Модель ядра реального времени для встроенных систем

В. В. Никифоров, Н.В.Гуцалов, М.П.Червинский

Аннотация. Демонстрируется метод формального представления архитектуры ядра реального времени для встроенных систем. В порядке представления метода строится объектно-ориентированная модель, задающая архитектуру компактного системного ядра, обеспечивающего обслуживание аппаратно-ориентированных задач реального времени во встроенных системах.

Введение

Увеличение объемов производства и разнообразия средств микропроцессорной техники, расширение сфер их применения приводит к необходимости разработок различных операционных систем реального времени - от компактных, рассчитанных на обслуживание одночиповых микроконтроллеров, до мощных сетевых систем. Путь к удовлетворению требований высокой эффективности и надежности этих систем лежит через повышение ясности и стройности их логической организации. Это обстоятельство выдвигает актуальные задачи разработки рационально организованных базовых структур, которые представляли бы в обобщенном виде ключевые принципы организации вариантов операционных систем, ориентированных на достижение того или иного типа эффективности.

1. Аппаратно-ориентированные задачи и приложения

С расширением сферы применения встроенных компьютерных систем растет разнообразие вариантов требований к обслуживающим их операционным системам или компактным ядрам реального времени. Прикладные программы встроенных систем работают в тесном взаимодействии с множеством внешних физических и информационных процессов. Многие встроенные системы реального времени (СРВ) генерируют данные, используемые элементами окружающего автоматического оборудования (аппаратуры). Результатом действия таких СРВ является надлежащее функционирование этого автоматического оборудования. Назначение данных, вырабатываемых такой системой, состоит в должном характере их влияния на поведение внешних (по отношению к компьютерной системе) объектов. Вырабатываемые с этой целью данные, как правило, не запоминаются и не представляются в каких-либо иных формах. Представление этих данных в форме, ориентированной на восприятие человеком, относится не к основным, а к вспомогательным функциям системы. В этом смысле системы, обеспечивающие работу внешнего автоматического оборудования (аппаратуры), уместно называть аппаратно-ориентированными СРВ.

Сложная СРВ характеризуется широким разнообразием связанных с ней внешних процессов. Каждый из внешних процессов генерирует для компьютерной системы входные данные со специфической частотой и специфическими требованиями к времени отклика (т.е. к срокам генерации компьютерной системой соответствующих выходных данных). В связи с этим комплекс прикладных программ СРВ формируется в виде множества задач реального времени. Каждая из задач реализует собственный последовательный алгоритм. Приоритеты на пользование задачами центрального процессора компьютерной системы распределяются с учетом требований к времени отклика. Задачи взаимодействуют друг с другом путем обмена данными и синхронизационными сигналами. Важнейшей чертой СРВ является координированное взаимодействие задач с процессами во внешней среде и требование высокой реактивности: допустимое время реакции СРВ на внешние события зачастую ограничивается временными интервалами в доли миллисекунды.

Механизмы, обеспечивающие работу аппаратно-ориентированных приложений (взаимодействие задач друг с другом и с процессами во внешней среде), реализуются операционными системами реального времени (ОСРВ) или компактными ядрами реального времени, предоставляющими прикладным задачам соответствующие сервисы. Показатели реактивности СРВ определяются, в первую очередь, свойствами ядра.

Наряду с аппаратно-ориентированными прикладными задачами компьютерные системы решают прикладные задачи другого рода. Это задачи, для которых конечным назначением вырабатываемых ими данных является восприятие этих данных человеком: результаты представляются в виде печатных или экранных форм, сохраняются в файлах, базах данных или пересылаются по сетям, чтобы в конечном счете предстать в том или ином виде на бумаге, экране или в звуковой форме. Такие задачи и приложения функционируют под управлением операционных систем общего назначения (ОСОН). Разработка приложений для ОСОН опирается на использование большого числа вспомогательных программных средств, поддерживающих графические интерфейсы пользователя, коммуникационные подсистемы, базы данных и т.п. Все эти вспомогательные средства создаются для работы в рамках конкретного типа ОСОН, поэтому и работа использующих их приложений привязана к этому типу. К ОСОН и поддерживаемым ими приложениям не предъявляется тех жестких требований реактивности, которые характерны для аппаратно-ориентированных приложений вместе с поддерживающими их операционными системами и ядрами реального времени.

В последнее время все более широкое распространение получают встроенные системы с программными приложениями комбинированного типа. В рамках таких комбинированных приложений аппаратно-ориентированные задачи должны решаться совместно с задачами, ориентированными на восприятие человеком. Примером таких систем комбинированного назначения являются компьютерные системы, встроенные в современные автомобили, где жесткие требования своевременных реакций на работу внешнего оборудования сочетается с требованиями поддержки графического взаимодействия с водителем, использования баз данных, доступа к глобальным коммуникационным системам. Одним из эффективных подходов к построению систем комбинированного назначения является использование двухъядерной операционной системы, в которую наряду с ОСОН включается ядро реального времени, поддерживающее аппаратно-ориентированные задачи. Известны реализации таких двухъядерных систем на базе ОСОН Linux [1, 2]. Однако аналитические и экспериментальные оценки показывают недостаточно высокий уровень качества реализации этих систем [3]. При этом качество реализации ядра реального времени существенно уступает качеству реализации ОСОН. Основная причина несоответствия их качества состоит в том, что ОСОН Linux прошла этап многолетнего бета-тестирования миллионами пользователей, в то время как ядро реального времени, входящее в состав двуядерной системы, такому тестированию не подвергалось. Одним из путей устранения указанного несоответствия является применение возможно более совершенной технологии разработки ядра реального времени. В частности, необходимо уделять по-

вышенное внимание этапу высокоуровневой разработки, этапу моделирования, формирования архитектуры и базовых структур ядра реального времени как самостоятельного программного изделия.

Ниже представлена формализованная объектная модель базовых структур для построения компактного ядра реального времени. Такое ядро может быть использовано для создания двухъядерной системы комбинированного назначения. Вместе с тем оно пригодно и для построения высоконадежной одноядерной СРВ, отвечающей жестким требованиям компактности и реактивности.

