret keys are the most basic example. Thus there must be a method to generate those random values in a secure way, to avoid weakening the whole system. A chain is as strong as its weakest link, so even indirect security issues should not be overlooked.

However, note that a PRNG is not needed by the target. It is only used on the manufacturer side, for the following operations:

• Generating private keys

• Generating initialization vectors

• Padding data when using RSA/OEAP or RSA/PSS

This means that in practice, there is neither real speed nor size constraint on the PRNG algorithm. Only its security matters.

PRNGs work by using a starting seed to generate successive random values. Initializing that seed is a core problem, which is referred to as gathering entropy. Consider the case where the current date & time are used to seed the PRNG. An attacker could obtain that information and thus reconstruct every random number generated using that seed: private keys, nonce, etc.

Operating systems usually provide a mechanism to provide entropy, eg, /dev/random on UNIX systems. They use, for example, the response time of devices such as hard disks to gather the required entropy.

Most PRNGs then rely on another cryptographic primitive (such as a block cipher or a hash function) to generate pseudo-random outputs. Here is a list of several secure PRNGs:

• Any block cipher in CTR mode

• Yarrow

• Fortuna

• Blum-Blum-Shub random number generator

2.6.2.6 Available Libraries

This section lists web sites providing libraries and/or reference implementations of several cryptographic primitives described in this document. Those are all open-source or freely available implementations.

• libTomCrypt (http://libtom.org)

• OpenSSL crypto (http://www.openssl.org)

• Crypto++ (http://www.cryptopp.com)

• Brian Gladman (http://www.gladman.me.uk)

• Libmcrypt (http://mcrypt.hellug.gr/#_libmcrypt)

2.6.2.7 Performances on an SAM3X8 Chip

Several aforementioned ciphers and modes have been tested using different implementations on a SAM3X8. Эта section presents the results obtained when compiled by EWARM 5.50.

The AES cipher has been tested using different implementations. The first one uses the libTomCrypt library, freely available from http://libtom.org , The second one is based on a standard implementation provided by Paulo Baretto and Vincent Rijmen.

Table 2-3. Performance of AES (128-bit key and 128-bit blocks)

шифрование Режим	Реализация Источник	Size Overhead (bytes)	Decryption Time for a 128KB File (ms)
ECB	libTomCrypt	8576	680.0
	Справка	implement	2632
CBC	libTomCrypt	8660	734.7
	Справка	implement	2708
	libTomCrypt	8820	752.7

CTR	Справка	implement	2096

Table 2-4. Performance of Triple-DES (168-bit key and 64-bit blocks)

шифрование Режим	Реализация Источник	Size Overhead (bytes)	Decryption Time for a 128KB File (ms)
ECB	libTomCrypt	4742	2006.9
CBC	libTomCrypt	4862	2065.3
CTR	libTomCrypt	5022	2085.3

2.6.3 Error Detection Codes

Since SAM microcontrollers are based on a 32-bit architecture, it seems logical to implement codes which are at least as long (as 32-bit). They will not cost more in terms of speed and size than an 8-bit or 16-bit variant.

Originally, simple checksums were used to detect errors. They operate by simply adding all the bytes in a piece of data to get a final value. However, they are very limited: they cannot, for example, detect that null bytes (0x00) have been appended or deleted. More reliable techniques are now available, so very simple checksums should be avoided.

There are two algorithms which are worth mentioning here. The first one is the well-known Cyclic Redundancy Check (CRC), which has strong mathematical properties and is quite fast. The 32-bit version, called CRC-32, is used in the IEEE ® 802.3 specification.

Adler-32 is an algorithm that is slightly less reliable but significantly faster than CRC-32. It has a weakness for very short messages (< 100 bytes), but this is not a concern if a whole page of data (≥256 bytes) is transmitted at one time.

2.6.4 Firmware File Format

Compilers support a wide variety of output file formats. The most basic of them is the binary format (.bin): it is simply a binary image of the firmware. Since no information apart from the application code is required, the firmware can simply be transmitted in binary format and directly written to memory by the bootloader. Since using other formats would mean adding the necessary code on the bootloader side to handle them, this may not be worth it.

