IST Knowledge

AVR BootLoader в вопросах и ответах. Часть 2

Эта статья является продолжением AVR BootLoader в вопросах и ответах. Часть 1 (/besplantye-uroki/avr-bootloader-v-voprosah-i-otvetah/)

Как загрузчику убедиться что в приложении нет изменений?

После сброса загрузчик, как правило, осуществляет переход к приложению. Но как он может быть уверен, что приложение является допустимым, не повреждена ли программа, которая должна начать исполняться? Что делать, если предыдущая попытка обновить программу аварийно завершилась на середине загрузки? В таком случае есть много возможных мер предосторожности, которые вы можете предпринять. Вот некоторые из них:

1. Если это возможно, сразу же после программирования неплохо бы убедиться, что обновление программы удалось. Это можно сделать, прочитав флэш-приложение, и сравнить результат с исходным файлом прошивки. Если вы не хотите, чтобы загрузчик выгружал прошивку, вы можете отправить загрузчику вашу прошивку два раза. Первый раз для записи, второй раз для того чтобы загрузчик сравнил с тем что только что записал. Конечно ни один подход не помогает если есть ошибки в канале связи или загрузчике. Большинство стандартных протоколов программирования поддерживают процедуру верификации, но такой подход не обнаруживает случайные искажения.
2. Перед тем как загрузчик начнет программирование, он должен удалить первую страницу приложения. Это позволит установить всю флэш-страницу в 0xFF. Затем прошивать страницы в обратном порядке от конца к началу. При запуске, загрузчик может проверить инструкции на правильность (обычно JMP или RJMP) по адресу 0, чтобы определить, все приложение было запрограммировано успешно. Этот подход, вероятно, потребует специального средства программирования, так как большинство стандартных средств не дают возможность обратного проектирования. Такой подход также не обнаруживает случайные искажения.
3. Вариант 2 прошивать все приложение как обычно, но резервировать последние несколько байт флэш-памяти для специального маркера, содержащий распознаваемый шаблон, например, 0xBEEF. Вы можете сделать это, изменив шестнадцатеричный файл прошивки с таким инструментом, как srec_cat. Когда загрузчик запускается, он читает маркер байт, и если он находит правильное значение, то понимает, что приложение было записано правильно. Реализовать данный вариант можно с помощью стандартных инструментов

программирования, при условии, что прошиваться программа будет с самого начала к концу (что, скорее всего так и будет, но тем не менее не гарантируется). Такой подход также не обнаруживает случайные искажения.

4. Более тщательный вариантом верификации является использование контрольных сумм CRC для проверки всего приложения каждый раз, когда запускается загрузчик. Этот вариант похож на вариант 3 за исключением того, что вместо маркера в последних нескольких байтах прошивки хранится вычисленное значение CRC. Это также вы можете проделать это с помощью инструмента srec_cat. Не забудьте, учесть оставшееся в конце свободное пространство в разделе RWW, заполнив его смещением или чем-то еще. Когда загрузчик запускается, он вычисляет CRC используя тот же алгоритм CRC как и srec_cat. Это значение сравнивается с хранимым значением и если они совпадают загрузчик считает, что приложение является не поврежденным. Реализовать данный вариант можно с помощью стандартных инструментов программирования, при этом данный подход позволяет обнаруживать случайные искажения. Недостатком является то что задержка перед запуском программы увеличивается.
5. Если вы очень обеспокоены возможными искажениями, вы также можете производить проверку контрольной суммы CRC загрузчика согласно варианту 4. Конечно, не так много чего можно предпринять в случае обнаружения искажений, например, попытаться просигнализировать о проблеме и выключиться. В таком случае настает время задействовать программатор ISP.
6. Одним хорошим эффектом использования опции перезагрузки, хранящейся в EEPROM (обсуждается в вопросе № 10) является то, что данный подход также способен обеспечивать автоматическое обнаружение искажений. Если приложение было записано наполовину или повреждено, оно не получит возможность установить байт APP_RUN и загрузчик будет продолжать исполняться при следующей перезагрузке. Вместо того чтобы искать искажения в приложении, этот подход позволяет оценить неудачное исполнение приложения как проблему. В некотором смысле это даже лучше, чем проверка CRC, потому что на самом деле CRC может быть вычислен у уже испорченного еще задолго до заливки образа прошивки. Конечно, подход с применением EEPROM не является идеальным, поскольку в приложении могут иметься небольшие искажения и сбой может возникнуть после того, как оно запишет байт APP_RUN.