2. Модель ядра реального времени

Представляемое ниже описание модели ядра реального времени состоит из двух разделов: формального описания семи классов системных объектов, включающего описания функциональных членов этих классов (используемые выразительные средства опираются, в основном, на стандартные средства языка Си++) и концептуального описания семантики спецификаторов процедур, отвечающих за обработку прерываний.

Формальная часть включает описания сервисных процедур, обеспечивающих:

- управление списками (в том числе упорядоченными по значению ключа),
- управление задачами, включающее синхронизацию задач посредством использования разделяемых считающих семафоров, приостановку задач на заданное число тактовых интервалов таймера реального времени,
- передачу сообщений между задачами.

Компактность модели достигнута за счет корректного распределения функций между классами системных объектов, тщательного выбора схемы наследования. Полученная объектная модель концентрированно выражает концепцию организации взаимодействия асинхронных процессов. В результате найденного распределения функций между классами системных объектов для описания двух десятков объявленных процедур (две трети из них непосредственно доступны прикладным программам) потребовалось всего несколько десятков операторов - в среднем чуть больше трех операторов на процедуру. Эти параметры свидетельствуют о высокой добротности модели.

Модель строилась с таким расчетом, чтобы ее можно было использовать как в системах, располагающих возможностью динамического порождения задач, так и в системах с самыми жесткими ограничениями на состав используемых ресурсов. В частности, в рамках модели реализуются ядра операционных систем реального времени, ориентированные на применение в приложениях со статическим порождением всех используемых объектов - задач, семафоров, почтовых боксов, а также способных мигрировать между различными списками узлов списочных структур.

Важным свойством модели, отражающим ее высокую гибкость, является отсутствие параметров настройки системы, задающих предельно допустимое количество системных объектов того или иного типа. Ограничения накладываются только характеристиками используемых физических ресурсов - разрядностью и объемом доступной памяти. Требование выдерживать это свойство в некоторых случаях может вступать в противоречие с отсутствием ресурсов на реализацию механизмов динамического распределения памяти. В ходе построения рассматриваемой минимальной модели пришлось столкнуться с примером такого противоречия, связанным с реализацией временных задержек. В разделе, поясняющем организацию соответствующего класса, описан способ разрешения этого противоречия.

Рассматриваемая модель интересна, в первую очередь, как модель статического типа, если в понятие статической модели вкладывать представление о текстах, схемах, служащих целям исследования структурных особенностей системы - особенностей внутренней организации отдельных ее частей и особенностей взаимосвязей между ними. Элементы статической модели анализируются

исследователем, конструктором "с карандашом в руке" и преобразуются вручную, они предназначены для выяснения свойств конструкции, для поиска вариантов ее модификации, а также в качестве исходного материала для построения действующих образцов системы или ее динамических моделей. Динамический (действующий) вариант модели получается, когда представления статической модели доводятся до уровня, удовлетворяющего требованиям автоматической трансляции - автоматической интерпретации или компиляции в исполняемый код. Аппаратура, используемая для функционирования динамической модели (модельный аппаратный комплекс), может существенно отличаться от целевого комплекса аппаратных средств. Чем меньше различия между модельным и целевым аппаратными комплексами, тем ближе динамическая модель к действующему варианту разрабатываемой системы.

3. Формальные и неформальные средства представления модели

В качестве основы для представления объектной модели микроядра операционной системы реального времени взяты выразительные средства языка Си++. В составе этого языка системного программирования имеются мощные и гибкие средства, представляющие основные особенности современного объектно-ориентированного подхода к конструированию программных систем. Однако составом понятий языка Си++ не охватывается ряд структурных особенностей вычислительного процесса, связанных с работой таких элементов архитектуры компьютерных систем как системные регистры, система прерываний. Однако функционирование микроядра в существенной мере опирается на использование этих элементов архитектуры компьютеров и без привлечения таких понятий невозможно построение достаточно полной модели микроядра операционной системы реального времени. Поэтому в состав используемых нами выразительных средств пришлось включить элементы, выводящие модель за рамки текста на формальном языке. В результате, построенная нами модель является текстом на псевдокоде, максимально приближенном к языку Си++. Основным отличием, выводящим модель за рамки языка Си++, является использование системных спецификаторов SYS, WSYS и HSYS. Имеется и ряд различий в деталях, которые отличают используемые нами формальные конструкции от языка Си++ и переводят их из разряда фрагментов программы в разряд фрагментов псевдокода. Так, например, остается недоопределенной структура STACK_FRAME, служащая для сохранения в стеке задачи соответствующих ей значений системных регистров:

```
struct STACK_FRAME {void *PC; < ..... > }.
```

Такая структура обязательно содержит значение счетчика команд PC, но она содержит и другие поля, состав которых зависит от конкретных особенностей архитектуры используемого процессора и принятых в системе соглашений об организации программных интерфейсов - в частности, соглашений, определяющих способ возврата значений, вырабатываемых функцией, в точку вызова этой функции.

Имеются и некоторые другие отличия используемого псевдокода от языка Си++. Так, все члены классов и родительские классы по умолчанию считаются специфицированными как public (спецификатор private вообще не используется). В нашем псевдокоде не требуется совпадения типов значений, вырабатываемых одноименными виртуальными функциями-членами потомков абстрактного класса.

Неформальную часть модели составляет концептуальное описание семантики спецификаторов SYS, WSYS и HSYS, используемых для обозначения процедур, отвечающих, соответственно, за обработку системных вызовов и внешних прерываний.

4. Формальная составляющая модели

Формальная модель системы включает в себя семь системных классов с перечнями их декларативных и процедурных членов, а также связывающие эти классы отношения наследования (Рис. 1). Предлагаемая модель опирается на использование цепных списков, поэтому корневым классом в структуре наследования является класс ListNode, позволяющий строить множества объектов, связываемые в цепочки за счет использования указателей.

