Cortex M3
ARM
Разработка встроенных приложений

Пересказал А. Косенко
aikos55@gmail.com

1

[1] Кому не лень, проставьте ссылки оглавления сами.

1. Cortex-M3

Здесь описаны основные качества процессора ARM Cortex™-M3 и разработка для него, включая миграцию проектов ARM на платформу Cortex-M3.

ARM Cortex-M3 это высокопроизводительный 32-бит RISC процессор низкой стоимости. Он выполняет только команды Thumb-2, а команды ARM не поддерживает. Имеет архитектуру ARM v7-M и ядро Harvard с 3-этажным конвейером. Поддерживает аппаратное деление и быстрый вход/выход в прерывание (ISR).

Кроме ядра CPU процессор Cortex-M3 имеет контроллер вложенных прерываний *Nested Vectored Interrupt Controller* (NVIC), необязательный блок защиты памяти *Memory Protection Unit* (MPU), таймер, порт отладки *Debug Access Port* (DAP) и необязательный *Embedded Trace Macrocell* (ETM). Карта памяти у Cortex-M3 фиксированная.

1.1. Nested Vectored Interrupt Controller (NVIC)

В зависимости от производителя NVIC может поддерживать до 240 внешних прерываний и до 256 динамически изменяемых уровней приоритетов. Поддерживаются источники прерываний по уровню и по фронту. При входе и выходе из прерывания состояние процессора сохраняется и восстанавливается автоматически. Также NVIC поддерживает сцепленные прерывания.

При этом таблица векторов Cortex-M3 весьма отличается от предыдущих ядер ARM. Она содержит адреса обработчиков прерываний и ISR, а не команды, как было раньше. Начальный адрес стека и обработчика Сброса лежат по адресам 0x0 и 0x4 соответственно. В нужные регистры CPU они загружаются при Сбросе.

1.2. Memory Protection Unit (MPU)

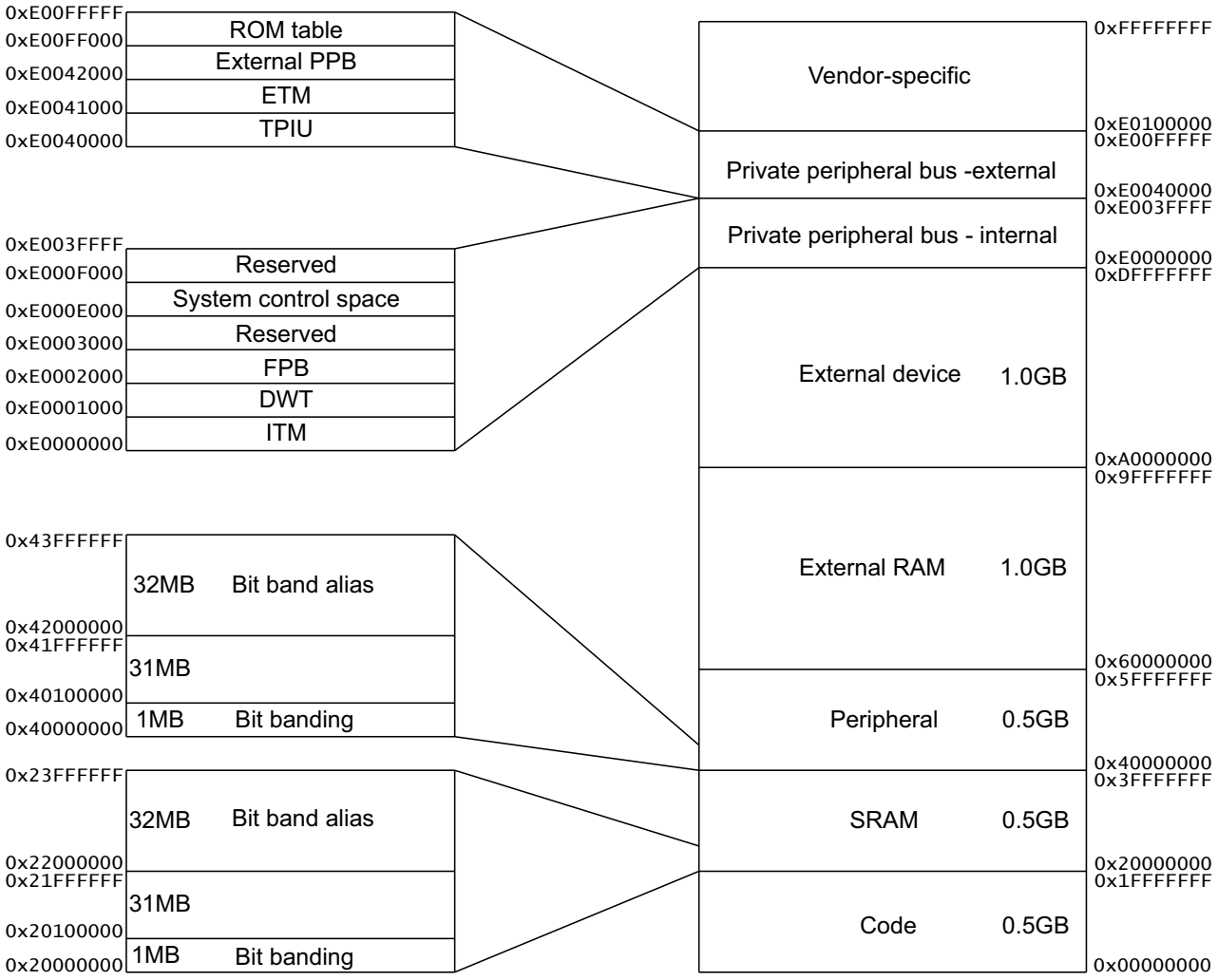
MPU это необязательный компонент Cortex-M3. Если есть, он поддерживает защиту регионов памяти через привилегии и правила доступа. Поддерживается до восьми регионов, каждый из которых может быть разделён на восемь под-регионов равного размера.

