И.В. ПЕТРОВ (Компания “Пролог”), В. ДОЛЛ (Компания “3S-Smart Software Solutions GmbH”)

Отладка прикладных ПЛК программ в CoDeSys (часть 9)[1]

Как и было запланировано в предыдущей статье, мы переходим к методам создания и отладки многозадачных проектов. Но прежде чем углубляться в тонкости, давайте поразмышляем над вопросами: "Что такое идеальный контроллер? Почему он не реализуем? Зачем нам многозадачность и можно ли обойтись без нее?" Время, потраченное на разбор этих вопросов, с лихвой окупится при практической работе, поскольку вы сможете правильно и технично использовать сильные стороны существующих программных инструментов, вместо того чтобы постоянно бороться с тем, что кажется неправильным, а по сути просто непонятым.

Идеальный ПЛК

Контроллер должен вырабатывать на своих выходах сигналы, которые определяются по заданному алгоритму на основании мгновенных значений входных сигналов и их предыстории. В идеальном мире время от изменения входов до соответствующей ответной реакции должно быть равно нулю. То есть идеальный контроллер должен обладать идеальным быстродействием, он должен успевать реагировать даже на импульсы бесконечно малой длительности. В современной математике создан аппарат, позволяющий моделировать и исследовать такие устройства, но физически изготовить такой контроллер нельзя. Даже если мы сделаем контроллер аппаратно, например, реализуем его алгоритм работы на интегральных микросхемах, то мы не достигнем абсолютного быстродействия (см. Рис 1). Мы сможем реагировать очень быстро и фиксировать очень короткие импульсы, но конечно, не бесконечно малые. Причем очевидно, что с ростом сложности алгоритма и соответственно числа элементов схемы, время реакции будет возрастать.

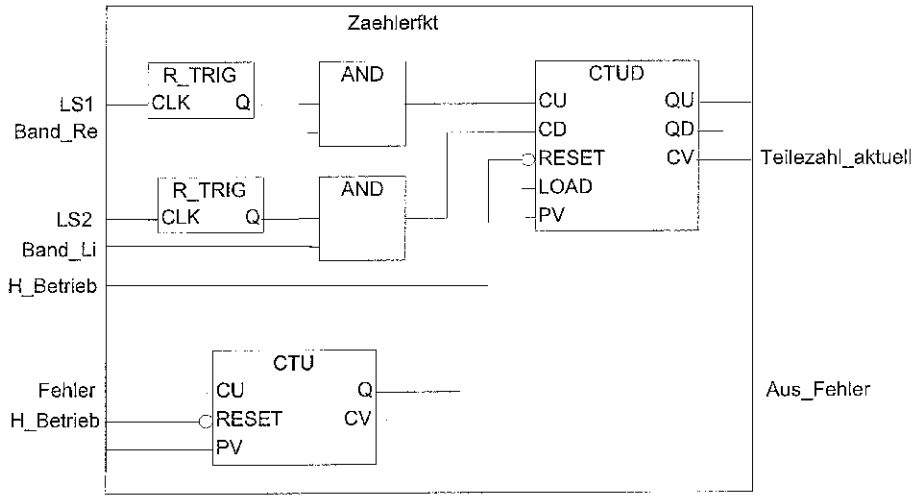

Рис. 1. Аппаратный контроллер

Образцовый ПЛК

К счастью, на практике от контроллера и не требуется сверхбыстрых реакций. Например, силовое реле нет смысла переключать чаще, один раз в 100мс, его контакты просто физически не успеют переключиться. То есть логично сразу отказаться от игр с бесконечно малыми временами и выделить контроллеру некое разумное время на принятие решения. Получается следующая модель: 1) снимаем мгновенные значения входов; 2) спокойно думаем, считая, что значения входов еще не изменились; 3) вырабатываем значения для выходов. Это есть ничто иное, как хорошо знакомый нам рабочий цикл контроллера. Очевидно, что если контроллер будет работать быстрее, чем управляемые им процессы, то он сможет полностью выполнять свои функции. Остается точно определиться насколько быстрее.

