Как избежать кошмара параллелизма в IoT: автоматы вместо корутин

Аннотация
В данной статье рассматривается библиотека для C++, предназначенная для реализации технологии параллельного автоматного программирования (АП), отвечающей концепции среды ВКПа, изложенной в [1]. Для полного понимания материала рекомендуется ознакомиться с основами теории АП, представленными в статьях [2, 3, 4], Взаимосвязь машины Тьюринга с конечными автоматами подробно рассмотрена в [5]. Вопросы применения корутин в контексте автоматного программирования анализируются в статьях [6-9]. Но в минимальном варианте достаточно общего представления о модели конечного автомата.

Цели работы:

1. Ознакомление разработчиков с универсальной технологией проектирования программного обеспечения.

2. Реализация технологии АП в виде библиотеки на C++ для микроконтроллеров ESP32, что позволяет применять передовые методы разработки, характерные для крупных платформ, в ресурсоограниченных средах.

Ключевые преимущества:

Объединение модульного, объектно-ориентированного и параллельного программирования в единую концепцию на основе строгой математической теории конечных автоматов кардинально меняет подход к проектированию, отладке и документированию ПО. Это устраняет зависимость от интуитивных и зачастую кустарных методик, характерных для традиционного embedded-программирования.

Без параллелизма не обойтись, но и с ним жизнь не предполагается безмятежной. Но тут, как «чертик из табакерки», выскакивают корутины, которые пышут оптимизмом. Но связи между верой в лучшее и знанием, как это произойдет, часто буквально никакой. Корутины ровно этого порядка. Тем не менее, в них есть нечто, что нам точно пригодится.

Корутины, как минимум, имитируют параллельную работу. Для этого нужно вручную расставить точки прерывания программных процессов. Далее, достигнув такой точки, процесс приостанавливает работу и передает управление другому процессу. Затем ему остается только надеяться, что когда-то он продолжит работу. Если все это происходит достаточно быстро, то и создается впечатление параллельной работы. Так организована работа в асинхронной программе.

Можно ли сделать корутинную модель программирования удобнее, проще и, главное, ~~истинно~~ параллельной? Это может быть ситуация, когда алгоритм содержит **обязательные** компоненты, которые можно считать точками прерывания. Как минимум, одна алгоритмическую модель, их содержащая, есть.

Но это не широко известная модель блок-схем (БС). Хотя можно представить множество БС, каждая из которых представляет корутину, а прерывания выполнять после каждой команды.

Реализовать подобное не так уж сложно, но будут весьма большие потери, связанные с переключением процессов. Оптимизация заключается в уменьшении числа прерываний. Так поступают в режиме многопоточности. Но поскольку они (прерывания) совершаются хаотично и в основном не тогда, когда это нужно, то возникает клубок проблем.

Расставляя точки прерывания, программист в какой-то мере микширует эти проблемы. Может, в этом и есть один из смыслов корутин? Но проблем даже в этом случае не становится меньше. А поскольку процесс их разрешения достаточно интуитивен, то это не способствует созданию качественного надежного программного кода.

Конечно-автоматные корутины

Однако, есть модель, которая перекрывает одновременно возможности корутин и потоков. Это модель конечного автомата (КА). Программистам она известна давно. На ней основан, например, метод введения переменной состояния, предложенный Ашкрофтом и Манной еще в 70-х годах прошлого века[12]. Причем он входит в число наиболее эффективных методов проектирования программ.

Конечные автоматы могут и должны быть не менее популярны, чем блок-схемы. Хотя бы потому, что эти модели эквивалентны друг другу. Например, давно известна простая процедура перехода от любой блок-схемы к конечному автомату. Поэтому, даже не имея представления об автоматах, но овладев данной процедурой, можно легко освоить автоматное программирование (программирование на базе автоматной модели). Но, к сожалению, программисты вряд ли о ней не знают.

Так почему конечные автоматы? Потому ли, что они ближе к машине Тьюринга (МТ)? Не только. Исторически МТ была создана одновременно с машиной Поста (МП), но это не сказалось на массовости ее применения. МП проще в реализации не только на аппаратном уровне, но и на уровне языков программирования. В этом смысле МП как бы взяла верх над МТ.

