Навигация

Глава III
Система тактирования, аппаратные прерывания и таймеры

Цель работы:

– настройка системы тактирования процессорного ядра;
– изучение аппаратных таймеров общего назначения;
– изучение сторожевого таймера;
– получение навыков использования аппаратных прерываний.

Оборудование:

– отладочный комплект для микроконтроллера 1986ВЕ92У;
– программатор-отладчик J-LINK (или аналог);
– персональный компьютер.

Программное обеспечение:

– операционная система Windows 7 / 8 / 10;
– среда программирования Keil µVision MDK-ARM 5.24;
– драйвер программатора J-LINK;
– примеры кода программ.

3.1. Система тактирования процессорного ядра

Система тактирования является одним из ключевых элементов микроконтроллера. Тактовая частота определяет производительность ядра, а также обеспечивает его синхронизацию со всеми периферийными блоками. Если какой-то из блоков не получает тактовых сигналов, то он просто бездействует. Поэтому работу с микроконтроллером всегда следует начинать с настройки системы тактирования.

Источниками тактовых импульсов в микроконтроллерах являются встроенные генераторы, а получение требуемых частот обеспечивается с помощью специальной схемы формирования тактовой частоты.

3.1.1. Генераторы тактовых импульсов

Микроконтроллеры серии 1986ВЕ9х могут тактироваться от четырех различных генераторов, реализованных внутри микросхемы.

1. Высокоскоростной внутренний RC-генератор (англ. High Speed Internal, **HSI**). Типовое значение тактовой частоты данного генератора составляет 8 МГц. В частности, значение может быть выбрано из диапазона 6.4...12.0 МГц с помощью подстройки частоты (рисунок 3.1).

Генератор автоматически запускается при появлении питания и **по умолчанию ядро микроконтроллера тактируется именно от него**. В дальнейшем можно переключиться на тактирование от других источников, а HSI отключить.

Достоинством HSI является отсутствие необходимости во внешних элементах: генератор расположен внутри кристалла. К недостаткам следует отнести низкую температурную и временную стабильность: при изменении температуры окружающей

среды происходят флуктуации частоты, вносящие ошибки в измерение временных промежутков. По этой причине HSI обычно используют только в простых и недорогих системах.

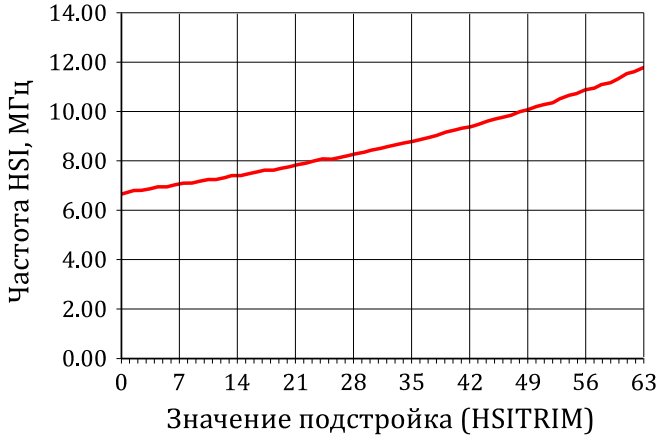

Рисунок 3.1 – Зависимость частоты HSI от значения регистра подстройки (HSITRIM)

2. Низкоскоростной внутренний RC-генератор (англ. Low Speed Internal, **LSI**). Типовое значение частоты – 40 КГц. Программная подстройка частоты позволяет выбрать значение из диапазона 32...70 КГц (рисунок 3.2).

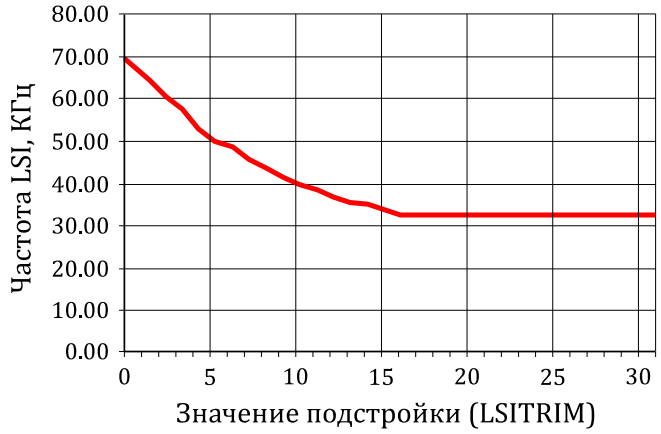

Рисунок 3.2 – Зависимость частоты LSI от значения регистра подстройки (LSITRIM)

Генератор также автоматически запускается при появлении питания и в начале работы используется для некоторых системных целей. Далее его можно отключить программным путем.

3. Высокоскоростной генератор, стабилизированный внешним кварцевым резонатором (англ. High Speed External, **HSE**). Генератор позволяет вырабатывать тактовую частоту в диапазоне 2...16 МГц. При этом частота задается и стабилизируется специальным элементом – **кварцевым резонатором** (рисунок 3.3). В обиходе его называют просто «кварцем».

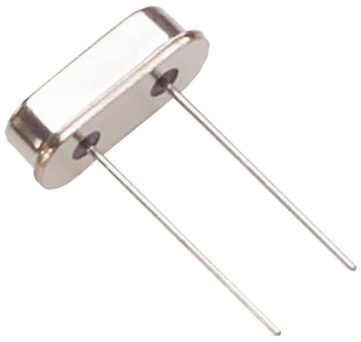

Рисунок 3.3 – Кварцевый резонатор

Кварцевый резонатор надежно стабилизирует частоту, не давая ей существенно изменяться под воздействием внешних факторов. По этой причине в серьезных приборах кварцевые резонаторы обязательно используют для тактирования процессоров. На отладочной плате также присутствует кварцевый резонатор; он расположен рядом с микроконтроллером, либо на обратной стороне платы (в зависимости от ревизии изделия).

Основным параметром любого кварцевого резонатора является **резонансная частота**. Ее обычно пишут прямо на корпусе элемента. Например, может быть написано «8.000 МГц». При этом частота гарантируется с точностью половины младшего разряда (±500 Гц).

На рисунке 3.4 показан фрагмент электрической схемы отладочной платы, относящийся к подключению кварцевых резонаторов.

