И.В. ПЕТРОВ (Компания “Пролог”)

Отладка прикладных ПЛК программ в CoDeSys (часть 8)[1]

Как уже знаем, CoDeSys – компилирующая система. Это означает, что из среды программирования в контроллер загружается машинный код. Благодаря информации из конфигурационных файлов CoDeSys узнает тип микропроцессора, объем памяти, стека и прочие детали, необходимые для генерации исполняемого кода. Все эти тонкости настраиваются изготовителем контроллера и нас интересовать не должны. В любой интерпретирующей системе, даже использующей промежуточный код, выполнение любого оператора МЭК языков – это всегда вызов подпрограммы с соответствующей подготовкой и передачей параметров. В CoDeSys многие операторы транслируются в единственную ассемблерную команду. Очевидно, по быстродействию выполнения прикладных программ этот подход вне конкуренции. Естественно, код, формируемый компилятором, не рассчитан на чтение человеком, в нем нет имен переменных и функций – только адреса. Это существенный минус. В случае возникновения ошибки при исполнении кода система может выдать только маловразумительное сообщение, наподобие знакомых всем пользователям PC. Естественно, нам хотелось бы узнать точное название шага или строки, где произошла ошибка, и желательно знать при каких условиях. Десять лет назад мы сказали бы, что подобные требования из области фантастики, но, к счастью, в современных профессиональных системах программирования компилирующего типа такое уже возможно. Учитывая претензии CoDeSys на роль лидера среди МЭК систем программирования высшего класса, мы вправе требовать от него таких средств.

Анализ условий перехода во время выполнения

Итак, возвращаемся к отладке SFC. В конце прошлой статьи мы рассмотрели технику выявления шагов, которые выполняются подозрительно долго. Это делается путем задания атрибутов контроля времени активности шагов. Обратите внимание, что переменная SFCErrorStep типа строка дает нам не адрес и не индекс, а именно название ошибочного шага. Безусловно, это очень сильная возможность. Но условие последующего перехода может быть достаточно сложным и включать десяток переменных. Поэтому для более детального анализа ситуации нужно знать наименования этих переменных и их значения на момент возникновения ошибки. Это возможно. Подключите к проекту библиотеку *AnalyzationNew.lib*. В разделе объявлений POU объявите массив:

SFCErrorAnalyzationTable : ARRAY [0..15] OF ExpressionResult.

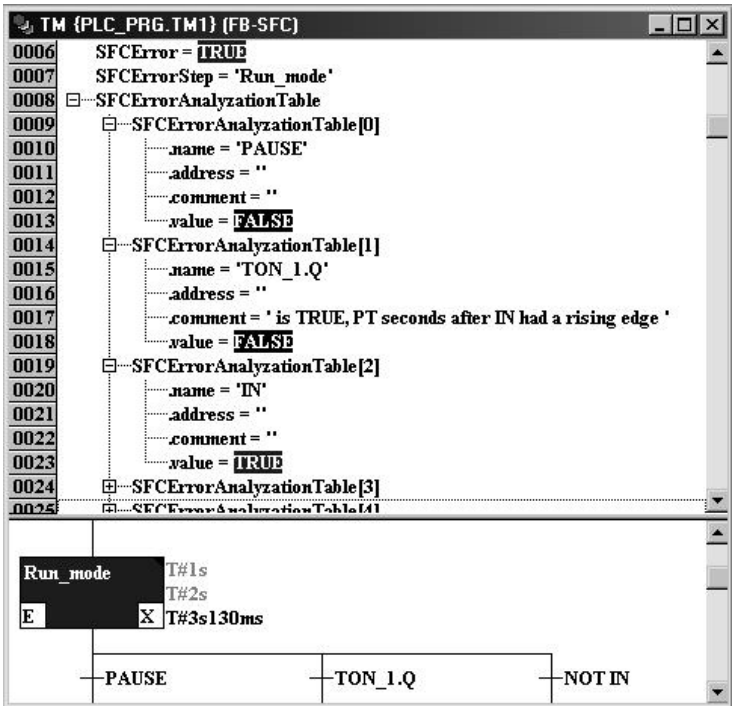

Рис. 1. Таблица анализа условного выражения в отладчике

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

 © НАУЧТЕХИЗДАТ 2007г.

Это массив структур типа ExpressionResult. Структура включает: название переменной, прямой адрес (если она размещена в областях входов, выходов или маркированной памяти), комментарий (если он есть) и логическое значение переменной на момент фиксации ошибки. Каждый элемент массива соответствует одному элементу условного выражения перехода и может включать до 16 таких элементов. Если диаграмма разветвляется по альтернативным условиям, то массив будет включать все условия слева направо.

На рис. 1 показан пример использования таблицы анализа условного выражения. В данном случае шаг Run_mode проработал более установленных ему 2 с и вызвал ошибку. В таблице видны имена и значения трех переменных, образующих условие перехода данного шага.

