Часть 7. MPLAB ASM30 Ассемблер

7.1 Обзор MPLAB ASM30 ассемблера

Теперь вы знаете как создавать и строить проект и использовать инструменты симуляции и отладки, давайте потратим немного времени, чтобы узнать как создавать код. Поскольку MPLAB ASM30 ассемблер включён в MPLAB IDE, мы обсудим несколько основ использования этого инструментального языка.

MPLAB IDE ассемблер основан на программном обеспечении с открытым кодом GNU, которое может казаться знакомым некоторым пользователям. Ассемблер интерпретирует инструкции и директивы в файле с исходным кодом и генерирует объектный код. Компоновщик используется для преобразования объектного кода в окончательный выходной (Hex) файл для программирования раздела (смотреть главу 9. "The MPLAB LINK30 компоновщик").

Инструкции выполняют время выполнения в устройстве dsPIC. Они являются родным языком процессора dsPIC. Тем не менее, система команд dsPIC не рассматривается в этом документе. Для детальной информации о системе команд dsPIC, обращайтесь к "Руководству по программированию dsPIC30F " (DS70030).

Директивы интерпретируются во время работы ассемблера и используются для определения секций памяти, инициализации констант, декларирования и определения символов (переменных, меток,
и т.п.), текста замены и тому подобное. Список деректив и их использование документировано в "Руководство пользователя по MPLAB® ASM30, MPLAB® LINK30 и утилитам" (DS51317). Точка (".") должна предшествовать каждой директиве.

Мы обсудим несколько наиболее часто используемых директив, так чтобы вы имели идею, что требуется при написании вашего кода. Много этих директив было использовано в примере кода из предыдущего описания.

Примечание:	Эта глава основана на MPLAB ASM30 ассемблере версии 1.31. Некоторая информация может становиться устаревшей при выходе новых версий.

7.1.1 Общий формат инструкций и директив

Инструкции и директивы требуют следующие основные формы:

[метка:] инструкция[операнды] [; комментарии]

[метка:] директива[аргументы] [; комментарии]

Метки используются для отметки позиций в коде. Во время компоновки, метки определяют адреса памяти; метки могут начинаться с "." (точки) и должны заканчиваться ":" (двоиточием).

Операнды используются в инструкциях для обеспечения информации об источнике и месте назначения. Они состоят:

● Литералы - Это шестнадцатеричные, восьмеричные, двоичные или десятичные значения. Признак числа "#" должен предшествовать всем литеральным значениям.

● Адреса регистров и памяти - Это есть рабочие регистры, аккумуляторы, универсальные регистры (GPRs) и регистры специальных функций (SFRs).

● Коды завершения - Это биты состояния, такие как Z (ноль) или C (перенос), используемые как операнды в инструкциях условного перехода.

Аргументы подобны операндам. Аргументы используются как источник и приёмник информации директив.

Синтаксические правила для инструкций и директив приведены в таблице 7-1.

Таблица 7-1: Синтаксические правила

Символ	Описание	Использование
.	Точка	Начало директивы или метки
:	Двоеточие	Конец метки
#	Решётка	Начало литерального значения
;	Точка с запятой	Начало однострочных комментариев
/*		Начало многострочных комментариев
***/**		Конец многострочных комментариев

7.2 Обычно используемые директивы

Несколько обычно используемых директив перечислены ниже. Они все даны как примеры в файлах шаблонах

dsPIC, которые вы можете найти в следующей директории:

C:\Program Files\Microchip\MPLAB ASM30 Suite\Support\templates\assembly

Так же вы можете найти первые пять директив в нашем руководстве из главы 4. "MPLAB SIM симулятор", где вы узнали как создавать и строить проект. Хотя другие пять директив были не представлены в нашем руководстве, вы можете найти самостоятельно в случае необходимости одно из них.

.equ	приравнивает значение символу
.include	включает другой файл в текущий файл
.global	создаёт глобально видимый символ
.text	начинает раздел выполняемого кода
.end	заканчивает сборку в течении файла
.section	начинает секцию (кода или данных, в программной памяти или памяти данных)
.space	распределяет пространство в пределах секции
.bss	добавляет переменные в неинициализированную секцию данных
.data	добавляет переменные в инициализированную секцию данных
.hword	объявляет слова данных в пределах секции
.palign	выравнивает код в пределах секции
.align	выравнивает данные в пределах секции

.equ

Одной из общих директив в любом ассемблерном исходном файле является .equ. Директива .equ используется для определения символа и присвоения ему значения. В примере 7-1, директива .equ используется для назначения литерального значения 7372800 символу FCY. В этом контексте, FCY является константой, которая может быть использована в коде для представления частоты цикла инструкции.