Теоретически доказано, что если произвольный сигнал не содержит частот выше f, то он может быть полностью восстановлен, если известны его мгновенные значения, взятые через равные промежутки времени f/2 (теорема Котельникова). Для логических сигналов это означает, что нам достаточно выбрать самый короткий

[1] *Продолжение. Начало в №№ 2-5,7,9,11. 2006 г., № 1. 2007г.*

 © НАУЧТЕХИЗДАТ 2007г.

импульс, на который мы должны реагировать и сделать наш рабочий цикл вдвое короче. То есть, если мы имеем дело с силовыми реле, то необходимо и достаточно иметь рабочий цикл равный 50мс. Делать его короче нет смысла.

Поскольку мы очень часто имеем дело с многомерными (взаимосвязанными) сигналами (например, положениями узлов или фазами сети), то возникает требование снимать значения входов в один момент времени и устанавливать значения на выходах столь же согласованно. На практике это опять же делается не мгновенно, но достаточно быстро, причем обеспечение нужной скорости является проблемой изготовителя контроллера, который должен применить соответствующие программно-аппаратные решения. Нас это волновать не должно.

Тот факт, что значения входов не изменяются в ходе выполнения прикладной программы, является очень важным. Это принципиально снимает проблему синхронизации различных частей программы. Один раз оценив значение некой входной переменной, мы можем гарантированно использовать ее во множестве вычислений без вспомогательных переменных, без повторных проверок, без риска вызвать ошибку исполнения или бесконечный цикл. Практически программа работает не напрямую с входами и выходами контроллера, а с их образом в памяти, своего рода защитным слоем. Такой подход позволят создавать надежные программы даже начинающим.

Как мы уже отметили, контроллер должен учитывать не только значения входов, но и предысторию или, иными словами, текущее состояние. Оно однозначно определяется множеством значений переменных. Очевидно, число состояний ПЛК может быть очень большим, но конечным. Один рабочий цикл контроллера – это один такт его работы. Переход из одного состояния в другое происходит только по тактам, то есть наш образцовый ПЛК является конечным автоматом. Очевидно, чем меньше состояний в программе, тем проще ее написать и верифицировать. Существует так называемый автоматный подход к программированию (мы использовали его в примере третьей части статьи), суть которого в четком выделении состояний их минимизации и кодировании при помощи возможно меньшего числа переменных, в идеальном случае всего одной. В языке SFC применяется несколько иной принцип, здесь шаги передают друг другу маркеры активности, которых может быть несколько. Это также происходит строго по тактам работы. Общая логика работы программы здесь четко отделена от деталей реализации и представлена исключительно наглядно.

Обратите внимание, в идеальном и в образцовом контроллерах мы не использовали привязку ко времени. В этом нет необходимости. Выдержка определенных интервалов в ПЛК делается при помощи таймеров. Таймеры могут быть сделаны программно или аппаратно, в зависимости от технических требований и возможностей. В любом случае для нас это как бы некое устройство, мы взводим таймер на нужное время и контролируем его выход готовности по единым правилам игры, аналогично входам контроллера. Подобным образом мы используем устройства (или программные блоки), измеряющие перемещение или другие физические величины.

Итак, образцовый контроллер отличается от идеального тем, что работает не мгновенно, но с необходимой и достаточной скоростью. Он должен работать строго по тактам, следующим с равными промежутками времени. То есть время рабочего должно быть детерминированным, оно не должно меняться в зависимости от того насколько сложные вычисления происходят в конкретном рабочем цикле. Мы не будем сейчас более углубляться в математику, нам достаточно понять, что циклическая модель функционирования взята не с потолка и не выдумана членами рабочей группы МЭК, она опирается на фундаментальные законы. Образцовый ПЛК позволяет создавать программы, обладающие необходимой реактивностью, сосредоточив свои усилия исключительно на прикладном алгоритме.