Может ли код загрузчика исполнять код, встроенный в приложение?

Говорят: «Нет, нет и 100% определенно нет».
Хоть технически это возможно, но ответ все равно — «Нет». Ваш загрузчик будет гораздо более надежным, если он имеет нулевую зависимость от приложения. Основная цель загрузчика — стереть и перепрограммировать приложение. Вы же не хотите выполнять вызовы участков кода приложения в то время как стираете его содержимое.
Плохой практикой также считается хранить код, относящийся к загрузчику в разделе RWW. Не существует полного доказательства или способа защиты от случайного стирания и перепрограммирования этой области. Одно полное стирание RWW и загрузчик потенциально становится бесполезным.

Может ли код приложения исполнять код, встроенный в загрузчик?

Да, сделать это довольно легко. Особенно если код для совместного использования не имеет доступа к глобальным переменным. Только не пытайтесь достичь общего кода путем создания загрузчика и приложения в качестве одного двоичного файла. Лучше всего строить их по отдельности и использовать указатели на функции общего кода. Простым решением в таком подходе будет создавать загрузчик обычным способом и затем искать в нем точки входа в общие функции через специальную карту адресов функций или в дизассемблированном файле. Затем можно создать жестко заданные функции указателей в приложении на основе этих адресов. Это будет работать, но при этом ваши два бинарных файла будут тесно связаны друг с другом. Каждый раз, когда загрузчик будет претерпевать изменения эти функции будут сдвигаться, и вам всякий раз будет требоваться обновить их адреса и перекомпилировать приложение. Это довольно узкое и трудное место в разработке загрузчика. Но если разработка вашего загрузчика завершена, при этом вы уверены, что это он больше не изменится, то это довольно быстрое (хотя немного грязное) решение.
Тем не менее существует лучший подход, устраняющий проблему перемещения общих функций в загрузчик. Механизм тот же что и для обработчиков прерываний. Как процессор определяет адреса обработчиков? Никак. Он знает только адрес таблицы переходов, который добавляется в двоичный код во время линковки проекта. Чтобы попадать в необходимое место процессору требуется только эта таблица переходов (так называемая таблица векторов). Вы можете использовать ту же технику чтобы обращаться к общим функциям из загрузчика. Есть несколько способов задать таблицу переходов. Удобнее это представить в
виде отдельного маленького файла сборки:

```
.section .jumps,"ax",@progbits
// The gnu assembler will replace JMP with RJMP when possible
.global _jumptable
_jumptable:
    jmp shared_func1
    jmp shared_func2
    jmp shared_func3
```

Ваша таблица переходов должна использовать имена общих функций, а не их адреса.
Это важно, потому что, когда вы перестроите загрузчик, вам потребуется, чтобы линковщик автоматически выставил корректные смещения к функциям в таблицу переходов. Тогда вам будет достаточно поместить таблицу переходов в заданное место загрузчика, а остальное у него может изменяться свободно. Чтобы поместить таблицу переходов в ваш загрузчик необходимо добавить .S в ваши фалы Makefile файл в строке ASRC. Затем нужно добавить компоновщику флаг позиционирования его на предопределенный адрес, который не будет меняться. Чаще всего это в конец загрузчика:

```
JUMPSTART = 0x3FE0 # 32 bytes from the end of the AT90USB162 4kb boot
section
LDFLAGS += -Wl,--section-start=.jumps=$(JUMPSTART)
```

Вам может понадобиться еще один флаг, чтобы предотвратить попытки компоновщика выбрасывать вашу таблицу переходов в случаях, когда она окажется не востребованной. Это возможно при использовании флага компилятора `-ffunction-sections` совместно с флагами линковщика `--gc-sections` и `--relax` . Поэтому если вы не уверены, то в любом случае это не помешает добавить:

```
LDFLAGS += -Wl,--undefined=_jumptable
```