2.6.5 Target Chips

Some of the chips in the SAM family have different IPs, such as the Flash controller. This means that they are programmed differently; therefore, several versions of the code must be written to accommodate all the microcontrollers.

Thus, the bootloader will be developed for a particular chip first, but in a modular way. This means that functions which are chip-dependent are wrapped in an abstraction layer. Porting the software to another chip is easy: only the necessary low-level functions have to be coded, without touching the bootloader core.

The following implementation example will use SAM3X8 as candidate, since it has no cryptographic accelerators, a software solution with libTomCrypt or Reference design is used to encrypt.

3. Bootloader Implementation

This section details several issues that one may encounter while implementing a safe & secure bootloader, along with some insight on how to approach them. Example code from a working implementation is given in the following sections to illustrate the solutions.

3,1 Bootloader Flow

3.1.1 Boot Sequence

The startup sequence of the bootloader is as follows:

- Initialization
- Trigger condition check
  - Firmware upgrade (if trigger condition is offered)
  - Firmware selection (if trigger condition is offered, for dual banked device)
- Firmware verification (optional)
- Firmware loading (if verification is ok or disabled)

The following is a simple bootloader example that supports only Firmware upgrade:

Figure 3-1. Basic Boot Sequence Diagram Example

Here is the corresponding code in C:

```
// Bootloader initialization
trigger_init();
memory_init();
media_init();
communication_init();
encryption_init();
// Check trigger condition
if (trigger_poll()) {
        // Upgrade firmware
        bootloader_load(APP_START_ADDRESS);
}
// Verify firmware
if (integrity_check() != OK) {
//
}
// Execute the firmware
binary_exec(APP_START_ADDRESS);
```

The function *binary_exec()* uses the application start address to start the firmware.

Page 26

3.1.2 Upgrade Sequence

The basic upgrade flow starts with the host sending the firmware to the target, which then programs it in memory. однажды the programming is done, the new application is loaded.

While, in theory, the “downloading” and “programming” steps are different, this is not the case in practice. Indeed, SAM microcontrollers usually have much more Flash memory than RAM. Thus, they cannot store the whole firmware in RAM before writing it permanently to the Flash.

This means that the code must be written to the memory while it is received, but not after. Since a Flash write operation takes approximately 6 ms (with a page erase), a communication protocol is needed to halt the transfer when the memory is being written and resume it afterwards.

Note also that the Flash memory is split up into fixed-size blocks called pages. A memory write operation can upgrade one page (or less) at a time; thus it seems logical to send packets containing one full page.

Finally, there are several optional post-processing features to take into account. If code encryption is activated, then each page must be decrypted before being written. If a digital signature or a message authentication code is present, it must be verified once the download is completed.

Figure 3-2. Basic Firmware Upgrading Process Example

Here are some sample C codes implementing the upgrade flow:

```
// Receive and write firmware
pCurrent = APPLICATION_STARTING_ADDRESS;
do {
        bytes = communication_receive(page);
        // If at least one byte is received, write the page
        if (bytes > 0) {
                // Decrypt firmware
                encryption_decrypt(page, page, bytes);
                // Pad page data
                while (bytes < MEMORY_PAGE_SIZE) {
                        page[bytes] = 0xFF;
                        bytes++;
                }
                // Write page
                memory_write(pCurrent, page);
                pCurrent += MEMORY_PAGE_SIZE;
        }
} while (bytes > 0);
```

In this example, the *communication_receive()* function waits for a whole page of data while detecting the transfer end. returned value “0” means that the transfer is finished.

3.1.3 Memory Partitioning

As described in Section 2.4.2 , using memory partitioning makes it possible to have at least one working version of the firmware in the device at anytime. This is useful to avoid firmware corruption if a problem occurs during an upgrade, such as a power loss or a connection loss.

Page 27

The following sections will describe the implements for single and dual banked memory.

