

Первоначальное знакомство с разработкой автокомпилируемых блоков в программном комплексе RDS

Обсуждается цель создания модуля автокомпиляции, облегчающего разработку собственных блоков, совместимых с программным комплексом RDS. Приводятся простейшие примеры создания шаблонов для решения ряда задач с помощью автокомпилируемых блоков. Рассматривается общая структура создаваемых модулем автокомпиляции файлов и обращается внимание на некоторые проблемы, возникающие в связи с разработкой автокомпилируемых блоков.

§1. Цель создания модуля автокомпиляции и блок-схема его взаимодействия с другими модулями программного комплекса RDS.

Приводятся требования к модулю автокомпиляции и блок-схема взаимодействия его основных файлов с модулями программного комплекса RDS.

Несмотря на наличие в RDS большого количества готовых блоков различного назначения, помогающих составлять блок-схемы систем управления, всегда существует потребность в разработке собственных блоков, модели которых отличаются от уже имеющихся в библиотеке. Действительно, невозможно заранее предусмотреть все разновидности блоков, необходимых для исследования поставленной задачи. Готовые блоки не всегда удовлетворяют пользователя, и ему хочется разработать собственную модель и включить ее в исследуемую систему.

Практически все современные программные продукты, позволяют подключать внешние, созданные пользователем модули. Они либо описываются внутри самой системы на каком-либо языке высокого уровня, либо подключаются как внешние исполняемые программы. Оба варианта имеют свои достоинства и недостатки. В специализированных языках описываются конструкции, специфические для каждой конкретной области. Они, как правило, уникальны и требуют внимательного изучения. Программные продукты, позволяющие описывать алгоритмы работы своих блоков на языках высокого уровня, могут использовать как специализированные языки, так и обычные универсальные языки программирования. Универсальные языки программирования более распространены, и многие пользователи предпочли бы пользоваться знакомыми языками программирования для описания моделей блоков. По этой причине в программном комплексе RDS было принято решение для создания собственных моделей блоков опираться на универсальные языки программирования (в частности, C++).

Разработка соответствующего программного модуля, названного модулем автокомпиляции, весьма трудоемкая задача. Хотелось, чтобы по фрагменту программы, задающему только алгоритм решения задачи, был автоматически построен файл DLL (динамически подключаемой библиотеки) модели, который не только органически вписался бы в комплекс RDS, но и обладал почти всеми свойствами профессионально создаваемых модулей. К этим свойствам относятся такие как блокировка неправильно введенных данных, выдача подсказок и предупреждений, определение синтаксических ошибок в программе с выдачей их списка на экран и многие другие. Все эти свойства важны, но они не имеют отношения к алгоритму работы блока RDS, поэтому желательно предоставлять разработчикам моделей возможность писать только те части программы, которые непосредственно описывают выполняемые блоком функции.

Чтобы разработчик, не будучи профессиональным программистом, смог, тем не менее, создавать собственные блоки, в комплекс инструментальных средств RDS были включены модули автоматической компиляции управляющих программ блоков. Такой модуль автоматически строит полный исходный текст программы по отдельным фрагментам, введенным разработчиком. Затем модуль вызывает один из внешних

компиляторов с выбранного языка программирования, который, обработав сформированный текст, создает файл блока для динамической библиотеки. Эта библиотека затем автоматически подключается к RDS и начинает обслуживать блоки системы.

Модуль автоматической компиляции должен:

1. Обеспечивать ввод и редактирование фрагментов исходного текста программы блока, отвечающих за его поведение и реакцию на события. Как минимум, должен обеспечиваться ввод функции расчета значений выходов блока при поступлении данных на его входы. Допустимо использовать для этого внешнюю программу-редактор.
2. Собирать из фрагментов текста, введенных разработчиком, полный исходный текст программы блока со всеми структурами данных и описаниями, необходимыми для доступа к собственным переменным блока, общим переменными системы и сервисным функциям комплекса.
3. Вызывать внешнюю программу-компилятор, которая преобразует исходный текст программы в исполняемый файл DLL.
4. Взаимодействуя с комплексом, отключать на время компиляции блоки RDS от обслуживающих библиотек. Если этого не сделать, файлы библиотек будут заблокированы, и компилятор не сможет заменить их новыми.