Многозадачность

В идеальном контроллере нет причин использовать более чем одну задачу. Любые действия выполняются мгновенно. В реальном ПЛК мы также можем помещать все, что нам нужно в один рабочий цикл. В большинстве прикладных программ так и делается. Если время, реально затрачиваемое на все вычисления в одном рабочем цикле, оказывается меньше заданного интервала его повторения, то система исполнения искусственно затягивает его. В противоположном случае, если мы затеяли в одном цикле слишком громоздкие вычисления, то естественно цикл растянется. По-другому сделать нельзя, этим любой реальный ПЛК отличается от образцового. Но обычно время рабочего цикла выбирается с большим запасом (единицы и десятки миллисекунд) в сравнении с тем, что может обеспечить процессорное ядро. В результате эта проблема не возникает и даже можно особенно не задумываться над оптимизацией программы. Но иногда мы сталкиваемся с тем, что есть некое устройство (или вход), которое требует значительно (в десятки раз) более быстрой реакции. Здесь возникают возможные варианты:

1) Задать в настройках контроллера достаточно малое время рабочего цикла, но тогда возникают сложности с выполнением громоздких вычислений, которые не вмещаются в такой цикл и не всегда возможно разделить их на несколько достаточно малых, ‘размазав’ их на несколько циклов.
2) Не пытаться поддерживать рабочий цикл постоянным, разрешить ему изменяться в зависимости от прикладной программы. В простых ПЛК так и сделано. Ответственность за реактивность полностью

отдается на уровень прикладной программы. Если, например, мы запустили некую операцию, ждем результат (опрашиваем вход) и должны очень быстро на него среагировать, то можно временно обойти вызов других частей и таким образом сделать достаточно короткий цикл. Фактически этот вариант хуже первого, поскольку время реакции отдельных ветвей программы оказывается трудно оцениваемым.

3) Сделать возможность прямого физического чтения значений входов. В результате в нужное время и в нужном месте мы сможем написать очень быстрый цикл (типа WHILE в ST) и дождаться необходимой реакции. В некоторых упрощенных системах программирования ПЛК все еще можно встретить такой подход, но все реже. В нем создание образа входов не используется, все входы и выходы доступны непосредственно. Выше мы разобрались с причинами, делающими этот вариант неудобным и даже опасным.
4) Использовать специализированные быстрые входы. Изменение значения входа должно отслеживаться аппаратно и мгновенно приводить к вызову определенной программы. Это обеспечивает максимально возможную скорость реакции. Вариант требует специальной аппаратной поддержки и особых предосторожностей при программировании, как работа с прерываниями на любом языке программирования. Он применим в определенных случаях, которые мы рассмотрим несколько позднее.
5) Сделать в ПЛК возможность создавать несколько рабочих циклов, то есть несколько задач, каждая из которых имеет свой образ входов/выходов и работает независимо. В терминологии стандарта МЭК это означает создание нескольких ресурсов, то есть нескольких контроллеров. Так мы сможем легко написать компактные быстрые задачи для обслуживания отдельных устройств и громоздкие медленные задачи, выполняющие менее требовательные к скорости вычисления.

Очевидно, последний вариант наиболее красив. С его помощью можно решать сложные задачи с гарантированной реактивностью. Использование многозадачности требует аккуратности даже в терминологии.