Пример 7-1: .equ

```
;Программные точные константы (литералы, используемые в коде)
    .equ FCY, #7372800 ;Темп цикла инструкции (Osc x PLL / 4)
;====================================================================
```

.include

Директива .include добавляет содержимое определённого файла в ассемблерный источник, точка эта быть использована, как показано в примере Example 7-2. Одно простое использование директивы .include есть добавление определений из стандартного процессорного include файла.

Пример 7-2: .include

```
    .equ 30F6014, 1
    .include "p30f6014.inc"
;--------------------------------------------------------------------
```

.global

Директива .global используется для того чтобы позволить меткам, определённым внутри файла, используваться в другом файле. В примере 7-3, символ __reset сделан глобальным с тем чтобы компоновщик мог использовать его как адрес для перехода туда из вектора сброса. Метка __reset: требуется, чтобы обозначить начало кода и должна быть представлена в одном из проектных объектных файлах (ассемблера, компилятора или библиотечных файлах).

Пример 7-3: .global

```
;Глобальные декларации
    .global __reset          ;Метка первой строки кода
    .global __OscillatorFail ;Декларация программы обработки ловушки ошибки осциллятора

    .global __AddressError   ;Декларация программы обработки ловушки ошибки адресации
```

.text

Это есть специальный случай директивы .section. Директива .text используется для информирования ассемблера что следующий далее код будет помещён в исполняемую секцию программной памяти (смотреть пример 7-4).

Пример 7-4: .text

```
;Начало кода
    .text      ;Начало кодовой секции
```

.end

Директива .end используется для обозначения окончания ассемблерного файла источника (смотреть пример 7-5).

Пример 7-5: .end

```
    .end       ;Конец кода в этом файле
```

.section

Директива .section декларирует секцию памяти. Эта секция может быть в RAM или в программной памяти, как определено атрибутами которые следуют за директивой. В примере 7-6, секция названная MyDataSection размещена в не инициализированной ближней памяти данных. Секция названная MyOtherSection размещена в памяти данных Y. Полный перечень типов секций содержится в "Руководство пользователя по MPLAB® ASM30, MPLAB® LINK30 и утилитам" (DS51317) и в "dsPIC30F Language Tools Quick Reference Card" (DS51322).

Пример 7-6: .section

```
;Переменные RAM
      .section MyDataSection, bss, near
Var1:  .space 1                    ;Распределение пространства (в байтах) в переменную
      .section MyOtherSection, ymemory
Array1:.space 20                   ;Распределение пространства (в байтах) в массив
```

.space

Директива .space указывает ассемблеру сколько надо зарезервировать пространства в данной секции. В примере 7-7, однобайтное пространство памяти зарезервировано для переменной с именем Var1.

Пример 7-7: .space

```
;Переменные RAM
      .section MyDataSection, bss, near
Var1:  .space 1                    ;Распределение пространства (в байтах) в переменную
```

.bss

Директива .bss есть специальный случай директивы .section. Вызывает добавление неинициализированных переменных данных в не инициализированную секцию данных. А примере 7-8, Var2 будет размещена в не инициализированной памяти данных.

Пример 7-8: .bss

```
;Переменные RAM
      .bss
Var2:  .space 2                   ;Распределение пространства (в байтах) в переменную
```

.data

Директива .data есть специальный случай директивы .section. Вызывает добавление инициализированных переменных данных в секцию инициализированных данных. В примере 7-9, массив MyRAM будет размещён в памяти данных и ассемблер разместит данные 0x1111, 0x2222 и 0x3333 в секции программной памяти.

Пример 7-9: .data

```
;Инициализированная RAM переменных
      .data
MyRAM: .hword 0x1111, 0x2222, 0x3333
```

Важно иметь в виду, что в порядке использования инициализированных данных, корректный код запуска нужно добавить в проект для копирования данных в RAM. Выполняемый модуль запуска включён в динамическую библиотеку, файл libpic30.a. Этот файл представлен в папке pic30_tools\lib.