Класс ListNode содержит единственный декларативный атрибут - указатель Next, предназначенный для указания следующего звена цепного списка (поскольку модель ориентирована на максимально возможную экономию памяти, формируемые в ней списки являются однонаправленными).

```
class ListNode {
    ListNode *Next;
    ListNode (ListNode *init_next = 0) {
        Next = init_next; };
    ListNode* Pop();
    void Push(ListNode*);
    ListNode* Find(ListNode *chain);
};
```

Конструктор `ListNode(ListNode *)` позволяет формировать исходные цепочки объектов статически (в тексте программы). Метод Pop:

```
ListNode* ListNode::Pop() {
    ListNode *ret_ptr = Next;
    if(ret_ptr != 0) Next = ret_ptr->Next;
    return(ret_ptr);
}
```

позволяет удалять следующее звено списка, а метод Push:

```
void ListNode::Push(ListNode *chain) {
    if(chain != 0) {
        chain->Next = Next;
        Next = chain;
    }
}
```

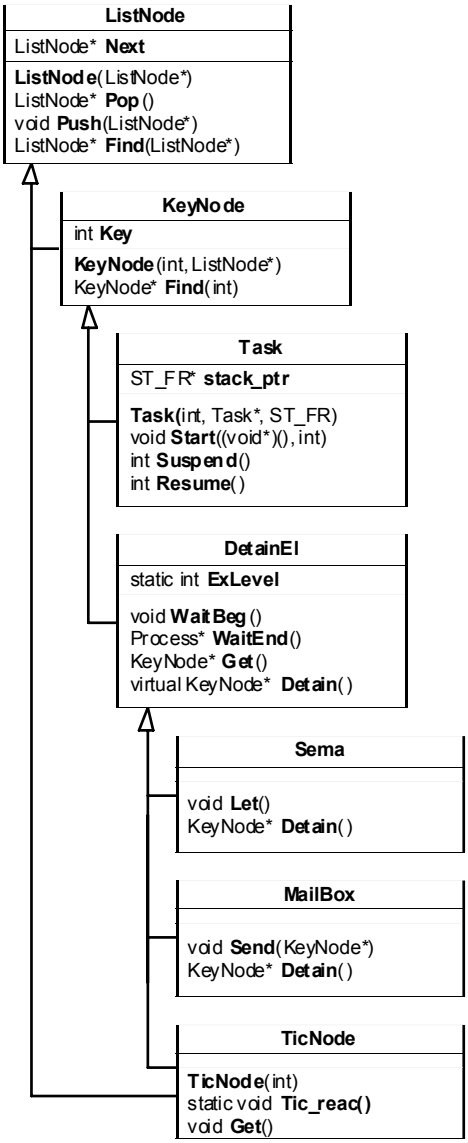

Рис. 1 Диаграмма классов модели ядра ОСРВ

обеспечивает выполнение обратной операции; совместное использование этих двух методов позволяет осуществить LIFO-буферизацию. Метод `Find`:

```
ListNode* ListNode::Find(ListNode *chain){
    ListNode *suc_ptr, *pre_ptr = this;
    for( ; ; ) {
        if((suc_ptr = pre_ptr->Next) == 0) return 0;
        if(suc_ptr->Next == chain)) return(pre_ptr);
        pre_ptr = suc_ptr;
    }
}
```

дает возможность найти место включения в список звена, размещенного по заданному адресу. Решение, обеспечившее повышение компактности модели, состоит в том, что методы поиска всегда выдают в качестве значения адрес звена, предшествующего искомому.

Объекты типа `ListNode` используются в нашей модели для представления трех системных списков:

```
ListNode RList, /* список задач, готовых к исполнению */
         SList, /* список приостановленных задач */
         DList; /* список заданий для таймера    */
```

Класс `ListNode` выступает в качестве прямого или косвенного базового класса для всех остальных классов модели.

Классы *KeyNode* и *Task*. Использование объектов, принадлежащих классу `KeyNode`, который определяется как непосредственный потомок класса `ListNode`

```
class KeyNode:ListNode {
   int Key;
   KeyNode(int n=0, ListNode *l=0):ListNode(l){Key = n;};
   KeyNode* Find(int);
};
```

позволяет строить списки, упорядоченные по значению ключа `Key`. Метод `Find`:

```
ListNode* KeyNode::Find(int prior) {
   KeyNode *suc_ptr;
   ListNode *pre_ptr = this;
   while((suc_ptr = (KeyNode *) pre_ptr->Next) != 0) {
      if(suc_ptr->Key >= prior) break;
      pre_ptr = suc_ptr;
   }
   return(pre_ptr);
}
```

обеспечивает поиск в списке места, соответствующего заданному значению ключа. Главным назначением упорядоченных списков в нашей модели является упорядочение (по приоритету) объектов, выступающих в качестве дескрипторов задач. При такой организации списка задач, готовых к исполнению, не возникает потребности сканирования всего списка каждый раз, когда необходимо выбрать задачу, которая будет владеть ресурсом процессора, достаточно взять лишь первый элемент списка. В качестве дескрипторов задач выступают объекты класса `Task`:

```
class Task:KeyNode {
   STACK_FRAME *stack_ptr;
   Task(int n, Task* p, STACK_FRAME *s):KeyNode(n, p)
      { stack_ptr = s; };
   SYS void Start(void (*mainprog)(), int st_sz);
   SYS int Suspend();
   SYS int Resume();
   static void Delay(int n);
};
```

Одним из условий корректной работы системы является наличие дескриптора инициализирующей задачи

```
Task MainP(MAX_INT /* максимальное целое */, 0, 0);
```

задача `MainP` выполняет все действия, которые необходимы для инициализации системы. Она является главной (или головной) задачей в том смысле, в каком программа Main является главной (головной) в традиционных системах программирования. Задача MainP запускает все остальные задачи системы (предварительно конструируя их дескрипторы, если это не сделано статически). Здесь необходимо отметить, что конструктор `Task(int, Task*, STACK_FRAME*)` лишь создает дескриптор задачи. Для включения задачи в состав претендентов на системные ресурсы (в первую очередь - на ресурс процессора) следует еще выполнить метод `Start`:

```
SYS void Start(void (*start_point)(),int st_sz) {
   stack_ptr += st_sz - sizeof(STACK_FRAME);
   stack_ptr->PC = start_point;
   KeyNode(RList).Find(Key)->Push(this);
}
```

который осуществляет необходимую настройку стека задачи и включает задачу в список претендентов на ресурс процессора. Указатель первого объекта этого списка помещается в поле `Next` объекта `RList`.