На рис.1 приведена блок-схема действий модуля автоматической компиляции в программном комплексе RDS.

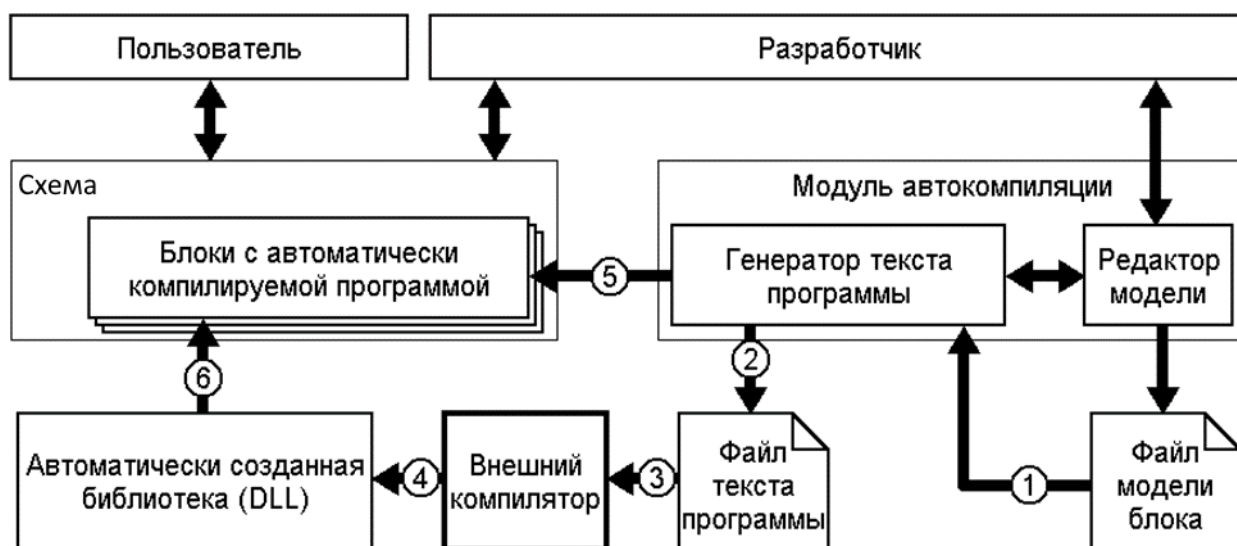


Рис.1. Блок-схема действий модуля автоматической компиляции

По схеме видно, что модуль автокомпиляции, взаимодействуя с фрагментом программы, описывающим алгоритм, создает **файл модели блока**. Далее модуль автокомпиляции автоматически создает полный файл (**файл текста программы**), который, используя внешний компилятор, переводит текст в исполняемый модуль DLL. Файл модели блока и исполняемый модуль сохраняются на компьютере соответственно с расширениями mdl и dll.

Файл модели блока является промежуточным файлом, который создается модулем автокомпиляции на основе данных, включенных в модуль блока пользователем. Этот файл формируется автоматически, и изменение его содержания вручную не рекомендуется. Он содержит ключевое слово, по которому модуль определяет файл модели, номер и дату версии файла во внутреннем формате, параметры и текст алгоритма, а также все параметры модели и окна редактора во внутреннем формате.

§2. Примеры создания автокомпилируемых блоков.

Приводятся три простых примера создания автокомпилируемых блоков, на которых прослеживаются этапы построения моделей и результаты их работы в системе.

