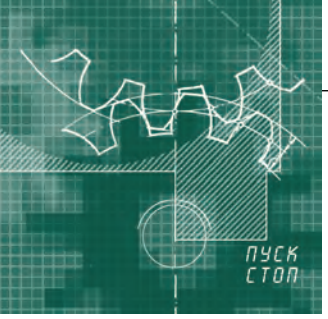

Разработка программно-аппаратного комплекса на основе систем реального времени

Алексей Марусов

В статье рассматривается применение ОС реального времени On Time RTOS-32 при проектировании системы числового программного управления (СЧПУ). Основой для статьи послужил опыт ООО НПО «Рубикон-Инновация», полученный в процессе разработки СЧПУ «Феникс».

Примеры задач

В настоящее время мы используем всё больше электронных устройств, к работе которых предъявляются требования жёсткого реального времени. Это и медицинское оборудование, где время работы может определять дозу полученного облучения, и металлообработка, где время движения рабочего органа станка определяет размеры полученных деталей, и атомная энергетика, где задержка при управлении реактором приводит к изменению вырабатываемой мощности реактора, к полной остановке или даже к тепловому разрушению.

Кроме того, постоянно появляются устройства для не столь ответственных применений, которые по своему назначению способны сделать нашу жизнь более комфортной, но без систем реального времени они просто не могли бы работать. Кому нужен MP3-плеер, который периодически останавливает проигрывание музыки? Понравится ли сотовый телефон, который прекращает передачу сигнала на время переключения между базовыми станциями? Конечно же, лучше, когда устройства работают стабильно.

Удобство и стабильность работы современных устройств во многом определяется тем, что они построены на базе операционных систем (ОС) реального времени: внутренние механизмы их работы не влияют на скорость выходного потока данных (музыка играет без задержек, на реактор сигнал управления передаётся с заданной частотой независимо от действий оператора и т.п.). Так давайте заглянем внутрь этих ОС, посмотрим, как они используются при проектировании устройств автоматизации на примере системы числового программного управления (СЧПУ).

Определение системы жёсткого реального времени

Для начала разберёмся, что же такое «реальное время». Этот термин часто встречается в литературе, но определение ему даётся редко. Причём он часто используется не для описания технических особенностей ОС, а для маркетинговых целей. Таким образом, появилось понятие системы «мягкого» реального времени. В дальнейшем, рассуждая об ОС реального времени, мы будем опираться на следующее определение, синтезированное из определений в [1]:

ОС реального времени – это ОС, в которой успешность работы любой программы зависит не только от её логической правильности, но и от времени, за которое она получила этот результат. Если система не может удовлетворить временным ограничениям, должен быть зафиксирован сбой в её работе.

Именно такие ОС часто называют системами жёсткого реального времени. Их мы и будем рассматривать далее.

Также, говоря об устройствах реального времени, будем иметь в виду устройства, которые обеспечивают результаты вычислений за фиксированный промежуток времени. Если результат предоставлен слишком поздно (или слишком рано в некоторых системах), должен быть зафиксирован сбой.

Кооперативная и вытесняющая многозадачность

Практически любое ядро многопоточной системы реального времени работает по типовому алгоритму: периодически вызывается планировщик – системная задача, которая определяет, какой из задач пользователя отдать следующий квант времени; затем управление передаётся задаче пользователя, где и производится непосредственно полезная работа; затем снова вызывается планировщик и т.д. Отличаются ОС реального времени по времени переключения задач, набору сервисных возможностей и моменту, когда происходит переключение задач. Если передача управления от потока к потоку может производиться ОС, говорят, что ОС работает в системе вытесняющей многозадачности. Если же переключение задач возможно только после вызова системной функции (одной из нескольких системных функций), имеет место кооперативная многозадачность. В каком из режимов лучше использовать ОС, зависит в первую очередь от проектировщика системы. И у того и у другого режима есть свои плюсы и минусы. При использовании кооперативной многозадачности все потоки нужно разрабатывать таким образом, чтобы они периодически передавали управление планировщику системы, а зависимые от стабильности

времени операции переносить в прерывания. При использовании вытесняющей многозадачности, как правило, повышаются накладные расходы на переключение задач ОС, а также значительно возрастает сложность программирования системы: для обращения к любому глобальному объекту нужно использовать критическую секцию или мьютекс [2]. Большинство ОС реального времени могут работать как в режиме вытесняющей многозадачности, так и в режиме кооперативной многозадачности, и выбор режима ложится на плечи проектировщика.

Конкретизация задачи – ЧПУ

Одной из востребованных задач в мире автоматизации в последние годы стала тема числового программного управления (ЧПУ). Действительно, раньше при разработке ЧПУ приходилось постоянно следить за объёмом вычислений. Контроллеры 80-90-х годов не могли похвастаться и десятой долей вычислительной мощности среднего современного контроллера, а частота выдачи управляющего сигнала определялась заданной точностью обработки и инерционностью приводов подач. Например, для известной ранее СЧПУ НЦ-31 эта частота составляла 100 Гц. По этой причине алгоритм работы СЧПУ требовалось жёстко оптимизировать, а набор сервисных функций ограничивать. Применять ОС в таких системах было просто нецелесообразно.