Терминология

В описаниях ОС мы сталкиваемся с терминами ‘задача’, ‘поток’ и ‘процесс’. В CoDeSys мы говорим только о задачах, правильно ли это? Действительно в Unix подобных ОС система исполнения CoDeSys SP является процессом, а отдельные МЭК являются потоками. Во встраиваемых ОС, таких как QNX, VxWorks и µCOS нет механизма защиты памяти между процессами или задачами. Говоря о потоках, мы однозначно имеем в виду элементы операционной системы, поддерживающие процессы. Говоря о многозадачности, мы имеем в виду абстрактный принцип разделения вычислительной мощности процессора для выполнения разной работы. Каждая из МЭК задач имеет определенный тип (циклическая, событийно управляемая, свободно выполняемая), приоритет и интервал вызова. Более приоритетная задача может прервать работу другой задачи. Такая схема называется *приоритетной вытесняющей многозадачностью*. Она поддерживается в самых мощных 32-х разрядных контроллерах с CoDeSys. В более простых ПЛК применяется так называемая *корпоративная многозадачность*, где каждая задача обязана доработать цикл, прежде чем управление будет передано другой задаче, даже если она имеет более высокий приоритет. Ее реализация существенно проще и позволяет более полно использовать вычислительные возможности процессора. Однако корпоративность подразумевает тесное сотрудничество системы исполнения с пользовательской программой для обеспечения правильной работы. Мы еще вернемся к этой теме более подробно, а пока перейдем к практике и рассмотрим вводный пример.

Пример

Пожалуй, самый простой случай, когда многозадачность может быть полезна – это моделирование объекта управления. Классическая простейшая задача на применение языка SFC – это стиральная машина, основные этапы работы которой: контроль закрытия дверки, ожидание нажатия кнопки пуск, наполнение бака водой, нагрев, стирка, слив воды. Мы не станем рассматривать реализацию примера детально, для тех кто читал предыдущие статьи данного цикла, он не представляет сложности. Подскажем только еще несколько полезных практических приемов программирования, попутно слегка усложнив пример. Добавим в нашу машину несколько режимов стирки, отличающихся температурой, временем стирки и количеством используемой воды. Для описания режима определим структуру *WashPara*:

```
TYPE WashPara :
STRUCT
  WaterIn: TIME;
  WashTime: TIME;
  WashTemp: WORD;
```

```
    WaterOut: TIME;
END_STRUCT
END_TYPE
```

Это ничто иное, как новый тип данных. Такие определения делаются в проекте централизованно на вкладке Data types (типы данных) менеджера проекта. Теперь мы можем объявлять в проекте переменные типа WashPara.

Создадим в проекте новый POU типа программа (PROGRAM), назовем его *Wash* и выберем упрощенный язык SFC. В разделе объявлений *Wash* объявим константу для числа режимов:

```
VAR CONSTANT
  maxProgNr :BYTE:= 5;
END_VAR
```

Идея состоит в том, чтобы везде, где потребуется количество режимов, использовать *maxProgNr*, это позволит при необходимости легко менять их число в единственном месте программы. Для хранения режимов объявим массив структур:

```
MyWashPara: ARRAY [0..maxProgNr] OF WashPara;
```

Обращение к элементам структуры происходит через точку после указания элемента массива, например:

```
MyWashPara[i].WaterOut := T#5s;
```

Далее мы хотим связать длительность выполнения определенных шагов SFC со значениями элементов массива структур, так чтобы мы смогли динамически влиять на работу диаграммы, изменяя только значения переменных. Вероятно, понадобятся таймеры или даже массив таймеров. Все гораздо проще, используем уже знакомые нам атрибуты шагов, а именно минимальное время активности шага. Привязка элементов массива к временам шага показана на рис. 2. Переходы после выполняемых по времени шагов всегда разрешены, то есть им присвоены константы TRUE. Этим собственно все программирование и исчерпывается. Мы практически готовы к демонстрации заказчику действующей модели.