3.1.3.1 Partitioning on Single Banked Memory

As described in Section 2.4.2.2 and Figure 2-10 , the whole Flash memory is divided into two distinct regions, A and B (excluding the bootloader region). The first one, A, is located right above the bootloader and contains the code which is to be loaded by the bootloader. B acts as a buffer during an upgrade: the code is downloaded to that region, verified, and finally copied to the first region if valid.

The boot and upgrade sequences are thus slightly modified. During the upgrade, the code is written to region B (the buffer zone). Several steps are then added to the boot sequence, regardless of whether or not a firmware upgrade has been requested. The bootloader first checks if the two codes present in regions A and B are identical (via code compare,

hash code compare or other ways). If they are, then the code in region A is loaded.
If two codes are not the same, this means that either an upgrade has just taken place, or there has been an error during a previous upgrade. The validity of the code in region B is verified. If it is indeed valid, then it is copied over region A. If not, it is deleted.

Memory partitioning also makes it possible to use a slightly different bootloading strategy. Since there is always a working firmware embedded in the device, some functionalities of the bootloader can be moved to the user application. By doing this they can be upgraded as well.

For example, the firmware transfer can be delegated to the user firmware. It simply downloads the new code in the buffer region and resets the chip. The bootloader then performs the remaining operations, ie, verify region B and copy it over region A.

Figure 3-3. Boot Sequence with Single Banked Memory Partitioning

Figure 3-4. Single Banked Memory Content during Upgrade Process

3.1.3.2 Partitioning on Dual Banked Memory

As there are two memory banks, more choices are available on memory partitioning:

- Choice #1, manage the whole memory as a single space, and accesses to different memory banks are translated to continuous memory address. In this way, the whole memory is looked as a single banked memory to partition. Для more details, please refer to Section 3.1.3.1 and Section 3.2.1 ,
- Choice #2, as described in Section 2.4.2.3 , manages the Flash by banks, and each bank has its bootloader and firmware region. Here the two banks have different physical addresses, but they are mapped to the same boot program address on boot time, so binary should use a link file that links to boot program address space. Когда boot memory layout is changed, different firmware version is chosen.

The first choice can apply if the two memory banks should be in a continuous address (such as SAM3X, SAM3SD8, SAM4SD16 and SAM4SD32, the Flash address of two banks are continuous), or the banks are able to be mapped to a continuous address (such as SAM3U, the Flash address of two banks are not continuous, but the bank in Flash area is mapped to reserved space to access, see Figure 3-5) .

The link file should use this continuous address to generate the application firmware. For more information about Flash banks, please refer to datasheet for SAM3/4 chips.

Figure 3-5. Map Two Memory Banks to a Continuous Address

Then the second choice: since Flash memory has two banks, the two firmware regions A & B are placed at the same offset of these banks; they are remarked as boot region and buffer region. Both of them are located right above the bootloader, on its location bank and contain the firmware code. The boot region A is loaded by bootloader on that bank when system starts up. The buffer region, B, is a buffer on upgrading: the code is downloaded to that region and verified. Finally, the roles of these regions can switch, by selecting different boot bank. That is, when booting from bank 0, the firmware region on it acts as boot region (A) and the firmware region on bank 1 acts as buffer region (B); when booting from bank 1, the firmware region on bank 0 acts as buffer region (B) and the firmware region on bank 1 acts as boot region (A).

In this case, when performing upgrade, the firmware is always loaded to buffer region (B) and after downloaded data in buffer region (B) is verified “OK”, the boot bank is switched and new firmware will be loaded when system restarts. Как result, the old firmware is still kept in buffer region but not used, so the two regions both contain workable firmware and the latest downloaded one is to be loaded.

Figure 3-6. Boot Sequence with Dual Banked Memory

Figure 3-7. Dual Banked Memory Content during Upgrade Process

Note that in SAM3SD8, SAM4SD16 and SAM4SD32, there is a slight difference when changing boot banks. It swaps the physical banks! Eg, before swap, the old firmware is in address area 0 and the new firmware is in address area 1, while after swap, the new firmware will be in address area 0 and the old firmware in address area 1. This feature keeps the boot firmware in the same physical address, whether Flash banks have been swapped or not.

