19.19.1 Визначити проблеми модульного програмування на всіх етапах життєвого циклу програми

Мы неоднократно подчеркивали один из существенных недостатков программ на языке ассемблера, а значит, и самого языка, — недостаточную наглядность. По про шествии даже небольшого времени программисту порой бывает трудно разобраться в деталях им же написанной программы. А о чужой программе и говорить не приходится. Если в ней нет хотя бы минимальных комментариев, то разобраться с тем, что она делает, довольно трудно. Причины этого тоже понятны — при написании программы на языке ассемблера человек должен запрограммировать самые элементарные действия или операции. При этом он должен учитывать и контролировать состояние большого количества данных. Из-за элементарности программи руемых операций реализация одного и того же алгоритма может быть произведена по крайней мере несколькими способами. Эта неоднозначность влечет за собой Непредсказуемость, что и затрудняет процесс обратного восстановления исходного алгоритма по ассемблерному коду.

По мере накопления опыта эти проблемы частично снимаются. Но одного опыта мало. Ситуация усугубляется, если работает коллектив разработчиков. Тут уже нужны специальные средства.

Концепцию модульного программирования можно сформулировать в виде нескольких понятий и положений. Основа концепции модульного программирования — *модуль*, который является продуктом процесса разбиения большой задачи на ряд более мелких функционально самостоятельных подзадач. Этот процесс называется *функциональной декомпозицией задачи*. Каждый модуль в функциональной декомпозиции представляет собой ≪черный ящик≫ с одним входом и одним выходом. Модули связаны между собой только входными и выходными данными. Модульный подход позволяет безболезненно производить модернизацию программы в процессе ее эксплуатации и облегчает ее сопровождение. Дополнительно модульный подход позволяет разрабатывать части программ одного проекта на разных языках программирования, после чего с помощью компоновочных средств объединять их в единый загрузочный модуль.

Так как отдельный модуль в соответствии с концепцией модульного программирования — это функционально автономный объект, то он ничего не должен знать о внутреннем устройстве других модулей, и наоборот, другим модулям также ничего не должно быть известно о внутреннем устройстве данного модуля. Однако должны быть какие-то средства, с помощью которых можно связать модули. Телевизор и видеомагнитофон

могут быть разными, но связь между ними одинакова. Та же идея лежит и в организации связи модулей. Внутреннее устройство модулей может совершенствоваться, они вообще могут в следующих версиях писаться на другом языке, но в процессе их объединения в единый исполняемый модуль этих особенностей не должно быть заметно. Таким образом, каждый модуль должен иметь такие средства, с помощью которых он извещал бы транслятор о том, что некоторый объект (процедура, переменная) видимым вне этого модуля. И наоборот, нужно объяснить транслятору, что некоторый объект находится вне данного модуля. Это позволит транслятору правильно сформировать машинные команды, оставив некоторые их поля не заполненными. Позднее, на этапе компоновки, программа TLINK (TASM) или программа компоновки языка высокого уровня произведут настройку модулей и разрешат все внешние ссылки в объединяемых модулях.

Если входные данные для модуля (аргументы) — переменные, то один и тот же модуль можно использовать многократно для разных наборов значений этих переменных. Но как организовать передачу значений переменных в модуль (процедуру)? При программировании на языке высокого уровня программист ограничен в выборе способов передачи аргументов теми рамками, которые для него оставляет компилятор. В языке ассемблера практически нет никаких ограничений на этот счет, и, фактически, решение проблемы передачи аргументов предоставлено программисту.

На практике используются следующие варианты передачи аргументов в модуль (процедуру):

через регистры; через общую
область памяти; через стек;

с помощью директив EXTRN и PUBLIC.

В каком виде можно передавать аргументы в процедуру? Ранее упоминалось, что

передаваться могут либо данные, либо их адреса (указатели на данные). В языке высокого уровня это называется передачей по значению и по адресу, соответственно.

В отличие от языков высокого уровня, в языке ассемблера нет отдельных понятий для процедуры и функции. Организация возврата результата из процедуры полностью ложится на программиста. Если исходить из того, что получение результата — частный случай передачи аргументов, то программисту доступны три варианта возврата значений из процедуры.

- С использованием регистров. Ограничения здесь те же, что и при передаче данных, — это небольшое количество доступных регистров и их фиксированный размер. Функции DOS используют именно этот способ. Из рассматриваемых здесь трех вариантов данный способ является наиболее быстрым, поэтому его есть смысл задействовать для организации критичных по времени вызова процедур с малым количеством аргументов.
- С использованием общей области памяти. Этот способ удобен при возврате большого количества данных, но требует внимательности в определении областей данных и подробного документирования, чтобы устранить неоднозначность при трактовке содержимого общих участков памяти.
- С использованием стека. Здесь, подобно передаче аргументов через стек, также требуется регистр BP.

Если программа не предназначена для решения каких-то системных задач, требующих максимально эффективного использования ресурсов компьютера, если к ней не предъявляются сверхжесткие требования по размеру и времени работы, если вы не ≪фанат≫ ассемблера — то, на мой взгляд, следует подумать о выборе одного из языков высокого уровня. Существует и третий, компромиссный путь — комбинирование программ на языке высокого уровня с кодом на ассемблере. Такой способ обычно используют в том случае, если в вашей программе есть фрагменты, которые либо вообще невозможно реализовать без ассемблера, либо ассемблер может значительно повысить эффективность работы программы.