Заметим, что метод `Start` помечен спецификатором `SYS`. Детальное пояснение семантики спецификатора `SYS` дается ниже при изложении неформальной части модели. Сейчас заметим только, что методам, помеченным этим спецификатором, соответствует особый механизм вызова. Особый механизм используется и для возврата из таких методов (SYS-методов).

В описании класса `Task` еще два члена помечены спецификаторами `SYS` - это возвращающие целые значения SYS-методы `Suspend()` и `Resume()`. Для возврата значений из SYS-методов в точку вызова также используется особый механизм, синтаксически представляемый оператором `SYSreturn`. SYS-метод `Suspend()`:

```
SYS int Suspend() {
   ListNode *p;
   if((p = RList.Find()) == 0) SYSreturn 0;
   SList.Push(p->Pop());
   SYSreturn 1;
}
```

обеспечивает приостановку задачи (перенесением ее дескриптора в список `SList` приостановленных задач). Приостанавливаться может только задача, размещенная в списке `RList`. Если выполняется попытка приостановить задачу, которой в текущий момент нет в этом списке, SYS-метод `Suspend()` выполняется без побочного эффекта и возвращает в точку вызова значение 0. В случае успеха возвращается значение 1. SYS-метод `Resume()`:

```
SYS int Resume() {
   if(((KeyNode*)p = (KeyNode*)SList.Find()) == 0) SYSreturn 0;
   (KeyNode(RList).Find(p->Key))->Push(p);
   SYSreturn 1;
}
```

обеспечивает возобновление нормального течения приостановленной задачи - возвращает дескриптор из списка SList обратно в список RList.

В описании метода Delay(int n), обеспечивающего задержку задачи на заданное число тактов таймера реального времени, используется объект еще не рассмотренного нами класса TicNode, поэтому его организация обсуждается ниже.

Синхронизирующие элементы. Для координации выполнения взаимодействующих задач используются синхронизирующие элементы. Общим свойством этих элементов является то, что операции доступа к ним могут приводить к приостановке задачи до момента возникновения событий, разрешающих дальнейшее развитие приостановленной задачи. Для выражения этого общего свойства синхронизирующих элементов в модели вводится абстрактный класс

```
class DetainEl:KeyNode {
   void WaitBeg();
   Task* WaitEnd();
   SYS virtual KeyNode* Detain()=0;
};
```

Условия приостановки различны для различных разновидностей синхронизирующих элементов, поэтому метод Detain(), выполнение которого может привести к приостановке, является виртуальным - для каждого из потомков абстрактного класса DetainEl этот метод определяется посвоему.

Центральными методами механизма синхронизации являются WaitBeg() и WaitEnd(). Первый выполняет операцию переноса дескриптора задачи из списка RList в список задач, задержанных синхронизирующим элементом

```
void DetainEl::WaitBeg() {
   Task *proc_ptr = (Task *)(Rlist.Pop());
   Find(proc_ptr->Key)->Push(proc_ptr);
   NOreturn;
}
```

Исполнение специального оператора NOreturn предотвращает попадание не по назначению результата выполнения SYS-метода, вызывающего WaitBeg(). Детали, связанные с ролью оператора Noreturn, обсуждаются ниже при рассмотрении неформальной части модели.

Второй из двух основных методов механизма синхронизации выполняет обратную операцию - возвращает дескриптор задержанной задачи в список RList

```
Task* DetainEl::WaitEnd() {
   Task *proc_ptr = (Task *)Pop();
   (Task(RList).Find(proc_ptr->Key))->Push(proc_ptr);
   return (proc_ptr);
}
```

Метод WaitEnd() возвращает указатель реактивируемой задачи в вызывающий его SYS-метод. Некоторыми вызывающими SYS-методами это значение используется, некоторыми игнорируется. Рассмотрим один из популярных примеров синхронизирующего элемента - считающий семафор

```
class Sema:DetainEl {
   WSYS void Let();
   SYS void Detain();
};
```

Спецификатор `WSYS`, которым помечен метод `Let()`, означает, что, как и в случае использования спецификатора SYS, вызов метода и возврат из него связаны с использованием специальных механизмов. Для SYS-методов и WSYS-методов эти механизмы схожи. Их особенности и различия между ними поясняются при рассмотрении неформальной части модели. Приостанавливающий задачу метод для семафора строится следующим образом:

```
SYS void Sema::Detain() { if(Key-- < 1) WaitBeg(); }
```

Заметим, что определенный в абстрактном классе универсальный метод `WaitBeg()` упорядочивает по приоритетам задержанные семафором задачи.

В нашей компактной модели метод `Detain()` класса `Sema` ничего не возвращает в точку вызова. Для более продвинутых систем модель этого метода может быть расширена так, чтобы в точку его вызова возвращалась информация об особенностях выполнения операции (пришлось ли ждать открытия семафора, как долго и т.п.).

Метод, открывающий семафор (и, возможно, освобождающий задержанную им задачу) использует другой универсальный метод абстрактного родительского класса:

```
WSYS void Sema::Let() { if(Key++ < 0) WaitEnd(); };
```

В описании `Sema::Let()` не определено, что будет происходить, если семафор (его поле Key) переполняется. При реализации модели необходимо решить этот вопрос.

Другим популярным примером синхронизирующего элемента является почтовый бокс.

```
class MailBox:DetainEl {
   WSYS void Send (KeyNode *);
   SYS KeyNode* Detain();
};
```

Эта разновидность синхронизирующих элементов обеспечивает не только синхронизацию задач, но и перенос информации между ними. Почтовый бокс хранит либо список задач, ожидающих сообщений, либо список сообщений, готовых для передачи обращающимся за ними задачам. Метод, способный (в случае отсутствия сообщений) приостановить выполняющую его задачу, строится так:

```
SYS KeyNode* MailBox::Detain () {
   if (Key-- < 1) { WaitBeg(); NOreturn; }
   else SYSreturn Pop();
 }
```

Если значение ключа оказывается меньше единицы (сообщений нет) и выполняется метод `WaitBeg()`, то в момент завершения выполнения метода `MailBox::Detain()` в той точке, где размещен оператор `NOreturn`, еще не известно значение указателя, который должен быть возвращен в точку вызова `MailBox::Detain()`. Ниже будет объяснено, как оператор `NOreturn` обеспечивает корректную обработку этой ситуации.