Возможности современных контроллеров таковы, что позволяют не только существенно расширить возможности СЧПУ, но и использовать ОС, что в значительной степени упрощает задачу проектирования. Большинство программируемых логических контроллеров даже реализуют задачу управления электроприводом. На рынке также присутствуют решения СЧПУ, выполненные на базе обычного персонального компьютера. Рассмотрим подробнее задачу проектирования системы ЧПУ на основе системы реального времени.

Сравнительные характеристики ОС реального времени (Windows, QNX, On Time), выбор системы

Выберем одну из наиболее популярных ОС реального времени для решения нашей задачи (подробнее с информацией об этих ОС можно ознакомиться здесь: http://embedded.prosoft.ru/products/types/3995/).

Начнём с Windows Embedded. Легко заметить преимущества этой ОС: написано огромное количество драйверов для самого разнообразного оборудования, и при желании подключить к нашей СЧПУ, скажем, принтер дополнительной разработки не потребуется. Даже после выхода системы для поддержки нового оборудования нужно лишь будет скачать драйвер с сайта производителя. Другое несомненное преимущество – развитая графическая подсистема и совместимость её с ОС для персональных компьютеров. Разрабатывать интерфейс с оператором можно в любой среде проектирования программ на персональном компьютере. К недостаткам этой ОС можно отнести высокие требования к ресурсам.

Рассмотрим QNX. Практически всё сказанное можно отнести и к этой ОС. Драйверов периферийного оборудования, конечно же, меньше, но все основные производители периферийного оборудования не забывают и про эту систему. Среда проектирования программ поставляется вместе с ОС. Развитая графическая подсистема, широкий набор сервисных возможностей – всё это позволило QNX занять довольно широкую нишу на рынке операционных систем реального времени. По производительности эта ОС превосходит Windows Embedded, но цена весьма существенна на фоне RTOS-32 (On Time Informatik).

RTOS-32 (On Time Informatik), или On Time RTOS-32, отличается низкими требованиями к ресурсам. Она представляет собой набор библиотек и не является средой для выполнения программ, как привычные современные ОС. Код компилируется в среде разработки программ на ПК, затем готовый двоичный файл записывается на платформу. Такая реализация ОС объясняет её основные преимущества и недостатки: с одной стороны, очень высокая производительность, заметно более низкая цена, низкие требования к процессору (нужно только иметь BIOS), а с другой, меньший набор сервисных возможностей. Кроме того, реализация драйверов всего периферийного оборудования, не поддержанного в ОС, ложится на плечи разработчика.

Таблица 1

Сравнительная характеристика ОС реального времени

КРИТЕРИИ СРАВНЕНИЯ	WINDOWS EMBEDDED	QNX	ON TIME RTOS-32
Удобство написания драйверов	Унифицированный интерфейс для драйвера – написание драйвера переходит на уровень реализации нескольких функций. Обмен между пользовательской задачей и драйвером при помощи механизма сообщений		Единое адресное пространство и прямой доступ к портам позволяют разработать драйвер с любым интерфейсом
Набор библиотек	Большое количество поддерживаемых устройств, функции работы с базами данных, простой доступ к файловым ресурсам (доступ к файлу на удалённой машине отличается только указанием полного пути), есть системный журнал ошибок, возможно применение любых библиотек C/C++		Низкоуровневые драйверы, минимальная поддержка пользовательского интерфейса (окна, графика)
Интерфейсы	USB 2.0, IEEE 1394, 802.11x, IrDA, UPnP, RDP, Bluetooth, CAN, PROFIBUS и др. Хорошая поддержка стека протоколов TCP/IP		USB, RS-232. Поддержка стека протоколов TCP/IP реализована в меньшем объёме по отношению к Windows Embedded и QNX. Остальные протоколы реализуются при помощи драйверов
Файловые системы	Практически все. Трудно найти устройство хранения данных, для которого нет драйвера под Windows	Широкий спектр файловых систем, включая образную файловую систему, файловую систему в ОЗУ или на флэш-диске, файловые системы QNX, Linux, DOS, CD-ROM, DVD, NFS, CIFS, пакетную файловую систему и файловую систему со сжатием	Модуль файлового ввода-вывода RTFiles 32 позволяет использовать стандартные функции работы с файловыми системами FAT 12, FAT 16 и FAT 32 и работать с любыми типами носителей (НГМД, НЖМД, флэш-диски, статическая память, USB-диски, CD ROM и DVD). Нестандартное оборудование может поддерживаться через драйвер с простым интерфейсом
Лицензии	Отчисления с каждой проданной копии с предварительным заказом	Отчисления с каждой проданной копии с предварительным заказом (2–4 недели)	Нет отчислений. Покупается набор разработки
Стоимость набора разработчика	Самая низкая из рассматриваемых систем	Довольно высокая	Комплектуется для конкретной задачи. Для проектирования СЧПУ вдвое ниже, чем у QNX
Поддерживаемые платформы	Только x86 (как в ПК)	ARM, PowerPC, MIPS, x86, SH	x86, любая платформа с BIOS

В табл. 1 сведены основные, с точки зрения проектирования автоматизированной системы, характеристики рассмотренных ОС.