1.3. Debug Access Port (DAP)

DAP использует интерфейс AHB-AP для связи с процессором и другой периферией. Debug Port имеет две реализации, *Serial Wire JTAG Debug Port* (SWJ-DP) и *Serial Wire Debug Port* (SW-DP). Это зависит от производителя Cortex-M3.

1.4. Карта памяти

В отличие от предыдущих ядер ARM, карта памяти Cortex-M3 фиксированная. Это позволяет легко портировать программы разных систем на Cortex-M3. Адресное пространство разделяется на несколько различных секций. См. ниже.



Регион	Описание	Доступ по шине
code	Память кода (flash, ROM или картированная RAM).	ICode и DCode
SRAM	Встроенная SRAM с битовым отображением.	system
peripheral	Нормальная периферия с битовым отображением.	system
external RAM	Внешняя память.	system
external device	Внешняя периферия или разделяемая память.	system
private peripheral	Системные устройства вроде MPU, NVIC, DAP и иных CoreSight устройств.	system
vendor specific	Дополнения от производителя.	—

2. Разработка для Cortex-M3

Все примеры сделаны с помощью *RealView Compilation Tools* (RVCT) 3.0 или позднее.

2.1. Обработка исключений

2.1.1. Таблица векторов исключений

Самое простое это разместить файл с массивом ссылок на C функции по адресу 0x0.

Пример C структуры с таблицей векторов

```
/* Filename: exceptions.c */
typedef void(* const ExecFuncPtr)(void) __irq;
/* Place table in separate section */
#pragma arm section rodata="exceptions_area"
ExecFuncPtr exception_table[] = {
(ExecFuncPtr)&Image$$ARM_LIB_STACKHEAP$$ZI$$Limit,
(ExecFuncPtr)__main, /* Initial PC, set to entry point */
NMIException,
HardFaultException
MemManageException,
BusFaultException,
UsageFaultException,
0, 0, 0, 0, /* Reserved */
SVCHandler,
DebugMonitor,
0, /* Reserved */
PendSVC,SysTickHandler,
/* Configurable interrupts start here...*/
InterruptHandler0,
InterruptHandler1,
InterruptHandler2
/* ... */
};
#pragma arm section
```

NB. Первые два элемента это начальный указатель стека и точка входа в программу. Указатель стека это предопределённый символ компоновщика, а точка входа стандартна для библиотеки C (`__main`).

Так все RO (read-only) данные между директивами `#pragma arm section rodata="exceptions_area"` и `#pragma arm section` определяются в собственной секции `exceptions_area`. И в файле размещения её засовывают по адресу `0x0`.

2.1.2. Драйверы исключений

Обработчикам исключений нет нужды сохранять регистры при входе и восстанавливать при выходе, это делается аппаратно. Так что их можно писать как обычные ABI-совместимые C функции. Однако рекомендуем снабжать их квалификатором `__irq` ясности кода для. По нему компилятор выполняет при необходимости выравнивание стека на границу 8 байт. См. *Восьмибайтовое выравнивание стека*.

Пример C обработчика прерываний

```
__irq void SysTickHandler(void) {
        printf("----- SysTick Interrupt -----");
}
```

NB. Очистка источника прерываний лежит на ISR.

На Cortex-M3 обработка приоритетов исключений, их вложенность и сохранение регистров полностью выполняется ядром. То есть, прерывания остаются разрешёнными при входе в любой обработчик исключений.

2.1.3. Размещение таблицы векторов

Это очень просто делается в файле размещения по адресу `0x0`.

Пример размещения таблицы векторов

```
LOAD_REGION 0x00000000 0x00200000 {
        ;; Maximum of 256 exceptions (256*4 bytes == 0x400) VECTORS 0x0 0x400
        {
                exceptions.o (exceptions_area, +FIRST)
        }
}
```

NB. `+FIRST` обеспечивает размещение `exceptions_area` в самом начале региона и предотвращает его удаление как неиспользованной секции при компоновке.