Для засылки сообщения в почтовый бокс используется метод

```
SYS void MailBox::Send(KeyNode *chain) {
   if (Key++ < 0) RetValue(WaitEnd(), chain);
   else (Find(chain->Key))->Push(chain);
}
```

Если оказывается, что в момент выполнения метода `Send()` у данного почтового бокса есть задачи, ожидающие поступления сообщений, то оператор `RetValue(Task*, KeyNode*)` обеспечивает передачу значения указателя chain в точку вызова SYS-метода `MailBox::Detain()`, задержавшего задачу в связи с отсутствием сообщений в почтовом боксе.

Для реализации элементов службы времени в нашей модели используется еще один из потомков абстрактного класса `DetainEl` - класс `TicNode`. Этому классу принадлежат объекты, используемые для формирования заявок типа "приостановить задачу на заданное время". Подобного рода заявки обычно хранятся в специальном списке, упорядоченном по возрастанию времени, на которое выполнение задачи отложено. Причем в качестве ключа используется не абсолютное системное время активации отложенной задачи, а разность между этим временем и временем активации задачи, являющейся предыдущей в списке отложенных. Для первой задачи в списке отложенных значение ключа составляет разность между временем ее активации и текущим системным временем. При каждом срабатывании таймера значение ключа для первой задачи из списка уменьшается на величину такта системного таймера, и когда это значение становится нулевым, первая задача переводится из списка отложенных задач в список готовых. Так же в список готовых переводятся следующие далее по списку задачи с нулевым значением ключа (для них время активации совпадает с временем активации предыдущей по списку задачи). В последующие свои срабатывания системный таймер продолжает свою работу с задачей, оказавшейся первой в списке отложенных.

Подобный механизм обработки отложенных задач получил название дельта-список [4]. Он позволяет обойтись работой с ключом первой в списке задачи, без просмотра при срабатывании таймера списка отложенных задач с целью поиска тех задач, которые необходимо активировать в данный момент времени. В представляемой модели данный механизм реализуется классом `TicNode`:

```
class TicNode:DetainEl, ListNode {
   TicNode(int n) {Key = n;}
   static HSYS void Tic_reac();
   SYS void Detain();
};
```

Конструктор класса обеспечивает занесение в поле `Key` значения, определяющего длительность приостановки. Текущая задача приостанавливается на заданное время, выполняя метод `Detain`:

```
void TicNode::Detain() {
   int d;
   ListNode *pre_ptr = &DList;
   TicNode  *suc_ptr = (TicNode *) DList.Next;
   while(suc_ptr != 0) {
      if((d = Key - suc_ptr->Key) > 0) Key = d;
      else { suc_ptr->Key = -d; break; }
      pre_ptr = suc_ptr;
      suc_ptr = suc_ptr->ListNode::Next;
   }
   pre_ptr->ListNode::Push(this);
   WaitBeg();
}
```

В результате выполнения `Detain` текущая задача заносится в список `ListNode DList` работ для статического метода `Tic_reac()`. Этот метод, помеченный спецификатором `HSYS`, выполняет обработку прерываний от таймера реального времени.

```
static HSYS void TicNode::Tic_reac() {
   TicNode *job = (TicNode *) DList.Next;
   if((job != 0) && (--(job->Key) == 0))
      while(job->Key == 0) {job->WaitEnd(); DList.Pop();}
}
```

Адрес точки входа в метод `Tic_reac()` размещается в векторе прерываний от таймера реального времени. Каждое срабатывание таймера приводит к уменьшению значения поля `Key` в первом элементе списка `DList`. Если это значение становится нулевым, работа завершается, соответствующий ей элемент изымается из списка `Dlist`, а связанная с ним задача переносится в список `RList` претендентов на процессорное время. Если в списке `Dlist` имеются другие задачи, время ожидания для которых истекло, они тоже переносятся из `Dlist` в `RList`.

Для предоставления пользователю сервиса, обеспечивающего задержку задачи на заданный временной интервал, необходимо для каждого обращения к этому сервису строить объект класса TicNone, связывающий указатель на задержанную задачу с указателем момента завершения интервала задержки. Построение такого объекта связано с необходимостью выделения соответствующей памяти. Неудобно возлагать заботу о выделении этого ресурса на разработчика приложений. Включать в систему некий резервный пул таких таймерных объектов тоже неудобно - ограниченность этого пула приводила бы к потере гибкости. Реализованное в нашей модели компромиссное решение состоит в том, что система сама строит нужный таймерный объект в стеке задерживаемой задачи. Это достигается за счет использования метода

```
static void Task::Delay(int n) {
   TicNode DNode(n);
   DNone.Detain();
}
```

По истечении интервала задержки выполняется выход из метода `Delay( )`, и память, занимавшаяся под объект `TicNode`, автоматически освобождается. При этом достигаются две цели: во-первых, сохраняется гибкость, во-вторых, разработчик приложений освобождается от необходимости вникать в детали организации таймерных объектов, следить за их порождением и использованием. Вместе с тем пользователю, вынужденному работать в рамках жестких ограничений на размеры стекового пространства задач, в рамках рассматриваемой минимальной модели оставлена возможность самому выделять статически или динамически память под таймерные объекты, использовать системный вызов `TicNode::Detain()`, не занимающий дополнительного места в стеке.

В рассмотренных выше описаниях классов и функций использовался ряд специфических служебных слов, раскрытие содержания которых, как мы поясняли, выходит за рамки формальной части модели. Далее следуют неформальные описания, поясняющие смысл этих служебных слов и подход к реализации соответствующих им механизмов.

5. Неформальная составляющая модели

Обработка внешних прерываний. Спецификаторам `HSYS` соответствуют обработчики прерываний - процедуры обработки ситуаций, вызываемых внешними (по отношению к логике выполнения программ) событиями. Такими событиями могут быть сигналы, поступающие от внешних устройств, нарушения в системе питания, сбои в работе аппаратуры и т.д.

Выше приведено описание процедуры, выполняющей задачу обработки внешних прерываний - метода `Tic_reac()`, обрабатывающего прерывания от таймера реального времени. Обращение к методу, помеченному спецификатором `HSYS`, его выполнение и завершение осуществляются по схеме, в которой можно выделить следующие фазы.

