# Изучение администратора ресурсов ОС QNX 6

*Операционная система реального времени QNX6 (www.qnx.com) пользуется заслуженным уважением программистов. Ее уникальная архитектура [1] позволила не только получить непревзойденные технические характеристики, но и заметно упростить программирование встраиваемых систем управления.*

*Все версии QNX базируются на обмене сообщениями между процессами, так называемом механизме Send-Receive-Reply (SRR). В шестой версии был добавлен механизм администратора ресурсов (PM)[1], который является надстройкой над базовым. Применение PM дает некоторые преимущества и настоятельно рекомендуется разработчиками ОС, особенно при написании драйверов.*

*Программирование администраторов ресурсов подробно описано в документации и в [2]. Данная статья является дополнением к этим материалам. Целью работы является попытка объяснить назначение администратора ресурсов, его идеологию и структуру. Это облегчит изучение PM и лучшее его понимание.*

*Статья предназначена программистам, осваивающим QNX6, но может быть полезна и более опытным разработчикам.*

## Понятие администратора ресурсов

QNX является POSIX-совместимой операционной системой. По традиции, восходящей к первым версиям UNIX, в этом стандарте внешние устройства представляются в виде файлов. Доступ к ним осуществляется путем вызова функций open(), close(), read(), write() и других. Так, например, последовательный порт для пользователя виден как файл /dev/ser1, параллельный как /dev/par и так далее. Работу с устройством обеспечивает драйвер – программа, написанная по специальным правилам и, как правило, являющаяся частью ядра.

QNX имеет микроядерную архитектуру. Микроядро ответственно только за обеспечение механизма SRR и диспетчеризацию потоков. Все остальные функции, относящиеся в других ОС к ведению ядра, в QNX выполняют отдельные независимые процессы. С точки зрения ОС QNX процессы, обеспечивающие работу с устройствами, ничем не отличаются от любых других процессов, в том числе и прикладных пользовательских. Чтобы подчеркнуть разницу между специальными процессами QNX и драйверами в других ОС был введен термин «администратор ресурсов».

Администратор ресурсов в QNX – процесс, обращение к которому производится посредством вызова функций для работы с файлами. Такой процесс организуется по определенным правилам с применением специальной библиотеки.

Применение администраторов ресурсов предоставляет все преимущества (и недостатки), связанные с использованием стандартов. Программы приобретают единый вид, что упрощает их понимание, облегчается портирование. Возможно обращение к процессам с помощью утилит ОС. Программные комплексы становятся более гибкими, так как возможно заменить один процесс другим, всего лишь задав другое имя файла.

Платой за стандартизацию являются некоторые накладные расходы. В данном случае расходы крайне незначительны, так как механизмы SRR и PM являются основой построения самой ОС и глубоко оптимизированы.

К настоятельному совету разработчиков QNX использовать администраторы ресурсов всюду, где возможно, следует прислушаться.

---

[1] Оригинальное название - resource manager (RM), жаргонное название - resmgr. Устоявшийся перевод термина - администратор ресурсов, но сокращение не введено. Предлагается использовать PM, что не является аббревиатурой русского названия, но которое можно считать «переводом» RM.

## Механизм Send-Receive-Reply

Изучение администратора ресурсов начнем с рассмотрения базового механизма Send-Receive-Reply, и структуры программ, написанных с его использованием. Затем выясним, как изменятся программы с введением некоторых допущений.

Механизм SRR обеспечивает обмен сообщениями между процессами[2]. Сначала один процесс организует канал приема информации, к этому каналу подключается второй процесс. Затем второй процесс вызывает функцию Send() с некоторым сообщением в качестве одного из параметров и блокируется до прихода ответного сообщения. Первый процесс асинхронно вызывает функцию Receive() и блокируется до получения сообщения. После получения сообщения выполняются необходимые действия и вызывается функция Reply(), посылающая ответное сообщение второму процессу. Второй процесс разблокируется. Цикл обмена завершен.

Сообщение является последовательностью байтов без предопределенного содержания. Интерпретация сообщений полностью возлагается на программиста.

Рис.1 иллюстрирует единичный цикл обмена. Тонкими линиями показано блокированное состояние процессов, широкими – активное.