2.1.4. Конфигурация System Control Space (SCS) регистров

Регистры `SCS` размещены по адресу `0xE000E000`. Их много, и надо бы разместить их в структуре, а её засунуть в файл размещения, как и таблицу векторов.

Пример структуры SCS

```
typedef volatile struct { int MasterCtrl;
int IntCtrlType;
int zReserved008_00c[2];
/* Reserved space */
struct {
int Ctrl;
int Reload;
int Value;
int Calibration;
} SysTick;

int zReserved020_0fc[(0x100-0x20)/4];
/* Reserved space */
/* Offset 0x0100 */
struct {
int Enable[32];
int Disable[32];
int Set[32];
int Clear[32];
int Active[64];
int Priority[64];
} NVIC;
```

```
int zReserved0x500_0xcfc[(0xd00-0x500)/4];
/* Reserved space */
/* Offset 0x0d00 */
int CPUID;
int IRQcontrolState;
int ExceptionTableOffset;
int AIRC;
int SysCtrl;
int ConfigCtrl;
int SystemPriority[3];
int SystemHandlerCtrlAndState;
int ConfigurableFaultStatus;
int HardFaultStatus;
int DebugFaultStatus;
int MemManageAddress;
int BusFaultAddress;
int AuxFaultStatus;

int zReserved0xd40_0xd90[(0xd90-0xd40)/4];
/* Reserved space */
/* Offset 0x0d90 */
struct {
int Type;
int Ctrl;
int RegionNumber;
int RegionBaseAddr;
int RegionAttrSize; } MPU;
} SCS_t;
```

NB. Эта структура может содержать не все регистры SCS вашего устройства.

2.1.5. Конфигурирование отдельных IRQs

Каждый IRQ имеет свой бит разрешения в Interrupt Set Enable Registers, из регистров NVIC. Для разрешения прерывания надо установить соответствующий бит в регистре Interrupt Set Enable. См. документацию производителя.

Пример функции разрешения IRQ

```
void NVIC_enableISR(unsigned isr) {
/* The isr argument is the number of the interrupt to enable. */
        SCS.NVIC.Enable[ (isr/32) ] = 1<<(isr % 32);
}
```

NB. Некоторые регистры в области SCS доступны только из привилегированного режима. Отдельные IRQs можно выключить установкой нужного бита в регистрах Interrupt Clear Enable.

2.1.6. Конфигурирование отдельных IRQs

Каждому прерыванию может быть назначен приоритет через регистры Interrupt Priority. В зависимости от реализации их может быть до 256 уровней. Уровни представлены 8 битами, которые хранятся группами по 4 на один регистр.

Высший приоритет представлен 0, а низший равен 255.

2.2. Memory Protection Unit (MPU)

В Cortex-M3 наличие MPU совсем необязательно. См. описание устройства.

2.2.1. Регистры MPU

Регистры MPU размещены по адресу `0xE000ED90`. Есть 5 базовых регистров MPU и регистры синонимов для каждого региона. Кратко см. ниже, а подробно в *Cortex-M3 Technical Reference Manual*.

Имя	Адрес	Описание
MPU type	0xE000ED90	Число регионов в битах [15:8] и 0 без MPU.
control register	0xE000ED94	Вкл./выкл. MPU, использование карты памяти по умолчанию для привилегированного доступа либо если MPU был включён обработчиками Hard Fault, NMI и Fault Mask.
region number	0xE000ED98	Номер конфигурируемого региона.
region base address	0xE000ED9C	Ставит или читает базовый адрес региона.
region attribute and size	0xE000EDA0	Ставит или читает размер и разрешения региона.

Для регистров базового адреса размера и атрибутов каждого региона существуют синонимы, расположенные последовательно начиная с адреса `0xE000EDA4`. Это полезно при быстрой конфигурации MPU по включению питания командой STM.

2.2.2. Конфигурирование MPU

При конфигурировании региона MPU его сначала нужно выбрать одним из двух методов. Можно записать нужное число в регистр номера региона или использовать биты 0 — 3 регистра базового адреса региона и установить бит VALID. Вторым методом можно одновременно установить и базовый адрес региона.

После выбора региона нужно записать его базовый адрес. Он должен быть кратен размеру региона, так для 64KB региона он может быть `0x00010000`, `0x00020000` и т.д.

Наконец, в регистре атрибутов и размера надо установить разрешения, размер и включить его. См. *Cortex-M3 Technical Reference Manual*.

NB. MPU нужно, конечно же, включить установкой бита 0 в регистре MPU Control Register.

2.2.3. Атрибуты и размер региона (тип и разрешения доступа)

MPU поддерживает несколько разных типов памяти, расширений и атрибутов для каждого региона.