Например, на любом современном языке программирования довольно просто запрограммировать фразу типа: «если случилось что-то, то сделать то-то». Выражению таких мыслей служит простой оператор IF-ELSE. Подобное мышление поощряют и другие управляющие операторы типа FOR, WHILE, SWITH и т.п. Но в таком разнообразии есть свой минус. Особенно, если вспомнить, что в категорию управляющих операторов попадает злополучный оператор GO TO.

Но, чем сложнее алгоритм, тем выигрышнее его автоматное представление. В этом убеждает уже упомянутый подход Ашкрофта-Манны. Параллелизм еще более изменяет отношение к модели блок-схем. Современный вектор такого движения далеко не в их пользу, а потому в «королевстве» БС все более и более тревожно.

То, что причиной упомянутых тревог является модель блок-схем, далеко не всеми осознается. И на то есть свои причины. Одна из них – мало рассматриваются конкурентные алгоритмические модели. И, заполняя этот пробел, мы далее этим и займемся.

Сопрограммы

Термин «корутины» ввел в оборот Мелвин Конвей в 1958 г. в статье "A Multiprocessor System Design". Первая их реализация появилась на языке COBOL спустя пять лет – в 1963 г. В целом же активная жизнь сопрограмм связана с 1960-1970-ми годами и языками Simula и Modula-2. Таким образом, процесс использования корутин имеет давнюю историю, и сейчас мы присутствуем при их ренессансе.

Итак, что же собой представляют корутины, которые советские инженеры назвали сопрограммами? На рис. 1. приведена картинка из книжки 80-х годов - Зелковиц М., Шоу А., Геннон Дж. «Принципы разработки программного обеспечения» [14]. Она более чем наглядно поясняет работу сопрограмм.

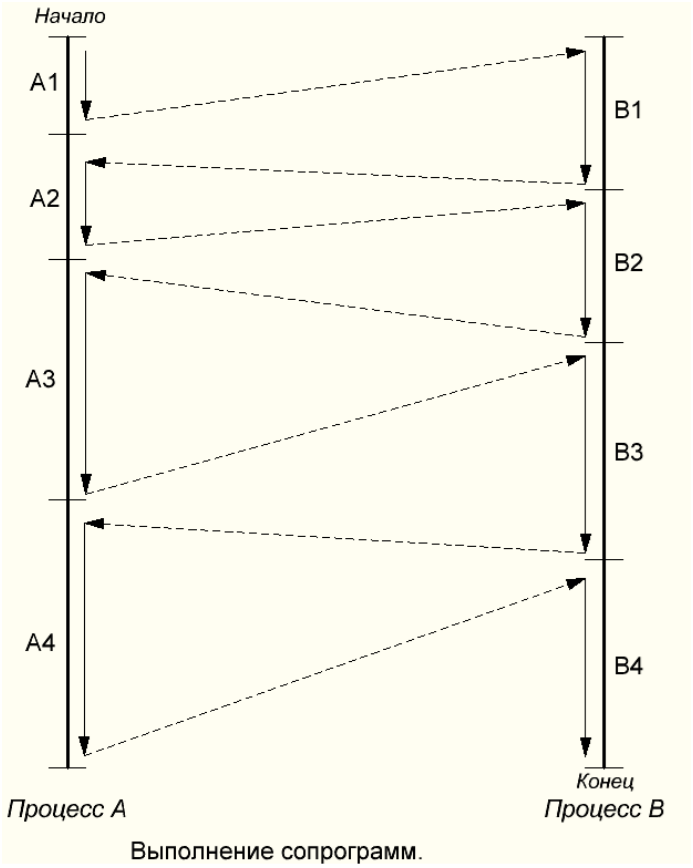

Рис.1. Сопрограммы

А если мы откроем «Толковый словарь по вычислительным системам» (под ред. Иллингоута и др.), переведенный на русский язык в 90-х годах прошлого века, то узнаем, что, во-первых, «сопрограммы, как правило, не встречаются в высокоуровневых языках», а, во-вторых, «представляют особый интерес, как средство моделирования параллельной работы в последовательных машинах» [15]. Но это отражение ситуации 90-х годов. Если бы авторы словаря только могли предположить, что ожидает высокоуровневые языки? Но с моделированием угадали, предсказав наше будущее.