Обращайтесь к "Руководство пользователя по MPLAB® ASM30, MPLAB® LINK30 и утилитам" (DS51317) для дальнейшей информации на функции модуля запуска динамической библиотеке.

.hword

Директива .hword декларирует слова инициализированных данных в пределах секции. Может так же декларировать данные константы в пределах програмной памятии. Слова данных, 0x0002, 0x0003 и 0x0005, могут быть загружены в смежные слова программной памяти. Поскольку программная память имеет ширину 24 бит, верхний байт каждого слова будет 0x0.

Пример 7-10: .hword

```
        .palign 2                   ;Выравнивает следующие слова к границе второго байта
MyData:.hword 0x0002, 0x0003, 0x0005
```

.palign

Директива .palign выравнивает данные в пределах секции программной памяти. В примере 7-11, переменная MyData начинается с чётного адреса (делимый без остатка на 2).

Пример 7-11: .palign

```
        .section .myconstbuffer, "x"
        .palign 2                   ;Выравнивает следующие слова к границе второго байта
MyData:.hword 0x0002, 0x0003, 0x0005
```

.align

Директива .align выравнивает данные в пределах секции. В примере 7-12, переменная Array3 начинается с адреса который точно делится на 8. Директива .align особенно полезна когда используется модульная адресация характеристики или процессор dsPIC30F.

Пример 7-12: .align

```
        .bss
        .align 8
Array3:.space 6                     ;Распределение пространства (в байтах) в переменную
```

7.3 Пример кода

Имея понятия о директивах и форматах инструкций, мы можем рассматривать пример, который покажет как всё работает. Здесь разъясняется код моргания светодиодами с использованием файла dsPIC30F6014.s из предыдущей главы.

Примечание:	Другие обучающие файлы, моргание светодиодами с dsPIC30F6012.s, моргание светодиодами с dsPIC30F2010.s и моргание светодиодами с dsPIC30F4011.s очень похожи и описание ниже прилагается.

7.3.1 Описание кода

Файл dsPIC30F6014.s моргания светодиодами начинается с комметариев. В этом файле, комментарии объясняют лицензионное соглашение и что делает программа (моргание светодиодами зависит от состояния ключа SW1). Комментариям всегда предшествует точка с запятой (;) или, альтернативный, C-подобный блок комментариев (/* */).

Комментарии сопровождающие определение __30F6014 метку позволяют включить файл проверки что корректный процессор использован. Стандартный include файл включает определение всех бит в различных SFRs.

Пример 7-13:

```
;=============================================================================
; Использовать Timer 1 для моргания LED1 когда ключ SW1 не нажат
; и моргать LED2 когда ключ SW1 нажат
;=============================================================================
     .equ __30F6014, 1
     .include "p30f6014.inc"
```

Следующая секция содержит глобальные описания метки __reset и меток переменных ловушки ошибок. Это позволяет определить компоновщику правильный адрес вектора сброса размещённый в инструкции goto. Это также позволяет компоновщику определить адреса для размещения в таблице векторов прерываний для программ ловушки ошибки. (Обратитесь к разделу 1.2.10 "Прерывания" для большей информации о ловушках ошибок и таблице векторов прерывания.)

Пример 7-14:

```
;-------------------------------------------------------------------
;Глобальные описания
    .global __reset          ;Метка первой линии кода
    .global __OscillatorFail ;Объявляет метку программы обработки ловушки ошибки
генератора
    .global __AddressError ;Объявляет метку программы обработки ловушки ошибки адресации
    .global __StackError   ;Объявляет метку программы обработки ловушки ошибки стека
    .global __MathError    ;Объявляет метку программы обработки ловушки математ. ошибки
```

В следующем разделе биты конфигурации определены с тем чтобы процессор был запрограммирован с корректным режимом генератора, установками сторожевого таймера, и т.п.. Адреса регистра конфигурации, такие как __FOSC и __FWDT, определены в скрипт файле компоновщика p30f6014.gld. Значения регистра конфигурации, такие как CSW_FSCM_OFF и XT_PLL4, определены в include файле соответствующего процессора, p30f6014.inc.