Поле	Имя	Описание
[28]	XN	Запрет выборки команд, 1= нельзя.
[26:24]	AP	Разрешения доступа к данным (чтение/запись в режимах User и Privileged).
[21:19]	TEX	Поле расширения типа, например, strongly ordered, peripheral.
[18]	S	Разделяемая.
[17]	C	Кэшируемая.
[16]	B	Буферизуемая.
[15:8]	SRD	Поле выключения подрегионов.
[5:1]	REGION SIZE	Размер региона, например 4K, 8K.
[0]	SZENABLE	Бит включения региона.

2.2.4. Подрегионы

Каждый регион памяти разделяется на восемь подрегионов (одна восьмая региона), каждый из которых может быть выключен отдельно в поле SRD в регистре атрибутов и размера. Младший бит соответствует подрегиону с меньшим адресом.

Подрегионы полезны при перекрытии областей памяти. Например, под-регион внутри большого региона может иметь свои атрибуты. Выключаем под-регион и назначаем туда другой регион с другими атрибутами.

NB. В регионах размером 32, 64 и 128 байт подрегионы невозможны.

2.3. Конфигурация стека и динамики

2.3.1. Стек и динамика

RealView Compilation Tools (RVCT) даёт несколько методов размещения стека и динамической памяти. Два основных метода это: переписать функцию `__user_initial_stackheap()` или поместить стек и динамику в особые регионы в файле размещения.

Также поддерживаются модели одного и двух регионов. В модели одного региона стек и динамика делят одну область. Динамика прирастает от младших адресов к старшим, а стек от старших к младшим.

В модели двух регионов они занимают каждый свою область, но направление прирастания остаётся прежним. См. *RVCT Developer Guide* и *RVCT Compiler and Libraries Guide*.

NB. Использование MPU при размещении регионов позволит обнаружить переполнение.

2.3.2. Один регион

Это режим по умолчанию.

Здесь самое простое это использовать регион с именем `ARM_LIB_STACKHEAP` файле размещения.

Пример одного региона

```
;; Heap and stack share 1MB
ARM_LIB_STACKHEAP 0x20100000 EMPTY 0x100000
{
}
```

Здесь `EMPTY` показывает, что при компоновке в регионе ничего не появляется.

В таблице векторов начальное значение указателя стека пишется с помощью предопределённого символа компоновщика `Image$$ARM_LIB_STACK$$ZI$$Limit`.

Вместо использования этого имени можно переписать функцию `__user_initial_stackheap()`. Но начальный указатель стека в таблице векторов нужен!

2.3.3. Два региона

Для этого нужно в файле размещения указать два региона с именами `ARM_LIB_HEAP` и `ARM_LIB_STACK`. Также надо добавить либо `IMPORT __use_two_region_memory` из ассемблера или `#pragma import(__use_two_region_memory)` из C.

Пример двух регионов

```
; Heap starts at 1MB and grows upwards
ARM_LIB_HEAP 0x20100000 EMPTY 0x100000-0x8000
{
}
; Stack space starts at the end of the 2MB of RAM
; and grows downwards for 32KB (indicated by the negative length)
```

```
ARM_LIB_STACK 0x20200000 EMPTY -0x8000
{
}
```

В таблице векторов начальное значение указателя стека пишется с помощью предопределённого символа компоновщика `Image$$ARM_LIB_STACK$$ZI$$Limit`.

Вместо использования этого имени можно переписать функцию `__user_initial_stackheap()`. Но начальный указатель стека в таблице векторов нужен!

2.3.4. Восьмибайтовое выравнивание стека

Application Binary Interface (ABI) архитектуры ARM требует восьмибайтового выравнивания стека во всех внешних интерфейсах вроде всех вызовов между функциями из разных исходных файлов. Но во внутренних делах такого выравнивания не требуется.

То есть при прерываниях выравнивания не требуется. Ревизия 1 кристалла Cortex-M3 и позже может сам выравнивать стек при исключениях. Это разрешается установкой бита `STKALIGN` (бит 9) в регистре Configuration Control Register по адресу `0xE000ED14`.

В ревизии 0 кристалла Cortex-M3 компилятор может добавлять в обработчики IRQ коды выравнивания стека. Для этого обработчик должен иметь префикс `__irq` и использовать ключ компилятора `--cpu=Cortex-M3-rev0` вместо `–cpu=Cortex-M3`.

NB. Ключ `--cpu=Cortex-M3-rev0` поддерживается только RVCT 3.0 SP1 (build 586) и дальше.

2.4. Поддержка набора команд

Некоторые команды набора Cortex-M3 компилятор сам не строит, они в языке C не выражаются. Но их можно строить встроенным ассемблером или в ассемблерных функциях.

2.4.1. Команды доступа к памяти

В языке C невозможно построить команды с эксклюзивным доступом `LDREX` и `STREX`. Они используются, например, для мьютексов между потоками.

2.4.2. Команды барьера

Cortex-M3 поддерживает несколько команд барьера, обеспечивающих завершение определённых событий до начала других команд или событий.

