Зачем использовать линкер

Повышение быстродействия и расширение периферийных возможностей микроконтроллеров приводит к использованию встраиваемых устройств в тех областях, где традиционно использовались мини-платформы, высокопроизводительные специализированные контроллеры или решения на базе ASIC. Соответственно растут объем и сложность программного обеспечения, что предъявляет новые требования к средствам и методологии разработки: минимум ошибок структурирования, минимум оверхеда (формальных нефункциональных участков кода), максимум реюзабилити (повторного использования) кода. Частично задача решается применением компиляторов высокого уровня, однако наиболее полноценного результата можно добиться при помощи ассемблера и линкера (компоновщика).

Традиционно используются три варианта организации проекта:

1) Сборка ассемблером, исходный код в виде одного файла. Отличный вариант для проектов, не превышающих нескольких страниц текста.

2) Сборка ассемблером, исходный код в виде основного файла и подключаемых через **#include** дополнительных модулей. В редких случаях такая структура может демонстрировать эффективность в проектах до 3..4К команд, однако попытка разделить проект на функциональные блоки упирается в ограничения ассемблера – невозможность генерации перемещаемого кода, как для программной области, так и для данных.

3) Сборка линкером, исходный код в виде основного файла и подключаемых линкером модулей и библиотек. Наиболее сложный вариант, при этом автоматический менеджмент памяти (как программного Flash, так и RAM, EEPROM) и сборка из перемещаемых объектных файлов позволяют, с одной стороны, разделять проект на логически законченные модули, а с другой – объединять в одном проекте готовые библиотеки и модули, в том числе, полученные при помощи разных компиляторов. При этом эффективно решаются все три проблемы, поставленные во введении – ошибки структурирования, эффективность и повторное использование кода.

Линкер-скрипт и логические секции

```
// $Id: 18f2520i.lkr,v 1.5 2005/04/18 15:11:24 kochars Exp $
// File: 18f2520i.lkr
// Sample ICD2 linker script for the PIC18F2520 processor

LIBPATH .

CODEPAGE    NAME=vreset       START=0x0000      END=0x0007        PROTECTED
CODEPAGE    NAME=vinthi       START=0x0008      END=0x0017        PROTECTED
CODEPAGE    NAME=vintlo       START=0x0018      END=0x001F        PROTECTED
CODEPAGE    NAME=seq          START=0x0020      END=0x00FF        PROTECTED
CODEPAGE    NAME=page         START=0x0100      END=0x7DBF
CODEPAGE    NAME=debug        START=0x7DC0      END=0x7FFF        PROTECTED
CODEPAGE    NAME=idlocs       START=0x200000    END=0x200007      PROTECTED
CODEPAGE    NAME=config       START=0x300000    END=0x30000D      PROTECTED
CODEPAGE    NAME=devid        START=0x3FFFFE    END=0x3FFFFF      PROTECTED
CODEPAGE    NAME=deeprom      START=0xF00000    END=0xF003FF      PROTECTED

ACCESSBANK  NAME=nvram        START=0x0         END=0x7
ACCESSBANK  NAME=accessram    START=0x8         END=0x7F
DATABANK    NAME=gpr0         START=0x80        END=0xFF
DATABANK    NAME=gpr1         START=0x100       END=0x1FF
DATABANK    NAME=gpr2         START=0x200       END=0x2FF
DATABANK    NAME=gpr3         START=0x300       END=0x3FF
DATABANK    NAME=gpr4         START=0x400       END=0x4FF
DATABANK    NAME=gpr5         START=0x500       END=0x5F3
DATABANK    NAME=dbgspr       START=0x5F4       END=0x5FF         PROTECTED
ACCESSBANK  NAME=accesssfr    START=0xF80       END=0xFFF         PROTECTED
```

```
SECTION     NAME=VRESET       ROM=vreset
SECTION     NAME=VINTH        ROM=vinthi
SECTION     NAME=VINTL        ROM=vintlo
SECTION     NAME=SEQ          ROM=seq
SECTION     NAME=PROG         ROM=page
SECTION     NAME=IDLOCS       ROM=idlocs
SECTION     NAME=DEEPROM      ROM=deeprom

SECTION     NAME=NVRAM        RAM=nvram
SECTION     NAME=ACCESS       RAM=accessram
SECTION     NAME=BANK0        RAM=gpr0
SECTION     NAME=BANK1        RAM=gpr1
SECTION     NAME=BANK2        RAM=gpr2
SECTION     NAME=BANK3        RAM=gpr3
SECTION     NAME=BANK4        RAM=gpr4
SECTION     NAME=BANK5        RAM=gpr5
```

Для понимания работы линкера необходимо ввести понятие логической секции: это область памяти определенного типа, имеющая жестко заданные адреса начала и конца, имя, атрибуты и предназначение, например: PROG (исполняемый код во Flash), SEQ (константные таблицы во Flash), NVRAM (область ACCESS BANK, не затираемая при «горячем» рестарте), CONFIG (конфигурационное слово) и т.д.