§2.1 Построение автокомпилируемого блока для расчета уравнений кинетики нейтронной мощности ядерного реактора

Упрощенная модель расчета кинетики нейтронной мощности ядерного реактора описывается тремя уравнениями. Входом блока является реактивность RO , выходом мощность P .

$$\begin{aligned} P &= (0,74P_1 + 0,26P_2)/(1 - RO), \text{ if } |1 - RO| > 0.01, \\ 0.288 \frac{dP_1}{dt} + P_1 &= P, \\ 0.0263 \frac{dP_2}{dt} + P_2 &= P, \end{aligned}$$

где P – нейтронная мощность реактора в относительных единицах, P_1 и P_2 – мощности групп запаздывающих нейтронов, RO – реактивность. Процесс построения модели прост и не требует от исследователя высокой программистской квалификации. Важно лишь знание предмета и алгоритма расчета.

Пользователь сначала создает новый стандартный пустой блок под названием «Кинетика». Далее заполняет поля в левой части модели редактора: список участвующих в программе переменных. В правой части записывает алгоритм изменения координат на одном шаге расчета. Разностный вид уравнений после аппроксимации по методу Эйлера следующий:

$$\begin{aligned} &\text{Если } |1 - RO| > 0.01, \text{ то (оператор защиты от деления на 0):} \\ &P = (0,74P_1 + 0,26P_2)/(1 - RO), \\ &P_1 = P_1 + \Delta t * (P - P_1)/0.288, \\ &P_2 = P_2 + \Delta t * (P - P_2)/0.0263, \\ &P_1(0) = P_{10}, P_2 = P_{20} \quad (\text{начальные условия}) \end{aligned}$$

Текст программы модели приведен на рис. 2, где step – шаг расчета. Переменная DynTime обозначает время, которое задается во всех блоках, участвующих в моделировании переходных процессов систем. Модель написана в синтаксисе языка C++ и автоматически переводится в необходимый DLL-файл. Таким образом строится блок модели, который может быть подключен к работе системы управления.

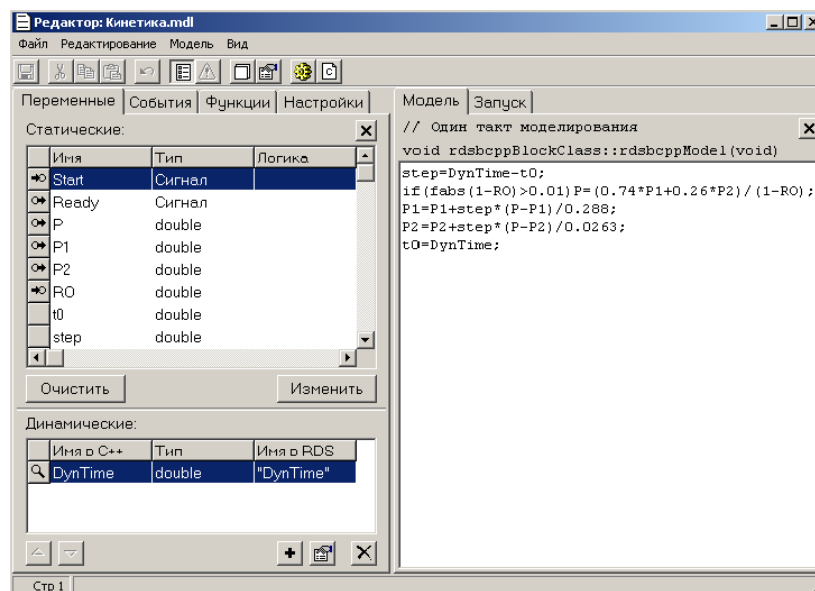


Рис.2. Разностная модель кинетики нейтронной мощности ядерного реактора

Точно также можно расписать алгоритм расчета произвольной системы дифференциальных уравнений, приведенных к виду Коши.

§2.2. Построение автокомпилируемого блока для отображения графика функции.

На практике довольно часто хочется получить графический вид функции, задаваемой аналитическим выражением или алгоритмически. Для этой цели в RDS нетрудно построить некоторый шаблон автокомпилируемого блока. Он представлен на рис.3. Здесь:

- x – значение аргумента функции;
- a и b – задают значения начала и конца области изменения аргумента;
- dx – шаг изменения аргумента;
- n – номер очередного шага;
- Fx – функция;

Последний оператор задает условие завершения расчета.