Есть ещё одна характеристика, которая в значительной степени влияет на выбор ОС, – это эффективность использования ресурсов. При её определении необходимо учитывать время переключения задач, минимальную конфигурацию системы для соответствующей ОС, время реакции на системное событие (семафор), время входа и выхода из прерывания. Как правило, ОС реального времени испытываются на разном оборудовании, которое вносит погрешность в анализируемые данные, поэтому точно оценить каждую систему не получится. Однако, ознакомившись с исследованиями (исследование Dedicated System Experts Windows CE 6.0 Embedded на Intel Pentium MMX 200 МГц, документ EVA-2.9-TST-CE-x86-62; исследование Dedicated System Experts QNX на PowerPC MPMC7410 333 МГц, документ EVA-2.9-TST-QNX-PPC-01-I01) и официальной документацией (http://www.on-time.com/rtkernel-32.htm), можно прийти к выводу, что у Windows Embedded сравнительно низкая эффективность, у QNX средняя, а у On Time RTOS-32 самая высокая эффективность использования ресурсов среди рассматриваемых систем.

Рассмотрев все три системы, можно сделать вывод: для проектирования ЧПУ, особенно на первом этапе, лучше всего подходит On Time RTOS-32. Вот основные качества этой ОС, повлиявшие в нашем случае на её выбор:

- поддержаны современные накопители и каналы передачи данных;
- низкие накладные расходы при вызове системных функций;
- невысокая, по сравнению с остальными ОС, стоимость;
- поддержка любого промышленного контроллера.

Разделение задачи на блоки, описание основных блоков с примерами использования механизмов ОС

Основой для реализации системы ЧПУ на базе On Time RTOS-32 может служить любой промышленный процессор. Процесс адаптирования операционной системы к конкретному процессору сводится к настройке физического канала отладчика (номер и скорость последовательного порта или адрес и порт для TCP/IP-соединения). Для работы с ОС, как, впрочем, и для разработки ЧПУ, потребуется среда разработки программ. Это может быть Microsoft Visual Studio либо Borland Developer Studio, так как On Time RTOS-32 поддерживает Win32-компиляторы от Borland и Microsoft (по данным официального сайта On Time Informatik).

Теперь все инструменты подготовлены, можно приступать к проектированию системы.

Рассмотрим работу абстрактной системы ЧПУ, обращая внимание (в тексте выделено курсивом) на входные и выходные данные проектируемой системы. Данные поступают или из *технологической программы* (работа в автоматическом режиме), или от оператора через *нажатие клавиш на пульте управления станком* (ручной режим). Эти данные преобразуются в *перемещения исполнительных механизмов* и *рабочего органа станка,* данные о текущем положении которого принимаются СЧПУ от *датчиков обратной связи*. *Алгоритмы управления электрооборудованием станка* работают постоянно и обеспечивают нормальную работу сис-

темы. Например, постоянно отслеживаются *входные сигналы* аварийных концевых переключателей, которые срабатывают при выходе рабочего органа за пределы рабочей зоны. В этом случае работа останавливается, *выдаётся сообщение об ошибке*. Работа СЧПУ определяется также *набором настраиваемых параметров*.

Теперь запишем входные и выходные данные и кратко рассмотрим каждый из этих наборов данных.

Входные данные:

- технологическая программа в кодах языка ISO 6983;
- датчики положения координатных осей;
- клавиатура;
- сигналы от электрооборудования станка;
- алгоритм управления сигналами электроавтоматики;
- набор параметров, адаптирующих ЧПУ к конкретному станку.

Выходные данные СЧПУ:

- сигналы управления электрооборудованием станка;
- сигналы управления приводами осей подач;
- информация, отображаемая на пульте оператора.

Технологическая программа представляет собой текстовый файл и определяет алгоритм движения рабочего органа станка, а также порядок выполнения технологических команд. Для разбора программы нам потребуется интерпретатор. Далее задание на перемещение нужно преобразовать в скорость движения рабочего органа станка – это типичный входной сигнал большинства приводов. Для этого нужно в соответствии с заданным максимальным ускорением рабочего органа станка рассчитать скоростную характеристику движения. Данной задачей занимается интерполятор.

Для контроля позиции рабочего органа станка и организации обратной связи используется модуль регулятора положения. Как правило, в системах ЧПУ регулирование проводится с постоянным системным тактом, так как при этом значительно упрощаются законы регулирования и легче избежать самовозбуждения системы. Обычно используется пропорционально-интегрально-дифференциальный регулятор положения (ПИД-регулятор).

Для всего периферийного оборудования нам понадобятся драйверы.

Для обработки данных с пульта оператора и индикации выводимой информации нужно предусмотреть модуль интерфейса пользователя.

Нужно помнить, что каждый станок – сложный механизм, который имеет множество характеристик, а чем механизм сложнее, тем меньше вероятность найти ещё один, в точности его повторяющий. Поэтому любая удобная система, спроектированная для конкретного станка, будет применяться только на этом станке (да и то в лучшем случае). Такая судьба проекта нас не устраивает, поэтому при проектировании будем ориентироваться на группу станков, а значит, придётся дать пользователю возможность адаптировать систему к конкретному станку при помощи определённого набора параметров.