Команда *Instruction Synchronization Barrier* (ISB) освобождает конвейер процессора так, что последующие команды выбираются из кэша или памяти после её завершения. Таким образом изменения в системе, например MPU, сразу вступают в действие.

Команда *Data Synchronization Barrier* (DSB) это специальный барьер доступа к памяти. Команда DSB завершится только после завершения всех отложенных видов доступа к памяти, запущенных до этой команды. Следующая команда будет загружена только после завершения команды DSB.

Команда *Data Memory Barrier* (DMB) это барьер памяти и немного отличается от команды DSB. Она обеспечивает завершение доступа к памяти, уже запущенного до выполнения команды DMB, до запуска доступа к памяти из команд, следующих за ней.

Ниже есть пример типичного использования этих команд с MPU. Эти микрофункции позже могут быть вставлены в виде команд компоновщиком.

Пример команд барьера

```
/* pseudo_intrinsics.c */
/* Small embedded assembly functions for barrier instructions*/
/* Link with armlink --inline ... */
__asm void __ISB(void) {
        ISB
        BX lr
}
__asm void __DSB(void) {
        DSB
        BX lr
}

/* scs.c - Initialize System Control Space registers */
void SCS_init(void)
{
/*
Code to configure the MPU regions inserted here …
*/
        /*  Enable the MPU */
        SCS.MPU.Ctrl |= 1;

        /* Force Memory Writes before continuing */
        __DSB();

        /* Flush and refill pipline with updated permissions */
        __ISB();
}
```

2.4.3. Условное выполнение

В отличие от команд ARM, большинство Thumb команд безусловные. Набор Thumb-2 к команде 16-бит условного перехода добавляет ещё три:

- 32-бит условный переход в диапазоне адресов +/- 1MB.
- 16-бит *сравнение и переход по нулю* (CBZ) и *сравнение и переход по не-нулю* (CBNZ) с переходом в диапазоне от +4 до +130 байт. Например, в C коде это может быть выход из цикла при счёте до нуля.
- 16-бит команда *if-then* (IT). Она делает до четырёх следующих команд условными. Часто используется вместо условного выполнения.

Ассемблер может автоматически генерировать нужные IT команды вместо условного выполнения.

2.4.4. Системные события

Есть несколько команд обработки системных событий, но они выполняются не всеми реализациями, а выполняются как NOP.

Команда	Действие	Описание
WFE	Wait For Event	Переводит процессор в режим низкого потребления до появления события. Программно вмешиваться нельзя.
WFI	Wait For Interrupt	Переводит процессор в режим низкого потребления до появления прерывания. Программно вмешиваться нельзя.
SEV	Send Event	Посылает событие всем процессорам многопроцессорной системы.

2.5. Атомарный доступ (bit-banding)

Атомарный доступ адресует целое слово в памяти (синонимов) на один бит в слове региона отображения. Например, запись в слово синонима очистит/поставит соответствующий бит в отображаемом регионе.

Так обеспечивается доступ со словным выравниванием одной командой `LDR` к отдельным битам. Теперь в языке С можно не использовать команды чтения-модификации-записи.

2.5.1. Трансляция адресов

Cortex-M3 имеет два 32MB региона, которые отображены на два 1MB региона битового отображения. Эти регионы раздельные, один в SRAM и один в области периферии.

Каждый бит регионе отображения последовательно адресуется на слова в 32MB регионе синонимов. Например, к восьмому биту региона отображения можно обратиться по восьмому слову 32MB региона синонимов.

2.5.2. Чтение и запись в регион отображения

Запись в бит [0] 32-разрядного слова региона синонимов изменяет бит в отображаемом регионе. Чтение слова региона синонимов возвращает значение нужного бита отображаемого региона в младшем бите результата, остальные биты очищаются.

К отображаемым регионам можно обращаться и обычном путём со словным, полусловным и байтовым доступом.

2.5.3. Атомарный доступ на С.

Можно использовать макросы препроцессора С.

Пример на языке С

```
#define BITBAND_SRAM_REF 0x20000000
#define BITBAND_SRAM_BASE 0x22000000
// Convert SRAM address
#define BITBAND_SRAM(a,b) ((BITBAND_SRAM_BASE + (a-BITBAND_SRAM_REF)*32 + (b*4)))
#define BITBAND_PERI_REF 0x40000000
#define BITBAND_PERI_BASE 0x42000000
// Convert PERI address
#define BITBAND_PERI(a,b) ((BITBAND_PERI_BASE + (a-BITBAND_PERI_REF)*32 + (b*4)))
#define MAILBOX 0x20004000
#define TIMER 0x40004000
// Mailbox bit 0
#define MBX_B0 *((volatile unsigned int *)(BITBAND_SRAM(MAILBOX,0)))
// Mailbox bit 7
#define MBX_B7 *((volatile unsigned int *)(BITBAND_SRAM(MAILBOX,7)))
// Timer bit 0
#define TIMER_B0  *((volatile unsigned char *)(BITBAND_PERI(TIMER,0)))
// Timer bit 7
#define TIMER_B7 *((volatile unsigned char *)(BITBAND_PERI(TIMER,7)))

int main(void) {
unsigned int temp = 0;
        MBX_B0 = 1;          // Word write
        temp = MBX_B7;       // Word read
        TIMER_B0 = temp;     // Byte write
        return TIMER_B7;     // Byte read
}
```