В исходных текстах секции адресуются по имени, при этом каждый исходный файл может размещаться в произвольном количестве секций (включая RAM и EEPROM), единственное ограничение – нельзя обращаться к одной и той же секции дважды из одного файла. Фактическим размещением объектных файлов по адресам памяти занимается линкер, что освобождает разработчика от ошибок перекрытия (особенно актуально для RAM, т.к. при объявлении переменных через **EQU** или **CBLOCK** какой-либо контроль над адресацией отсутствует) и позволяет использовать память наиболее полно благодаря размещению фрагментов кода и данных вплотную в пределах каждой секции. Перемещение кода по памяти осуществляется редактированием секций в линкер-скрипте. Например, при переходе от «фабричной» прошивки к варианту с бутлоадером (загрузчиком) требуется правка только нескольких строк:

```
CODEPAGE    NAME=blreset      START=0x0000      END=0x0007      PROTECTED
CODEPAGE    NAME=blinthi      START=0x0008      END=0x0017      PROTECTED
CODEPAGE    NAME=blintlo      START=0x0018      END=0x001F      PROTECTED
CODEPAGE    NAME=seq          START=0x0020      END=0x00FF      PROTECTED
CODEPAGE    NAME=bootldr      START=0x0100      END=0x07FF      PROTECTED
CODEPAGE    NAME=vreset       START=0x0800      END=0x0807      PROTECTED
CODEPAGE    NAME=vinthi       START=0x0808      END=0x0817      PROTECTED
CODEPAGE    NAME=vintlo       START=0x0818      END=0x081F      PROTECTED
CODEPAGE    NAME=page         START=0x0900      END=0x7DBF
```

Как видно из комментариев в заголовке файла, за основу взят пример скрипта из комплекта MPASM, рассчитанный на работу с внутрисхемными отладчиками – для этой цели резервируются кодовые страницы **debug** (Flash) и **debugspr** (RAM). В принципе, даже без определений пользовательских секций (выделено жирным) скрипт может использоваться в проекте, однако конкретные адреса в таком случае задаются принудительно в исходных текстах или остаются на усмотрение линкера, при этом первое откатывает структуру проекта к варианту 2), а второе дает неудовлетворительный результат, т.к. линкер не способен угадать полет мысли разработчика. Проще говоря, секции – ключ к успеху.

В целом построение и смысл скрипта интуитивно понятны, и на директивы есть подробные описания в справке по линкеру.

Структура asm-файла

Для размещения кода в определенных секциях наиболее часто применяются следующие директивы:

<sectname> **code** - исполняемый код, размещение во Flash
<sectname> **udata** - данные, размещение в банке общего назначения
<sectname> **udata_acs** - данные, размещение в банке ACCESS (PIC18)
<sectname> **udata_shr** - данные, размещение в банке SHARED (PIC16)

Используемые директивы (**code** / **udata** / …) должны совпадать с типом области памяти (**CODEPAGE** / **DATABANK** / …), на которую ссылается секция **<sectname>** в линкер-скрипте.

Примером исходного файла, работающего с приведенным скриптом, может послужить следующий фрагмент:

```
;*****************************************************************
;
;     Project:    txtlib
;     File:       int.asm
;
;*****************************************************************
      #define     _INT
      #include    "int.inc"
;*****************************************************************
ACCESS      udata_acs

IDiv10m     res   1
IDiv100m    res   1
IDiv1s      res   1
KLatch      res   1
KSwitch     res   1

;*****************************************************************
; Interrupt Service Routine (ISR) - TMR0 overrun
;*****************************************************************
VINTH code

      bra         Int

;*****************************************************************
VINTL code

      bra         $

;*****************************************************************
;
;*****************************************************************
PROG  code

Int
      bcf         T0IF

;*****************************************************************
; 1 ms
      bsf         F1m

;*****************************************************************
; 10 ms
      decfsz      IDiv10m, f
      bra         IntRet

      bsf         F10m
      movlw       .10
      movwf       IDiv10m

      rcall       IntKbd
      ...
      ...
```

Смысл приведенного кода значения не имеет, а структура довольно прозрачна: файл содержит четыре блока, размещающихся в секциях **ACCESS** (данные), **VINTH** (вектор высокоприоритетного прерывания), **VINTL** (вектор низкоприоритетного прерывания) и **PROG** (основная область исполняемого кода). Новая директива **RES** резервирует под каждую переменную необходимое количество байт, причем на этапе компиляции известны только размер переменной и секция, в которой она располагается. Отсюда вывод: если требуется расположить код или переменные в определенной области памяти, необходимо создать отдельную секцию – это касается всех областей памяти: Flash, RAM, DEEPROM.