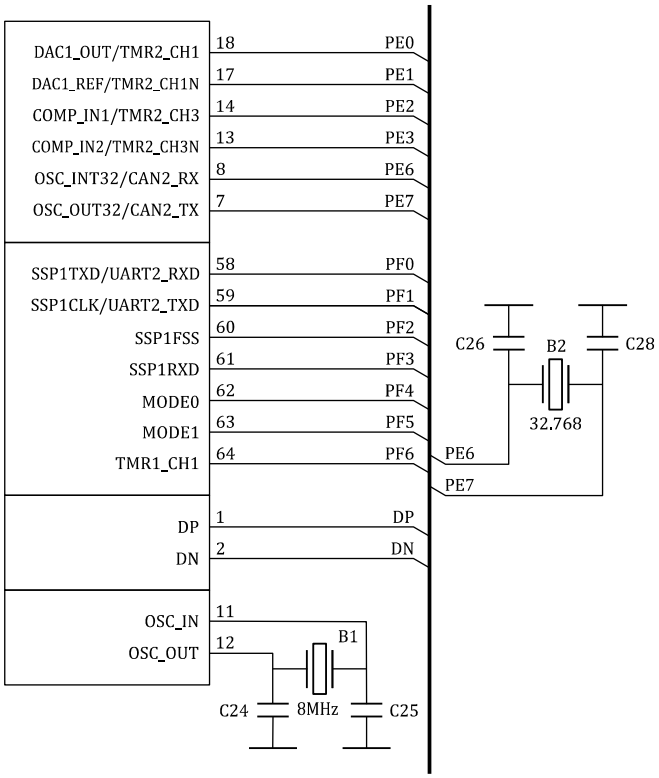

Рисунок 3.4 – Схема подключения кварцевых резонаторов к микроконтроллеру

На схеме присутствуют сразу два кварцевых резонатора: B1 и B2. Генератор HSE использует кварцевый резонатор B1. Рядом с ним расположены небольшие (по емкости и габаритам) конденсаторы C24 и C25. Они необходимы для надежного запуска генератора.

4. Низкоскоростной генератор, стабилизированный внешним кварцевым резонатором (англ. Low Speed External, **LSE**). Генератор LSE вырабатывает частоту 32 768 Гц и предназначен в основном для тактирования часов реального времени. Странное, на первый взгляд, значение частоты объясняется просто: для получения частоты 1 Гц (с целью отсчета секунд) достаточно поделить исходную частоту на 2^{15}, что очень просто сделать как аппаратными, так и программным средствами. Генератор LSE может работать при отсутствии основного питания при условии использования батарейного домена.

Итак, из данного раздела можно заключить, что использование генератора, стабилизированного кварцевым резонатором, для тактирования процессорного ядра позволяет повысить надежность и качество системы. Именно такой генератор и будет использован в рамках данного практикума.

3.1.2. Формирование тактовой частоты

Как можно заметить, генераторы обеспечивают сравнительно невысокие тактовые частоты; в частности, резонансная частота кварцевого резонатора B1 равна 8 МГц. Для получения иных частот используется схема формирования тактовой частоты (рисунок 3.5), состоящая из мультиплексоров (MUX), делителей (/) и блоков **фазовой автоподстройки частоты** (англ. Phase-Locked Loop, **PLL**), которые позволяют значительно повысить тактовую частоту путем умножения ее исходного значения на коэффициент в диапазоне 2...16.

В качестве примера рассмотрим, как получить тактовую частоту 80 МГц, используя генератор HSE. Подобная настройка системы тактирования реализована в общем модуле *clk.c* в виде процедуры с сигнатурой `void CPU_Init(void)` и в дальнейшем используется во всех программных проектах.

Итак, для получения тактовой частоты 80 МГц достаточно умножить исходную частоту 8 МГц на 10. На это и будем ориентироваться.

В первую очередь рекомендуется выполнить сброс системы тактирования:

```
// Сброс настроек системы тактирования
MDR_RST_CLK->PLL_CONTROL    = 0;
MDR_RST_CLK->HS_CONTROL     = 0;
MDR_RST_CLK->CPU_CLOCK      = 0;
MDR_RST_CLK->USB_CLOCK      = 0;
MDR_RST_CLK->ADC_MCO_CLOCK = 0;
MDR_RST_CLK->RTC_CLOCK      = 0;
MDR_RST_CLK->CAN_CLOCK      = 0;
MDR_RST_CLK->TIM_CLOCK      = 0;
MDR_RST_CLK->UART_CLOCK     = 0;
MDR_RST_CLK->SSP_CLOCK      = 0;

// Отключение тактирования всех блоков,
// кроме системы тактирования и батарейного домена
MDR_RST_CLK->PER_CLOCK = (1 << RST_CLK_PCLK_RST_CLK_Pos)
                       | (1 << RST_CLK_PCLK_BKP_Pos);
```

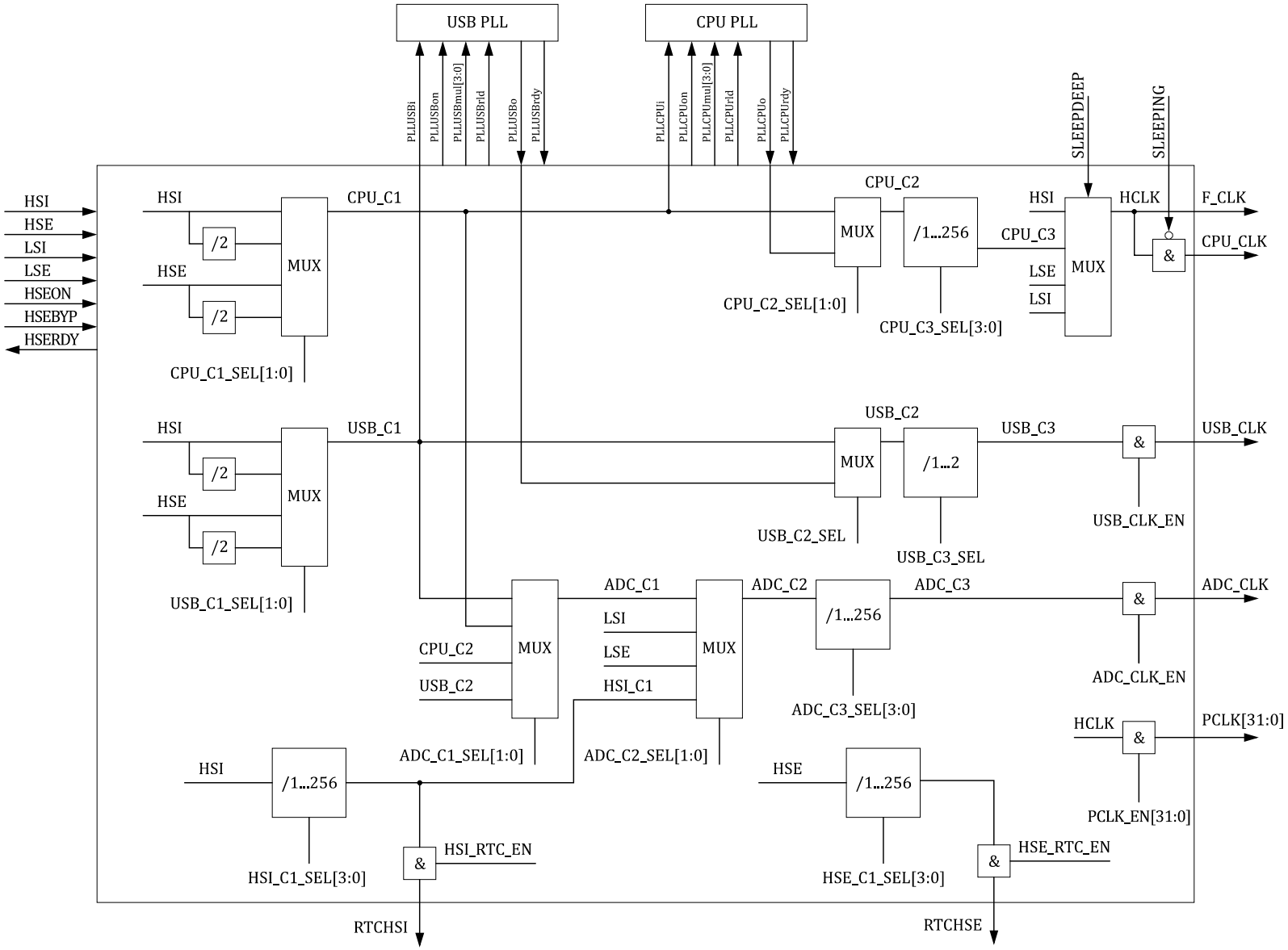

Рисунок 3.5 – Структурная схема формирования тактовой частоты

Далее следует запустить генератор HSE. Генератор запускается не мгновенно, поэтому обязательно нужно дождаться его входа в рабочий режим путем проверки соответствующего флага:

```
// Включение генератора HSE
MDR_RST_CLK->HS_CONTROL = (1 << RST_CLK_HS_CONTROL_HSE_ON_Pos);

// Ожидание входа генератора HSE в рабочий режим
while (MDR_RST_CLK->CLOCK_STATUS & RST_CLK_CLOCK_STATUS_HSE_RDY == 0);
```

