# Приемы эффективного проектирования

## Логическая организация проекта

Разделение проекта на логически законченные модули упрощает проектирование и повторное использование кода. Чтобы безошибочно разделять и связывать между собой отдельные модули, необходимо сформулировать основные принципы.

В общем случае проект можно разделить на три уровня по степени специфичности кода и принципам использования:

| название | Описание | представление | степень адаптации |
|---|---|---|---|
| Алгоритмический | уникален, инкапсулирует основную функциональность проекта | исх. тексты | полная |
| Вспомогательный | набор типовых решений, адаптированных под текущую задачу | исх. тексты | умеренная |
| Системный | набор типовых решений, не требующих адаптации под проект | библиотеки / исх. Тексты | без адаптации |

Алгоритмический уровень может разбиваться на подуровни и/или отдельные модули, но смысл приведенной классификации не изменится. Например, логически завершенные части алгоритмического модуля (таблицы, математические алгоритмы, прерывания, …) целесообразно вынести в отдельный файл, однако подключать его имеет смысл через #include, а не в виде модуля, и уровень файла остается прежним – алгоритмический. Системный уровень используется, в первую очередь, для драйверов периферии. Правильно спроектированный системный уровень допускает замену программного модуля на альтернативный (например, драйвер аппаратного устройства на программный эмулятор) без доработок модулей более высокого уровня.

Методом исключения оставшиеся модули следует отнести к вспомогательному уровню. Как правило, это алгоритмические библиотеки, допускающие адаптацию под текущую задачу (способы передачи и типы параметров, использование памяти и периферии) и вызовы из смежных модулей. Типичные функции – задержки, преобразования типов, операции с памятью, первичная обработка данных.

Связи проводятся сверху вниз, от более специфичного модуля к менее. Допустимы и по горизонтали между модулями одного уровня, однако одноранговыми связями желательно не злоупотреблять, т.к. это может приводить к неочевидным зависимостям, как следствие, к ошибкам.

Наиболее предпочтительным уровнем является системный, т.к. подразумевает высокую инкапсуляцию и максимальные возможности повторного использования кода. Следовательно, каждый модуль проекта должен принадлежать максимально низкому уровню.

Использование префиксов для обозначения принадлежности меток к модулям и имен, раскрывающих суть подпрограммы или назначение переменной, повышают самодокументируемость кода, что в предельном случае дает возможность разобраться в работе программы по алгоритмическому уровню, не углубляясь во вспомогательный и системный.

**Стек данных**

Проблема хранения временных переменных эффективно решается с применением стека данных. Поскольку архитектура 8-разрядных микроконтроллеров PIC не подразумевает существования такого стека, реализовать его можно при помощи косвенной адресации. Наиболее подходит для этого семейство PIC18, поскольку содержит три регистра косвенной адресации с автоинкременом-декрементом.

Описание:

```
STACKLEN  = 0x20
Stack     res STACKLEN
```

Инициализация:

```
lfsr x, Stack-1
```

Запись:

```
push  WREG      movwf  PREINCx
push  GPR       movff  GPR, PREINCx
```

Чтение:

```
pop  WREG       movf   POSTDECx, w
pop  GPR        movff  POSTDECx, GPR

; x = номер используемого FSR
```

Применение пары PREINC-POSTDEC делает возможным обращение к вершине стека без извлечения значения. Пример – построение цикла:

```
      movlw     .10             ; подготовка счетчика
      movwf     PREINC2         ; запись в стек
Loop:
      …                         ; тело цикла
      decfsz    INDF2, f        ; счет без извлечения из стека
      bra       Loop
      movf      POSTDEC2, f     ; освобождение стека
```