Также при помощи конструкции вида

```
BANK0 idata
ID          db    “TXTLib v0.80”
DFlags      dw    b’1100010000010111’
            ...
```

можно создавать блоки инициализированных переменных. При этом линкер размещает в программной памяти секцию **.cinit** типа **romdata**, содержащую таблицу начальных значений. Однако необходимость подключения внешнего файла с процедурой инициализации (**C18i.o** или **IDASM16.asm** для PIC18 и Mid-Range соответственно) перечеркивает всю красоту исходной задумки, т.к. написать свою подпрограмму оказывается проще и нагляднее.

Заголовочные файлы

Следующим шагом к модульному проекту служит написание заголовочного файла, позволяющего корректно работать с переменными и вызывать подпрограммы данного модуля из внешней среды. Директивы **global** и **extern** позволяют, как и следует из значений слов, объявить метку в глобальном пространстве имен (что открывает доступ внешним модулям) и объявить метку как внешнюю (чтобы компилятор корректно обрабатывал обращение к метке, определенной в другом модуле и не представленной в локальном пространстве имен). Проще говоря, если из одного модуля необходимо обратиться к метке **Entry** в другом, то в фалах должны быть сделаны следующие определения:

вызывающий	**extern Entry**
вызываемый	**global Entry**

Самый простой и очевидный вариант – прописать в заголовочном файле каждого модуля все необходимые внешние и глобальные метки как **extern** или **global** соответственно, однако при этом нарушается принцип модульности проекта, т.к. определения расползаются по нескольким файлам. При этом внесение изменений во внешние интерфейсы модулей (т.е. список доступных снаружи меток) или подключение новых модулей приводит к многократным исправлениям схожих фрагментов текста, что само по себе довольно утомительно, и предоставляет отличную возможность ошибиться. Идеальный вариант – подключение внешнего модуля через единый заголовочный файл – требует неочевидной организации. Например, такой:

```
;*****************************************************************
;
;     Project:    txtlib
;     File:       int.inc
;
;*****************************************************************
      #ifdef      _INT
      #undefine   _INT
```

```
        #include    "txtlib.inc"
        #include    "rtc.inc"

        global      Int, IntInit
        global      KSwitch, KLatch

        #define     KMASK b'00111100'

;*****************************************************************************
        #else

        extern      Int, IntInit
        extern      KSwitch, KLatch

        #endif
;*****************************************************************************

        #define     KLeft       KSwitch, 0
        #define     KRight      KSwitch, 1
```

Это заголовочный файл к приведенному выше **int.asm**, и смысл его структуризации сводится к следующему: поскольку один и тот же текст должен содержать определения как для своего модуля, так и для внешнего (его подключающего), то, очевидно, он должен состоять из двух частей, каждая из которых компилируется в соответствующем случае. Для распознавания подключения к своему или к внешнему модулю используется символ **_INT**, который устанавливается только в **int.asm** – при этом компилируется часть, выделенная синим. При подключении извне метка **_INT** не определена, компилируется зеленая часть. Все, что идет после **#endif**, компилируется независимо от ситуации – в этой области располагаются определения констант и символов, общих для всех участвующих файлов. Удаление метки **_INT** директивой **#undefine** необходимо, чтобы при последующих обращениях к файлу **int.inc** из внешних модулей компилировалась «внешняя» часть. Файл **txtlib.inc** в данном примере служит общим заголовочным файлом всего проекта и содержит определения, необходимые для компиляции каждого модуля (подключение **18F2520.inc**, установки **errorlevel**, общие константы и символы, …); **rtc.inc** – пример внешнего модуля, вызываемого из **int.asm**.

Библиотеки

При сборке библиотеки вместо линкера (**MPLINK**) вызывается утилита **MPLIB**, и основное отличие с точки зрения структуризации заключается в отсутствии проверки внешних меток, что в дальнейшем не только позволяет вызывать библиотечные функции, но и дает доступ библиотеке к внешним модулям. Т.е. объявление несуществующих переменных и подпрограмм как **extern** позволяет успешно собрать библиотеку, однако очевидно, что при компиляции конечного проекта в hex-файл все использованные метки должны существовать в глобальном пространстве имен. Приведенный шаблон заголовочного файла годится и для библиотек; с другой стороны, подключение «внутреннего» блока требуется только один раз (на этапе сборки библиотеки), поэтому публичная версия заголовочного файла может, в целях простоты и наглядности, содержать только «внешний» и «общий» блоки, без условной компиляции.