Затем нужно инициализировать блок *CPU PLL*. Процесс инициализации состоит из выбора коэффициента умножения и включения блока. Входная частота блока должна принадлежать диапазону 2...16 МГц, а выходная не должна превышать 100 МГц. Как и в случае с генератором, блок *CPU PLL* требует некоторое время на стабилизацию.

```
// Инициализация блока PLL
MDR_RST_CLK->PLL_CONTROL =
    (1 << RST_CLK_PLL_CONTROL_PLL_CPU_ON_Pos)    // Включение блока PLL
  | (9 << RST_CLK_PLL_CONTROL_PLL_CPU_MUL_Pos);  // Коэф. умножения (9 + 1 = 10)

// Ожидание входа блока PLL в рабочий режим
while (MDR_RST_CLK->CLOCK_STATUS & RST_CLK_CLOCK_STATUS_PLL_CPU_RDY == 0);
```

Последующий процесс настройки системы тактирования проиллюстрирован рисунком 3.6; целевые сигналы на нем выделены красным цветом.

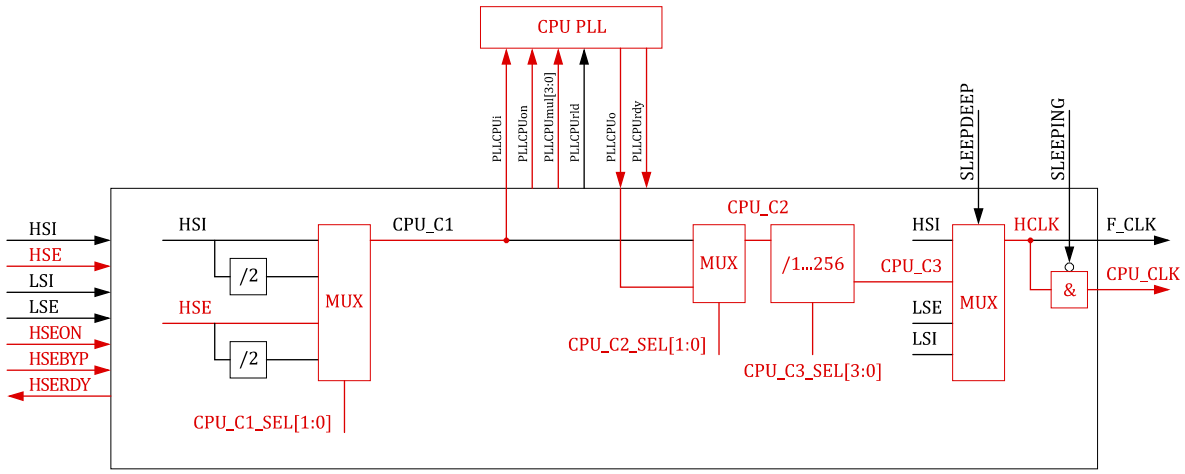

Рисунок 3.6 – Схема настройки системы тактирования

В первую очередь выбирается сигнал *CPU_C1*. В данном случае он должен соответствовать сигналу генератора HSE с частотой 8 МГц.

Сигнал *CPU_C2* должен быть получен с выхода блока CPU PLL. После операции умножения частота сигнала составит 80 МГц.

Сигнал *CPU_C3* зависит от выбранного значения делителя из ряда 1, 2, 4, 8, 16, 32, 64, 128, 256. Поскольку требуемое значение частоты уже получено, то следует выбрать делитель равный единице.

Делитель позволяет понизить частоту тактирования. Понижение частоты может применяться для уменьшения энергопотребления системы, т.к. ток потребления системы снижается почти пропорционально с понижением тактовой частоты ядра.

Последний мультиплексор определяет итоговый сигнал, который должен соответствовать ранее сформированному сигналу *CPU_C3*. Возможны также варианты использования одного из трех генераторов (с целью пропуска тракта).

С этого момента процессорное ядро начнет работать на частоте 80 МГц, поскольку сигнал *CPU_C3* заменит собой сигнал с генератора HSI, который до сих пор использовался.

Конфигурация описанного выше тракта аппаратно реализована в регистре `MDR_RST_CLK->CPU_CLOCK` (таблица 3.1):

Таблица 3.1 – Описание регистра `MDR_RST_CLK->CPU_CLOCK`

№ битов	**Функциональное имя битов**	**Дескрипция**
31…10	–	–
9…8	HCLK_SEL[1:0]	Источник сигнала *HCLK*: 00 – *HSI*; 01 – *CPU_C3*; 10 – *LSE*; 11 – *LSI*.
7…4	CPU_C3_SEL[3:0]	Источник сигнала *CPU_C3* (делитель сигнала *CPU_C2*): 0XXX – *CPU_C2*; 1000 – *CPU_C2* / 2; 1001 – *CPU_C2* / 4; 1010 – *CPU_C2* / 8; … 1111 – *CPU_C2* / 256.
3	–	–
2	CPU_C2_SEL	Источник сигнала *CPU_C2*: 0 – *CPU_C1;* 1 – *PLLCPUo*.
1…0	CPU_C1_SEL[1:0]	Источник сигнала *CPU_C1*: 00 – *HSI*; 01 – *HSI* / 2; 10 – *HSE*; 11 – *HSE* / 2.

```
// Настройка тракта формирования тактовой частоты
MDR_RST_CLK->CPU_CLOCK =
    (2 << RST_CLK_CPU_CLOCK_CPU_C1_SEL_Pos)  // CPU_C1 = HSE
  | (1 << RST_CLK_CPU_CLOCK_CPU_C2_SEL_Pos)  // CPU_C2 = PLLCPUo
  | (0 << RST_CLK_CPU_CLOCK_CPU_C3_SEL_Pos)  // CPU_C3 = CPU_C2
  | (1 << RST_CLK_CPU_CLOCK_HCLK_SEL_Pos));  // HCLK   = CPU_C3
```

После этого рекомендуется отключить внутренние генераторы для снижения энергопотребления:

```
// Отключение генераторов HSI и LSI
MDR_BKP->REG_0F &= ~((1 << BKP_REG_0F_LSI_ON_Pos)
                   |   (1 << BKP_REG_0F_HSI_ON_Pos));
```

В заключение процесса настройки тактирования необходимо вычислить новое значение тактовой частоты, используя следующую процедуру:

```
// Обновление значения тактовой частоты
SystemCoreClockUpdate();
```

Это очень важно при использовании операционной системы реального времени, так как ее временная база опирается на изменяемую процедурой переменную `SystemCoreClock`.