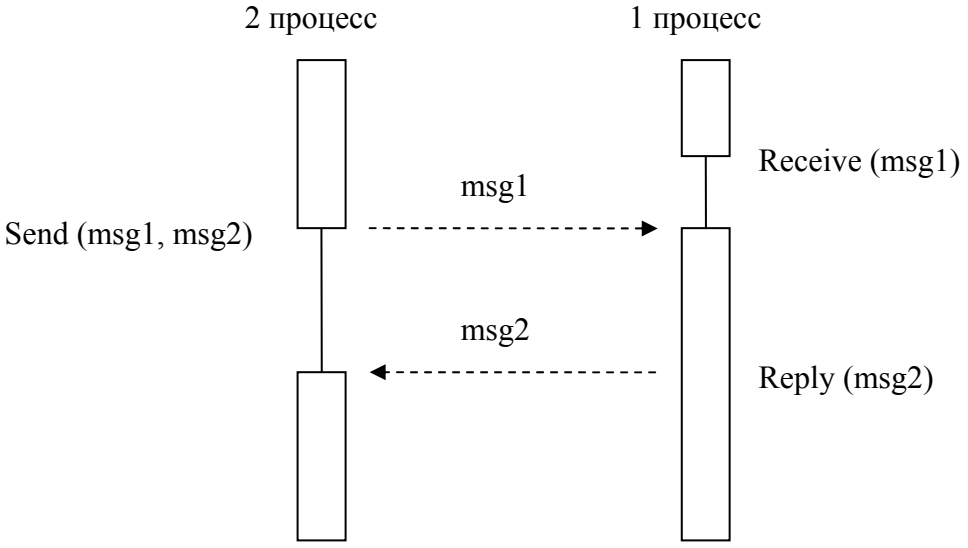


Рис.1 Иллюстрация межпроцессного обмена

Как правило, программы организуются по технологии «клиент-сервер». Сервер в цикле принимает сообщения, обрабатывает их и посылает ответ клиенту. Клиент запрашивает сервис у сервера и ожидает ответа.

Сервер может обрабатывать сообщения нескольких типов. В этом случае, очевидно, сообщение должно описываться объединением структур, каждая из которых содержит код команды и, если необходимо, данные. Псевдокод программ показан на рис.2. Инициализация программ, организация канала и описание структур опущены.

Для расширения функциональности следует описать новую структуру, добавить новый код команды, связать его с функцией в операторе switch и написать функцию-обработчик.

[2] В шестой версии QNX обмен сообщениями может происходить и между потоками. Но по традиции говорят о межпроцессном взаимодействии.

клиент

```
msg1.type = c1;
msg1.data = data;
Send (msg1, msg2);
```

сервер

```
while (1) {
   Receive (msg1);
   switch (msg1.type) {
      c1: rc = f1 (msg1, msg2);
      c2: rc = f2 (msg1, msg2);
          …
      cn: rc = fn (msg1, msg2);
   }
   msg2.status = rc;
   Reply (msg2);
}
int f1 (msg1, msg2) {…};
int f2 (msg1, msg2) {…};
…
int fn (msg1, msg2) {…};
```

Рис.2 Псевдокод программ, выполненных по технологии «клиент-сервер»

Для удобства использования вместе с сервером может поставляться библиотека, в функциях которой заполняются поля сообщений и вызывается Send(). Для каждого кода команды, то есть, для каждого предоставляемого сервиса создается соответствующая функция. На рис.3 показано использование такой библиотеки.

клиент

```
rc = GetData (data);
```

библиотека

```
int GetData (data) {
   msg1_t msg1;
   int rc;

   msg1.type = getdata;
   msg1.data = data;
   Send (msg1, msg2);
   rc = msg2.status;

   return rc;
}
```

Рис.3 Использование библиотеки доступа к серверу

Предположим, что количество команд фиксировано и неизменяемо. Тогда обработка кода команды также будет неизменяемой, ее можно извлечь из текста сервера, поместив в библиотеку. Программа сервера примет вид, показанный на рис. 4.

сервер

```
while (1) {
   Receive (msg1);
   Handler (msg1);
}

int f1 (msg1, msg2) {…};
int f2 (msg1, msg2) {…};
…
int fn (msg1, msg2) {…};
```

библиотека

```
int Handler (msg1) {
   switch (msg1.type) {
      c1: rc = f1 (msg1, msg2);
      c2: rc = f2 (msg1, msg2);
          …
      cn: rc = fn (msg1, msg2);
   }
   msg2.status = rc;
   Reply (msg2);
   return rc;
}
```

Рис.4 Введение библиотеки сервера

Сделаем еще одно допущение. Пусть для каждой команды существует обработчик по умолчанию. Конкретные функции-обработчики могут выполнять или только действия по умолчанию, или добавлять к ним какую-либо обработку, или полностью заменять обработчик по умолчанию. Текст сервера и библиотеки видоизменятся, как отражено на рис. 5.