Пример 7-15:

```
;-----------------------------------------------------------------------------
;Биты конфигурации
      config __FOSC, CSW_FSCM_OFF & XT_PLL4
      config __FWDT, WDT_OFF
      config __FBORPOR, PBOR_OFF & BORV_27 & PWRT_16 & MCLR_EN
      config __FGS, CODE_PROT_OFF
```

Метка FCY приравнена к значению с тем чтобы можно было легко осуществить изменения частоты. Теперь С помощью одной строки можно адаптировать код к различным опциям генератора.

Пример 7-16:

```
;-----------------------------------------------------------------------------
;Программные специфические константы (литералы используемые в коде)
      .equ FCY, #7372800     ;Темп цикла инструкции (Osc x PLL / 4)
```

Директива .text сообщает ассемблеру что код, который следует должен быть установлен в секцию кода по умолчанию.

Пример 7-17:

```
;=============================================================================
;Начало кода
      .text                  ;Начало кодовой секции
```

Компоновщик распознаёт __reset: как стандартную метку и добавляет код ветвления сюда после сброса. Это должно быть глобальной для компоновщика использующего метку.

После размещения всех переменных RAM, компоновщик обнаруживает самое большое доступное пространство для стека и присваивает начальный метке __SP_init. Код загружает эту метку в регистр указатель стека (Stack Pointer register), W15, и это настраивает программный стек. Обратите внимание, что "#" признак указывает значение литерала.

Компоновщик так же обеспечивает адрес конца пространства доступного стеку и код загружает это значение из метки __SPLIM_init в регистр указатель предела стека (Stack Pointer Limit register),

SPLIM. Это устанавливает проверку ошибок для переполнений стека. Ошибка стека произойдёт если указатель стека, W15, сравняется с адресом в SPLIM. Обратите внимание, что эта операция делается в две инструкции. Регистр W0 загружен значением __SPLIM_init и это значение затем перемещается в регистр SPLIM. Это потому что невозможно закодировать 16-битый литерал и 13-битный адрес памяти в одной 24-битной инструкции.

Пример 7-18:

```
;-----------------------------------------------------------------------------
;Инициализировать указатель стека и регистр предела
__reset:   mov #__SP_init, W15     ;Инициализировать регистр указателя стека
           mov #__SPLIM_init, W0   ;Получить адрес конца области стека
           mov W0, SPLIM           ;Загрузить регистр указатель предела стека
           nop                     ;Добавить NOP следую SPLIM инициализации
```

После настройки стека, код инициализирует порт I/O для управления LEDs на PORTD. LEDs на битах от 0 до 3 PORTD и включаются когда ножки едут вниз. Код устанавливает эти биты в регистре защёлке порта, LATD, с тем чтобы когда ножки I/O были настроены на выход, LEDs были отключены. Коде затем очищает соответствующие биты в регистре TRISD, с тем чтобы ножки I/O были настроены на вывод. Наконец код очищает бит 0 LATD чтобы включить один LED.

Пример 7-19:

```
;-------------------------------------------------------------------------
;Инициализировать выводы LED на битах 0-3 PORTD
           mov #0xffff, W0 ;Инициализировать данные ножек LED в выключенное состояние
           mov W0, LATD
           mov #0xfff0, W0 ;Настроить ножки LED как выходы
           mov W0, TRISD
           bclr LATD, #0   ;Включить LED1
```

Timer1 инициализирован для 1/5-секундного периода с тем чтобы эта фигня бодро моргала. Код очищает регистр управления таймером, T1CON, останавливает таймер и и регистр счётчика Timer1, TMR1, очищен с тем чтобы старт счёта произошёл с нуля.

Регистр периода Timer1, PR1, загружен числом подсчётов в 1/5 секунд. Ассемблер вычисляет это значение поскольку мы имеем инструкцию определения темпа (FCY), делит на значение предделителя (256), умножает на время (1/5 секунд). Предделитель делить тактовый темп, который инкрементирует таймер.

Наконец, регистр управления Timer1, T1CON, записан чтобы включить таймер и используя внутренний тактовый источник с предделителем 1:256.