Управление тактированием периферийных блоков осуществляется через регистр `MDR_RST_CLK->PER_CLOCK`. Каждый бит регистра соответствует некоторому блоку (таблица 3.2), при этом значения битов интерпретируются таким образом:

– 0 – тактирование блока отключено;

– 1 – тактирование блока включено.

Тактирование всех периферийных блоков (за исключением блока *RST_CLK*) по умолчанию отключено, поэтому для работы с каждым отдельным блоком необходимо обеспечить его тактовыми импульсами *HCLK*, записав единицы в соответствующие биты регистра `MDR_RST_CLK->PER_CLOCK`. Так, например, для включения тактирования портов A и D можно написать:

```
MDR_RST_CLK->PER_CLOCK |= (1 << RST_CLK_PCLK_PORTA_Pos)   // Тактирование порта A
                        | (1 << RST_CLK_PCLK_PORTD_Pos);  // Тактирование порта D
```

Опыт аппаратного программирования однозначно показывает, что если какой-либо **периферийный блок не работает**, то первым делом следует **проверить наличие его тактирования**.

Стоит также отметить, что не следует включать тактирование неиспользуемых блоков, так как это ведет к увеличению тока, потребляемого микроконтроллером.

*
**

Обратите внимание, что при конфигурации системы тактирования постоянно использовались **позиционные константы битовых полей**, разговор о которых поднимался в разделе 2.3. Эти константы содержатся в глобальном заголовочном файле *1986VE9x.h*. Таких констант очень много: они предусмотрены для всех битовых полей каждого регистра микроконтроллера.

Чтобы узнать значение константы (да и любого другого программного элемента – переменной, функции, макроса) можно кликнуть по нему правой кнопкой мыши и выполнить команду *Go To Definition Of '...'*; это перенесет текстовый курсор в область определения элемента. Данная процедура работает **только после компиляции проекта**.

Таблица 3.2 – Управление тактирование периферийных блоков;
описание регистра `MDR_RST_CLK->PER_CLOCK`

№ битов	**Функциональное имя битов**	**Дескрипция (периферийный блок)**
0	PCLK_EN[31:0]	CAN1
1		CAN2
2		USB
3		EEPROM_CNTRL
4		RST_CLK
5		DMA
6		UART1
7		UART2
8		SPI1
9		–
10		I2C
11		POWER
12		WWDG
13		IWDG
14		TIMER1
15		TIMER2
16		TIMER3
17		ADC
18		DAC
19		COMP
20		SPI2
21		PORTA
22		PORTB
23		PORTC
24		PORTD
25		PORTE
26		–
27		BKP
28		–
29		PORTF
30		EXT_BUS_CNTRL
31		–

3.2. Аппаратные прерывания

Прерывание (англ. Interrupt) – это изменение нормального хода выполнения программы с целью обработки какого-либо события. Если прерывание инициировано на аппаратном уровне, то оно, соответственно, именуется **аппаратным**. Когда какое-то устройство формирует аппаратное прерывание, процессор останавливает программу, выполняет специальную функцию (обработку прерывания) и возвращается в точку останова. Такой подход обеспечивает обработку приоритетных событий с минимальной задержкой. Так, например, системы повышенной надежности используют прерывание при аварии в питающей сети для выполнения процедур записи содержимого регистров в энергонезависимую память, чтобы после восстановления питания продолжить работу с того же места.

Периферийные устройства способны только формировать запрос на прерывание, а управление ими осуществляется **контроллером вложенных векторных прерываний** (англ. Nested Vector Interrupt Controller, **NVIC**). Данный контроллер позволяет задать каждому прерыванию некоторый приоритет, в соответствии с которым будет определяться порядок обработки запросов на прерывания, поступивших одновременно. Приоритет может принимать значения от 0 до 7, при этом 0 – самый высокий приоритет, 7 – самый низкий. При равных значениях приоритетов прерывания обрабатываются по принципу FIFO.

Для каждого периферийного блока предусмотрен только один обработчик прерываний. Однако переход в этот обработчик может быть инициирован по разным причинам. Например, для некоторого интерфейса обмена данными прерывания могут происходить при завершении приема, завершении передачи или возникновении ошибки на линии. Возможные причины прерываний закладываются на аппаратном уровне в виде битовых флагов в определенном регистре. При этом нередко требуется формировать прерывание лишь при одном условии, а не всех возможных; в таком случае остальные прерывания могут быть **замаскированы**, тогда флаги, соответствующие этим прерываниям, будут игнорироваться. В микроконтроллерах все прерывания (кроме системных) по умолчанию замаскированы, поэтому для их использования потребуется снять маску. Делается это обычно установкой определенного бита в регистре конфигурации аппаратных прерываний конкретного периферийного блока.

Обработчик прерываний (англ. Interrupt Request (**IRQ**) **Handler**) – это обычная процедура с детерминированным именем (таблица 3.4). Ее можно разместить в любом месте программного проекта; обычно это делают в том же модуле, где описывается функция инициализации периферийного блока.

Обработчик должен быть кратким и простым; не следует помещать в него требовательные к вычислительной мощности операции, т.к. это приведет к снижению стабильности и надежности системы.

При входе в обработчик необходимо сбросить запрос на обработку прерывания. Это очень важно, потому что в противном случае система зависнет в обработчике.

Таблица 3.4 – Описание обработчиков аппаратных прерываний

№ п/п	Имя обработчика прерываний	Дескрипция
Системные		
1	Reset_Handler	Сброс системы
2	NMI_Handler	Немаскируемое прерывание
3	HardFault_Handler	Сбой обработки прерываний
4	MemManage_Handler	Защита памяти
5	BusFault_Handler	Сбой шины данных
6	UsageFault_Handler	Сбой выполнения инструкций
7	SVC_Handler	Функция Supervisor Call
8	PendSV_Handler	Запрос сервисов системного уровня
9	SysTick_Handler	Системный таймер
Периферийные		
10	CAN1_IRQHandler	Контроллер интерфейса CAN1
11	CAN2_IRQHandler	Контроллер интерфейса CAN2
12	USB_IRQHandler	Контроллер интерфейса USB
13	DMA_IRQHandler	Контроллер прямого доступа к памяти
14	UART1_IRQHandler	Контроллер интерфейса UART1
15	UART2_IRQHandler	Контроллер интерфейса UART2
16	SSP1_IRQHandler	Контроллер интерфейса SSP1
17	I2C_IRQHandler	Контроллер интерфейса I2C
18	POWER_IRQHandler	Детектор напряжения питания
19	WWDG_IRQHandler	Сторожевой таймер
20	Timer1_IRQHandler	Таймер 1 общего назначения
21	Timer2_IRQHandler	Таймер 2 общего назначения
22	Timer3_IRQHandler	Таймер 3 общего назначения
23	ADC_IRQHandler	Аналого-цифровой преобразователь
24	COMPARATOR_IRQHandler	Контроллер схемы компаратора
25	SSP2_IRQHandler	Контроллер интерфейса SSP2
26	BACKUP_IRQHandler	Батарейный домен
27	EXT_INT1_IRQHandler	Внешнее прерывание 1
28	EXT_INT2_IRQHandler	Внешнее прерывание 2
29	EXT_INT3_IRQHandler	Внешнее прерывание 3
30	EXT_INT4_IRQHandler	Внешнее прерывание 4