Для того чтобы ЧПУ можно было адаптировать не только к разным станкам одной серии, но и к любому станку фрезерной или токарной группы, нужно учитывать, что в разных сериях станков используются разные электрические схемы, наборы сигналов и алгоритмы управления электрооборудованием. Следовательно, нам необходима возможность задавать этот алгоритм и менять его по мере необходимости. Блок, решающий эту задачу, называет-

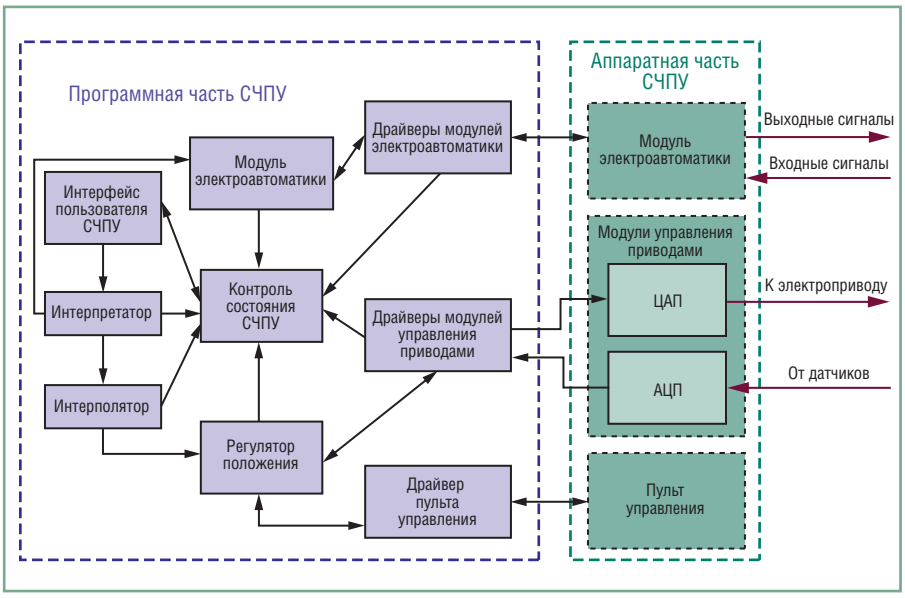

Рис. 1. Структурная схема системы ЧПУ

ся программируемым контроллером электроавтоматики (или модулем электроавтоматики). Программа для контроллера электроавтоматики готовится на одном из языков стандарта IEC 61131.

На рис. 1 приведена упрощённая структурная схема системы ЧПУ, построенной по описанным принципам.

При работе СЧПУ циклически выполняется программа цепей электроавтоматики, результат которой передаётся драйверам плат электроавтоматики в виде выходных сигналов. С этих же плат через драйвер на модуль электроавтоматики приходит информация о состоянии входов.

Положение рабочего органа станка контролируется регулятором положения: сигнал с датчика обратной связи через АЦП и драйвер модуля управления приводами поступает на вход регулятора, вычисляется значение ошибки положения рабочего органа. Далее, в зависимости от параметров регулятора, для каждой из осей системы рассчитывается сигнал управления. Этот сигнал через драйвер передаётся на ЦАП и далее на привод.

Модуль контроля состояния СЧПУ собирает данные о состоянии системы и информирует о нём оператора через интерфейс пользователя.

Перечисленные модули полностью описывают систему в состоянии равновесия, пока не поступит запрос на выполнение команды или программы.

Рассмотрим выполнение одной команды. Оператор формирует команду на пульте управления, при этом реакция на нажатие кнопок и отображаемый результат полностью находятся в зоне ответственности интерфейса пользователя СЧПУ. Когда ввод закончен и оператор нажал кнопку «Пуск», команда поступает на вход интерпретатора, где преобразовывается в удобную для системы структуру данных (вычисляются заданные перемещения по координатным осям, вектор подготовительных функций, действующих при отработке команды, отделяется информация для системы электроавтоматики). Разобранные технологические команды поступают в модуль электроавтоматики. Данные о новом положении рабочего органа поступают в интерполятор, где с учётом реальных ограничений станка (максимальной скорости движения рабочего органа, закона изменения ускорения, максимального значения ускорения и т.п.) вычисляются задания на перемещения для каждого системного такта. Именно интерполятор отвечает за то, что после отработки команды рабочий орган станка окажется в положении, заданном оператором. Задания на перемещения поступают на вход ПИД-регулятора в виде изменившейся заданной координаты, и система приводится в движение.

Рассмотрим, каким образом выбранная ОС может помочь реализовать приведённую схему.

Очевидным решением для блоков, которые выполняют циклические вычислительные задачи, является применение потоков. Под такое описание в программной части системы подходят все модули. При этом драйверы лучше отнести в один поток с наивысшим приоритетом – стабильность такта выдачи управляющей информации и приёма сигнала обратной связи определяет точность СЧПУ.

On Time RTOS-32 предоставляет весьма простой интерфейс для создания и управления потоками:

- RTKCreateThread и RTKRTLCreateThread – создание потока (вторая функция использует многопоточную версию библиотечных функций при создании потока);
- RTKResume – запуск;
- RTKSuspend – приостановка выполнения;
- RTKTerminateTask – уничтожение.

В листинге 1 приведён пример создания и управления потоком при помощи этих функций.