Фаза 1. Аппаратно реализованный механизм вызова процедуры обработки внешних прерываний обеспечивает следующие действия:

- снимает признак разрешения прерываний (внешние прерывания блокируются либо полностью, либо частично);

- формирует структуру STACK_FRAME, отражающую состояние контекста программы на момент возникновения прерывания - значение счетчика команд и некоторых других системных регистров (возможные варианты организации STACK_FRAME определяются аппаратной организацией механизма прерываний в целевой системе; для некоторых платформ возможно даже отсутствие аппаратного формирования STACK_FRAME, тогда эти действия реализуются программно на следующей фазе обработки); структура STACK_FRAME формируется в пользовательском стеке;

- из вектора прерываний извлекает адрес точки входа в головной блок процедуры обработки прерываний; выполняет передачу управления в этот блок.

Фаза 2. Головной (системный) блок процедуры обработки прерываний обеспечивает следующие действия, общие для всех HSYS-процедур:

- вносит в структуру STACK_FRAME ту информацию о контексте текущей задачи, которая не была в ней зафиксирована аппаратно в фазе 1 (либо формирует структуру целиком, если это не реализуется аппаратно);

- переключает процессор на работу в системном стеке (если это не делается аппаратно в фазе 1);

- увеличивает на единицу значение системного счетчика уровня вложенности прерываний (обозначим этот счетчик идентификатором SYSlevel); назначение счетчика SYSlevel - предотвращать непредсказуемые результаты от попыток выполнять приостанавливающие операции из процедур обработки прерываний (эти процедуры не являются частью задач и потому их приостановка по доступу к синхронизирующим элементам недопустима);

- выполняет переход на исполнение основного (проблемного) блока обработки внешнего прерывания.

Фаза 3. В ходе исполнения основного (проблемного) блока процедуры обработки внешних прерываний происходит следующее:

- реализация действий, специфицированных описанием HSYS-процедуры; эти действия могут привести к смене текущей задачи (к смене значения системной переменной RList);

- переход на исполнение завершающего блока обработки прерываний.

Фаза 4. В блоке завершения обработки прерываний выполняются следующие действия:

- уменьшается на единицу значение счетчика SYSlevel уровня вложенности прерываний;

- при ненулевом значении счетчика SYSlevel процессор переключается на работу в пользовательском стеке (возможно, это окажется уже не тот стек, с которым выполнялась работа в фазе 2);

- восстанавливаются значения тех системных регистров, которые были программно занесены в структуру STACK_FRAME в фазе 2;

- выполняется машинная команда выхода из обработки прерываний.

Условное переключение на работу в пользовательском стеке иллюстрирует второе назначение SYSlevel - предотвращение инверсии приоритетов при исполнении обработчиков прерываний. Планировщик не выполняет переключение контекста задач, если значение SYSlevel не равно нулю. Переключение осуществляется только тогда, когда все обработчики прерываний завершены (точнее, завершен самый внешний обработчик).

Фаза 5. Аппаратно реализованный механизм выхода из обработки прерываний обеспечивает:

- восстановление уровня разрешенных прерываний, восстановление контекста прерванной программы (значений системных регистров) по содержанию соответствующих полей структуры SYSlevel в пользовательском стеке;

- переход к продолжению исполнения той задачи, на дескриптор которой указывает RList.Next.

Все действия, специфические для конкретной HSYS-процедуры, выполняются в фазе 3. Действия, выполняемые в фазах 2 и 4, - общие для всех процедур обработки внешних прерываний. Поэтому в целях экономии памяти целесообразно строить систему так, чтобы не повторять идентичные фрагменты HSYS-процедур, соответствующие фазам 2 и 4. Действия, соответствующие фазе 2, должны выполнятся общим головным блоком обработки прерываний, а действия, соответствующие фазе 4, - общим блоком завершения обработки прерываний.

Модель предполагает, что все задачи всегда имеют приоритет ниже, чем приоритет обработчиков прерываний. Планирование обработчиков прерываний выполняется аппаратно. Модель гарантирует, что при выполнении обработчиков прерываний не происходит инверсии приоритетов.

Обработка командных прерываний. Спецификаторам `SYS` и `WSYS` соответствуют сервисные системные процедуры, реализация которых связана с использованием механизма "командных" прерываний. Обращения к таким процедурам принято называть "системными вызовами".

Вопросы, связанные с детализацией способа выполнения системных вызовов, стоят в ряду важнейших системных решений. Ответы на эти вопросы, полно и точно раскрывающие семантику спецификатора `SYS` и `WSYS`, должны привязываться к конкретной архитектуре компьютерной системы. Но и в рамках одной и той же архитектуры возможны варианты, ориентированные на различные критерии эффективности функционирования системы.

Например, возможны варианты либо с одним, либо с несколькими векторами прерываний для системных вызовов. Простейший вариант опирается на использование единого, общего для всех системных вызовов, вектора прерываний. В этом случае системные вызовы нумеруются и составляется таблица точек входа в процедуры обработки системных вызовов. Длина такой таблицы равна числу процедур, помеченных спецификаторами `SYS` или `WSYS`. В этой же таблице может размещаться информация о типе процедуры, о ее параметрах. В точке обращения к системной процедуре ее номер и, возможно, значения фактических параметров засылаются либо в стек, либо в системные регистры - в зависимости от принятых системных соглашений. Дальнейшая обработка обращения к процедуре, помеченной спецификатором `SYS`, выполняется по схеме, в значительной мере повторяющей схему обработки внешних прерываний. Ниже перечислены отличия в порядке обработки командных прерываний от обработки внешних прерываний по каждой из пяти рассмотренных выше фаз.

Фаза 1. Блокируются все прерывания от внешних устройств.

Фаза 2. Производятся дополнительные действия:

- в специальной системной переменной (обозначим ее для определенности идентификатором `CurProc`) фиксируется значение указателя дескриптора текущей задачи (то есть запоминается значение поля `RList.Next`);

- для SYS-процедур в самом начале фазы 2 проверяется значение уровня вложенности прерываний: если `SYSlevel>1`, то выполнение SYS-процедуры блокируется, осуществляется переход сразу на фазу 5, для WSYS-процедур такая проверка и такая блокировка не производятся, то есть при обработке внешних прерываний выполнение SYS-процедур блокируется, а выполнение WSYS-процедур нет (в этом состоит вся разница между системными вызовами SYS и WSYS).