Например, процедура обработки прерывания от таймера 1, которая ведет подсчет количества перезагрузок таймера, выглядит следующим образом:

```
// Обработчик прерывания от таймера TIMER1
void Timer1_IRQHandler(void)
{
  // Сброс запроса на обработку прерывания
  MDR_TIMER1->STATUS &= ~(1 << TIMER_STATUS_CNT_ARR_EVENT_Pos);

  // Счетчик прерываний
  static uint32_t k;

  // Инкремент счетчика
  k++;
}
```

Исходя из вышесказанного, для использования аппаратных прерываний необходимо выполнить следующие действия:

1. Разрешить периферийному блоку направлять запросы на обработку прерываний.
2. Разрешить контроллеру NVIC обрабатывать прерывания от периферийного блока и задать их приоритет.
3. Сформировать процедуру обработки прерываний от периферийного блока.

Программная реализация указанных действий будет рассмотрена в разделе 3.3.2.

3.3. Аппаратные таймеры

В данном разделе речь пойдет о таком функциональном блоке микроконтроллера, как таймер. На деле этому блоку больше подходит название «счетчик», так как его основной функцией является подсчет входных импульсов.

Таймеры могут иметь разное назначение. При кажущейся тривиальности, таймеры обладают очень непростой структурой и предоставляют массу возможностей. В микроконтроллерах серии 1986ВЕ9х реализованы следующие таймеры:

– системный таймер *SysTick*;
– 3 таймера общего назначения (*TIMER1*, *TIMER2*, *TIMER3*);
– сторожевые таймеры *IWDG* и *WWDG*.

3.3.1. Системный таймер

Системный таймер (**SysTick**) является 24-разрядным вычитающим счетчиком с автоматической загрузкой начального значения при достижении нуля. В качестве тактовых импульсов таймера может быть использован либо сигнал *HCLK* (с частотой, в частном случае, равной частоте ядра), либо сигнал с генератора LSI. Таймер способен формировать аппаратное прерывание при каждой перезагрузке.

Данный таймер является примитивным, и обычно используется для создания задержек. В нашем случае, однако, этот таймер задействован операционной системой для формирования квантов времени. Из этого следует, что **недопустимо задействовать в проекте системный таймер при использовании ОСРВ**, т.к. это гарантированно приведет к сбою системы.

3.3.2. Таймеры общего назначения

В состав микроконтроллеров серии 1986ВЕ9х входит три 16-разрядных перезагружаемых таймера общего назначения. Все они имеют одинаковую структуру и функциональные возможности. Каждый таймер содержит 16-разрядный счетчик, 16-разрядный предделитель и 4-канальный блок захвата/сравнения. Счет таймеров может быть прямым, обратным или двунаправленным (прямым до определенного значения, затем обратным). Источниками событий для изменения счета таймера могут служить либо внутренние тактовые импульсы *HCLK*, либо внешние сигналы.

Иллюстрация примера работы таймера приведена на рисунке 3.7.

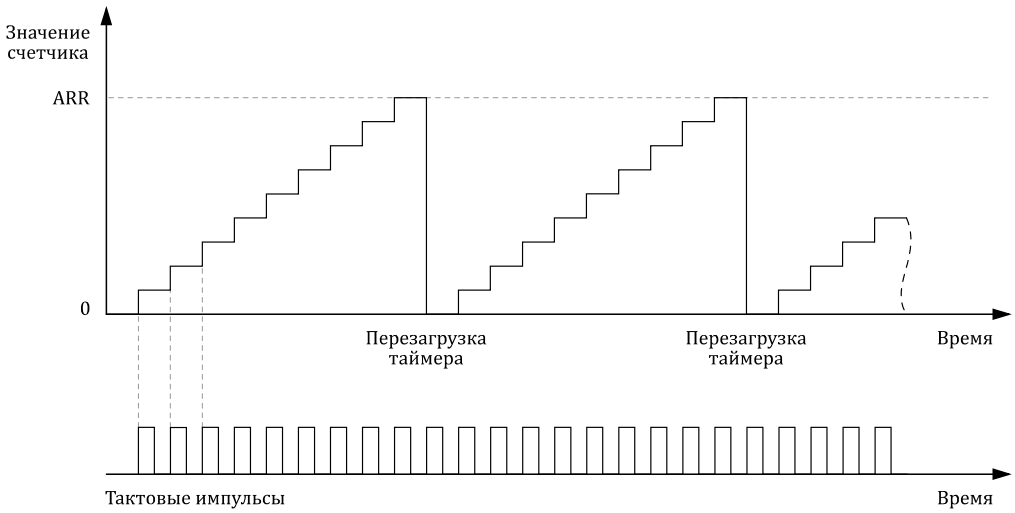

Рисунок 3.7 – Работа таймера в режиме прямого счета

В данной главе рассмотрим базовую функцию таймера – измерение временных интервалов.

Инициализация всех периферийных блоков, в том числе аппаратных таймеров, в общем случае имеет следующую структуру:

1. Включение тактирования.
2. Деинициализация.
3. Конфигурация.
4. Настройка аппаратных прерываний.

Разберем ее на примере инициализации первого таймера.

1. Включение тактирования таймера производится в двух регистрах:

```
// Включение тактирования таймера TIMER1
MDR_RST_CLK->PER_CLOCK |= (1 << RST_CLK_PCLK_TIMER1_Pos);
MDR_RST_CLK->TIM_CLOCK |= (1 << RST_CLK_TIM_CLOCK_TIM1_CLK_EN_Pos);
```

Регистр `MDR_RST_CLK->PER_CLOCK`, как уже говорилось, управляет включением периферийного блока в работу. Значение битового сдвига должно соответствовать таблице 3.1.

Регистр `MDR_RST_CLK->TIM_CLOCK` контролирует подачу тактовых импульсов на таймер и определяет делитель частоты этих импульсов (стр. 166 Спецификации).