Page 30

Figure 3-8. Dual Banked Memory Content during Upgrade Process for SAM3SD8, SAM4SD16, SAM4SD32

3,2 Bootloader Programming on SAM3/4 Chips

In practice, for SAM3 and SAM4 chips, there are two types of flash controller:

- The Enhanced Embedded Flash Controller (EEFC): Used by most of SAM3 and SAM4 chips
- The FLASHCALW: Used by SAM4L

Here we take EEFC as example to show how internal flash of SAM3/4 chips are programmed. For operations that using FLASHCALW, please refer to SAM4L documents (can be found Вот).

3.2.1 Flash Programming

Usually the Flash of the SAM3 chip is organized in one or two planes (banks) that are made of pages of 256 bytes. SAM4 chip has the same organization while the page size is 512 bytes. The EEFC is used to manage the Flash access. The Fast Flash Programming Interface (FFPI) is used to program the device. Depending on the device, there might be one or two Flash Controllers, and the physical address of two Flashes (banks) can be continuous or not. Table 3-1 gathers the rough information. For more detailed information, please refer to chip's data sheet.

Заметка:

- The Flash memory of SAM microcontrollers cannot be read and written at the same time for one single bank case.
- For SAM3 the programming command can perform erase and program automatically.
- For SAM4, before writing the Flash memory, an erase operation must be performed,
- If flash erase block size are bigger than write block size (this is a usual situation for flash devices, eg, in SAM4 writing bases on one page but minimum erasing size is eight pages), the partition must be aligned with erase block size so that the unexpected area will not be erased.

Page 31

Table 3-1. Flash Organizations for SAM3/4 Chips

устройство	вспышка организация	Number of Flash Controllers	Flash (bank) Address is continuous	вспышка Controller
SAM3U2/1	Single Bank	1	-	
SAM3U4	Dual Bank	2	нет	
SAM3S4/2/1	Single Bank	1	-	
SAM3N Series	Single Bank	1	-	

SAM3X(A) Series	Dual Bank	2	да	
SAM3S8	Single Bank	1	-	EEFC
SAM3SD8	Dual Bank	1	да	
SAM4S16/8	Single Bank	1	-	
SAM4SA16	Single Bank	1	-	
SAM4SD32/16	Dual Bank	2	да	
SAM4E Series	Single Bank	1	-	
SAM4L4/2	Single Bank	1	-	FLASHCALW

3.2.1.1 Single Banked Flash Programming

Single banked Flash cannot be read and written at the same time since it is single plane. Therefore, a program executing from the Flash cannot perform a memory write. Since the bootloader is located inside the Flash, it must be copied to the RAM prior to execution.

A way is available for certain toolchains such as IAR Embedded Workbench ®. Using the *__ramfunc* attribute for a function will tell the compiler that it is supposed to run from the RAM, not the Flash. The C-initialization routine copies the function at runtime. Note that you must disable interrupts during a Flash write if this solution is used, since exception vectors will still be read from the Flash by the ARM core.

Example function performs a Flash command using the *__ramfunc* attribute:

```
__ramfunc void flash_cmd(Efc* pEfc, uint32_t dwCmd, uint32_t dwArg) {
        // Start Flash operation and wait for completion
        // ITs are masked during this.
        while (!(efc->EEFC_FSR & EEFC_FSR_FRDY));
        __disable__irq();
        efc->EEFC_FCR = EEFC_FCR_FKEY(0x5A)
                | EEFC_FCR_FARG(dwArg)
                | EEFC_FCR_FCMD(dwCmd)
        while (!(efc->EEFC_FSR & EEFC_FSR_FRDY));
        __enable__irq();
}
```

3.2.1.2 Dual Banked Flash Programming

Dual banked Flash enables Flash writing to another bank while code is running on one bank since it is in different Flash bank. So there are several different ways to write.