При проектировании нам потребуются только две функции: создание потока и запуск его на выполнение. Управление потоками при помощи внешних функций RTKSuspend/RTKResume лучше не использовать. Например, представим такую ситуацию. Поток, вычисляющий закон изменения координаты (интерполятор), был остановлен при помощи функции RTKSus-pend. Перед его запуском в системе уменьшилось значение максимально допустимого ускорения. После запуска интерполятор может начать генерировать несогласованные данные для регулятора положения, или не успеет остановить рабочий орган станка вовремя, или в обычно верных формулах возникнет ошибка (деление на 0, корень из отрицательного числа и т.п.). В общем случае поведение системы в такой ситуации не определено. Чтобы избежать этого, нужно либо индексировать все параметры, от которых зависит поток в каждом его цикле (хотя даже это не избавит от всех перечисленных проблем), либо забыть про управление потоками при помощи внешних функций управления состоянием. Отмечу, что приведённые рассуждения касаются любой ОС.

Альтернативой внешнему управлению потоками является управление внутреннее, когда поток изменяет своё состояние в заранее определённой им точке, а извне его только запускают или «просят» остановиться при помощи какого-либо механизма синхронизации. В On Time RTOS-32 для синхронизации потоков предусмотрены следующие механизмы:

- почтовые ящики;
- семафоры.

При проектировании системы удобно пользоваться и теми и другими. Вот как выглядит безопасный механизм

управления потоком при использовании семафора:

- поток ждёт семафора;
- когда семафор установлен, поток начинает работать.

И это всё. Проще не бывает. Для управления таким потоком достаточно устанавливать и сбрасывать соответствующий семафор. Изменим предыдущий пример, используя управление потоком при помощи семафора (см. листинг 2).

Примерно такой же механизм можно использовать при синхронизации с помощью почтовых ящиков:

- поток вызывает синхронную функцию получения сообщения из почтового ящика;
- поток обрабатывает полученные данные.

Почтовые ящики, очевидно, также являются идеальным средством для обмена структурированными данными между потоками и позволяют организовать ориентированную на события модель обработки информации.

On Time RTOS-32 позволяет использовать следующие типы семафоров:

- бинарный – семафор с автосбросом (для того чтобы тело потока в описанном механизме выполнилось второй раз, нужно ещё раз подать сигнал на этот семафор; после каждого прохода поток остановится сам);
- счётный – сколько раз подадим сигнал на семафор, столько раз он и сработает;
- событие – после установки такого семафора он остаётся в активном состоянии до тех пор, пока его не сбросят вручную (если рассматривать применение такого семафора для управления потоком, то, устанавливая семафор, мы запускаем работу потока, а сбрасывая – останавливаем, но только перед следующим циклом);
- ресурс – этот тип семафора специально предназначен для управления ресурсами; ОС отслеживает, чтобы захват и освобождение таких семафоров производились одним и тем же потоком; также гарантируется, что приоритет потока, занявшего ресурс, будет по крайней мере максимальным из приоритетов потоков, этот ресурс ожидающих;
- мьютекс – тип ресурсного семафора, единственное отличие которого заключается в том, что владеющий мьютексом поток может захватить мьютекс ещё раз.

Следующие функции предоставляют удобный согласованный интерфейс для использования семафоров:

- RTKCreateSemaphore – создание семафора;
- RTKSignal – запись в семафор одного сообщения;
- RTKWait – синхронное ожидание семафора;
- RTKWaitCond – асинхронное ожидание семафора;
- RTKWaitTimed – ожидание семафора в течение заданного промежутка времени;
- RTKPulse – запуск всех потоков, ожидавших семафор, и немедленный сброс этого семафора;
- RTKResetEvent – сброс семафора;
- RTKDeleteSemaphore – удаление семафора.

Для работы с почтовыми ящиками используются следующие функции:

- RTKCreateMailbox – создание почтового ящика;
- RTKDeleteMailbox – удаление почтового ящика;
- RTKClearMailbox – очистка почтового ящика.

Для управления сообщениями есть такие функции:

- RTKPut – синхронная функция передачи сообщения (поток, в котором была вызвана эта функция, останавливается до тех пор, пока в почтовом ящике не появится место для сообщения);
- RTKPutCond – асинхронная функция отправки сообщения (если в почтовом ящике места нет, то сообщение не передаётся; эту функцию можно использовать для передачи данных из прерываний драйверов);

ЛИСТИНГ 1

```
void RTKAPI threadInterpolator(void*)
{
        while(true)
        {
                // выполнение работы
                ...
        }
}
void taskControl(void)
{
        // инициализация данных
        ...
        RTKTaskHandle handleInterpolator = RTKRTLCreateThread(
                // тело
                threadInterpolator,
                // приоритет
                INTERPOLATOR_PRIORITY,
                // размер стека. Если 0 - размер стека
                // определяется настройками системы
                0,
                // после создания поток "спит"
                TF_SUSPENDED,
                // параметр, который передается в тело потока
                NULL,
                // имя потока
                "Thread interpolator");
        RTKResume(handleInterpolator);
        ...
        RTKSuspend(handleInterpolator);
        ...
}
```

ЛИСТИНГ 2

```
RTKSemaphore semaphoreInterpolator;

void RTKAPI threadInterpolator(void*)
{
        while(true)
        {
                RTKWait(semaphoreInterpolator);
                // выполнение работы
                ...
        }
}
void taskControl(void)
{
        // инициализация данных
        ...
        semaphoreInterpolator = RTKCreateSemaphore(
                ST_EVENT, 0, "Interpolator control");
        RTKTaskHandle handleInterpolator = RTKRTLCreateThread(
                // тело
                threadInterpolator,
                // приоритет
                INTERPOLATOR_PRIORITY,
                // размер стека. В данном случае 0 -
                // размер стека определяется настройками системы
                0,
                // после создания поток сразу работает
                0,
                // параметр, который передается в тело потока
                NULL,
                // имя потока
                "Thread interpolator");
        // запуск потока
        RTKSignal(semaphoreInterpolator);
        ...
        // "просьба" потоку остановиться.
        // остановится поток только в начале цикла
        RTKResetEvent(semaphoreInterpolator);
}
```