Сопрограммы привлекают детским примитивизмом, но напрягают необходимость ручной расстановки точек прерывания. А можно ли это исправить? Ведь, сама идея сопрограмм удачна для реализации параллелизма. Пусть даже в режиме имитации. А когда-то, ведь, и выхода-то

другого не было. Только представьте - о многоядерности элементарно не знали, а потоки только-только зарождались! Т.е. крутись, как хочешь. Вот и докрутились до … корутин. Круг, как говорится, замкнулся в прямом и переносном смыслах.

Сторонний взгляд на модель конечного автомата выявляет, что его состояния могут быть точками прерывания. А после этого недалеко и до идеи управления автоматными потоками. Это может быть ядро, которое интерпретирует автоматную форму и при смене состояний ему доверяется управление программными потоками.

В идеале автоматный язык должен соответствовать формальному определению модели. Он должен быть эффективно реализуемым и иметь иерархическую (вложенную) модель для реализации аналога программных функций. Заметим, что в модели КА с подобной иерархией есть проблемы, которые необходимо обязательно решить. В нашем случае они были решены (см., например, [2]).

Кандидат, обладающий подобными качествами, на момент создания сопрограмм уже был. Это таблицы решений (ТР). В книге Э.Хамби «Программирование таблиц решений» они описаны весьма подробно [13]. Поэтому не так уж сложно догадаться, что ТР это те же автоматы, но без состояний. Или, другими словами, любая таблица решений – это автомат с одним состоянием, по которое просто забыли.

Таким образом, налицо связь между таблицами решений и табличной формой представления автоматных процессов. Отметим также один замечательный факт: **одна** табличная форма КА заменяет **все** разнообразие управляющих операторов любого языка программирования. Таблицы допускают также весьма эффективную не только программную, но и аппаратную реализацию (например, использование ассоциативной памяти). У какой модели вы еще такое видели?!

Внедряя автоматы, удобно изменить представление о структуре программы, разделив ее на три части – память, управление и операторы, где операторы разделить на предикаты и действия. Первые – функции, предназначенные делать любой анализ. Они возвращают булевское значение. Вторые – делают конкретную работу, не возвращая значений. Кстати, так или примерно так поступают при реализации ТР.

Довести идею автоматных корутин до рабочего состояние поможет объектно-ориентированное программирование (ООП). В этом случае каждый поток представляется объектом, у которого предикаты и действия – суть его методы, а свойства объекта – память автоматной программы. Остается только внедрить в тело объекта управление в форме таблицы переходов (ТП) автомата. Собственно это и есть описание того задания, которое он должен реализовать. Или по-другому – интерпретировать.

Есть от автоматного программирования польза и объектам - у них появляется поведение. Подобные объекты называются акторами. Только теперь в основе их поведения лежит строгая математическая модель. Такие объекты мы будем назвать программными автоматами, автоматными процессами или, когда это понятно из контекста, просто автоматами. Понимая при этом, что такой объект - это программа, представленная активным программным объектом, имеющим управление в форме КА.

Таким образом, можно представить, как должно измениться мышление программиста, использующего автоматы. В рамках технологии АП он оперирует строгими математическими

понятиями, где отдельный процесс (а по существу алгоритм его работы) конечный автомат, а их множество – сеть взаимодействующих автоматов. Из этого следует, что применение теории конечных автоматов (ТКА), весьма востребованной в практике проектировании цифровых схем, принесет огромную пользу и программированию.

Процесс внедрения какой-либо концепции в иную для нее область приложения представляет часто долгую, а для кого-то, порой, даже мучительную процедуру. Здесь нужно будет не просто переучиться, а изменить свое мышление. Подобная «ломка» происходит, как правило, при переходе от обычного программирования к объектному. Детали подобного процесса перехода на технологию автоматного программирования (АП) рассмотрим на конкретном примере.