2.6. Режимы выполнения

2.6.1. Режимы работы

Cortex-M3 поддерживает Привилегированный (с полным доступом) и Пользовательский (доступ ограничен) режимы работы. Ограничения касаются некоторых команд, вроде `MSR`, доступа к системным устройствам и конфигурации MPU.

Процессор поддерживает два режима работы, Потока и Обработчика. Режим Потока включается по сбросу и нормально при выходе из исключения. В этом режиме выполнение может быть Привилегированным и Непривилегированным (Пользовательским).

Режим Обработчика включается при входе в исключение. Здесь выполнение всегда Привилегированное.

Переключение между Привилегированным Потоком и Пользовательским Потоком выполняется изменением значения `EXC_RETURN` в `R14` при выходе из исключения. Из Привилегированного в Пользовательский режим можно перейти очисткой бита `CONTROL[0]` командой `MSR`. Однако обратно можно переключиться только из исключения, например, командой `SVC`.

2.6.2. Стеки (Главный и Процесса)

Cortex-M3 поддерживает два стека (Главный и Процесса). Для этого у Cortex-M3 есть два указателя стека (`R13`). Единовременно виден только один `R13`. Но командам `MRS` и `MSR` доступны оба указателя стека.

Главный стек используется при сбросе и всегда используется в режиме Обработчика. Стек процесса доступен как текущий указатель стека только в режиме Потока. Указатель стека для режима Потока выбирается изменением `EXC_RETURN` при возврате из режима Обработчика или записью в `CONTROL[1]` командой `MSR` из Привилегированного Потока.

NB. Указатель стека процесса нужно инициализировать при переключении контекстов или инициализации программы.

2.7. Вызовы супервизора (SVC)

Обычно `SVC` (бывшая `SWI`) используется когда надо выполнить привилегированные действия или обратиться к системным ресурсам.

В команде `SVC` хранится её номер, используемый для выяснения смысла запроса с операционной системе. В предыдущих ядрах ARM код `SVC` надо было извлекать используя регистр возврата, а аргументы команды `SVC` уже были в регистрах `R0` — `R3`.

На Cortex-M3 регистры аргументов сохраняются в стеке, поскольку следующее прерывание, возникающее до выполнения первой команды обработчика, может разрушить содержимое регистров `R0` — `R3`. Возвращаемое значение также должно передаваться через содержимое стека. Для этого в начале обработчика `SVC` нужен кусочек ассемблерного кода. Он определяет стек с сохранными регистрами, извлекает номер `SVC` и передаёт его и ссылку на аргументы главному обработчику, написанному на C.

Пример обработчика `SVC` по значению `EXC_RETURN` определяет использовавшийся стек. В большинстве систем в этом нет нужды, поскольку в типичных системах используется вызов из кода со стеком процесса. В этом случае ассемблерный код может состоять из одной команды `MSR` с последующим переходом на тело обработчика на языке C.

Пример обработчика на языке C

```
__asm void SVCHandler(void) {
IMPORT SVCHandler_main
        TST         lr, #4
        MRSEQ       r0, MSP
        MRSNE       r0, PSP
        B SVCHandler_main
}
```

```
void SVCHandler_main(unsigned int * svc_args) {
unsigned int svc_number;
/*
* Stack contains:
* r0, r1, r2, r3, r12, r14, the return address and xPSR * First argument (r0) is svc_args[0]
*/
        svc_number = ((char *)svc_args[6])[-2];
        switch(svc_number)
        {
        case SVC_00:        /* Handle SVC 00 */
                break;
        case SVC_01:        /* Handle SVC 01 */
                break;
        default:            /* Unknown SVC */
                break;
        }
}
```

А вызывать можно используя ключевое слово компилятора `__svc`.

Пример вызова из языка C

```
#define SVC_00 0x00
#define SVC_01 0x01
void __svc(SVC_00) svc_zero(const char *string);
void __svc(SVC_01) svc_one(const char *string);
int call_system_func(void)
{
        svc_zero("String to pass to SVC handler zero");
        svc_one("String to pass to a different OS function");
}
```

2.8. Системный таймер (SysTick)

2.8.1. О SysTick

SCS содержит системный таймер (SysTick), обычно используемый операционной системой. SysTick можно программно читать или получать от него прерывания. Он имеет свою строку в таблице векторов и свой обработчик.

Конфигурируется SysTick через четыре регистра.