ЛИСТИНГ 3

```
RTKSemaphore semaphorePIDRegulator;
RTKSemaphore magneticsSemaphore;
RTKMailbox consoleBox;
RTKMailbox interpretatorBox;
RTKMailbox interpolatorBox;
RTKMailbox regulatorBox;
RTKMailbox magneticsBox;
// время системного такта - параметр СЧПУ
unsigned machineCycle;

void RTKAPI userInterface(void*)
{
        TConsoleMessage data;
        while(true)
        {
                // обработка нажатия клавиши
                TUserFrame userFrame;
                RTKGet(consoleBox, &data);
                ...
                if (frameExecuteRequest(data))
                        RTKPut(interpretatorBox, &userFrame);
                ...
        }
}
void RTKAPI interpretator(void*)
{
        while(true)
        {
                TUserFrame userFrame;
                RTKGet(interpretatorBox, &data);
                TInternalFrame* frame = parse(data);
                ASSERT(frame != NULL);
                if (!frame.geometry().Empty())
                        RTKPut(interpolatorBox, &frame);
                if (!frame.magnetics().Empty())
                        RTKPut(magneticsBox, &frame);
        }
}
void RTKAPI interpolator(void*)
{
        while(true)
        {
                TInternalFrame* frame = NULL;
                RTKGet(interpolatorBox, &frame);
                // расчет эпюр скоростей
                calculate(frame->geometry());
                do {
                        TRegulatorTickTask* tickTask =
                                new TRegulatorTickTask;
                        frame->geometry().nextTick(tickTask);
                        RTKPut(regulatorBox, &tickTask);
                } while (
                        !frame->geometry().Empty() &&
                        !frame->Stopped() );
                // контроль кадра, освобождение ресурсов
                ...
        }
}
void RTKAPI PIDRegulator(void*)
{
        while(true)
        {
                RTKWait(semaphorePIDRegulator);
                TRegulatorTickTask* data = NULL;
                if (RTKGetCond(regulatorBox, &data))
                {
                        // обработка задания
                        ...
                }
                // обработка обратной связи,
                // запись в данные драйвера
                ...
        }
}
void RTKAPI magnetics(void*)
{
        while(true)
        {
                TInternalFrame* frame;
                RTKWait(magneticsSemaphore);
                if (RTKGetCond(magneticsBox, &frame))
                {
                        runTech(frame->magnetics());
                        // индексирование кадра,
                        // освобождение ресурсов
                        ...
                }
                techStep();
                chainStep();
                // обработка данных от электроавтоматики,
                // если произошли ошибки - генерация сообщений
                ...
        }
}
typedef void (RTKAPI *TinterruptHandler)(void);
void RTKAPI timerInterrupt(void)
{
        static unsigned ticks = 0;
        ++ticks;
        if (ticks >= machineCycle)
        {
                // обмен данными с оборудованием
                refreshSignals();
                RTKSignal(semaphorePIDRegulator);
                RTKSignal(magneticsSemaphore);
        }
        // вызов стандартного обработчика прерывания
        ((TinterruptHandler)OldHandlerTimer.D.Local)();
}
void initResources(void)
{
        // инициализация данных
        ...
        semaphorePIDRegulator =
                RTKCreateSemaphore(ST_BINARY, 0, "Regulator");
        magneticsSemaphore =
                RTKCreateSemaphore(ST_BINARY, 0, "Magnetics");

        consoleBox = RTKCreateMailbox(
                sizeof(TConsoleMessage),
                CONSOLE_BUFFER,
                "Console messages");
        interpretatorBox = RTKCreateMailbox(
                sizeof(TUserFrame),
                FRAMES_BUFFER,
                "ISO frames");
        interpolatorBox = RTKCreateMailbox(
                sizeof(TInternalFrame*),
                MOVE_DATA_BUFFER,
                "Interpolator data");
        regulatorBox = RTKCreateMailbox(
                sizeof(TRegulatorTickTask*),
                REGULATOR_BUFFER,
                "Regulator data");
        magneticsBox = RTKCreateMailbox(
                sizeof(TInternalFrame*),
                MAGNETICS_BUFFER,
                "Magnetics data");

        RTKRTLCreateThread(userInterface,
                INTERFACE_PRIORITY,  0, 0,
                NULL, "Interface thread");
        RTKRTLCreateThread(interpretator,
                INTERPRETATOR_PRIORITY, 0, 0,
                NULL, "Interpretator");
        RTKRTLCreateThread(interpolator,
                INTERPOLATOR_PRIORITY, 0, 0,
                NULL, "Interpolator");
        RTKRTLCreateThread(PIDRegulator,
                REGULATOR_PRIORITY, 0, 0,
                NULL, "PIDRegulator");
        RTKRTLCreateThread(magnetics,
                MAGNETICS_PRIORITY, 0, 0,
                NULL, "Magnetics thread");

        // сохранили дескриптор старого прерывания по таймеру
        RTKSaveIRQHandlerFar(IRQTimer, &OldHandlerTimer);
        // запретили прерывания по таймеру
        RTKDisableIRQ(IRQTimer);
        // установили новый обработчик прерываний по таймеру
        RTKSetIRQHandler(IRQTimer, TimerInt);
        // разрешили прерывания по таймеру
        RTKEnableIRQ(IRQTimer);
}
```

- RTKPutTimed – сообщение не попадает в очередь, если по истечении заданного времени в очереди так и не появилось свободного места (функция возвращает управление, как только поставит сообщение в очередь или когда истечёт заданное время).