Алгоритм фактически не требует комментариев. Первые два оператора формируют значение аргумента функции, третий – само значение функции.

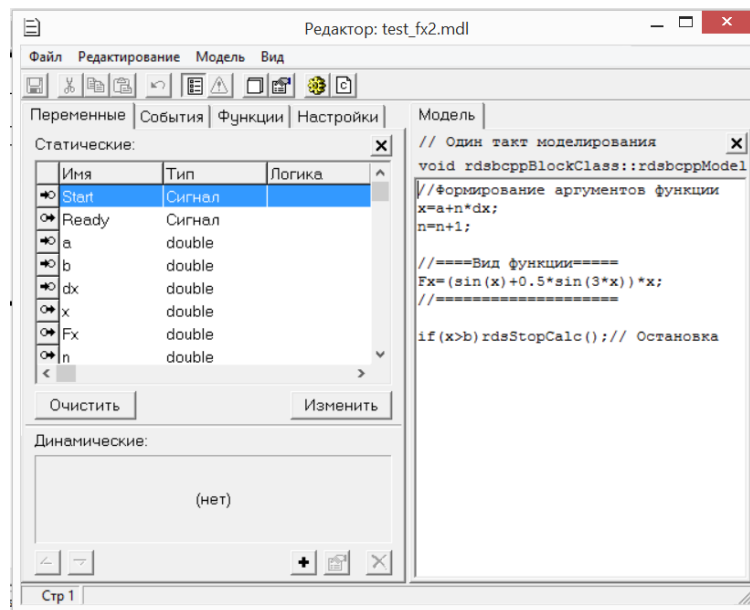


Рис.3. Шаблон автокомпилируемого блока для получения графика функции

На рис.4 показана подсистема, решающая задачу вывода графического вида функции.

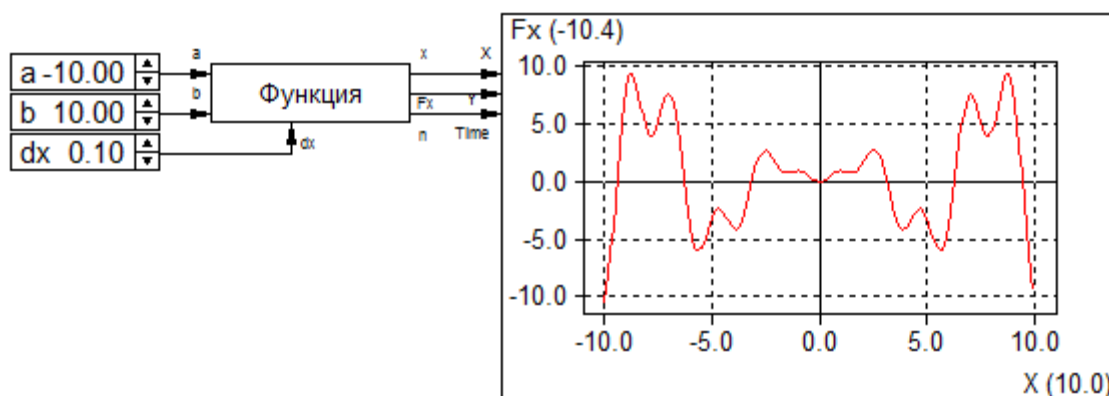


Рис.4. Результат работы схемы с функцией, заданной в блоке рис.3

Подсистема не меняется в случае, если необходимо получить график другой функции. Следует только задать новые значения a , b , dx и вставить в шаблон автокомпилируемого блока выражение новой исследуемой функции. Например, для функции

$$\begin{aligned}
 & \text{if}(x < 0.01 \ \&\& \ x \geq 0) \ x = 0.01; \\
 & \text{if}(x > -0.01 \ \&\& \ x \leq 0) \ x = -0.01; \\
 & Fx = \sin(x) + \frac{\sin(3 * x)}{x},
 \end{aligned}$$

где первые 2 оператора добавлены, чтобы исключить деление на 0, необходимо вместо функции в алгоритме на рис.2, подставить приведенные выше выражения. После запуска расчета получим график функции, представленный на рис. 5.