Количество переходов, которые будут помещаться в таблице зависит от того, какой объем пространства, вы зарезервируете, а также это зависит от размера переходов. Размер переходов зависит от того, насколько далеко адресованы переходы. В загрузчиках по 8Кб или менее на командой переходов будет RJMP, требующая только 2 байта. Самый простой способ узнать точный размер — посмотреть на дизассемблированный код загрузчика. С другой техникой, основанной на принципе таблиц прерываний можно ознакомиться в вопросе №18. Кроме того, некоторые предлагают добавить еще одну прослойку перехода, чтобы также можно было перемещать саму таблицу векторов, но необходимость в этом довольно редкая.
После создания таблицы переходов, следующим шагом является определить указатели на функции, которые упростят вызов общих функций через таблицу переходов. Вы можете сделать это с помощью макросов или встроенных функций. Возможно, встроенные функции предпочтительнее для дополнительной безопасности типизации. Для создания указателей на функции необходимо открыть для дизассеблированный код загрузчика и найти «_jumptable». Запишите адрес байта каждого перехода, указанного в таблице. Если у вас есть адреса, то необходимо создать заголовочный файл вроде этого:

```
typedef void (*PF_VOID)(void);
typedef void (*PF_WHATEVER)(uint8_t);
static __inline__ void call_func1(void)
            { ((PF_VOID) (0x3FE0/2))(); }
static __inline__ void call_func2(void)
            { ((PF_VOID) (0x3FE2/2))(); }
static __inline__ void call_func3(uint8_t arg)
            { ((PF_WHATEVER) (0x3FE2/2))(arg); }
```

Шестнадцатеричные числа байт-адреса, которые извлекаются из дизассемблированного файла. Они должны быть поделены пополам, чтобы создать слово-адреса, так как GCC с такой ситуацией автоматически не справляется (это возможно баг GCC). Включите этот заголовок в приложение и осуществляете вызов функций.

```
call_func3(1);
```

Почему глобальные переменные не доступны из общих функции?

Так как у вас два совершенно разных двоичных файла, то каждый будет иметь свое собственное распределение памяти. Позиции глобальных переменных в приложении никак не связаны с позициями глобальных переменных в
загрузчике.
Допустим, общая функция загрузчика по ошибке производит чтение и запись в глобальную переменную напрямую.
Когда загрузчик был отлинкован, пространство глобальных переменных по сути было жестко запрограммированно в его исполняемый код. Когда загрузчик исполняется, то его позиция правильная, и все работает нормально. Но когда запускается приложение, которое вызывает те же функции, то жестко заданные позиции глобального пространства уже совсем не те. Создается неправильная ситуация, т.к. когда приложение исполняется, то ее распределение памяти отличается от распределения памяти загрузчика. Таким образом, вы не можете получить непосредственный доступ к глобальным переменным из общих функций
без потенциальной угрозы уничтожения данных в приложении. Общие функции должны принимать адреса нелокальных данных во время исполнения. Сделать это проще, чем кажется — достаточно функции передавать указатели на глобальные данные в качестве аргументов. Когда загрузчик исполняется, он должен передать адрес глобальной переменной, которая была определена в загрузчик. А когда приложение запущено, то оно должно тоже передавать адрес глобальной переменной, которая определена в
приложении. Если необходимо передать несколько переменных, то их удобно группировать в структуру, а функции передавать на нее указатель. Что-то вроде этого:

```
// In a shared header file
typedef struct {
    uint8_t val1;
    uint16_t val2;
} globals_t;
// The shared function
void func4(globals_t *vars) {
    vars->val1 = 0;
    vars->val2 = 512;
}
// Globally defined in each binary
globals_t g_vars;
// Calling the shared function
call_func4(&g_vars);
```

Если по каким-то причинам вы не можете передать параметр в общую функцию см. вопрос № 17 ниже.

Может ли приложение использовать IRS встроенный в загрузчик?