2. Деинициализация таймера состоит из сброса значений всех его регистров, коих в целом достаточно много. В рамках данной главы нас будут интересовать только 6 из них, однако для надежной работы системы следует сбросить их все. По этой причине рекомендуется выносить деинициализацию в отдельную функцию:

```
// Деинициализация аппаратного таймера
void TIMER_Reset(MDR_TIMER_TypeDef *MDR_TIMERx)
{
  MDR_TIMERx->CNT          = 0;
  MDR_TIMERx->PSG          = 0;
  MDR_TIMERx->ARR          = 0;
  MDR_TIMERx->CNTRL        = 0;
  MDR_TIMERx->CCR1         = 0;
  MDR_TIMERx->CCR2         = 0;
  MDR_TIMERx->CCR3         = 0;
  MDR_TIMERx->CCR4         = 0;
  MDR_TIMERx->CCR11        = 0;
  MDR_TIMERx->CCR21        = 0;
  MDR_TIMERx->CCR31        = 0;
  MDR_TIMERx->CCR41        = 0;
  MDR_TIMERx->CH1_CNTRL    = 0;
  MDR_TIMERx->CH2_CNTRL    = 0;
  MDR_TIMERx->CH3_CNTRL    = 0;
  MDR_TIMERx->CH4_CNTRL    = 0;
  MDR_TIMERx->CH1_CNTRL1   = 0;
  MDR_TIMERx->CH2_CNTRL1   = 0;
  MDR_TIMERx->CH3_CNTRL1   = 0;
  MDR_TIMERx->CH4_CNTRL1   = 0;
  MDR_TIMERx->CH1_CNTRL2   = 0;
  MDR_TIMERx->CH2_CNTRL2   = 0;
  MDR_TIMERx->CH3_CNTRL2   = 0;
  MDR_TIMERx->CH4_CNTRL2   = 0;
  MDR_TIMERx->CH1_DTG      = 0;
  MDR_TIMERx->CH2_DTG      = 0;
  MDR_TIMERx->CH3_DTG      = 0;
  MDR_TIMERx->CH4_DTG      = 0;
  MDR_TIMERx->BRKETR_CNTRL = 0;
  MDR_TIMERx->STATUS       = 0;
  MDR_TIMERx->IE           = 0;
  MDR_TIMERx->DMA_RE       = 0;
}
```

3. Конфигурация таймеров выполняется через четыре регистра.

Регистр **`MDR_TIMER1->PSG`** содержит 16-битный предделитель тактовой частоты. Фактически, этот регистр определяет частоту изменения значения счетчика по формуле (3.1):

$$F_{TMR} = \frac{SystemCoreClock}{PSG + 1}. \tag{3.1}$$

Обратите внимание, что записываемое в регистр значение предделителя должно быть на единицу меньше желаемого. Чтобы помнить об этом и не запутаться, рекомендуется использовать следующую форму записи:

```
// Предделитель тактовой частоты таймера
MDR_TIMER1->PSG = 80 - 1;
```

В соответствии с формулой (3.1), таймер с таким предделителем будет изменять свое значение раз в одну микросекунду:

$$\Delta T = \frac{1}{F_{TMR}} = \frac{PSG + 1}{SystemCoreClock} = \frac{79 + 1}{80 \times 10^6 \text{ Гц}} = 10^{-6} \text{ с.}$$

Регистр **MDR_TIMER1->CNT** содержит 16-битное значение счетчика. Регистр может быть прочитан в любой момент для получения текущего значения. Запись данных в этот регистр не только изменяет текущее значение счетчика, но и задает точку отсчета после перезагрузки таймера. Так, например, если указать:

```
// Начальное значение счета
MDR_TIMER1->CNT = 100;
```

то таймер каждый цикл будет начинать отсчет со ста.

Регистр **MDR_TIMER1->ARR** определяет период перезагрузки таймера. Иными словами, таймер будет увеличивать значение *CNT* до тех пор, пока оно не станет равным значению *ARR*, а затем выполнит перезагрузку.

Опираясь на формулу (3.1), время перезагрузки таймера можно определить, как:

$$T_{RLD} = \frac{PSG + 1}{SystemCoreClock} \times ARR. \quad (3.2)$$

Регистр предполагает 16-битное значение. Не следует записывать в него значение больше, чем $2^{16} - 1 = 65535$, т.к. биты старше 15-го будут проигнорированы и поведение таймера будет отличаться от ожидаемого.

Регистр **MDR_TIMER1->CNTRL** содержит основные настройки счетчика (таблица 3.5).

Таблица 3.5 – Описание регистра MDR_TIMER1->CNTRL

№ битов	Функциональное имя битов	Дескрипция
31…12	–	–
11…8	EVENT_SEL[3:0]	Источник событий для счета: 0000 – передний фронт тактовых импульсов; 0001 – *CNT* = *ARR* в *TIMER1*; 0010 – *CNT* = *ARR* в *TIMER2*; 0011 – *CNT* = *ARR* в *TIMER3*; 0100 – импульс на первом канале; 0101 – импульс на втором канале; 0110 – импульс на третьем канале; 0111 – импульс на четвертом канале; 1000 – импульс на канале *ETR*.
7…6	CNT_MODE[1:0]	Режим счета: 00 – тактовые импульсы с фиксированным направлением; 01 – тактовые импульсы с изменяемым направлением; 10 – внешние события с фиксированным направлением; 11 – внешние события с изменяемым направлением.

Продолжение таблицы 3.5

№ битов	Функциональное имя битов	Дескрипция
5...4	FDTS[1:0]	Частота выборки данных: 00 – каждый импульс; 01 – каждый второй импульс; 10 – каждый третий импульс; 11 – каждый четвертый импульс.
3	DIR	Направление счета: 0 – прямой счет (0...*ARR*); 1 – обратный счет (*ARR*...0).
2	WR_CMPL	Флаг работы с регистрами CNT, ARR, PSG: 0 – регистры готовы к записи; 1 – идет запись в регистры.
1	ARRB_EN	Режим обновления регистра ARR: 0 – мгновенное обновление; 1 – обновление после перезагрузки.
0	CNT_EN	Запуск таймера: 0 – отключен; 1 – включен.

Программное заполнение регистра выглядит следующим образом:

```
// Общая конфигурация таймера
MDR_TIMER1->CNTRL =
    (1 << TIMER_CNTRL_CNT_EN_Pos)      // Работа таймера (включен)
  | (0 << TIMER_CNTRL_ARRB_EN_Pos)     // Режим обновления регистра ARR (мгновенное)
  | (0 << TIMER_CNTRL_DIR_Pos)         // Направление счета (прямой счет)
  | (0 << TIMER_CNTRL_FDTS_Pos)        // Частота выборки (не используется)
  | (0 << TIMER_CNTRL_CNT_MODE_Pos)    // Режим счета (такт. импульсы с фикс. напр-ем)
  | (0 << TIMER_CNTRL_EVENT_SEL_Pos);  // Триггер счета (тактовые импульсы)
```

4. Настройка аппаратных прерываний таймера, согласно разделу 3.2, состоит из трех этапов.

Во-первых, необходимо разрешить таймеру отправлять запросы на обработку прерываний при определенных событиях. Делается это в регистре MDR_TIMER1->IE (стр. 304 Спецификации). В регистре предусмотрено достаточно много событий для разных каналов. Например, чтобы таймер формировал запрос на прерывание при перезагрузке нужно установить первый бит этого регистра:

```
// Настройка запросов на обработку прерываний от таймера
MDR_TIMER1->IE = (1 << TIMER_IE_CNT_ARR_EVENT_IE_Pos);  // Прерывание при CNT = ARR
```

Во-вторых, нужно назначить приоритет запросов от таймера в контроллере NVIC и разрешить их обработку. Это проще всего выполнить с использованием стандартных библиотечных процедур ядра Cortex-M3:

```
// Назначение приоритета прерывания от таймера
NVIC_SetPriority(Timer1_IRQn, 1);

// Разрешение обработки прерывания от таймера
NVIC_EnableIRQ(Timer1_IRQn);
```

Наконец, в-третьих, надо описать обработчик прерываний от таймера. Содержание обработчика полностью зависит от задач проекта.