сервер

```
while (1) {
   Receive (msg1);
   Handler (msg1);
}

int f1 (msg1, msg2) {
//обработка по умолчанию
// «заглушка»
  rc = f1_def(msg1, msg2);
  return rc;
};

int f2 (msg1, msg2) {
// добавлена обработка
…
  rc = f2_def(msg1, msg2);
  return rc;
};
…
int fn (msg1, msg2) {
// обработка заменена
…
   return rc;
};
```

библиотека

```
int Handler (msg1) {
   case (msg1.type) {
      c1: rc = f1 (msg1, msg2);
      c2: rc = f2 (msg1, msg2);
          …
      cn: rc = fn (msg1, msg2);
   }
   msg2.status = rc;
   Reply (msg2);
   return rc;
}
int f1_def (msg1, msg2) {…};
int f2_def (msg1, msg2) {…};
…
int fn_def (msg1, msg2) {…};
```

Рис.5 Добавление обработчиков по умолчанию

Если команд много, и обработчики по умолчанию заменяются редко, то иметь большое число функций-заглушек не эффективно. Хотелось бы иметь возможность добавлять конкретные функции-обработчики, только в случае необходимости. Это нетрудно сделать, введя таблицу вызовов. Таблица представляет собой массив указателей

на функции. При инициализации она заполняется указателями на обработчики по умолчанию. Затем, если требуется, значения некоторых элементов заменяются указателями на конкретные функции-обработчики.

Программа сервера, которую приходится писать прикладному программисту, значительно упростится, что отражено на рис. 6.

сервер

```
Server () {
   InitTable (Table);

// замена указателей
   Table[2] = f2;
   Table[5] = f5;

   while (1) {
      Receive (msg1);
      Handler (msg1);
   }
}

int f2 (msg1, msg2) {
// добавлена обработка
…
  rc=f2_def(msg1, msg2);
  return rc;
};

int f5 (msg1, msg2) {
// обработка заменена
…
   return rc;
};
```

библиотека

```
int Handler (msg1) {
 switch (msg1.type) {
   c1:rc=(*Table[1]) (msg1, msg2);
   c2:rc=(*Table[2]) (msg1, msg2);
…
   cn:rc=(*Table[n]) (msg1, msg2);
 }
   msg2.status = rc;
   Reply (msg2);
   return rc;
}
int f1_def (msg1, msg2) {…};
int f2_def (msg1, msg2) {…};
…
int fn_def (msg1, msg2) {…};
```

Рис.6 Введение таблицы указателей на функции-обработчики

Заметим, что введение таблицы возможно только в том случае, если все обработчики имеют один тип, то есть одинаковый список параметров и возвращаемое значение.

Еще раз отметим, что приведенные тексты имеют условный вид. В них опущены важные моменты, присутствующие в реальных программах. Так, например, не показана организация канала приема сообщений и инициализация необходимых переменных.

## Администратор ресурсов

Вспомним, что обращение к администратору ресурсов производится путем вызова функций для работы с файлами. Вспомним также, что все в QNX построено на механизме SRR. По сути, функции для работы с файлами являются клиентской библиотекой для обращения к серверу. Внутри функций происходит составление сообщения и вызов Send().

Кому посылаются сообщения? При работе с файлами сначала вызывается функция open(), содержащая в качестве параметра имя файла и возвращающая файловый дескриптор. Последующие вызовы используют полученный дескриптор. В теле open() выполняется обращение (путем посылки сообщения, разумеется) к администратору

процессов QNX. Администратор процессов по имени файла находит номер канала обмена сообщениями, куда следует направлять последующие запросы.

Если открываемый файл находится на жестком диске, то за дальнейшую работу с ним будет отвечать администратор файловой системы, если файл представляет устройство, то запросы будут направлены соответствующему администратору ресурса (драйверу).

Такой механизм называется *отображением пространства имен путей*.

При инициализации администратор ресурса должен зарегистрировать некоторый путь, называемый *точкой монтирования*. Путь может быть именем файла или директорией. Допускается регистрация нескольких путей.

После регистрации пути процесс и становится администратором ресурса. Он заявляет о зоне своей ответственности в пространстве имен путей. Все дальнейшие обращения к зарегистрированному пути будут направляться зарегистрировавшему РМ.

Какие при этом будут выполняться действия, зависит только от программиста. РМ может работать с физическим устройством, тогда он будет драйвером в общепринятом понимании. Но он может быть и не связан с оборудованием, а быть одним из процессов в большом программном комплексе. В последнем случае возможности, предоставляемые администратором ресурсов, используются для организации каналов и обмена сообщениями между процессами. Преимущества такого подхода рассмотрены выше.