Автоматные корутины на микророцессоре

Чем привлекает микроконтроллер? Своей доступностью, а в последнее время и возможностями. С появлением серии ESP32 у микроконтроллеров появился полноценный C++ (даже с библиотекой STL). Теперь легко заимствовать созданный для «больших» ПК код. На переносе ядра среды визуального проектирования автоматных программ - ВКПа все это успешно было проверено.

В результате была создана библиотека для ESP32. Она позволяет без проблем переносить на микроконтроллер автоматы, ранее созданные на C++ на ПК. Более того, теперь можно саму разработку и отладку вести на ПК (хотя бы в Qt Creator или Microsoft Visio, где была реализована среда ВКПа). А это многократно эффективнее, чем проектирование в том же редакторе VS Code. При этом финишную доводку кода доверять уже микроконтроллеру.

Итак, предположим, необходим процесс, регистрирующий три уровня жидкости. Датчики, регистрирующие их, подключены к входам (пинам) микроконтроллера с номерами 18, 19, 21 (пример взят из рабочего проекта). Процедуру чтения входов и отображения их значений отражает листинг 1. На листинге 2 показана установка режима входов микроконтроллера – в функции setup() и организация периодического опрос датчика – в функции loop(). Функция setup() выполняется только раз при запуске микроконтроллера, а потом постоянно «крутится» loop(), которая содержит периодический опрос датчика (один раз в секунду). Это пример типичного проектирования программ для микроконтроллера.

```
#include <Arduino.h>
// Номера входных пинов датчика
const int gpioLevel1{18};
const int gpioLevel2{19};
const int gpioLevel3{21};
// текущее состояние входов
char levelStatus1{0x0};
char levelStatus2{0x0};
char levelStatus3{0x0};
// Чтение/отображение состояния датчика уровня
void LevelView()
{
  char gpioNewStatus1 = digitalRead(gpioLevel1);
  if (levelStatus1 != gpioNewStatus1) { levelStatus1 = gpioNewStatus1; };
  char gpioNewStatus2 = digitalRead(gpioLevel2);
  if (levelStatus2 != gpioNewStatus2) { levelStatus2 = gpioNewStatus2; };
  char gpioNewStatus3 = digitalRead(gpioLevel3);
  if (levelStatus3 != gpioNewStatus3) { levelStatus3 = gpioNewStatus3; };
```

```
}
```

Листинг 1. Опрос и отображения состояния входов датчиа уровня

```
#include <Arduino.h>

void setup() {
  // Инициализация последовательного порта
  Serial.begin(115200);
// конфигурирование пинов датчиков уровней
  pinMode(18, INPUT); pinMode(19, INPUT); pinMode(21, INPUT);
}

unsigned long lastUpdateTimeLevel = 0;
const long updateIntervalLevel = 1000;
void loop() {
  if (millis() - lastUpdateTimeLevel >= updateIntervalLevel) {
    lastUpdateTimeLevel = millis();
    // читаем входы датчика уровня
    void LevelView();
    LevelView();
  }
}
```

Листинг 2. Установка режимов входов и цикл их опроса

Поскольку мы используем редактор VSCode и платформу PlatformIO, то нужен файл инициализации проекта – platformio.ini, который определяет 1) текущую платформу, 2) тип платы, 3) параметры порта для подключения микроконтроллера 4) подключаемые библиотеки и другие параметры проекта. Его вид приведен на листинге 3.

```
 [platformio]
default_envs = esp32dev

[env:esp32dev]
platform = espressif32
board = esp32dev
framework = arduino
monitor_speed = 115200
```

Листинг 3. Файл инициализации проекта

Вот фактически весь проект. Его мы далее будем называть исходным.