На самом деле это проще сделать, чем организовать общие функции, поскольку векторы прерывания по умолчанию уже обеспечивают переход чтобы найти процедуру обработки прерывания (ISR). Достаточно написать и откомпилировать привычным способом в загрузчике ISR (только без доступа напрямую к глобальным переменным). После этого необходимо в дизассемблированном файле загрузчика найти таблицу прерываний. Записать адрес вектора ISR, который необходимо сделать общим. Вам не потребуются адреса обработчиков из этого вектора. Вам нужет адрес самого вектора прерываний. Его адрес будет в начале загрузчика. Затем необходимо добавить обрабочик прерывания в вашем приложении. Данный обработчик сам по себе ничего не делает — он осуществляет безусловный переход на функцию — обработчик прерывания загрузчика. Например:

```
// This must be declared "naked" because we want to let the
// bootloader function handle all of the register push/pops
// and do the RETI to end the handler.
void USB_GEN_vect(void) __attribute__((naked));
ISR(USB_GEN_vect)
{
    asm("jmp 0x302C");
}
```

Это не будет работать если не пометить функцию атрибутом «naked». Без этого атрибута созданная продедура ISR в приложении будет толкать значения в стек, которые никогда не будут извлечены, что означает, что обратные адреса будут теряться. Это произойдет, потому что вы делаете не вызов ISR загрузчика, а безусловный переход на него и функция RETI в конце ISR загрузчика не вернет курсор к ISR приложения, а вместо этого курсор попадет обратно в код, который исполнялся во время вызова ISR. Также отметим, что в ассемблерных вставках GCC автоматически преобразуются байт-адреса в слово-адреса, так что вам не нужно их делить пополам.
Если вы не хотите забивать голову деталями, вы можете заменить все объявления с атрибутами «naked» на макросы или организовать встроенную функцию для вызова общих обработчиков через указатели. Указатель на функцию будет выглядеть так же как и в вопросе № 14. Но вам придется поплатиться за эту простоту задействованием кучей бесполезных регистров которые появятся на входе и выходе из ISR приложения.

Как организовать доступ к глобальным данным внутри общих обработчиков ISR?

Такая же ситуация как и в вопросе №15, но на этот раз решение не такое простое, поскольку вы не можете поместить в стек обработчика ISR указатель. Есть целый ряд возможных решений, но, поскольку это руководство уже довольно обширно об этом будет сказано не так подробно. Есть два варианта, таких как использование регистров GPIO для передачи указателей на глобальные или резервирование части SRAM с известным адресом, где расположены глобальные данные. Если вам нужен иной способ, то вам сюда: http://tinyurl.com/q3fpud (http://tinyurl.com/q3fpud) . Идея с зарезервированной областью SRAM, кажется неплохой.

Можно ли сэкономить место в загрузчике на использовании обработчиков ISR?

Да, зачастую таким образом можно сэкономить 100 и более байт загрузчика. Эта также относиться и к обычным приложениям, но такая экономия для загрузчика более существенна чем к приложению. Некоторые архитектуры AVR имеют 40 или более прерываний, каждый из которых в таблице векторов прерываний принимает по 4 байта. Вы можете не только незначительно сэкономить, но и переопределять неиспользуемые вектора как переходы на общие функции (обсуждается в вопросе № 14). Первая запись в таблице прерываний — это вектор сброса. Загрузчики и приложения часто пользуются вектором сброса, т.к. исполняемый код никогда не начинается с начала прошивки. Даже если вектор сброса в настоящее время не требуется, он может понадобиться в будущем, например, после того, как кто-то добавит новую переменную PROGMEM. Поэтому вместо того чтобы, полностью удалять таблицу прерываний из загрузчика, лучше просто его уменьшить. Минимальная таблица прерываний может быть записать таким образом:

```
.section .blvects,"ax",@progbits
.global __vector_default
__vector_default:
    jmp  __init
```