## Структура администратора ресурсов

Администратор ресурса должен обрабатывать все функции работы с файлами. Набор таких функций фиксирован. Обработка немногих вызовов изменяется для различных РМ. Например, функция chown() должна выполнять одни и те же действия для любого файла независимо от того, является ли он файлом на диске, соответствует устройству или процессу в программном комплексе. Для всех функций существуют обработчики по умолчанию.

Проницательный читатель сразу понял, что мы неспроста проводили надуманные преобразования сервера. Так мы постепенно подходили к структуре администратора ресурсов.

Минимальный полностью рабочий код РМ, приводимый в документации к ОС, занимает около 50 строк. Какая другая система может сравниться с QNX по простоте и компактности текста драйверов!

Для изучения структуры оставим в тексте минимального РМ только вызовы функций, убрав описания, проверки и комментарии.

```
int main (int argc, char** argv)
{
    dpp = dispatch_create ();
    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                      _RESMGR_IO_NFUNCS, &io_funcs);
    iofunc_attr_init (&attr, S_IFNAM | 0666, 0, 0);
    memset (&resmgr_attr, 0, sizeof resmgr_attr);
    id = resmgr_attach (dpp, &resmgr_attr, "/dev/sample",
                        _FTYPE_ANY, 0, &connect_funcs, &io_funcs, &attr);
    ctp = dispatch_context_alloc (dpp);
    while (1) {
        ctp = dispatch_block (ctp);
        dispatch_handler (ctp);
    }
}
```

Первой вызывается функция dispatch_create(). Не будем ее рассматривать, просто скажем, что так надо.

Обратим внимание на функцию iofunc_func_init(). Она инициализирует две таблицы указателей на функции. Дело в том, что все функции работы с файлами разбиты на две группы – функции установления соединения и функции ввода-вывода. Но принцип их использования и подстановки своих обработчиков такой же, как и рассмотренный выше.

Функция iofunc_attr_init() инициализирует атрибутная запись. Структуры данных РМ будут рассмотрены позже. Опять скажем, что так надо делать.

Вызов memset () в данном случае обнуляет еще одну структуру.

И наконец, вызывается функция resmgr_attach(), которая и есть функция регистрации пути ответственности. Данный вызов превращает процесс в администратор ресурсов.

Функция dispatch_context_alloc() распределяет память для обмена сообщенями.

Затем в бесконечном цикле вызывается функция dispatch_block(), которая внутри себя содержит вызов Receive(), и значит, блокирует процесс до получения сообщения, которое передается для обработки в dispatch_handler().

Как видно структура программы администратора ресурсов очень проста. Сначала вызывается ряд специальных функций, инициализируются структуры данных, регистрируется путь. Потом следует бесконечный цикл приема и обработки сообщений.

Данная статья рассматривает общие вопросы построения администраторов ресурсов. Для изучения всех деталей следует обратиться к литературе и документации.

## Структуры данных

Вопрос об используемых структурах данных весьма важен для понимания и использования РМ.

Представим систему, осуществляющую сбор аналоговой информации с помощью плат АЦП. Платы бывают разных типов, в компьютер может быть установлено несколько однотипных плат. Оцифрованные значения поступают в буфер, откуда затем считываются потребителем (прикладной программой).

Какие структуры данных должен иметь администратор ресурса (драйвер) платы АЦП?

Потребителей информации может быть несколько. Они считывают информацию из буфера независимо друг от друга. Драйвер обязан запоминать позицию в буфере для каждого потребителя. Значит, нужна структура для каждого потребителя, открывшего файл.

Плат может быть несколько, каждая со своими параметрами, например, адрес платы, усиление в канале и частота оцифровки. Следовательно, требуется структура данных, относящаяся к конкретной плате.

Платы могут быть различных типов. Каждый тип имеет свои характеристики. Например, массив возможных значений усиления и частоты оцифровки. И для такой информации необходима структура.

Все рассмотренные три типа структур данных предусмотрены в библиотеке администратора ресурсов.

Они называются блок управления открытым контекстом (open control block – OCB), атрибутная запись (attributes structure) и запись точки монтирования (mount structure).

Блок управления открытым контекстом создается для каждого открытия файла, атрибутная запись для каждого устройства, запись точки монтирования для каждого типа устройств. Последняя запись является опциональной.

Указатель на структуру OCB передается в функцию-обработчик. Структура OCB содержит указатель на атрибутную запись, которая в свою очередь содержит указатель на запись точки монтирования. Таким образом, внутри обработчика можно получить доступ ко всем структурам данных администратора ресурсов.