Имя	Адрес	Описание
SysTick Control and Status	0xE000E010	Управление SysTick - разрешение, источник, прерывания
SysTick Reload Value	0xE000E014	Перезагружаемое значение Current Value про достижении 0
SysTick Current Value	0xE000E018	Текущее значение счётчика
SysTick Calibration Value	0xE000E01C	Может содержать число тиков для генерации 10 ms интервала и прочее, зависящее от реализации

2.8.2. Конфигурация SysTick

Для этого нужно записать в регистр *SysTick Reload Value* интервал между событиями SysTick. При переходе счётчика из 1 в 0 ставится бит `COUNTFLAG` (в регистре *SysTick Control and Status*) или вызывается прерывание и данное значение перегружается в счётчик. То есть он активируется каждые *n+1* тиков. Например, если требуется делитель 100, то в *SysTick Reload Value* нужно записывать число 99. *SysTick Reload Value* поддерживает значения от 1 до `0x00FFFFFF`.

Регистр *SysTick Calibration Value* помогает масштабировать значение для регистра *SysTick Reload Value*. Он доступен только по чтению. Поле `TENMS` (биты 0 — 23) содержит число импульсов для интервала 10ms. Также бит `SKEW` (бит 30) показывает, что число импульсов в поле `TENMS` для интервала 10ms неточно из-за колебаний тактовой частоты. Бит 31 показывает, что используется опорная частота.

Регистр *Control and Status* позволяет выбрать между чтением флага `COUNTFLAG` (бит 16) и генерацией прерывания.

По умолчанию SysTick установлен в режим опроса. В этом режиме нужно читать регистр *Control and Status*, где установленный бит `COUNTFLAG` показывает истечение периода таймера. Чтение регистра *Control and Status* очищает бит `COUNTFLAG`. Для генерации прерывания нужно установить бит `TICKINT` (бит 1 регистра *SysTick Control and Status*). Кроме того, нужно разрешить соответствующее прерывание в NVIC. Запись единицы в `CLKSOURCE` (бит 2) выбирает такты ядра, а 0 внешнюю опорную частоту.

Таймер включается установкой бита 0 в регистре *SysTick Status and Control*.

2.9. Опции RVCT 3.0

2.9.1. Опции компилятора и ассемблера

По умолчанию RVCT 3.0 строит код для ядра ARMv4T, так что нужно обязательно указывать ваше ядро опцией `--cpu name`, где `name` равно `Cortex-M3` или `Cortex-M3-rev0`. Опцию `--thumb` добавлять не нужно, она и так работает.

2.9.2. Опции компоновщика

Специальных опций для компоновщика нет, он определяет всё по объектным файлам в командной строке.

Однако мы рекомендуем использовать файл размещения, (`--scatter filename`), поскольку в опциях компоновщика определить карту памяти для Cortex-M3 ой как не просто. Также нужно указать точку входа ключом `--entry` switch. Обычно это те же имя, что и в таблице векторов.

3. Перенос проекта ARM на Cortex-M3

В примере приведена миграция с ARM7TDMI на платформу Cortex M-3.

Лучшая стратегия это пошаговое наращивание функциональности с минимальной версии используя комментарии или макросы препроцессора.

3.1. Общая содификация кода

Большинство платформ-независимых секций в модификации не нуждаются, но кое-что всё-таки менять придётся.

3.1.1. Изменения в C коде

Для начала нужно перекомпилировать весь код с подходящей опцией `--cpu`, включая сторонние библиотеки.

Старый код Thumb-1 двоично совместим с Cortex-M3 и может выполняться как есть. Однако, компоновщик RVCT 3.0 может прикомпоновать файлы с командами ARM, так что благоразумно перекомпилировать всё, что можно, для Thumb-2, да и вручную перепроверить библиотеки на наличие команд ARM.

Сам код тоже может потребовать изменений, в частности, некоторые директивы (`#pragma arm` и `#pragma thumb`) нужно удалить. Кроме того, построчный ассемблер не поддерживает Thumb-2, и нужно переписывать.

3.1.2. Изменения в ассемблере

Тут нужно быть внимательным.

Директивы для команд ARM (`ARM` или `CODE32`) нужно удалить или заменить на `THUMB`. Если правильно выбрана опция `--cpu`, то всё будет хорошо, если не использовались некоторые редкие команды ARM.

Если есть директивы `CODE16`, замену их на `THUMB` должно оттранслировать без предупреждений, поскольку `CODE16` использует правила синтаксиса Thumb-1. Например, в правилах синтаксиса `CODE16` команды без суффикса `S` превращаются в знаковые варианты, а синтаксис `THUMB` требует наличия суффикса `S`.

Ассемблер по мере необходимости вставляет команды `IT` перед условными командами. Например, одинокая команда `ADDSNE r0,r0,r1` (с последующей безусловной командой) превратится в: `IT NE` с последующей `ADDS r0,r0,r1`.