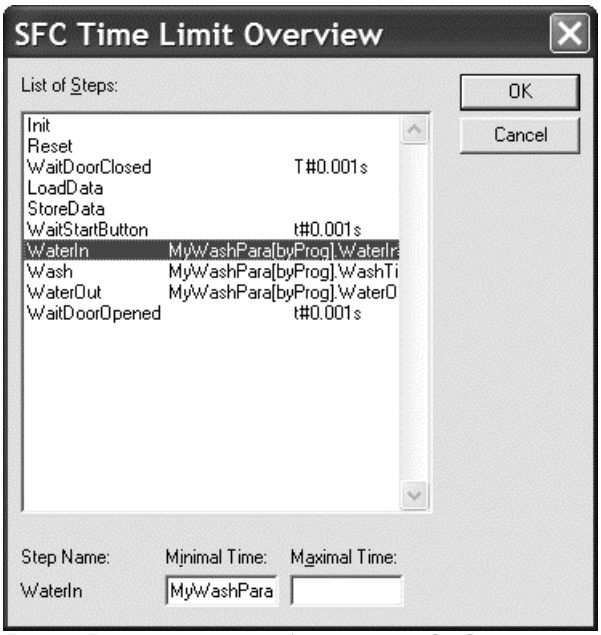

Рис. 2. Временные атрибуты шагов SFC

Для наглядной демонстрации работы нашей машины удобно использовать встроенную графическую визуализацию CoDeSys. Ее возможный вариант показан на рис. 3.

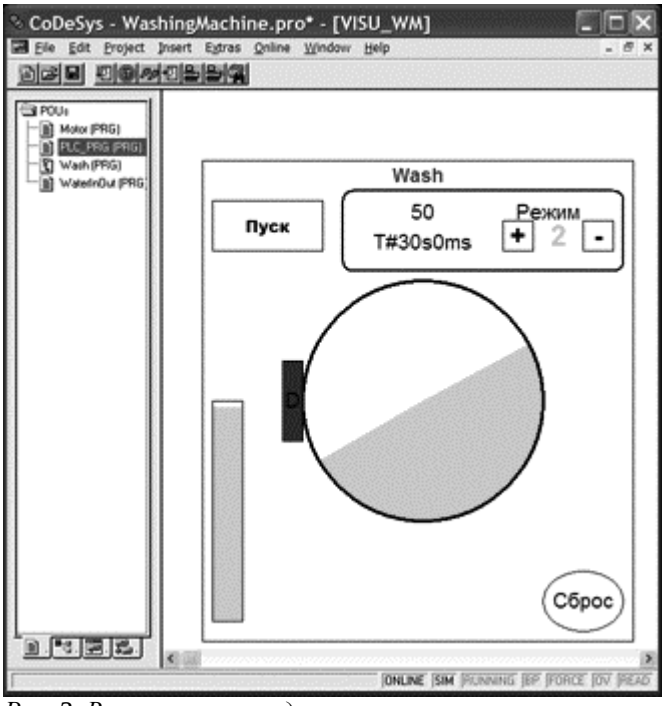

Рис. 3. Визуализация модели

Наша визуализация слишком уж статична, хотелось бы отобразить вращение барабана. Это можно сделать при помощи закрашенного сегмента. Напишем отдельную программу *Motor*, ее задача плавно циклически изменять переменную *wAngle* от 0 до 360 градусов, которую мы свяжем с углом сектора соответствующего элемента визуализации. Например, так:

```
IF xMotOn THEN
    wAngle := wAngle + 5;
    IF wAngle > 359 THEN
        wAngle := 0;
    END_IF
ELSE
    wAngle := 180;
END_IF
```

Как видите, здесь нет таймеров, обеспечивающих достаточно медленную для зрительного восприятия скорость вращения барабана. Мы можем поступить проще, создадим для целей моделирования циклическую задачу с достаточно медленным циклом работы, который легко регулировать. После отладки эту задачу можно будет просто отключить, не изменяя ничего в основных программах.