На рис.7 показано назначение и связь между структурами данных. Иллюстрируется ситуация, при которой в системе существует два однотипных устройства, причем первое используется двумя потребителями, а второе только одним.

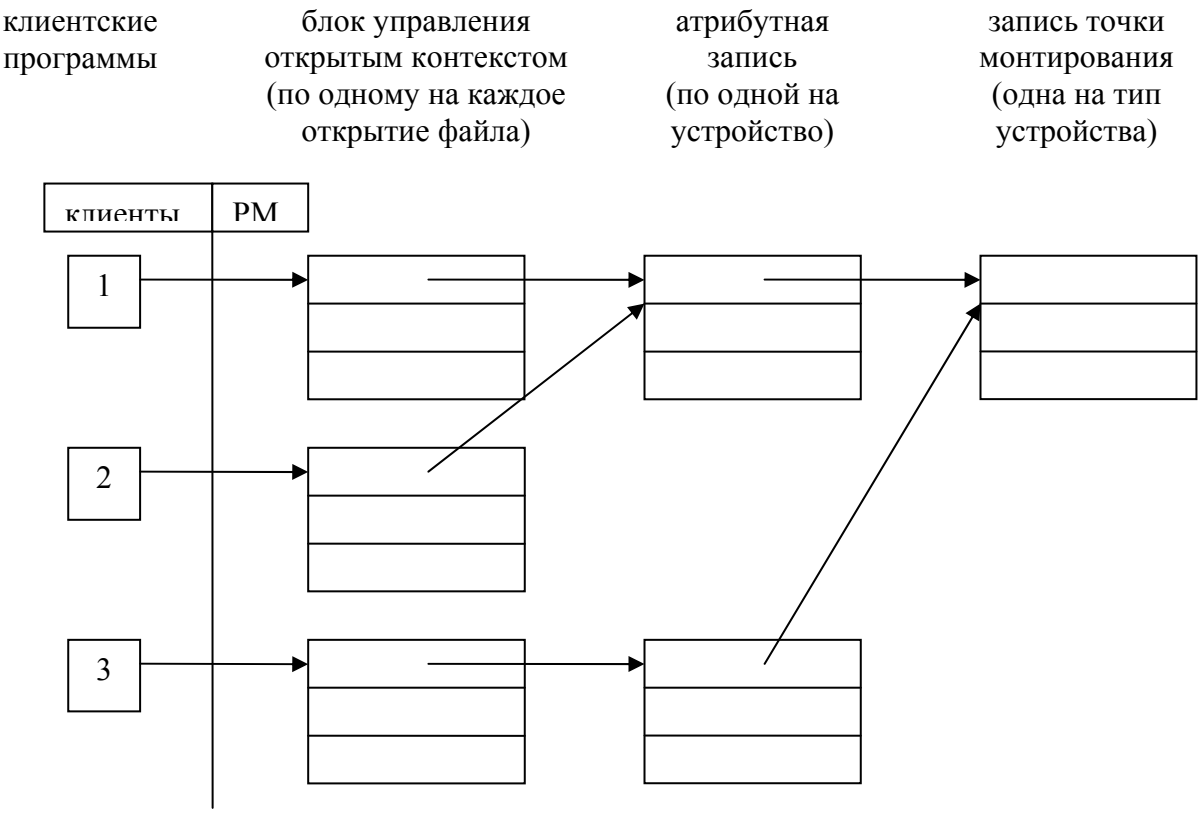


Рис.7 Назначение и связь между структурами данных

Все структуры данных допускают добавление полей прикладным программистом.

При расширении структуры OCB надо вызывать специальные функции, распределяющие и освобождающие память под расширенные структуры и выполняющие все необходимые действия. Указатели на эти функции должны быть занесены в специальные поля в записи точки монтирования.

## Что дальше

В настоящей работе рассмотрены самые общие вопросы создания администраторов ресурсов. Полная информация и детали реализации описаны в документации. Очень полезной является [2].

Опыт применения PM показал, что работу с ним можно упростить. Автор создал шаблон администратора ресурсов. Ознакомиться с ним можно на сайте http://resmgr.narod.ru.

Ну и конечно, лучшим способом освоения PM является его использование. Применяйте администраторы ресурсов и решайте возникающие вопросы по мере их появления. Надеюсь, данная статья поможет вам в вашей деятельности.

Игорь Желтиков (resmgr@narod.ru)

Литература

1. Операционная система реального времени QNX Neutrino 6.3. Системная архитектура - СПб.: БХВ-Петербург, 2005
2. Кертен Р. Введение в QNX/Neutrino 2. – СПб.: Петрополис, 2001