Фаза 3. В отличие от HSYS-процедур SYS-процедуры могут быть процедурами-функциями, то есть могут вырабатывать значения, предназначенные для передачи в точку вызова. При составлении описаний SYS-функций необходимо заботиться, чтобы вырабатываемые значения попадали по назначению, то есть в точку вызова. Проблема состоит в следующем.

Выполнение SYS-функции осуществляется не в стеке вызвавшей задачи, а в системном стеке. Поэтому механизмы, ассоциируемые с выполнением оператора return, здесь не работают. С целью акцентировать на этом внимание введен оператор `SYSreturn`, работа которого сводится к размеще-

нию вырабатываемого SYS-функцией значения в нужном поле структуры STACK_FRAME, сформированной в пользовательском стеке.

К моменту завершения обработки системного вызова текущей может стать не вызвавшая SYS-функцию, а какая-то другая задача. Поэтому-то и нужно сохранять в переменной CurProc указатель дескриптора той задачи, которая вызвала SYS-функцию. Так, например, в результате выполнения SYS-функции Resume(), возобновляющей некую высокоприоритетную задачу, текущая задача может смениться. Но, между тем, целое значение 1, свидетельствующее об успешном выполнении операции возобновления высокоприоритетной задачи, необходимо передать в точку вызова SYS-функции Resume(). Поэтому выход из Resume() выполняется не оператором return, а оператором SYSreturn, использующим сформированное в фазе 2 значение переменной CurProc.

В тех случаях, когда в момент вызова SYS-функции вырабатываемое ею значение еще не известно (например, при обращении за сообщением к пустому почтовому боксу), вызвавшая задача приостанавливается, поскольку указатель требуемого сообщения еще не выработан. В этом случае выполнение SYS-функции завершается оператором NOreturn, который выполняется так, как если бы завершалась процедура типа void. Когда же SYS-процедура Send() получит указатель требуемого сообщения, она передает его ожидающей задаче с помощью оператора RetVal.

Фаза 4. Для SYS- и WSYS-процедур действия, выполняемые в 4-ой фазе, в точности совпадают с соответствующими действиями для HSYS-процедур. В случае же выполнения SYS-функций в фазе 4 выполняются те действия по формированию возвращаемых значений, которые передаются в эту фазу из фазы 3 через аргументы операторов SYSreturn и RetVal.

Фаза 5. Аппаратно реализуемая фаза выхода из обработки прерываний обычно выполняется идентично для всех типов прерывающих процедур.

Эффективная реализация представленой модели ядра реального времени требует тщательного учета архитектурных особенностей целевой аппаратной платформы.

6. Использование модели

Тщательная разработка высокоуровневой модели дает возможность выделить сущность рассматриваемого класса систем, объективно выразить его фунциональную природу, его место в ряду систем родственного типа.

Использование формализованной модели ядра реального времени позволяет конструктору ясно видеть систему в целом, эффективно отслеживать особенности основных внутрисистемных связей. Добиваясь компактности высокоуровневой модели, конструктор, как правило, обеспечивает и компактность итогового кода.

Предложенная статическая модель может быть положена в основу разработки действующей (динамической) модели, открывающей возможность полной эмуляции целевых систем средствами инструментальных вычислительных комплексов.

Представленная модель может быть использована как пример формализованного описания архитектуры ядра реального времени. Ниже приведены варианты постановок задач переработки модели для встраивания в систему различных служебных функций, модификации интерфейсов и поддерживаемых ядром механизмов.

Встраивание дополнительных функций ядра. Использование формальной модели ядра реального времени позволяет упорядочить встраивание в систему различных служебных функций. Это может быть, например, трассировка. Трассировка состоит в протоколировании системой заданных событий, сохранении информации о них в протоколе [5]. Составляющими этой информации, как правило, являются:

- тип события;

- время возникновения;
- вспомогательная информация (например: параметры системных вызовов, номер прерывания, количество байтов, прочитанных из очереди).

По протоколу трассировки могут быть восстановлены актуальные аспекты поведения системы, особенности ее состояний в различные моменты времени. Методы трассировки применяются, в частности, при отладке сложных многозадачных приложений в тех случаях, когда символьная отладка является недостаточно эффективной либо когда требуется анализировать поведение системы на протяжении длительных интервалов времени.

Для реализации трассировки в код ядра операционной системы помещаются специальные обращения к компоненте трассировки с целью сохранения необходимой информации. Конкретные точки в коде ядра, куда помещаются подобные обращения, определяются составом протоколируемых событий. Если ядро слабо структурировано, то задача выявления всех необходимых точек обращения к модулю трассировки сводится к сплошному анализу всего кода целиком. При высоких объемах исходного кода системы эта задача является чрезвычайно трудоемкой, а появление в различных фрагментах кода обращений к модулю трассировки затрудняет и без того недостаточно ясное восприятие кода системы.

При использовании высокоуровневой модели ядра задача локализации точек обращения к трассировщику упрощается. Допустим, что необходимо отслеживать все события, связанные с объектом "задача". Тогда нам достаточно объявить в системе новый класс `TracingTask`, являющийся наследником класса `Task`:

```
class TracingTask: Task {
    TracingTask(int n, Task *p, ST_FR *s):Task(n, p, s) { DoTracing(…);};
    SYS void Start(void (*mainprog)(), int st_sz); {
        DoTracing(…);
        Task::Start(mainprog, st_sz);
    }
    SYS int Suspend(); { DoTracing(…); Task::Suspend();};
    SYS int Resume(); { DoTracing(…); Task::Resume();};
    static void Delay(int n); { DoTracing(…); Task::Delay(n);};
};
```

Семантика вызова `DoTracing` остается недоопределенной, так как параметры вызова могут быть различными в зависимости от требований, предъявляемых к протоколируемым данным. Но уточнение этих требований не приводит к изменению структуры программы. При использовании данного класса в системе мы добиваемся того, что все события, происходящие с объектом "задача", протоколируются. При этом добавление функций трассировки в ядро системы не влияет на прозрачность восприятия его структуры. Более того, этот класс может быть описан в другом файле и использоваться в системе опционально, в зависимости от режима, в котором работает система в данный момент времени.