По умолчанию CoDeSys создает новый проект с единственной программой PLC_PRG, которая неявно включена в единственную циклическую задачу. Чтобы сделать многозадачный проект, необходимо использовать конфигуратор задач (Task configuration), расположенный на вкладке ‘Ресурсы’ менеджера проектов. Щелкните правой кнопкой мыши по элементу Task configuration и в появившемся контекстном меню выберете команду Append task. Сначала создадим главную задачу, она будет называться *Main*, ее тип – циклический (cyclic) и интервал работы T#10ms. Аналогичным способом откройте контекстное меню для нашей новой задачи и добавьте в нее (Append program call) программу PLC_PRG();. Теперь создайте еще одну циклическую задачу *MotorSim* с интервалом T#100ms, в нее вставьте вызов программы Motor();. Теперь конфигурация задач должна выглядеть так, как показано на рис.4.

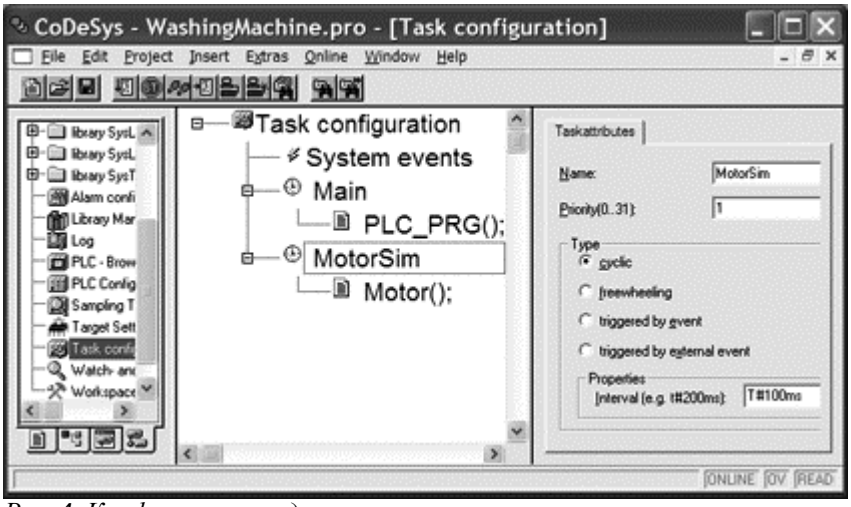

Рис. 4. Конфигурация задачи

Готовый проект для данной задачи вы можете найти в разделе ‘Публикации’ на сайте www.codesys.ru. Его выполнили участники тренинг-курсов по CoDeSys под руководством Ральфа Хольца. Они развили наше решение, дописав еще одну программу для моделирования уровня воды в баке. Кроме того, добавлена возможность сохранения массива режимов стирки в файле. Моделирование позволило понять недостатки предложенного интерфейса управления машиной и ‘обкатать’ варианты его усовершенствования.

Заключение

Мы начинали этот цикл статей с простых вещей, применимых и для других МЭК систем программирования, незаметно дошли до границ их возможностей и продолжаем движение дальше. В CoDeSys действительно пишут программы, нехарактерные для ПЛК, от редакторов и многозадачных интерпретаторов специализированных языков управления машинами до графических интерфейсов (в CoDeSys 3.0). Работа с разделяемыми ресурсами, точная синхронизация задач, их динамическое создание и отладка требуют применения специальных приемов, не предусмотренных стандартом МЭК. Необходимые для этого функции в CoDeSys реализованы в виде дополнительных библиотек. Их состав разработан и утвержден совместно компаниями-членами CoDeSys Automation Alliance. Ряд новых компонентов самого комплекса CoDeSys реализован его собственными средствами на языках МЭК и неявно использует несколько задач. Например, в SoftMotion точное управление приводами и интерполяцию траекторий перемещения невозможно сделать в одной задаче. Поэтому ваш проект может оказаться многозадачным, даже если вы об этом не знаете. Вы можете создавать подобные расширения комплекса и самостоятельно. От этого нас отделяет всего несколько шагов.

Продолжение следует.