Очевидно, что все операции записи и чтения стека должны быть парными. Проверить правильность работы подпрограмм на этапе отладки можно довольно просто: установить точки останова на обращения к ячейкам Stack-1 и Stack + STACKLEN. Помимо этого хорошо принять практику написания кода таким образом, чтобы минимизировать риск рассогласования пар записи-чтения. В вышеприведенном примере операция освобождения стека является явным потенциальным источником ошибок, и описать цикл более надежно можно так:

```
      movlw     .10             ; подготовка счетчика
LOOP:
      movwf     PREINC2         ; запись в стек
      …                         ; тело цикла
      decfsz    POSTDEC2, w     ; счет с извлечением из стека
      bra       LOOP
```

Здесь операция освобождения стека отсутствует, цикл выглядит привычно и прозрачно. Однако есть и минусы – тело цикла становится длиннее на 1 инструкцию, и при декременте счетчика задействуется WREG. Для всех случаев, где возможно, рекомендуется использование именно второй конструкции.

Предлагаемая реализация стека не накладывает ограничений на применение из прерываний, в т.ч. для сохранения контекста.

**Синхронные системы**

Привязка задачи к реальному времени и синхронизация процессов между собой требуются практически в каждом проекте. Очевидно, что программные задержки не являются оптимальным решением, т.к. зависят от тактовой частоты, синхронизируются относительно текущего момента вместо абсолютных временных меток, и не позволяют выполнять фоновые задачи. Этих недостатков лишен программно-аппаратный метод синхронизации, в котором таймер используется в качестве независимого опорного генератора высокой точности. Например, используя Timer0 для генерации прерывания с периодом 1 мс и каскад счетчиков-делителей в прерывании, можно получить набор временных меток с интервалами 1-10-100-1000 мс, которые в дальнейшем могут использоваться подпрограммами генерации задержек и процессами при взаимной синхронизации. Поскольку при таком подходе требуется только ожидать установки нужного флага, то подпрограммы задержек могут вызывать фоновые подпрограммы, имеющие недетерминированное время завершения:

```
Delay10m                ; задержка WREG * 10 мс
     rcall     Idle     ; низкоприоритетные фоновые задачи
     btfss     F10m     ; проверка 10-мс флага
     bra       Delay10m

     bcf       F10m
     addlw     -1       ; декремент счетчика в WREG
     bz        Delay10m ; повтор, если не 0
     return
```

Строго говоря, максимальное время выполнения не должно превышать период опорных меток, иначе может возникать пропуск тактов – из этого следует, что с фоновыми процессами предпочтительнее использовать задержки с максимально грубыми метками: 5 x 10 мс лучше, чем 50 x 1 мс. С другой стороны, максимальная погрешность задержки с несинхронным началом равна периоду меток, т.о. использование высокочастотной опоры повышает точность.

Генераторы разнообразных тайм-аутов (например, отключение подсветки при бездействии, ожидание ответа от медленного устройства и другие, измеряющиеся сотнями мс и более) целесообразно строить на основе выделенного инкрементального счетчика, работающего в прерывании, и устанавливающего флаг при достижении заданного значения. Основному процессу необходимо периодически обнулять значение таймера (на это требуется всего одна инструкция) и проверять состояние флага переполнения. Если действие при тайм-ауте не требует использования существенных ресурсов и вызова подпрограмм алгоритмического уровня (см. «Логическая организация проекта»), то выполняться оно может непосредственно в прерывании (например, отключение подсветки).

Максимальная автоматизация процессов через прерывания – ключ к надежности и быстродействию системы.

**#define**

Принцип действия директивы предельно прост:

```
#define   [ident]   [string]
```

при компиляции заменяет все совпадающие идентификаторы [ident] на строку [string]. Выгоды очевидны:

- самодокументируемость: название идентификатора говорит о его назначении;
- единство: модификация определения = модификация всех обращений;
- безошибочность обращений: адресуемые биты привязаны к своим регистрам.

Хорошей практикой может стать доработка процессорных inc-файлов с применением директивы #define для всех регистров специальных функций. Готовые к использованию примеры таких файлов можно найти в разделе, посвященном данной статье.