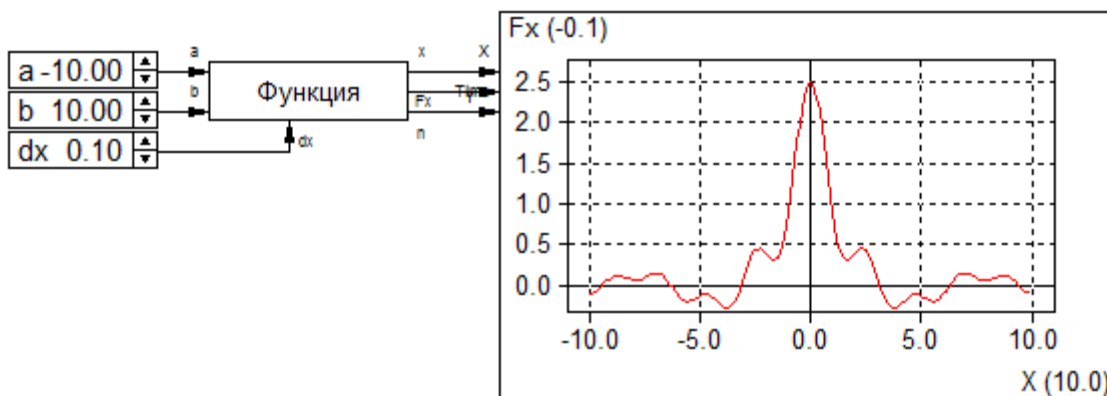


Рис.5. Результат работы схемы со вновь заданной функцией.

Очень важно, что в автокомпилируемых блоках алгоритм расчета доступен для анализа и изменения – ведь закрытость алгоритма часто вызывает сомнения в правильности выбранного блока для исследования задачи или в корректности его применения.

§2.3. Построение автокомпилируемого блока для вычисления определенного интеграла функции на отрезке $[a, b]$.

Для вычисления определенного интеграла создается шаблон автокомпилируемого блока, в который встроен алгоритм, представленный на рис.6. В нем:

- x – значение аргумента функции;
- a и b – задают значения начала и конца области изменения аргумента;
- dx – шаг изменения аргумента;
- Fx – функция;
- Int – значение интеграла;
- n – номер очередного шага.

Последний оператор задает условие завершения расчета.

В качестве конкретной функции в шаблоне была выбрана функция $\sin(x)$. Комментарии в тексте достаточно ясно характеризуют алгоритм (в примере применен самый простой алгоритм ступенчатой аппроксимации функции).

Интеграл вычислялся на отрезке $[0, \pi \approx 3.14]$. На рис. 7 показан вид системы после расчета интеграла.