Аналогичная группа функций есть для приёма сообщений: RTKGet, RTKGetCond, RTKGetTimed.

Для управления сообщениями есть ещё две функции, которые также могут быть очень полезны:

- RTKPutFront – сообщение кладётся в начало очереди (функция синхронная, её можно использовать для передачи срочных сообщений);
- RTKNextCond – просмотр сообщения из почтового ящика (эта функция отличается от RTKGetCond тем, что не выбирает сообщение, а только создаёт его копию).

Рассмотрим, где для управления потоком лучше использовать семафор, а где больше подходят почтовый ящик и система сообщений.

Если при передаче данных драйверу воспользоваться системой сообщений почтового ящика, то нужна будет дополнительная обработка. Вероятно, нужно обработать все сообщения и, соответственно, пересчитать результат как суперпозицию информации, которая находится в них. А делать это придётся уже после получения данных. Лучше готовить данные заранее, а по сигналу системного таймера (сигналим семафором из прерывания) сразу начинать обмен. Это позволит сделать жёстким системный цикл и перенести

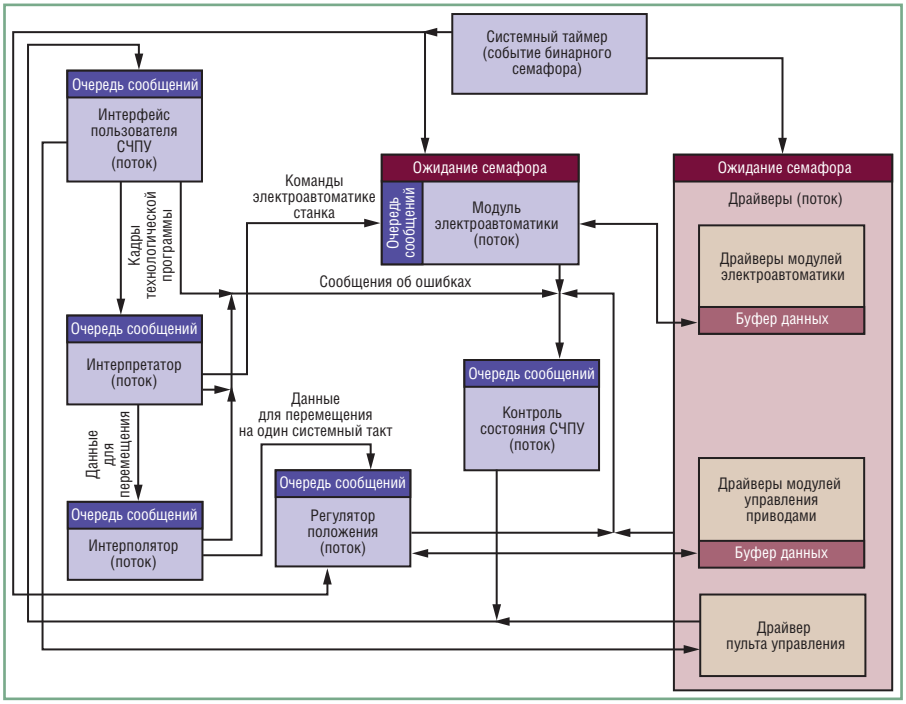

Рис. 2. Структурная схема системы СЧПУ «Феникс», построенной на базе On Time RTOS-32

сложную обработку данных в код с меньшими требованиями к быстродействию.

Данные от драйвера пульта потоку интерфейса с пользователем лучше передавать в виде сообщений, ведь пока оператор не произвёл никаких действий, не нужна и ответная реакция.

Потоком регулятора положения лучше управлять при помощи семафора, так как его необходимо запускать регулярно. Данные для этого потока логично передавать в виде сообщений. В этом случае семафор – синхронизирующий элемент, а почтовый ящик используется только как механизм передачи данных.

Интерпретатор нужен лишь тогда, когда требуется разобрать команду, которая поступит от интерфейса с пользователем, а значит, этим потоком лучше управлять при помощи очереди сообщений и почтового ящика.

Запуск интерполятора нужно производить при перемещении рабочего органа станка. Такое перемещение может быть результатом выполнения технологической программы или работы программы электроавтоматики станка. Данные для интерполятора удобно передавать в виде очереди сообщений, этой же очередью удобно синхронизировать его работу.

В коде это будет выглядеть так, как показано в листинге 3.