Преобразуем проект к автоматному виду. Для этого нам необходимо: 1) подключить библиотеку, интерпретирующую автоматы, 2) создать и подключить к ядру автомат, реализующий работу с датчиком. Подключение библиотеки к проекту демонстрирует листинг 4, где курсивом выделено дополнение к исходному варианту (ср. с листингом 3).

```
 [platformio]
default_envs = esp32dev

[env]
lib_extra_dirs = ./lib/VCPa
[env:esp32dev]
```

```
platform = espressif32
board = esp32dev
framework = arduino
monitor_speed = 115200
```

Листинг 4. Подключение библиотеки к проекту

Рассмотрим превращение обычной функции (см. выше `LevelView`) в автомат. Но прежде создадим файл, определяющий текущую конфигурацию проекта. Он показан на листинге 5. Здесь константа IF_SENSOR_SRC формирует исходный проект (см. выше), константа IF_USING_FSM подключает код автоматного ядра, IF_SENSOR_1 подключает 1-й датчик, а при отсутствии IF_USING_FSM создает код исходного проекта. Определение константа IF_SENSOR_2 создает еще автоматный процесс, аналогичный первому автомату. Все это сделано с целью оценки объема кода проекта в разных конфигурациях.

```
#define IF_USING_FSM      // автоматный проект
#define IF_SENSOR_1       // датчик 1
#define IF_SENSOR_2       // датчик 2
```

Листинг 5. Файл конфигурации проекта

Граф автомата для работы с датчиком уровня приведен на рис. 2. Создан он в Microsoft Visio и служит примером документирования автоматов. На его примере также видно отличие «автоматного мышления» от обычного. Если в исходном проекте мы создавали функцию обслуживания датчика, то в рамках АП мы сначала проектируем процесс в форме автомата.

Программный автомат – это процесс с заданным поведением. Здесь он анализирует величину задержки и исполняет функции опроса входов микроконтроллера. Он создает, если задано, задержку между опросами. И все это – процесс и его задержка реализуется автоматным ядром.

Рис.2. Автоматная модель управления датчиком уровня

Коды заголовка автоматного класса и его реализация приведены соответственно на листингах 6 и 7. Здесь курсивом выделено то, что отличает новую программу от ее исходного варианта, а подчеркиваем – то, что относится к специфике библиотеки.

Данный объект инкапсулирует свойства и поведение автоматного процесса. В этом и есть качественное отличие объектного автоматного подхода от классического.

В программировании микроконтроллеров часто избегают использования объектов. Возможно, потому, что программированием занимаются те, кто лучше разбирается в железе и им ближе по духу простой Си? Другой причиной может быть желание ужать код. И, конечно, отсутствие поведения у классических объектов также не способствует популярности объектов у тех, кто много внимания уделяет «умным» алгоритмам и увлечен идеями IoT.

```
#ifndef __SENSORLEVEL_H
#define __SENSORLEVEL_H

#if defined(IF_USING_FSM)
#include <Arduino.h>
#include "lfsaappl.h"
class TAppProcesses;
class SensorLevel:public LFsaAppl
{
public:
    void LevelView();
    SensorLevel(string strNam = "SENSORLEVEL");
    ~SensorLevel() {};
    // Номера выводов для цифровых входов сенсора
    const int gpioLevel1{18};
    const int gpioLevel2{19};
    const int gpioLevel3{21};
    int nDelay{0};
protected:
    int x1(); int x2();
    void y1(); void y2(); void y3(); void y4();
    char levelStatus1{0x0};
    char levelStatus2{0x0};
    char levelStatus3{0x0};
    bool bIfViewError{false};
};
#endif
#endif // __SENSORLEVEL_H
```