Пример 7-20:

```
;-------------------------------------------------------------------------
;Инициализация Timer1 для периода 1/5 секунды
           clr T1CON          ;Выключить таймер, очистив регистр управления
           clr TMR1           ;Старт Timer1 с нуля
           mov #FCY/256/5, W0 ;Получить значение регистра периода для 1/5 секунды
           mov W0, PR1        ;Загрузить регистр периода Timer1
           mov #0x8030, W0    ;Получить установки Timer1 (1:256 предделитель)
           mov W0, T1CON      ;Загрузить установки Timer1 в регистр управления
```

Главный кодовый цикл стартует с метки MainLoop:. Бит T1IF флага прерывания Timer1 в регистре IFS0 проверен чтобы независимо видеть, что счётчик таймера достиг значения регистра периода. Если период ещё не прошёл, тогда код ветвится обратно на MainLoop.

Как только Timer1 установит бит T1IF, код пропускает инструкцию ветвления с тем чтобы с тем чтобы пойти и моргануть светодиодом. Бит T1IF очищается с тем чтобы его можно было использовать ещё для определения конца следующего периода таймера.

Ключ подключен к ножке RA12, таким образом код тестирует бит 12 регистра PORTA, чтобы видеть если ключ будет нажат. Если ключ нажат, инструкция ветвления выполняет переключение LED2; в противном случае, код пропускает инструкцию ветвления и взамен переключает LED1.

LED1 подключен к ножке RD0, таким образом бит 0 LATD переключает состояния вкл/выкл LED. LED2 выключен очисткой бита 1 LATD в случае если LED зажжён. Обратите внимание что когда порт I/O используется на вход, регистр PORTx использован и когда порт используется как выход, используется регистр LATx. После изменения любого LEDs, код ветвится обратно на MainLoop.

Пример 7-21:

```
;-------------------------------------------------------------------------
;Циклить пока ожидается Timer1 и переключить LED1 или LED2 когда это случится
MainLoop:      btss IFS0, #T1IF  ;Проверить, если флаг прерывания Timer1 установлен
               bra MainLoop      ;обратно циклить пока установлен
               bclr IFS0, #T1IF  ;Очистить флаг прерывания Timer1
               btss PORTA, #12   ;Проверить ключ SW1 (низкий, когда нажат)
               bra SwitchPressed
               btg LATD, #0      ;Переключить LED1 когда SW1 не нажат
               bset LATD, #1     ;Выключить LED2
               bra MainLoop      ;Циклить обратно
SwitchPressed: bset LATD, #0     ;Выключить LED1
               btg LATD, #1      ;Переключить LED2 когда SW1 нажат
               bra MainLoop      ;Циклить обратно
```

Программы ловушек ошибок следующий остаток кода. Если код терпит неудачу благодаря катастрофической ошибке, как например отказ генератора или ветвление в несуществующую память, тогда аппаратное средство включит выполнение соответствующей программы ловушки ошибки. Каждая программа имеет глобальную адресную метку, как например __OscillatorFail: и компоновщик использует эти адреса для создания таблицы векторов прерываний. Каждая из этих программ ловушки ошибки включает LED и бесконечный цикл.

Пример 7-22:

```
;=============================================================================
;Ловушки ошибок
;-----------------------------------------------------------------------------
;Программа ловушки отказа генератора
   .text                         ;Начало кодовой секции
_OscillatorFail:
          bclr LATD, #3          ;Включить LED4
          bra __OscillatorFail   ;Циклить, когда происходит событие отказа генератора
;-----------------------------------------------------------------------------
;Программа ловушки ошибки адреса
__AddressError:
          bclr LATD, #3 ;Turn LED4 on
          bra __AddressError     ;Циклить, когда происходит событие ошибки адресации
```

После программ ловушек ошибок, должна быть директива .end, которая показывает что далее нет кода, который должен быть ассемблирован в этом файле.

Пример 7-23:

```
    .end                          ;Конец кода в этом файле
```

Ассемблер всегда генерирует объектный файл, который должен быть скомпонован. Узнать о LINK30 компоновщике и как получить код и данные из объектных файлов и создать финальный выходные файлы, пожалуйста прыгайте вперёд к главе 9. "MPLAB LINK30 компоновщик". Если вы будете использовать компилятор MPLAB C30, тогда приступайте к главе 8. "MPLAB C30 C компилятор".