- Put function codes in bank 0 to perform writing in bank 1 and vice versa. When writing to a Flash bank, judge which function to use by the writing address.
- Use RAM functions, the same as single banked Flash programming in Section 3.2.1.1 .

3.2.1.3 Flash Programming via IAP Function

SAM3/4S/4E chips offer IAP function in ROM code to send the Flash command to EEFC and wait for the Flash to be ready. Since the code is executed from ROM, it allows Flash programming to be done while the code is still running in Flash, to any of the banks.

Example function performs a Flash command using the IAP function:

```
#define CHIP_FLASH_IAP_ADDRESS (IROM_ADDR + 8)
void flash_cmd(uint8_t ucFlashID, uint32_t dwCmd, uint32_t dwArg) {
        // Obtain Pointer on IAP function in ROM
        static uint32_t (*IAP_PerformCommand)( uint32_t, uint32_t ) ;
        IAP_PerformCommand = (uint32_t (*)( uint32_t, uint32_t ))
        *((uint32_t*)CHIP_FLASH_IAP_ADDRESS ) ;
        // Start Flash operation and wait for completion
        // ITs are masked during this.
        __disable__irq();
        IAP_PerformCommand( ucFlashID, EEFC_FCR_FKEY(0x5A)
                | EEFC_FCR_FARG(dwArg)
                | EEFC_FCR_FCMD(dwCmd) ) ;
        __enable__irq();
}
```

3.2.2 Vector Table Relocate & Application Execute

Atmel SAM3/4 chips are based on an ARM ® Cortex ® core. When the core boots, it always starts from address 0x0, where a vector table exists. The first word of vector table is used to initialize stack (SP) and the second word is used as a start entry (PC). The other words in this vector table offer exception vectors to the ARM ® Cortex ® core.

The vector table location can be changed by modifying the VTOR register to a different memory location, in the range from 0x00000080 to 0x3FFFFF80.

Before executing the firmware, the vector table must be set to the address of firmware vector table.

Then SP is loaded from the first word of vector table. After that, PC is loaded from the second word of vector table, to jump to the application.

Example code for vector table relocating and application executing:

```
// -- Disable interrupts
// Disable IRQ
__disable_irq();
// Disable IRQs
for (i = 0; i < 8; i ++) NVIC->ICER[i] = 0xFFFFFFFF;
// Clear pending IRQs
for (i = 0; i < 8; i ++) NVIC->ICPR[i] = 0xFFFFFFFF;
// -- Modify vector table location
// Barriars
__DSB();
__ISB();
// Change the vector table
SCB->VTOR = ((uint32_t)vStart & SCB_VTOR_TBLOFF_Msk);
// Barriars
__DSB();
__ISB();
// -- Enable interrupts
__enable_irq();
// -- Execute application
__asm ("mov r1, r0 \n"
       "ldr r0, [r1, #4] \n"
       "ldr sp, [r1] \n"
       "blx r0"
);
```


3.2.3 Boot Code Remapping

SAM3/SAM4S/SAM4E chips can boot from ROM or Flash. If there are two banks, each of the banks can be the boot bank. The boot selection is changed by modifying boot code mapping.

Eg, for SAM3X8, there are two 256-Kbyte Flash banks; the boot mapping is changed by configuring GPNVM bits 1 & 2. Table 3-2 and Figure 3-9 are table and diagram for reference:

Table 3-2. Boot Code Mapping for Dual Banked Device

GPNVM[1]	GPNVM[2]	Boot From
Clear	Clear/Set	ПЗУ
Задавать	Clear	Flash Bank0
Задавать	Задавать	Flash Bank1

Figure 3-9. Memory Organization Before and After Boot Code Selection

Page 34

Note that for SAM3SD8 and SAM4SD32/16, setting GPNVM bit 2 will swap the two Flash banks, so the memory mapping change is as follows:

Figure 3-10. Memory Organization Before and After Boot Code Selection SAM3SD8, SAM4SD32/16

3.2.4 Application Code Linking

The next sections focus on application code linking in different cases.