Листинг 6. Файл заголовка автомата датчика уровня

```
#include "Config.h"
#include <Arduino.h>
#if defined(IF_USING_FSM)
#include "stdafx.h"
#include "SensorLevel.h"
LArc TBL_SENSORLEVEL[] = {
  LArc("ss",    "ss","^x1^x2",  "y1"),
  LArc("ss",    "s1","x1",      "y1"),
  LArc("ss",    "er","x2",      "y3"),
  LArc("s1",    "ss","--",      "y2"),
  LArc("er",    "er","--",      "y4"),
  LArc()
};
SensorLevel::SensorLevel(string strNam):
  LFsaAppl(TBL_SENSORLEVEL, strNam)
```

```
{
// конфигурирование пинов датчика уровня
  pinMode(gpioLevel1, INPUT);
  pinMode(gpioLevel2, INPUT);
  pinMode(gpioLevel3, INPUT);
  bIfViewError = false;
}
int SensorLevel::x1() { return nDelay > 0; }
int SensorLevel::x2() { return nDelay < 0; }
// чтение и отображения состояния входов датчика
void SensorLevel::y1() { LevelView(); }
// задержка
void SensorLevel::y2() { FCreateDelay(nDelay); }
// ошибка в задании задержки
void SensorLevel::y3() { Serial.printf("%s(%s):error nDelay=%d\n",
FGetNameVarFSA().c_str(), FGetState().c_str(), nDelay); }
void SensorLevel::y4() {
  if (!bIfViewError) {
Serial.printf("%s(%s)\n", FGetNameVarFSA().c_str(), FGetState().c_str());
bIfViewError = true; }
}
// Отображение состояния датчика уровня
void SensorLevel::LevelView()
{
  char gpioNewStatus1 = digitalRead(gpioLevel1);
  if (levelStatus1 != gpioNewStatus1) { levelStatus1 = gpioNewStatus1; };
  char gpioNewStatus2 = digitalRead(gpioLevel2);
  if (levelStatus2 != gpioNewStatus2) { levelStatus2 = gpioNewStatus2; };
  char gpioNewStatus3 = digitalRead(gpioLevel3);
  if (levelStatus3 != gpioNewStatus3) { levelStatus3 = gpioNewStatus3; };
}
#else
#if defined(IF_SENSOR_1)
// Номера входных пинов датчика
const int gpioLevel1{18};
const int gpioLevel2{19};
const int gpioLevel3{21};
// текущее состояние входов
char levelStatus1{0x0};
char levelStatus2{0x0};
char levelStatus3{0x0};
// Отображение состояния датчика уровня
void LevelView()
{
  char gpioNewStatus1 = digitalRead(gpioLevel1);
  if (levelStatus1 != gpioNewStatus1) { levelStatus1 = gpioNewStatus1; };
  char gpioNewStatus2 = digitalRead(gpioLevel2);
  if (levelStatus2 != gpioNewStatus2) { levelStatus2 = gpioNewStatus2; };
  char gpioNewStatus3 = digitalRead(gpioLevel3);
  if (levelStatus3 != gpioNewStatus3) { levelStatus3 = gpioNewStatus3; };
}
#endif
#endif
```

Листинг 7. Файл реализации автомата процесса оценки уровня жикости

Напомним, что автоматный код будет создан при определении константы IF_USING_FSM, но если будет определена константа IF_SENSOR_SRC, то будет создан код исходного проекта (см. выше листинг 1). Если не будет определена ни одна из этих констант, то файлы будут фактически пусты.

Созданный автоматный код мало отличается от кода обычного классического объекта. Подчеркнутого текста, относящегося к автомату, совсем немного. В рамках создания автомата определен базовый класс – LFsaAppl, которому передается ссылка на таблицу переходов автомата - TBL_SENSORLEVEL. У объекта есть методы с определенными именами, которые ассоциируются с именами в данной таблице. Во всем этом чего-то необычного для ООП нет.

Как все это работает? Имеется ядро в форме библиотеки, которому передается информация об автоматном объекте. За создание ссылок к ядру и реализацию дискретного времени, отвечает код, представленный на листинге 8. Здесь код, относящийся к автоматному ядру (он подчеркнут), расположен между условными операторами с константой IF_USING_FSM.

```
#include "Config.h"
#include <Arduino.h>
#if defined(IF_USING_FSM)
#include "TAppProcesses.h"        // ядро FSM
#include "SetVarSetting.h"
#include "netfsa.h"
extern TAppProcesses *pTAppProcesses;
#endif
void setup() {
  // Инициализация последовательного порта
  Serial.begin(115200); Serial.println();
  Serial.println("Demo project - SensorLevelFSM");
#if defined(IF_USING_FSM)
// создаем среду для автоматов
  TAppProcesses* VCPaCore init();
  VCPaCore init();                // создание ядра FSM
  void LoadingFsaProsesses();
// загружаем автоматные процессы
  LoadingFsaProsesses();
  string str = "Setting1";
  CVarSetting *pViewVar = pTAppProcesses->pSetVarSetting->GetAddressVar(str);
  if (pViewVar) { pViewVar->dDeltaTime = 2; }
#endif
}
void loop() {
#if defined(IF_USING_FSM)
// моделирование дискретного времени FSM
  void VCPaCore TimerEvent(int nId);
  VCPaCore TimerEvent(0);
#else
#if defined(IF_SENSOR_SRC)
unsigned long lastUpdateTimeLevel = 0;
const long updateIntervalLevel = 1000;
  if (millis() - lastUpdateTimeLevel >= updateIntervalLevel) {
    lastUpdateTimeLevel = millis();
    // читаем входы датчика уровня
    void LevelView();
    LevelView();
  }
```

```
#endif
#endif
}
```