Безусловно, все вышеописанные флаги доступны программно. Это позволяет организовать в ней контроль работы критичных SFC шагов. Значения в таблице выражений фиксируются в момент возникновения ошибки и хранятся в ней вплоть до принудительного сброса обнулением флага SFCError. Так мы можем добавить в проект страничку визуализации, которая будет отображать все необходимые нам сведения для анализа работы системы. При этом нам не нужно запускать CoDeSys, мы можем воспользоваться панелью контроллера, web-визуализацией либо открыть доступ к этим данным из SCADA-системы верхнего уровня.

Библиотека *AnalyzationNew.lib* реализована в недрах системы исполнения, редактировать ее мы не можем, в некоторых типах контроллеров она может не поддерживаться. При работе с SFC диаграммами нам достаточно только включить ее проект, входящие в нее POU будут вызваться системой неявно.

Возможно, методика работы флагами SFC показалась вам несколько сложной и даже ненужной. Действительно, она относится к специальным расширенным возможностям, предназначенным для весьма опытных пользователей. Составляя программу на SFC, постарайтесь забыть о существовании флагов. Это дополнительный инструмент для локализации сбоев в системе, как осциллограф или графическая трассировка, а не средство программирования.

Полезный совет. Часто возникает необходимость иметь несколько наборов или модификаций POU для разных режимов работы. Например, с целью отладки. Если отладочные изменения небольшие, то в текстовых языках их можно просто заключать в комментарии. Конечно, делать это вручную неудобно. Есть универсальный прием, применимый в любых МЭК системах. Перед фрагментом отладочного текста ставим комментарий (*<*), в конце фрагмента: (*>*). Если возникнет необходимость отключить данные фрагменты, то это можно автоматизировать при помощи операции поиска и замены на (*< и >*) соответственно. В итоге, все фрагменты будут заключены в комментарии, и при необходимости можно будет провести обратную замену. В CoDeSys вы можете исключить ненужные компоненты из процесса компиляции, не удаляя их из проекта. Для этого в меню Project, Options нажмите кнопку Build Exclude object, выберите в дереве объектов отключаемые и поставьте галочку Exclude. Но удобнее другой способ. Перенесите изменяющиеся POU в отдельный проект, сделайте на его основе два или более проектов с наборами одноименных компонентов и сохраните их как внутренние (Save as … Internal library) библиотеки. Теперь достаточно просто подключить к основному проекту рабочую или отладочную библиотеку.

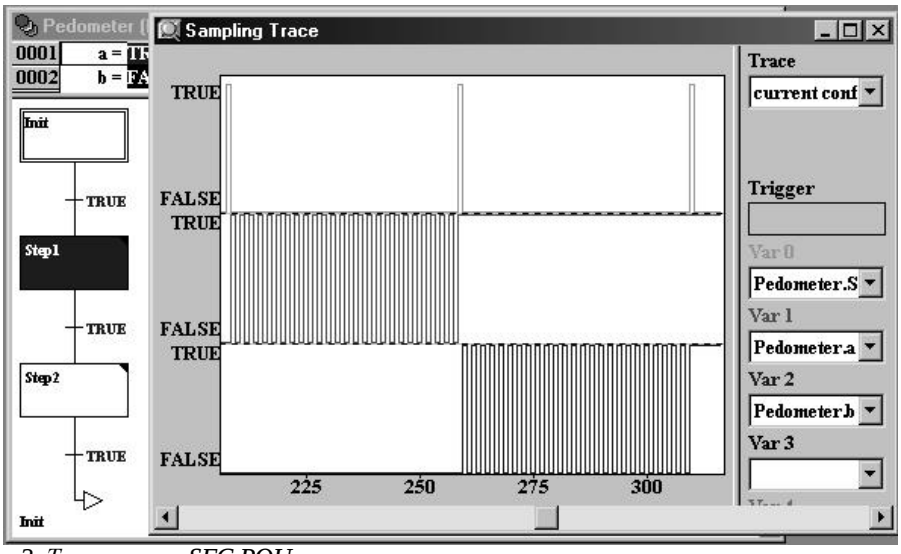

Рис. 2. Трассировка SFC POU, выполняемого по тактам

Синхронизация и выполнение SFC по тактам

Как мы уже обращали внимание, разрешение переходов происходит однократно в цикле вызова SFC компонента. Если мы не вызываем POU, то, естественно, его состояние не изменится, но и действия активных шагов вызываться не будут. Иногда как раз нужно бывает заморозить только переходы, чтобы все шаги и действия работали, но переходы происходили строго по команде. Иными словами, мы хотим заставить работать SFC компонент син-

хронно, как автомат, по тактам, задаваемым внешними импульсами. Это может пригодиться и для реализации режима наладки оборудования. Такой эффект можно сделать программно, если добавить в компонент детектор импульсов и включить его выход по “И” во все условия переходов. Безусловно, это не слишком красивое решение. На тот случай в CoDeSys есть специальные флаги: SFCTip и SFCTipMode. Первый служит входом тактовых импульсов, второй разрешает либо отключает режим выполнения по тактам. Как и прочие SFC флаги, они требуют объявления, оба имеют тип BOOL. Для доступа извне компонента их нужно объявить в разделе VAR_INPUT. Переходы происходят по фронту импульсов SFCTip.