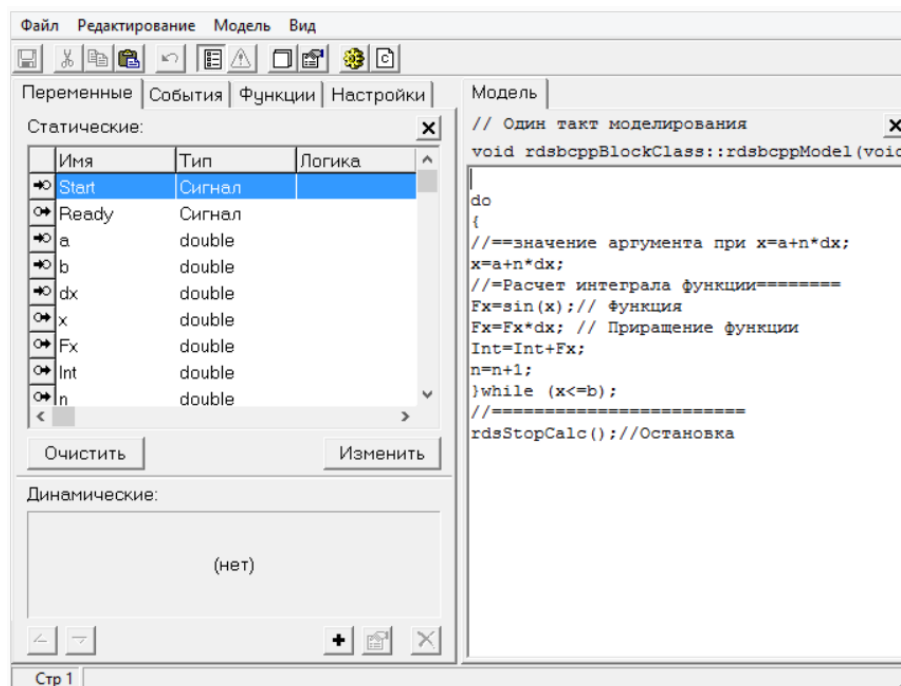


Рис.6. Шаблон автокомпилируемого блока для вычисления интеграла функции.

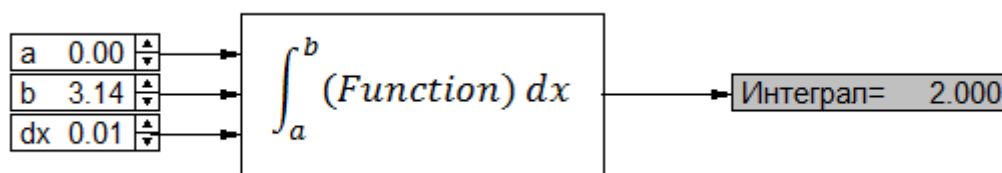


Рис.7. Результат вычисления интеграла функции $\sin(x)$ на отрезке $[0, \pi = 3.14]$.

§3. Этапы формирования автокомпилируемого блока и общее представление о структуре файла текста программы.

Дается общее представление о структуре файлов, создаваемые модулем автокомпиляции, на примере, рассмотренном в §2.2. Обращается внимание на некоторые проблемы, возникающие в связи с созданием и применением автокомпилируемых блоков.

Создание текста файла программы даже для самых простых автокомпилируемых блоков, типа приведенных в §2, требует осторожности. Необходимо предусмотреть массу малозаметных, но возможных ошибок при составлении и исполнении программы. Только опытный программист, не используя модуль автокомпиляции, мог бы подготовить такой файл. В этом параграфе, для примера, рассмотренного в §2.2, приводятся тексты файлов, создаваемых модулем автокомпиляции с краткими комментариями к отдельным разделам. Целью такого рассмотрения не является изучение их структуры (об этом подробно идет речь в описании пользователя), а только предварительное указание на трудности, с которыми столкнулся бы пользователь при отсутствии программы, создающей такой файл автоматически.