The example uses a linker script that is based on address 0x0, which satisfies both single banked and dual banked situation on SAM3X chips.

3.2.4.1 Single Banked Code Linking

Ordinarily, the code segment of a user application is linked at the beginning of the Flash. This is because the ARM core of SAM microcontrollers starts fetching program code at address 0x0, where the beginning of Flash is mapped. However, this cannot be the case when using a bootloader, since it will be located at the beginning of the Flash.

The solution is to modify the address at which the application is linked, depending on the reserved bootloader size. ĸ do that, you have to modify the linker script of your project. Simply define a “bootloader_size” constant which is equal to the size of the compiled bootloader, and the first available ROM address to “flash_start+bootloader_size”. Here is an example for SAM3X8 chip using IAR Embedded Workbench 5.50:

```
/*--- Bootloader size (32K) ---*/
define symbol __BOOTLOADER_size__ = 0x8000;
...
/*--- Code region 3X8 ---*/
define symbol __region_ROM_start__ = 0x00080000+__BOOTLOADER_size__;
define symbol __region_ROM_end__ = 0x00040000-1;
```


Page 35

```
...
```

Code segments are then defined between *__region_ROM_start__* and *__region_ROM_end_ _*.

Note that when single banked Flash is selected to boot, the whole Flash is mapped to address 0x0, so it's also possible to arrange bootloader and application based on this mirrored address. It also works for a dual banked Flash with

continuous bank addresses. But when a dual banked Flash without continuous bank addresses is used (as choice #1 described in Section 3.1.3.2), the linking map must be based on the virtual continuous Flash start address (see Figure 3-5) but not on this boot mirror, since the boot mirror might only include the single bank that is used for boot.

3.2.4.2 Dual Banked Code Linking

In the choice #2 case in Section 3.1.3.2 , the code segment should be placed on different physical bank, but arranged in the same mapping. Since the code will never exceed one bank size, and all banks can be mapped to address 0x0, which is used as a base to place bootloader and application, so the application linker script can be like the following:

```
/*--- Bootloader size (32K) ---*/
define symbol __BOOTLOADER_size__ = 0x8000;
...
/*--- Code region 3X8 ---*/
define symbol __region_ROM_start__ = 0x00000000+__BOOTLOADER_size__;
define symbol __region_ROM_end__ = 0x00040000-1;
...
```

The code segments start address is modified to place code based on 0x0 but not on physical Flash address.

Note that for some dual-bank chips such as SAM3SD8, SAM4SD32/16, thanks to its Flash banks swapping feature (Figure 3-8 and Figure 3-10 , two Flash banks are swapped, but Flash start address is always mapped to address 0x0, no matter whether it's banks are swapped or not), it's also possible to put code segments at Flash start address. В этом way, the code can be debugged directly in IAR Embedded Workbench.

3.2.5 Boot Region Locking

To avoid having the bootloader region accidentally erased by the user application, it is safe to lock it with the Flash контроллер. Any write operation to a locked Flash region will be aborted automatically.

A command is available in the Enhanced Embedded Flash Controller (EEFC) for locking a specified memory region. The problem is that the region is quite large: between 4K and 16K bytes. Since only the bootloader region must be locked, this means that the application will have to start at the beginning of the next region.

In practice, the users have to set the bootloader segment size to a multiple of a region size. This effectively means that you waste the region space which is not used by the bootloader.

Note that for dual bank, the bootloader region for both banks should be locked.

The security bit, used to protect the internal memory (Flash + SRAM) from external access, can also be set in the EEFC.

Page 36

4. Example Implementation

This section describes a sample implementation of the safe & secure bootloader presented in this document. It is made up of three programs: the bootloader itself and two program tools. One program tool is used to transfer the firmware between a PC and the bootloader. The other one is necessary to encrypt the firmware before sending it. The three pieces of software are detailed below.

4,1 Bootloader

It's general implementation for device with single banked Flash or dual banked Flash.