MAP-файл и ошибки компоновки

Контроль над работой линкера и поиск ошибок компоновки были бы практически невозможны, если бы в процессе сборки не создавался подробный отчет о проделанной работе. По умолчанию он отключен, поэтому для начала предстоит сходить в опции

сборки проекта, далее – настройки линкера, и установить галочку «**Generate map file**» (а заодно и «**Suppress COD-file generation**»).

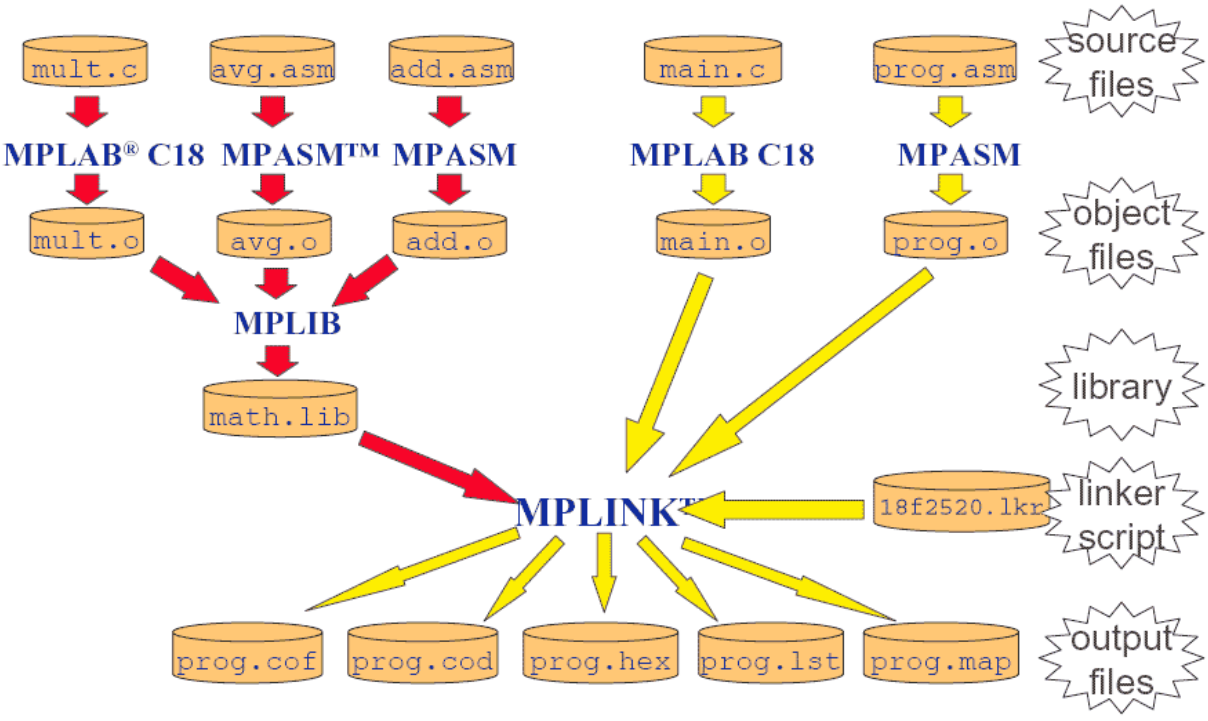
MAP-файл открывается любым текстовым редактором, и содержит сводную информацию относительно сборки проекта:

- адреса и фактические размеры логических секций;
- подробную статистику по заполнению программного Flash;
- списки всех меток в алфавитном и адресном порядке.

Если в процессе компоновки возникает ошибка, то получить дополнительные сведения (помимо сообщения линкера) также можно из map-файла, т.к. его генерация будет остановлена на проблемном месте.

Итоги

На примере небольшого проекта видно что, использование простых приемов компоновки позволяет добиться высокой степени инкапсуляции отдельных задач в виде модулей, что упрощает работу с крупными сложными проектами, уменьшает количество ошибок сопряжения отдельных частей проекта и экономит время благодаря повторному использованию модулей и библиотек в новых проектах.



Автоматический менеджмент памяти со стороны линкера и ограничение области видимости меток минимизируют вероятность появления ошибок структуризации проекта, при этом, в отличие от языков высокого уровня, ассемблер позволяет сохранить полный контроль над кристаллом и достичь максимальной эффективности кода, как по объему, так и по производительности.