Будьте осторожны со специфичными для ядра и архитектуры командами. Это:

- Команды сопроцессора.
- Изменение состояния или режимов при доступе к `PSR`.
- Команда `SWP`, заменённая на `LDREX` и `STREX`.
- Некоторые режимы адресации не поддерживаются командами `LDM` и `STM`.
- Дополнительные ограничения в Thumb-2, например команды `LDR` и `STR` с регистровым и непосредственным смещением.

Может понадобиться переписать эти части кода под модель Cortex-M3. На несовместимости ассемблер выдаст предупреждения. В Cortex-M3 управляющие регистры часто отображены в памяти и поддерживаемые режимы сильно отличаются от ARM7TDMI. Коды изменения режимов должны быть изменены или удалены. Также коды доступа к сопроцессору должно удалить, если они не эмулируются в обработчике, исключения Usage Fault.

Будьте осторожнее с использованием PC в адресной арифметике. Поскольку в ней Thumb-2 использует смешанные 16- и 32-бит команды, значение PC всегда указывает на текущую команду плюс 4.

3.2. Изменения в стартовом коде

Стартовый код состоит из кода сброса и инициализации приложения и аппаратуры перед запуском основного тела программы. Он зависит от ядра и назначения.

В простой программе бывает достаточно использовать точку входа из библиотеки C (`__main()`) в векторе сброса и выполнять инициализацию из функции `main()`. Но при наличии аппаратуры, требующей критической инициализации, перед входом в `__main()` может понадобиться крохотная ассемблерная программка. Также будьте внимательны при работе с некоторыми устройствами, например с MPU, где нужны команды барьера, и при работе с MPU или MMU связанный код нужно пересмотреть.

Для всех проектов Cortex-M3 нужно создать новую таблицу векторов с адресами стека и обработчика сброса.

3.3. Изменения в обработке исключений

Для Cortex-M3 обработчики исключений нужно адаптировать.

Ассемблерный обработчик нижнего уровня обычно не нужен, ядро само обеспечивает повторную входимость. Если обработчик нижнего уровня чего-то там выполняет, то лучше выделить это в функции и соваться туда.

Не забудьте отметить обработчик IRQ ключевым словом `__irq` для порядка в выравнивании стека и в мозгах.

У Cortex-M3 нет входа FIQ. Периферию с таким сигналом надо переместить на приоритет повыше или вообще на NMI и обязательно просмотреть обработчики этих сигналов на использование регистров FIQ, их теперь надо совать в стек.

И в конце концов для инициализации NVIC нужна новая функция. Теперь разрешать прерывания можно и до входа в головную функцию.

3.3.1. Критические секции и исключения

На Cortex-M3 приоритеты, вложенность исключений и сохранение регистров полностью обеспечиваются ядром. То есть после входа в обработчик исключения прерывания остаются разрешёнными. Кроме того, если при выходе из исключения прерывания запрещены, то процессор их автоматически не разрешит. Запрещенные в обработчике прерывания разрешать нужно самим непосредственно перед выходом из исключения, но оно может возникнуть и перед самой командой выхода.

Такое дело может повлиять на работу критических секций, для которых работа с выключенными прерываниями жизненно необходима, например, переключение контекстов в ОС. Старые коды могут полагаться на автоматическое запрещение прерываний при входе в обработчик, а этого нельзя оставлять без внимания и придётся переписывать обработчики.

3.4. Переход на новое устройство

Для этого нужны дополнительные шаги, например, определение новой карты памяти в файле размещения. Это создание структуры для регистров в System Control Space и пространства для стека и динамической памяти. Если вы используете таймер, то может понадобиться изменение кода для SysTick.

3.5. Свойства Cortex-M3

Большинство новых команд ARMv7M будут использованы автоматически при перекомпиляции для Thumb-2. Однако кое-что надо переделывать вручную, это:

- Изменить код инициализации для включения MPU.
- Обеспечить использование режимов низкого потребления.
- Рассмотреть использование атомарного доступа для ускорения работы с битами памяти и периферии.

4. Отладка на Cortex-M3

Возможности такие:

- Установка контрольных точек в флэш и ROM регионах.
- Поддержка контроля данных и триггеров ETM.
- Необязательный ETM для отслеживания команд и данных.
- *Instrumentation Trace Macrocell* (ITM), программное отслеживание событий в стиле printf.

Кроме того, доступ отладчика ведётся через *Debug Access Port* (DAP) через матрицу шин к работающему процессору. Например, для доступа к внешней памяти не надо останавливать процессор. RVD 3.0 поддерживает это при подключении к аппаратуре, но он не поддерживает функции слежения Cortex-M3.

RVDS 3.0 также включает *Instruction Set System Model* (ISSM) в Cortex-M3. Это симуляция ядра, NVIC и других качеств процессора. Также моделируется один UART и три дополнительных таймера. Но модель симулирует только контрольные точки команд и данных. Компоненты слежения не моделируются.