Рис. 3. Станок СМ1М-Ф3 на Рославльском автоагрегатном заводе АМО ЗИЛ

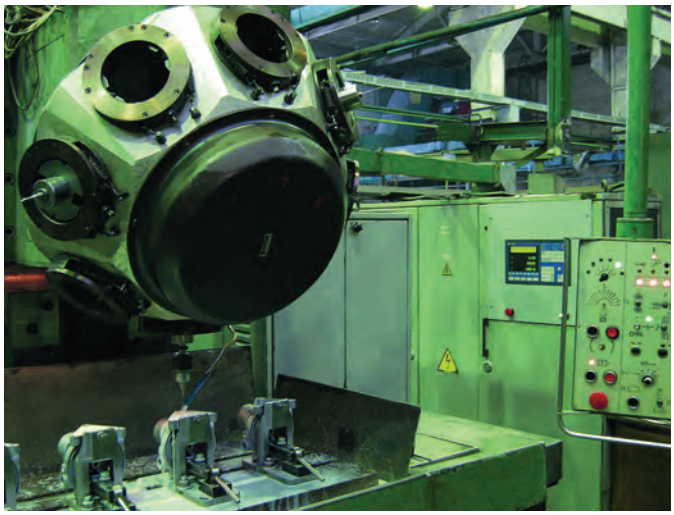

Рис. 4. Станок МА655А3 с заменённой системой управления «Курс»

Описание СЧПУ «Феникс», спроектированной на основе On Time RTOS-32

По описанным принципам в НПО «Рубикон-Инновация» была разработана система ЧПУ «Феникс». Упрощённая структурная схема этой системы показана на рис. 2.

Стоит отметить, что наша организация уже более 10 лет успешно работает на рынке изделий с повышенными требованиями по надёжности и устойчивости к климатическим условиям: это и космические бортовые приборы, которые успешно эксплуатируются на спутниках серии «Ямал», и приборы для авиации. Среди наших заказчиков такие организации, как РКК «Энергия», «Российская корпорация ракетно-космического приборостроения и информационных систем», НИИ авиационного оборудования, ЗАО «РААЗ АМО ЗИЛ». Применение опыта, накопленного за время работы в этой сфере, позволило минимизировать стоимость разработки аппаратной части СЧПУ.

Разбиение задачи на модули и активное использование системных объектов On Time RTOS-32 позволили проектировать программную часть СЧПУ «Феникс» относительно небольшими силами. Поддержка и развитие ОС On Time RTOS-32 даёт возможность постепенно наращивать функциональность СЧПУ. Графическая подсистема, поддержка различных файловых систем, стек протокола USB – всё это и многое другое является частью операционной системы.

При разработке СЧПУ «Феникс» особое внимание было уделено изучению существующих наработок и сложившихся правил в автоматизации металлообработки. В частности, это отразилось в реализации свободно программируемой электроавтоматики на языке ST (Structured Text) стандарта МЭК 61131-3, а также в развитой структуре системных параметров. Это позволило не только применять СЧПУ для замены старых систем управления токарными и фрезерными станками, но и проводить модернизацию уникальных станков. Примером может служить запуск станка СМ1М-Ф3 (рис. 3), который проводился совместно работниками предприятий ЗАО «РААЗ АМО ЗИЛ» (монтаж и подключение СЧПУ) и НПО «Рубикон-Инновация» (настройка СЧПУ и запуск станка). Время, потребовавшееся для настройки параметров станка, а также для написания и отладки программы электроавтоматики, составило две недели.

Использование СЧПУ «Феникс» в металлообработке

СЧПУ «Феникс» применяется при модернизации таких существенно отличающихся друг от друга станков, как ФП17, МА655А3 (рис. 4), МС12-250, КД-46, 16Б16, 16К20. Стандартный язык программирования электроавтоматики, развитая система сообщений об ошибках и простая интеграция с CAM-системами позволяет строить недорогие производственные участки с малым временем изготовления готовых изделий.

В качестве примера можно привести производственную цепочку изготовления корпуса разъёма стандарта D SUB (DB-9) со специализированными размерами для переходника DB-9 – USB на базе чипа FTDI, преобразующего RS-422 от промышленного устройства в виртуальный COM-порт компьютера (преобразователь встроен в DB-9, отсюда специализированные размеры корпуса разъёма). Описание оборудования: горизонтальный фрезерный станок МС12-250 под управлением СЧПУ «Феникс»; система EdgeCam для подготовки технологических программ. Если нет трёхмерной модели детали и её необходимо изготовить по чертежу, то модель строится (на этом этапе обычно выявляется бóльшая часть ошибок, если они есть). У нашего технолога эта операция заняла 1 час. Далее в системе EdgeCam определяются инструменты, которыми будет вестись обработка, указывается последовательность технологических операций (ещё 2 часа). После этого EdgeCam генерирует технологическую программу, которая загружается в СЧПУ «Феникс». Этот процесс отнимает 5 минут. Запускается обработка (две установки, каждая рассчитана приблизительно на 22 минуты обработки), и деталь готова! Общее время обработки, как нетрудно посчитать, составило чуть менее 4 часов. И это время, которое прошло от приёма чертежа до выдачи готовой детали.

Таким образом, использование готовых решений, в частности ОС On Time RTOS-32, позволяет значительно сократить время разработки программно-аппаратного комплекса, а также облегчает его поддержку и развитие. ●

Литература

1. Горошко Е. Операционные системы реального времени [Электронный ресурс]. – Режим доступа : www.qnxclub.net/files/articles/rtos/rtos.html.
2. Александреску А. Современное проектирование на C++. – М. : Издательский дом «Вильямс», 2008.

E-mail: lex_rubicon@bk.ru