4.1.1 Features

The following features have been implemented:

- Basic bootloading capabilities using a USART to transmit the firmware
- XON/XOFF protocol for flow control during downloading/Flash programming
- Boot region locking
- Code encryption using AES or Triple DES
- Memory partitioning

The software has been developed for IAR Embedded Workbench 5.50.

4.1.2 Configurability

The bootloader has been designed in a way which makes it easy to add new components to it, like a new media or a

new communication protocol.
The /inc/config.h file controls the configuration of both mandatory components (such as which media to use) and optional ones (security, safety). A particular component can be selected by defining the following constant:

```
#define USE_COMPONENTNAME
```

The COMPONENTNAME is the name of the components (eg, USART0, UART, and BUTTONS for SAM3/4). Обратите внимание, что only one component can be selected for each mandatory category, which are:

- Communication protocol
  - XON/XOFF (*USE_XON_XOFF*)
- Media layer
  - USART (*USE_USART0*)
- Debug
  - DBGU (*USE_DBGU*)
  - UART (*USE_UART*)
- Memory type
  - Flash (*USE_Flash*)
- Trigger condition
  - Dummy (*USE_DUMMY*)
  - Switches (*USE_SWITCHES*)
  - Button (*USE_BUTTONS*)

- Timing measurement
  - Timer0 (*USE_TIMER0*)

Both the debug and timing categories are not truly mandatory. If no debug driver is defined, for example, then the bootloader does not output debug messages. The timing module can be used to perform benchmarks on several features.

The following optional parameters are available:

- Boot region locking (*USE_BOOT_REGION_LOCKING*)
- Code encryption (*USE_ENCRYPTION*)
- Memory partitioning (*USE_MEMORY_PARTITIONING*)

4.1.3 Code Location

4.1.3.1 Main Bootloader Algorithm

The /src/main.c file contains the core bootloader algorithm. This includes the boot sequence (Section 3.1.1), upgrade sequence (Section 3.1.2) as well as the modified boot & upgrade sequence for memory partitioning (Section 3.1.3) . Memory locking of the bootloader region is also done during initialization of this bootloader.

4.1.3.2 Vector Table Relocating & Application Execute

The code is located in /src/main.c, implemented in function *binary_exec()* .

4.1.3.3 Flash Programming

Algorithms for programming the Flash as well as locking/unlocking memory regions are located in the /src/media/flash.c файл.

4.1.3.4 Linking Code

All the files used for both bootloader and user application linking and startup are located in the /resources directory. Это includes:

- bootloader_sam3x8_flash_0.icf: startup and linker file for the bootloader without boot code remap (based on Flash beginning address)
- bootloader_sam3x8_flash_remap.icf: startup and linker file for the bootloader with boot code remap (based on address 0x0, the mapped address of Flash)
- firmware_sam3x8_flash_0.icf: startup and linker file for a firmware without remapped boot code (based on Flash beginning address)
- firmware_sam3x8_flash_remap.icf: startup and linker file for a firmware with remapped boot code (based on address 0x0, the mapped address of Flash)

4,2 Dual Banked Bootloader

As described in previous sections, dual bank gives the chance of modifying data on another bank while code is running on one bank, and also allows booting from different banks.

- It allows customers to implement the FW upgrade in application without bootloader: let application run in one bank

and buffer FW upgrade data in another bank.

• It adds the advantage to let the same code work on both banks without modifying the binary mapping, since both banks can be mapped to boot program area.

– It allows 2 firmware versions in 2 banks and selection of boot firmware version.

Page 38

This simple example adds implementation of dual banked bootloader as described in Section 3.1.3.2 , It's added as an alternative partitioning method. The following constant should be defined to enable it:

```
#define PARTITION_DUAL_BANK
```

When using this option, the two firmware region size (for region A & B, defined as *REGION_SIZE*) should also be decreased, because the bootloader code is duplicated.

4,3 Firmware Updater

This program is used to transmit the firmware from the host PC to the bootloader. Since an RS232 (through a PC COM port) link is implemented at the moment, a standard terminal application (like HyperTerminal) could be used to perform the same operation. However, it will be necessary to develop a program if another media or communication protocol is implemented and not supported by standard programs.