Эта таблица содержит только вектор сброса, который ведет на функцию __init среды исполнения C (это то, что делает вектор сброса по умолчанию). Предполагая, что JMP можно заменить на RJMP, мы имеем вектор прерываний размером в 2 байта. Добавьте этот .S файл в строку ASRC вашего Makefile. Более правильный способ замены таблицы векторов по умолчанию заключается в использовании специального сценария линкера. Сначала необходимо узнать из какого места берет скрипты компоновщик AVR. Теоретически сценарии компоновщика должен быть назван именем архитектуры AVR. Но оказывается, что иногда это не так, поэтому обнаружить скрипт можно переименованием папки, в которой он содержится (чаще всего это C:WinAVRavrlibldscripts на Windows, и /usr/local/avr/lib/ldscripts на Unix-подобных ОС), и последующей перекомпиляцией. Компоновщик будет жаловаться «Не удается открыть файл сценария линковщика ldscripts/avr3.x». Это подскажет вам правильное имя файла сценария. Восстановите исходные имена директориям и скопируйте найденный файл в каталог проекта. Затем добавьте следующую строку в Makefile в место, где прописываются флаки компоновщика:

```
LDFLAGS += -T bootloader.x   # Or whatever you named the linker script
```

Теперь нужно изменить скопированный скрипт компоновщика используя ваш уменьшенный вариант таблицы прерываний, а то что по умолчанию убрать. Откройте скрипт компоновщика и найдите «»vectors». Вы должны найти текст со следующим содержанием:

```
.text   :
  {
    *(.vectors)
    KEEP(*(.vectors))
```

Добавьте строчку DISCARD и замените «vectors», на ваше имя секции:

```
/DISCARD/ : { *(.vectors); } /* Discard standard vectors */
  .text   :
  {
    *(.blvects)   /* Position and keep custom vectors */
    KEEP(*(.blvects))
```

Если у вас не используются прерывания и таблица ISR пуста, то это все что вам нужно сделать. О другом подходе можно упоминуть вскользь — необходимо использовать флаги компоновщика -nostartfiles и -nodefaultlibs совместно с пользовательской процедурой инициализации. Это позволяет исключить из прошивки таблицу прерываний по заданную по умолчанию, также как и саму среду исполнения C. Если вам необходима более детальная информация по данной теме, то вы можете обратиться за ними по следующим ссылкам:

- http://tinyurl.com/pkkwzt (http://tinyurl.com/pkkwzt)
- http://tinyurl.com/qrsjjp (http://tinyurl.com/qrsjjp)
- http://tinyurl.com/ptshkp (http://tinyurl.com/ptshkp)

Если загрузчик использует таблиуц прерываний ISR, вы можете сэкономить на размере прошивки заменив полную таблицу прерываний на уменьшенную версию. Допустим, нам необходимо только 11-е прерывание (USB на AT90USB162), поэтому мы усечем таблицы и повторно задействуем слоты до прерывания 11 в качестве переходов на общую функцию. При этом убедитесь, что положение оставшихся векторов прерываний правильно. Каждый должен быть нацелен на адрес указанный для этого прерывания в даташите (‘nop’ы в примере ниже). Повторно задействуя часть таблицы векторов вы можете избежать необходимости в таблице с отдельными переходами, как рассказано в вопросе №14. Вам просто потребуются указатели на функции для повторного вызова векторов. См. комментарии ниже для более подробной информации:

```
.section .bootvect,"ax",@progbits
; Custom vector table that eliminates wasted space after the last used
; vector (__vector_11, usb general). Also re-purpose the unused space
; between the reset vector and the usb vector for the jumps to shared
; code.
;
.global __vector_default
; There are 21 "word" spaces between __init and __vector_11. This fits
; 21 RJMPs or 10 JMPs. Since the bootloader is only "2K words" long,
; use RJMPs.
; - Don't change the order of these (unless it is before any devices
;   shipped)!
; - Add new entries by replacing nop's
; - Remove entries by replace them with nop's (without reordering)
__vector_default:
    rjmp  __init          ; 0x3000  !used interrupt!
    rjmp shared_func1     ; 0x3002
    rjmp shared_func2     ; 0x3004
    rjmp shared_func3     ; 0x3006
    rjmp shared_func4     ; 0x3008
    rjmp shared_func5     ; 0x300a
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    rjmp __vector_11      ; 0x302C  !used interrupt!
```

Автор: Брэд Шик <schickb@gmail.com (mailto:schickb@gmail.com)> При участии и редактирования Клифф Лоусон. Перевод: Валеев Денислам.

Микроконтроллеры AVR (https://istknowledge.wordpress.com/category/mikrokontrollery-avr/)

Zashibon
31 августа, 201219 декабря, 2019

AVR /
bootloader /
C /

загрузчик /
уроки