Если рассмотреть проект *Sample 3.1*, то в нем реализовано измерение фактического времени задержки операционной системы. Значение регистра ARR задано максимальным – 65535. В таком случае прерывание будет свидетельствовать о **переполнении счетчика**. Иными словами, о том, что измеряемый период времени больше того, на который настроен таймер. Об этом, конечно, нужно сообщить системе (через глобальную переменную или сервисы ОСРВ), иначе результат измерений будет неверным. Такая процедура обработки приведена ниже:

```
// Псевдоним флага переполнения таймера
#define EVENT_TIMER_OVERFLOW 0x00000001U

// Обработчик прерывания от таймера TIMER1
void Timer1_IRQHandler(void)
{
  // Сброс запроса на обработку прерывания
  MDR_TIMER1->STATUS &= ~(1 << TIMER_IE_CNT_ARR_EVENT_IE_Pos);

  // Установка флага переполнения таймера
  osEventFlagsSet(EventId_Timer, EVENT_TIMER_OVERFLOW);
}
```

Инициализация таймера на этом завершена. С момента включения таймер работает автономно.

3.3.3. Сторожевые таймеры

Сторожевой таймер (англ. **Watchdog**) – это специальный блок контроля над зависанием системы. Идея состоит в том, что сторожевой таймер должен периодически сбрасываться контролируемой системой. Если сброса не происходит в течение некоторого интервала времени, то делается вывод о зависании системы и выполняется ее принудительная перезагрузка (рисунок 3.8).

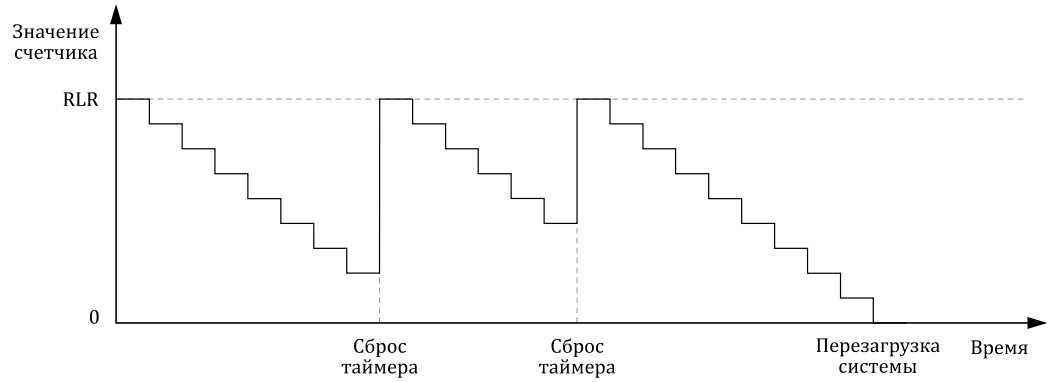

Рисунок 3.8 – Временная диаграмма сторожевого таймера

Микроконтроллеры серии 1986ВЕ9х содержат два сторожевых таймера – независимый и оконный.

Независимый сторожевой таймер (англ. Independent Watchdog, **IWDG**) представляет собой 12-разрядный вычитающий счетчик. Он тактируется от генератора LSI, благодаря чему продолжает работу даже при отсутствии основного тактового сигнала. С другой стороны, это не позволяет предъявлять высокие требования к временным параметрам таймера ввиду низкой стабильности данного генератора.

Для работы сторожевого таймера необходимо включить его тактирование, задать период работы и предусмотреть регулярный программный сброс.

Для подачи тактовых импульсов на сторожевой таймер в первую очередь следует включить генератор LSI:

```
// Включение генератора LSI
MDR_BKP->REG_0F |= (1 << BKP_REG_0F_LSI_ON_Pos);

// Ожидание входа генератора LSI в рабочий режим
while ((MDR_BKP->REG_0F & BKP_REG_0F_LSE_RDY) == 0);
```

Затем рекомендуется выполнить подстроку тактовой частоты генератора в примерном соответствии с графиком, изображенном на рисунке 3.2. Согласно графику, тактовой частоте 40 КГц соответствует значение подстройки, равное 11:

```
// Подстройка тактовой частоты генератора LSI
MDR_BKP->REG_0F &= ~BKP_REG_0F_LSI_TRIM_Msk;         // Сброс битов
MDR_BKP->REG_0F |= (11 << BKP_REG_0F_LSI_TRIM_Pos);
```

После этого можно включить тактирование сторожевого таймера:

```
// Включение тактирования сторожевого таймера
MDR_RST_CLK->PER_CLOCK |= (1 << RST_CLK_PCLK_IWDT_Pos);
```

Далее нужно определить период работы сторожевого таймера, т.е. интервал времени, по прошествии которого таймер будет перезагружать систему. Он зависит от частоты тактовых импульсов, предделителя частоты (регистр `MDR_IWDG->PR`) и значения перезагрузки (регистр `MDR_IWDG->RLR`):

$$T_{RLD} = \frac{2^{PR+2}}{LSICLK} \times RLR. \qquad (3.3)$$

Для примера рассмотрим, как выбрать значения параметров, чтобы получить значение перезагрузки, равное 5 секундам.

Значение частоты тактовых импульсов известно и равно 40 КГц, поэтому имеем уравнение с двумя параметрами. Делитель частоты определим из условия, что значение перезагрузки не должно выходить за границы диапазона, определяемого разрядностью таймера. Тогда по формуле (3.3) получаем:

$$\frac{LSICLK}{2^{PR+2}} \times T_{RLD} < 2^{R};$$

$$2^{PR+2} > \frac{LSICLK}{2^{R}} \times T_{RLD};$$

$$2^{PR+2} > \frac{40 \times 10^3}{2^{12}} \times 5 = 48.83.$$

Выберем ближайшее допустимое значение делителя, удовлетворяющее этим условиям: оно равно 64 ($PR = 4$).

В таком случае требуемое значение перезагрузки по формуле (3.3) составит:

$$\mathrm{RLR} = \frac{\mathrm{LSICLK}}{2^{\mathrm{PR}+2}} \times T_{RLD} = \frac{40 \times 10^3}{2^{4+2}} \times 5 = 3125.$$

Регистры MDR_IWDG->PR и MDR_IWDG->RLR изначально защищены от записи из соображений надежности. Для доступа к ним необходимо записать ключевое значение 0x5555 в регистр MDR_IWDG->KR. Кроме того, к данным регистрам нельзя обращаться в период их обновления, которое сопровождается установкой специального флага. Поэтому работа с указанными регистрами должна выполняться следующим образом:

```
// Разрешение записи данных в регистры PR и RLR
MDR_IWDG->KR = 0x5555;

// Установка делителя частоты сторожевого таймера
while ((MDR_IWDG->SR & IWDG_SR_PVU) != 0);  // Проверка флага обновления
MDR_IWDG->PR = 4;                           // Задание делителя (40 КГц / 64)

// Установка значения перезагрузки сторожевого таймера
while ((MDR_IWDG->SR & IWDG_SR_RVU) != 0);  // Проверка флага обновления
MDR_IWDG->RLR = 3125;                       // Задание значения перезагрузки
```

Для запуска таймера следует записать ключевое значение 0xCCCC в регистр MDR_IWDG->KR:

```
// Запуск сторожевого таймера
MDR_IWDG->KR = 0xCCCC;
```

С этого момент его можно отключить только путем запрета тактирования.