Листинг 8. Код реализации ядра и дискретного времени автоматов на ESP32

Создание автоматных объектов происходит в фунгции LoadingFsaProcesses(). Ее код приведен в листинге 9. Здесь може быть создано два автоматных процесса. При этом втрой процесс – копия первого. Но работать они могут по-разному. Например, первый датчик имеет задержку – 500 мсек, а второму задано -1 (см. листинг 9), что равносильно ее отсутствию (см. граф автомата). Поэтому 2-й автомат, обнаружив отрицательную задержку, выдаст сообщение об ошибке.

```
#include "Config.h"
#if defined(IF_USING_FSM)
#include "TAppProcesses.h"
extern TAppProcesses *pTAppProcesses;
#include "SensorLevel.h"
#if defined(IF_SENSOR_1)
SensorLevel *pSensorLevel{nullptr};              // ссылка на процесс 1
#endif
#if defined(IF_SENSOR_2)
SensorLevel *pSensorLevel2{nullptr};             // ссылка на процесс 2
#endif
void LoadingFsaProsesses() {
#if defined(IF_SENSOR_1)
// 0. SensorLevel
    pSensorLevel = new SensorLevel("sensor1");  // создание объекта процесса
    pTAppProcesses->arLibraryFsa[0] = pSensorLevel;// передача ссылки на
процесс ядру FSM
    string vFSA1 = "sensor1;0;0;0;0;InitFsaWorld;1;1";
    static CVarFSA var1(pTAppProcesses,vFSA1);  // описатель процесса
    bool bRet1 = var1.LoadFsa(pTAppProcesses->arLibraryFsa[0]); // загрузка
процесса
    if (bRet1) Serial.println("Load sensor1");
// устанавливаем задержки датчикам уровня
    pSensorLevel->nDelay = 500;   // задержка датчика 1
#endif
#if defined(IF_SENSOR_2)
//    pSensorLevel->FSetSleep(100);              // установка длительности
такта процесса
// 1. SensorLevel
    pSensorLevel2 = new SensorLevel("sensor2"); // создание объекта процесса
    pTAppProcesses->arLibraryFsa[1] = pSensorLevel2;// передача ссылки на
процесс ядру FSM
    string vFSA2 = "sensor2;0;0;0;0;InitFsaWorld;1;1";
    static CVarFSA var2(pTAppProcesses,vFSA2);   // описатель процесса
    bool bRet2 = var2.LoadFsa(pTAppProcesses->arLibraryFsa[1]); // загрузка
процесса
    if (bRet2) Serial.println("Load sensor2");
// устанавливаем задержки датчикам уровня
    pSensorLevel2->nDelay = -1;   // задержка датчика 2
#endif
}
#endif
```

Листинг 9. Код подключения автоматных объектов к автоматному ядру

В заключении осталось озвучить размер библиотеки. Для микроконтроллера ESP32 с объемом памяти 4Mb он будет чуть более 24%. Заметный, но не запредельный объем, если сравнивать его с наиболее объемными библиотеками ESP32. Тем не менее, практика убеждает, чтобы лучше использовать конфигурации ESP32 с большим объемом оперативной памяти.

А что касается приведенного выше кода примеров (исходного и автоматного), то это полный код двух проектов, которые можно попробовать «тут и сейчас» в рамках редактора VSCode в связке с PlatformIO и C++.