Подобным образом протоколирование может быть организовано и для других системных объектов. При этом, если протоколируется исполнение только части методов какого-либо класса, то в классе-потомке, реализующем трассировку, нет необходимости переопределять все родительские методы, а необходимо переопределить только те, которые подвергаются трассировке. Такая организация процесса трассировки делает его хорошо структурированным, прозрачным и масштабируемым. Подобной эффективности невозможно добиться при организации трассировки в системах с рыхлой архитектурой.

Использование программируемого таймера. Рассмотренный вариант модели ядра ориентирован на использование периодического таймера. Ряд аппаратных платформ для встроенных приложений содержит программируемые таймеры, т.е. таймеры, которые можно перепрограммировать

на срабатывание через определенный промежуток времени. Рассмотренную модель несложно модифицировать для такой аппаратной платформы. В этом случае таймер настраивается на очередное срабатывание в момент времени, соответствующий головному элементу дельта-списка, а в случае прерываний не должен всякий раз уменьшать значение поля Key, так как гарантируется, что прерывание произойдет именно в тот момент, когда значение поля Key равно нулю. Этим достигается существенное снижение накладных расходов, связанных с обработкой прерываний от таймера.

Варианты интерфейсов прикладных программ. Продемонстрированный выше способ формализованного представления архитектуры ядра реального времени может быть использован для построения ядер операционных систем, реализующих различные типы интерфейсов прикладных программ, например, таких как стандартные интерфейсы POSIX, OSEK [6,7] или интерфейс ядра реального времени RTAI для двухъядерных операционных систем на базе ОСОН Linux [8].

Механизмы предотвращения инверсии приоритетов и тупиков. Современные ОСРВ для встроенных систем требуют наличия механизмов, которые предотвращают явления, известные как неограниченная инверсия приоритетов и тупики. В отличие от приложений, выполняемых под управлением ОСОН, встроенные приложения не "видны" пользователю и, если ОС не имеет защиты от подобных ситуаций, могут происходить незаметно. Например, тупик при доступе к записям базы данных в ОСОН сопровождается системным сообщением пользователю с подсказкой, как можно устранить проблему (например, выполнить откат и повторить операцию). Во встроенном приложении пользователь не сможет обнаружить возникший тупик и, соответственно, не сможет принять меры по его устранению. Осложняет положение и тот факт, что тупики и инверсия приоритетов могут остаться незамеченными на этапе тестирования, так как их возникновение зависит от временного поведения. Поэтому даже протестированное приложение может содержать подобные ошибки, которые порой являются фатальными.

Существует хорошо разработанный способ устранения инверсии приоритетов и тупиков, известный под названием "протокол пороговых приоритетов". Этот механизм применяется практически во всех современных системах реального времени, включая расширения POSIX для приложений реального времени, OSEK/VDX OS, спецификации расширений реального времени для Java. Механизмы предотвращения инверсии приоритетов и тупиков [9, 10] могут быть представлены средствами формальной части модели ядра.

Заключение

Представленный метод формализованного описания архитектуры может использоваться для различных вариантов организации ядер систем реального времени, отвечающих различным конфигурациям исходных требований. Объекты и процедуры, составляющие формальную часть рассмотренного примера построения ядра реального времени, образуют задел архитектурных решений, ориентированных на создание предельно компактных встроенных систем.

Вместе с тем представленный метод и элементы иллюстрирующего его примера могут использоваться для представления ядер встроенных систем, ориентированных на другой тип эффективности или систем с существенно расширенными интерфейсами прикладных программ. Рассмотренный пример наглядно демонстрирует преимущества, получаемые в результате применения формализованных описаний на этапах высокоуровневого проектирования базисов систем рассматриваемого класса. Компактное формализованное описание играет роль статической модели ядра системы. Наличие компактной статической модели позволяет разработчику охватить систему в целом, расширяет возможности поиска эффективных путей ее модификации в случае изменения требований.

Использование конструктивных решений, представляемых рассмотренной моделью ядра реального времени, позволяет разрабатывать добротные базисы для реализации системных услуг, обес-

печивающих выполнение аппаратно-ориентированных задач во встроенных системах. Наличие высокоуровневой модели ядра системы упрощает построение адекватных динамических моделей и, соответственно, позволяет осуществлять глубокую отладку комплекса аппаратно-ориентированных задач средствами инструментальных машин.

Ядро реального времени рассмотренного класса может обслуживать либо все задачи встроенной системы, либо только аппаратно-ориентированные задачи в рамках совместной работы с ядром операционной системы общего назначения.

Литература

1. Yodaiken V. RTLinux Manifesto. - Socorro, NM: Dept. of Computer Science, New-Mexico Institute of technology, 1999.
2. Embedix-RealTime. Programming Guide. 1.0. - Lindon, UT: Lineo Inc., 2000.
3. Никифоров В.В., Павлов В.А., Данилов М.В., Глазков А.В., Гуцалов Н.В. Адаптация ОС Linux к требованиям поддержки программных приложений во встроенных системах. // Программные продукты и системы - 2001. - №4.
4. Comer D. Operating System Design. - NJ: Prentice Hall, Inc., 1984.
5. Yaghmour K., Dagenais M. R. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. - Montreal, Quebec, CANADA: Departement de genie electrique et de genie informatique Ecole Polytechnique de Montreal, 2000.
6. Information Technology. - Portable Operating System Interface. (POSIX). International Standard ISO/IEC 9945-1:1996(E), IEEE Std 1003.1, 1996 Edition.
7. OSEK/VDX Operating System. - November, 2000.
8. Cloutier P., Mantegazza P., Yaghmour K., etc. DIAPM-RTAI Position Paper. - 21thR IEEE Real-Time Systems Symposium, Orlando, FL - November, 2000.
9. Данилов М.В. Методы планирования выполнения задач в системах реального времени. // Программные продукты и системы - 2001. - №4.
10. Сорокин С. В. Системы реального времени: операционные системы. // Современные технологии автоматизации. - 1997. - №2.