Figure 4-1. Firmware Updater Main Window

The main window of the application enables the user to perform the following operations:

• Choose the firmware file to send to the bootloader

• Select the media and associated parameters

• Select the communication protocol and associated parameters

• Launch an upgrade

Note that you must select the same parameters in the Firmware Updater as the ones which have been selected for the bootloader. For example, if the bootloader is configured to connect using a USART configured at 115200 bps, no parity and 1 stop bit, select those parameters in the Firmware Updater.

4,4 Firmware Packager

The Firmware Packager is used to prepare the firmware prior to sending it to customers, including encrypting it, generating a signature or MAC tag, etc.

Note that when the bootloader enables code encryption (*USE_ENCRYPTION*), the Firmware Packager must be used, with the same encryption settings, to generate encrypted firmware for Firmware Updater to download.

Page 39

Figure 4-2. Firmware Packager Main Window

Currently, the following functionalities are implemented:

- Select the firmware file to package
- Select the output file
- Select an encryption method (AES or Triple DES) with associated parameters
  - Encryption mode, key and Initialization Vector (IV)
- Launch the firmware packaging

Note that you must select the same parameters in the Firmware Packager as the ones selected for the bootloader. Для example, if the bootloader is configured to accept a firmware encryption in AES-CBC with a particular key and IV, enter the same parameters in the Firmware packager.

5. Related Documents

[1] Atmel Corp., 2006, Safe and Secure Firmware Upgrade for AT91SAM Microcontrollers, literature no. 6253.

[2] Atmel Corp., 2006, Safe and Secure Bootloader Implementation, literature no. 6282.

6. лист регистраций изменений

Док. Rev.	Дата	Комментарии
42141A	06/2013	Initial revision

Atmel Corporation
1600 Технология Drive
Сан-Хосе, CA 95110
США
Tel: (+1)(408) 441-0311

Atmel Asia Limited
Блок 01-5 и 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Мюнхен
ГЕРМАНИЯ

Atmel Japan GK
16F Shin-Osaki Kangyo Building
1-6-4 Osaki
Shinagawa-ku, Tokyo 141-0032
ЯПОНИЯ

Fax: (+1)(408) 487-2600
www.atmel.com

ГОНКОНГ
Тел: (+852) 2245-6100
Факс: (+852) 2722-1369

Тел: (+49) 89-31970-0
Fax: (+49) 89-3194621

Tel: (+81)(3) 6417-0300
Fax: (+81)(3) 6417-0370

© 2013 Atmel Corporation. Все права защищены. / Rev.: 42141A-SAM-06/2013

Atmel®, Atmel logo and combinations thereof, Enabling Unlimited Possibilities®, and others are registered trademarks or trademarks of Atmel Corporation or дочерние компании. ARM® and Cortex® are registered trademarks of ARM Ltd. Other terms and product names may be trademarks of others.

Отказ от ответственности: информация в этом документе предоставляется в связи с продуктами Atmel.
Документа или в связи с продажей продукции Atmel.
КОСВЕННЫЙ, ШТРАФНЫЙ, СПЕЦИАЛЬНЫЙ ИЛИ СЛУЧАЙНЫЙ УЩЕРБ (ВКЛЮЧАЯ, БЕЗ ОГРАНИЧЕНИЙ, УБЫТКИ ПОТЕРИ И ПРИБЫЛИ
ИНФОРМАЦИЯ), ВОЗНИКАЮЩИЕ В РЕЗУЛЬТАТЕ ИСПОЛЬЗОВАНИЯ ИЛИ НЕВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ЭТОГО ДОКУМЕНТА, Д
Представлений или гарантий в отношении точности или полноты содержания настоящего документа и оставляет за собой право в любое время вноси
без предупреждения. Atmel не делает никаких обязательств по обновлению информации, содержащейся в настоящем документе.
Atmel не делает
Автомобильных приложений.
Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in,
Изделия Atmel не предназначены, не разрешены или не гарантированы для использования в качестве компонентов в приложениях, предназначенных