Выводы

Реализацию автоматного ядра может быть другой. На базе тех же потоков, где каждый поток представлен автоматом. Но параллелизм в одном потоке лучше. В этом случае у вас не будет проблем с отладкой, синхронизацией, гонками, производительностью и т.д. и т.п.

Но обязательно найдется кто-то, кто возразит по поводу связи одного потока с параллелизмом. Но ее и не должно быть, т.к. вопрос параллелизма целиком лежит во власти алгоритмической модели. Если модель параллельная и корректно реализована, то параллельны и представленные ею процессы. Таков простой, краткий и, как представляется, ясный ответ, рассеивающий подобные сомнения.

Современные микроконтроллеры и ПК хорошо дополняют друг друга. Возникает уникальная ситуация, когда среду проектирования для микроконтроллеров можно использовать лишь на финишном этапе проектирования, а разработку вести на ПК. Именно в среде Qt Creator была создана автоматная библиотека для ESP32. И только после полной уверенности в работоспособности она была перенесена на микроконтроллер, где и прошла окончательную доводку. Это позволило многократно ускорить процесс проектирования и, если честно, вообще осуществить сам перенос.

Необходимо упомянуть характеристики реального проекта на базе ESP32, из которого заимствован рассмотренный выше пример. Он содержит кроме датчиков уровня жидкости датчики температуры, влажности, освещенности, включая их цифровые фильтры, несколько реле и т.д. и т.п. Каждый из них представлен автоматом, которые работают в жестком реальном времени (дискретность такта 10 мсек). Проект поддерживает подключение по WiFi, MQTT, Home Assistant и имеет свою web-страницу. При этом, как уже сказано выше, не использует среду FreeRTOS, к которой стандартно прибегают для создания множества процессов.

С учетом сторонних библиотек, библиотеки автоматов и прикладных процессов загрузка памяти на базе микроконтроллера ESP32-WROOM-32 приближается к 100%, где библиотеки занимают почти 85% из 4Mb памяти (библиотека автоматов занимает, напомним, 24%) Поэтому предполагается переход на микропроцессор ESP-32-S3-WROOM-1, который имеет «на борту» 16Mb. А этого уже достаточно для реализации гораздо более масштабных проектов в рамках технологии АП.

Вариант устройства на базе ESP32l, созданный по канонам АП, уже доступен на маркетплейсе OZON. При его реализации технология АП подтвердила еще раз свою эффективность при решении задач, объединяющих множество параллельно и исполняемых в жестком реальном времени процессов, но уже на базе микроконтроллеров.

Литература

1. ВКПа. Введение, ч.1. Визуальное проектирование автоматов. https://habr.com/ru/articles/794498/
2. Автоматная модель управления программ. https://habr.com/ru/articles/484588/
3. Модель параллельных вычислений. https://habr.com/ru/articles/486622/
4. Автоматное программирование: определение, модель, реализация. https://habr.com/ru/articles/682422/
5. Машина Тьюринга, как модель автоматных программ. https://habr.com/ru/articles/481998/
6. Параллелизм, корутины, событийные автоматы,… живая математика. https://habr.com/ru/articles/499460/
7. Параллелизм и эффективность: Python vs FSM. https://habr.com/ru/articles/506604/
8. Мир без корутин. Итераторы-генераторы. https://habr.com/ru/articles/512348/
9. Мир без корутин. Костыли для программиста — asyncio. https://habr.com/ru/articles/513512/
10. Автоматное программирование в SimInTech и ВКПа. https://habr.com/ru/articles/717190/
11. Практика применения автоматов в ПЛК. https://habr.com/ru/articles/694078/
12. Йодан Э. Структурное проектирование и конструирование программ. М.: Мир,1979 - 415с
13. Хамби Э. Программирование таблиц решений. М.: Мир, 1976. – 86 с.
14. Зелковиц М., Шоу А., Геннон Дж. «Принципы разработки программного обеспечения». М.: Мир, 1982. –368 с.
15. Толковый словарь по вычислительным системам/Под ред. В. Иллингоута и др.: Пер. с англ. А.К. Белоцкого и др.; Под ред. Масловского. – М.: Машиностроение, 1991. – 560 с.