Теперь все, что остается сделать – это организовать регулярный сброс таймера путем записи ключевого значения 0xAAAA в регистр MDR_IWDG->KR:

```
// Перезагрузка сторожевого таймера
MDR_IWDG->KR = 0xAAAA;
```

Оконный сторожевой таймер (англ. Windowed Watchdog, **WWDG**) отличается от независимого в первую очередь тем, что его сброс должен производиться в заданный интервал времени – ни раньше, ни позже.

Данный таймер обычно используется для обнаружения некорректной работы программного обеспечения в виде опережения или запаздывания, вызванного внешним вмешательством или непредвиденными логическими условиями.

Оконный таймер представляет собой 7-разрядный вычитающий счетчик, тактируемый частотой *HCLK* с фиксированным аппаратным делителем 4096 и программно конфигурируемым делителем 1...8. Из этого следует сделать вывод, что оконный сторожевой таймер может быть настроен лишь на небольшие временные

интервалы. При частоте ядра, равной 80 МГц, период таймера будет составлять от 3.3 до 26.2 миллисекунд в зависимости от выбранного значения делителя.

Работа оконного сторожевого таймера проиллюстрирована рисунком 3.9. После запуска таймер начинает обратный отсчет со значения *T*. Для правильной работы системы он должен быть сброшен момент, когда значение его счетчика находится в диапазоне от 0x40 до *W*. Если значение счетчика достигает 0x3F, то сторожевой таймер перезагружает систему. С другой стороны, если сброс таймера происходит при значении счетчика выше *W*, то также производится перезагрузка системы.

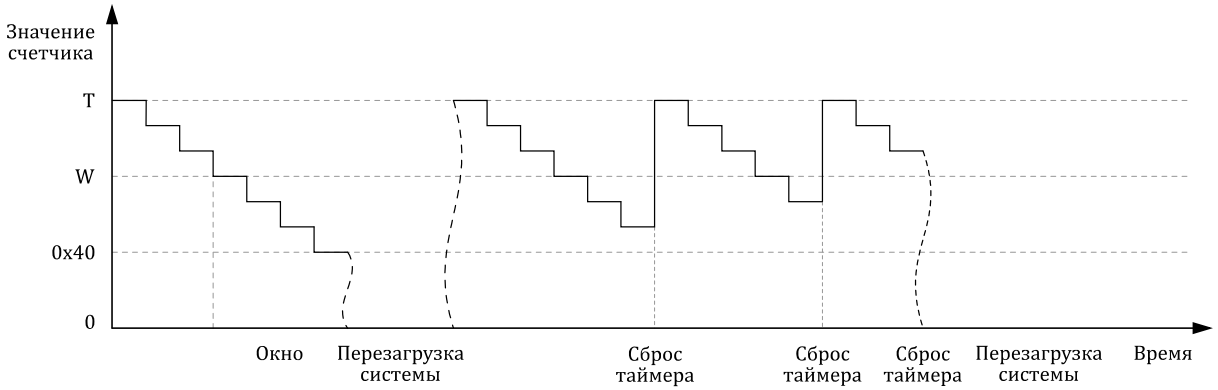

Рисунок 3.9 – Временная диаграмма оконного сторожевого таймера

Оконный сторожевой таймер может генерировать предупреждающее аппаратное прерывание при значении счетчика 0x40. В обработчике прерывания, помимо прочего, таймер может быть сброшен для предотвращения перезагрузки.

Инициализация оконного сторожевого таймера в целом относительно проста: достаточно включить его тактирование, задать значение окна, и при необходимости настроить аппаратные прерывания:

```
// Включение тактирования WWDG
MDR_RST_CLK->PER_CLOCK |= (1 << RST_CLK_PCLK_WWDT_Pos);

// Конфигурация WWDG
MDR_WWDG->CR  = (1 << WWDG_CR_WDGA_Pos);  // Перезагрузка системы (разрешена)

MDR_WWDG->CFR = (0x60 << WWDG_CFR_W_Pos)  // Окно таймера
              | (3 << WWDG_CFR_WGTB_Pos)  // Делитель частоты (2^3 = 8)
              | (1 << WWDG_CFR_EWI_Pos);  // Предупреждающее прерывание (разрешено)

// Сброс запроса на обработку прерывания
MDR_WWDG->SR = 0;

// Назначение приоритета прерывания от WWDG
NVIC_SetPriority(WWDG_IRQn, 1);

// Разрешение обработки прерывания от WWDG
NVIC_EnableIRQ(WWDG_IRQn);
```

Однако есть несколько важных нюансов, на которые необходимо обратить внимание:

1. При первичной подаче тактовых импульсов на сторожевой таймер, значения его счетчика и окна равны 0x7F, а также установлен запрос на обработку прерывания.

2. Разрешение перезагрузки системы таймером производится путем записи единицы в седьмой бит регистра MDR_WWDG->CR. Однако сбросить этот бит программно нельзя. Поэтому единственный способ отключить сторожевой таймер – запретить его тактирование.

3. Биты 0...6 регистра MDR_WWDG->CR хранят значение счетчика. Запись любых данных в эту область сбрасывает значение счетчика в исходное (0x7F).

4. Не следует записывать в указанный выше регистр значения бóльшие, чем величина окна, т.к. это приведет к перезагрузке системы. Безопаснее всего проводить сброс таймера записью нуля:

```
// Сброс таймера
MDR_WWDG->CR = 0x00;
```

5. Значение окна следует выбирать из диапазона 0x41...0x7F.

Особое внимание нужно уделить пунктам 3 и 4, т.к. они официально не документированы. Это может привести к трудностям при освоении оконного сторожевого таймера.

Описание программных проектов

В проекте **Sample 3.1** выполняется измерение фактического времени задержки, создаваемой операционной системой, с использованием аппаратного таймера. Заданное и измеренное значения отображаются на дисплей.

Проект **Sample 3.2** содержит конфигурацию независимого сторожевого таймера, перезагружающего систему каждые пять секунд. Отсчет времени работы отображается на дисплей. Сброс таймера производится при нажатии кнопки *SEL*.

Задачи для самостоятельной работы

1. Используя аппаратный таймер, организуйте возможность измерения временных интервалов до полутора секунд.
2. Создайте из двух 16-разрядных таймеров один 32-разрядный. Для этого первый таймер должен инкрементироваться от тактовых импульсов, а второй – от перезагрузки первого. На дисплей отображайте текущие значения всех трех таймеров.
3. Сконфигурируйте сторожевой таймер таким образом, чтобы он перезагружал систему с интервалом в 12 секунд. За три секунды до перезагрузки выводите на дисплей предупреждение.

Контрольные вопросы

1. Что такое тактирование?
2. Какие источники тактовых импульсов использует микроконтроллер?
3. Как настроить тактовую частоту микроконтроллера?
4. Что такое прерывание?
5. Как организовать работу с прерываниями?
6. Что такое аппаратный таймер?
7. Какие аппаратные таймеры реализованы в микроконтроллере?
8. Чему равен максимальный период перезагрузки 10-разрядного таймера, тактируемого частотой 1 КГц?
9. Что такое сторожевой таймер?
10. Чем независимый сторожевой таймер отличается от оконного?