На рис. 2 показан простой пример. Здесь SFC компонент (программа) имеет 2 шага, каждый из которых включает действие, генерирующее меандр на выходах “A” и “B”. Все условия переходов имеют значение TRUE, но компонент тактируется импульсами из основной программы PLC_PRG. Порядок работы компонента четко иллюстрируется графиками трассировки тактовых импульсов и выходов “A” и “B”.

Оптимизация SFC и вложенные диаграммы

Общее время работы SFC шага или действия задается условиями и классификаторами. Как минимум, это один цикл вызова POU, но чаще всего это время несоизмеримо больше. Говоря об оптимизации по времени SFC компонента, мы имеем в виду сокращение времени выполнения одного цикла его вызова. Очевидно, оно будет разным в зависимости от того, какие действия выполняются в данный момент. Для оптимизации нужно выявить наиболее сложные шаги и разбить их на несколько простых. В отличие от всех других МЭК языков в SFC за один цикл вызова POU выполняются только активные шаги и действия. Это уже само по себе способствует уменьшению времени цикла. Очевидно, есть смысл использовать большее число коротких действий, каждое из которых оптимизируется по правилам языка, на котором оно реализовано. С другой стороны, SFC исполнитель обязан в каждом цикле вызова выбирать все активные шаги диаграммы и проверять условия переходов. Кроме того, очень большие диаграммы тяжелы для восприятия. Поэтому не стоит забывать о возможности создания вложенных диаграмм. Если определенная ветка диаграммы выполняет некоторую автономную операцию, то есть смысл целиком вложить ее в одно действие.

В некоторых МЭК системах программирования действия SFC должны описываться на других языках. Чтобы в шаге использовать другой POU, на SFC приходится создавать отдельные программы (дочерние) и управлять ими посредством специальных флагов. В CoDeSys таких ограничений нет. Вы можете писать действия на SFC. Причем действия вложенной диаграммы тоже можно писать на SFC. Глубина вложений ограничивается только возможностями аппаратных средств контроллера (размер стека). Возможность вложений позволяет создавать компактные, красивые и эффективные диаграммы.

Измерение скорости работы POU

В завершение рассмотрим еще один пример функционального блока на SFC. Он позволяет измерять время выполнения любых POU. Для оценки быстродействия контроллеров принято указывать время выполнения 1000 команд, мы поступим подобным образом. Наш блок будет вычислять время тысячи вызовов указанного POU в миллисекундах. К сожалению, штатно в МЭК нет таймеров, позволяющих с высокой точностью измерять миллисекундные интервалы времени. Стандартные таймеры сообщают при очередном опросе о том, что заданное время вышло, но не более того. Поэтому мы поступим следующим образом: в первом шаге EmptCycle сделаем вызов 1000 пустых программ EMPT_PRG и подсчитаем, сколько таких циклов наш контроллер успеет сделать за 1 с. Обратите внимание на инструкции:

dwSumCnt := dwSumCnt +1;

IF dwSumCnt = 0 THEN bErr := TRUE; END_IF

Они осуществляют контроль переполнения счетчика. Если при очередном инкременте он будет обнулен, то взводится признак ошибки. Это дополнительная мера контроля корректности работы нашего блока.

Пустая программа написана на ST и состоит из единственной инструкции ';'. Следующий шаг называется SumCycle, он работает совершенно аналогично, но считает число вызовов программы TST_PRG, в которую мы будем вкладывать испытуемые POU или фрагменты алгоритмов. Полный пример решения показан на рис. 3. В качестве тестируемой мы используем функцию вычисления синуса.

Оба шага выполняются ровно по 1 с. Далее происходит переход на начальный шаг, в котором рассчитываются времена работы пустых циклов, циклов с вызовом тестовой программы и вычисляется разница, то есть чистое время ее выполнения. В нашем примере (рис. 3) время выполнения 1000 пустых циклов составило 0,10 мс, а 1000 операций вычисления синуса заняло 0,23 мс.

Время выполнения пустого рабочего цикла – достаточно показательная величина. Контроллер обязан опрашивать входы/выходы и выполнять другие системные задачи, на это нужно время. На первый взгляд, чем меньше рабочий цикл, тем лучше. Если отдать на его выполнение все свободное время процессора, то он получится плавающим. Если наша прикладная программа проводит некие громоздкие вычисления, то цикл соответственно увеличивается. Так сделано в большинстве ПЛК. Именно на этом эффекте работает описанный пример.

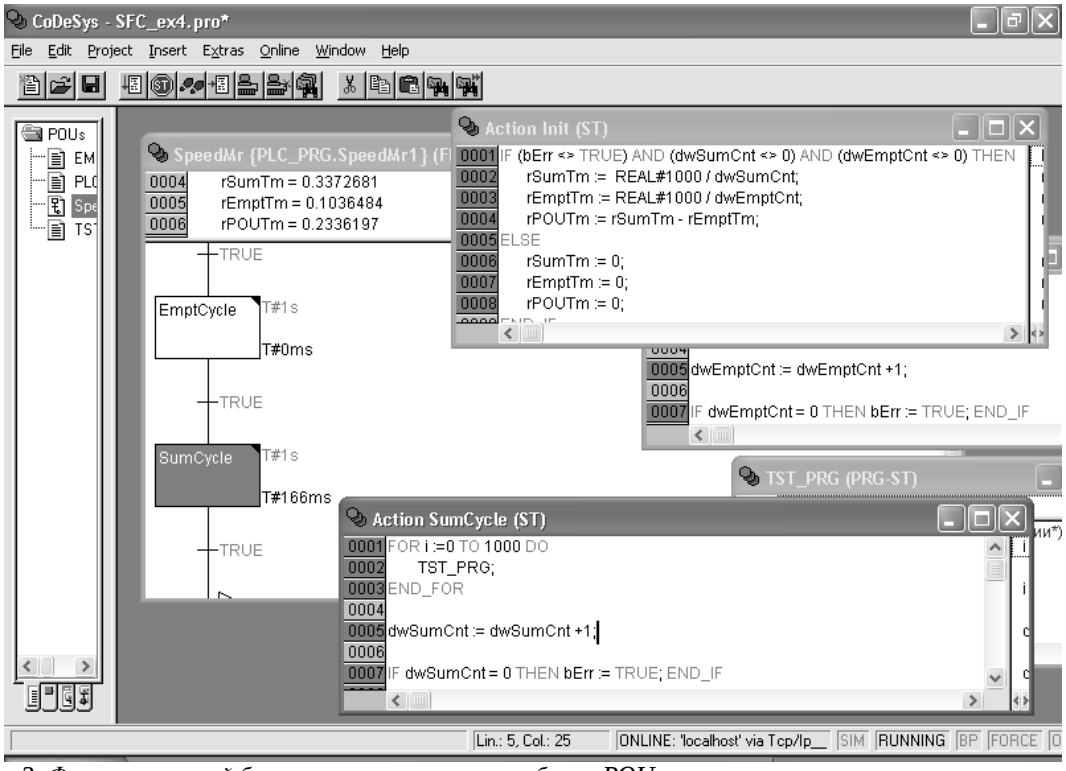

Рис. 3. Функциональный блок измерения времени работы POU

Но в некоторых контроллерах высшего класса с CoDeSys вы можете заметить странную вещь: изменение программы не меняет времени рабочего цикла. Независимо от того, выполняем ли мы сложные вычисления или простые быстрые операции, рабочий цикл всегда остается стабильным. Это сделано умышленно и позволяет получать гарантированное время реакции, не особенно задумываясь об оптимизации программы. Например, в SP RTE по умолчанию PLC_PRG вызывается ровно через 10 мс. Чтобы наш пример работал в этой системе, необходимо поместить его в свободно выполняемую (freewheeling) задачу, но об этом речь еще впереди.

Заключение

До сих пор мы считали, что программа в ПЛК всегда выполняется циклически. Это нельзя отнести к их уникальным свойствам. Цикличность лежит в основе любых цифровых систем. Вся хитрость только в том, чтобы измерять входные значения и вырабатывать выходные воздействия достаточно быстро. Можно подсчитать, насколько быстро это нужно делать. На этот вопрос дает ответ теорема Котельникова. В современных контроллерах используются достаточно быстродействующие процессоры, обеспечивающие этот минимум с большим запасом. Мы даже можем разделить процессорное время на доли, отдав их для выполнения нескольких задач. Если каждая из них будет удовлетворять минимальным требованиям по скорости, то можно законно говорить о параллельности выполнения. Мы можем просто взять SFC диаграмму, создать в ней параллельные ветви, вложить в их шаги другие (параллельные) программы, даже сделать главную программу, управляющую поведением подчиненных. Стоит ли тратить на это силы? Можно ли сразу запустить много задач? Можно ли в ПЛК обрабатывать очень быстрые события без всяких там циклов? Как минимум, давайте попробуем.

Продолжение следует.