§3.1. Файл текста программы

Файл текста программы – это окончательно сформированный модулем автокомпиляции файл, который, используя внешний компилятор, переводит текст в исполняемый модуль DLL. Его структура более подробно рассмотрена в §3.6.1 описания пользователя.

```
//-----  
// Автоматически сформированный исходный текст для модели "test_fx2.mdl"  
// Модуль автокомпиляции: Borland C++ 5.5  
//-----  
#line 7 "-1"  
/* Директива «#line» используется модулем автокомпиляции для разбора сообщений об ошибках, выдаваемых компилятором. Директивы «#include» - для включения в текст программы стандартных файлов заголовков, в которых содержатся описания, необходимые для программ Windows и для использования функций стандартных библиотек языка C. Часть директив «#define» описывают специфичные для RDS константы, и директивы включения файлов заголовков RDS. */  
#include <windows.h>  
#include <stdlib.h>  
#include <math.h>  
#include <float.h>  
  
#define RDSBCPP_MODELNAME "test_fx2.mdl" // Имя модели для индикации  
  
#include <RdsDef.h>  
#define RDS_SERV_FUNC_BODY rdsbcppGetService  
#include <RdsFunc.h>  
#include <CommonBl.h>  
#include <CommonAC.hpp>  
  
/* Макросы обработки исключений используются в функции модели, если в параметрах модели будет установлен флажок «перехватывать все исключения, возникшие в модели» Они помогают отлавливать и перехватывать возникающие при исполнении программы ошибки. */  
// Обработка исключений включена  
#define RDSBCPP_EXCEPTIONS  
// Оператор try//  
#define RDSBCPP_TRY __try  
// Оператор catch  
#define RDSBCPP_CATCHALL __except (EXCEPTION_EXECUTE_HANDLER)  
//-----  
  
/* Объявляется весьма важная вещественная глобальная переменная double rdsbcppHugeDouble. В ней всегда хранится специальная константа HUGE_VAL из стандартных описаний языка C, в RDS обозначающая ошибку выполнения математической операции. Это значение может поступить на вход модели вместо вещественного числа, если блок, соединенный с этим входом, не смог выполнить какое-либо действие (например, в нем возникло переполнение или деление на ноль). */  
double rdsbcppHugeDouble;  
//-----  
  
/* Поскольку в параметрах модели по умолчанию установлен флажок «перехватывать ошибки математических функций», в текст программы вставлена функция обработки математических ошибок _matherr. Она устроена очень просто: при потере точности результат считается нулем, а при всех прочих ошибках в качестве результата операции возвращается значение-индикатор ошибки из глобальной переменной rdsbcppHugeDouble. */
```



```
//-----
int _matherr (struct _exception *a)
{ a->retval=(a->type==UNDERFLOW)?0.0:rdsbcppHugeDouble;
  return 1;
}
//-----

//-----
/* После функции перехвата математических ошибок располагается автоматически
формируемая функция DllEntryPoint – главная функция DLL, которую должна иметь
каждая динамическая библиотека Windows. Эта функция вызывается при загрузке
DLL в память RDS. Автоматически формируется функция, служащая для получения
доступа к сервисным функциям RDS. */
//-----
#pragma argsused
int WINAPI DllEntryPoint(HINSTANCE hinst,unsigned long reason,void
*lpReserved)
{ if(reason==DLL_PROCESS_ATTACH)
  { if(!RDS_SERV_FUNC_BODY())
    { if(rdsIncompatibleDll) rdsIncompatibleDll();
    }
    else // Есть доступ к сервисным функциям
    { rdsGetHugeDouble(&rdsbcppHugeDouble);
    }
  }
  return 1;
}
//-----
/* Сразу за главной функцией DLL следуют макросы вида RDSBCPP_*CLASS,
разворачивающиеся в описания классов для доступа к статическим и динамическим
переменным, использованным в модели блока.*/
//-----
// Переменная double ("D")
RDSBCPP_STATICPLAINCLASS(rdsbcstDouble,double);
// Переменная char ("S")
RDSBCPP_STATICPLAINCLASS(rdsbcstSignal,char);
//-----

//-----
// Класс блока
//-----
class rdsbcppBlockClass
{ public:
  // Структура данных блока
  RDS_PBLOCKDATA rdsbcppBlockData;
  // Статические переменные
  rdsbcstSignal Start;
  rdsbcstSignal Ready;
  rdsbcstDouble a;
  rdsbcstDouble b;
  rdsbcstDouble dx;
  rdsbcstDouble x;
  rdsbcstDouble Fx;
  rdsbcstDouble n;

  /* В класс блока автоматически вставляется функция, инициализирующая объекты
  доступа к переменным, в которой эти объекты настраиваются на конкретные
  переменные блока. Эта функция автоматически добавляется модулем перед вызовом
  функций реакции блока на любое событие. */
  void rdsbcppInitVars(void *base)
  {
```

```

    // Статические переменные
    Start.Init(base,0);
    Ready.Init(base,1);
    a.Init(base,2);
    b.Init(base,10);
    dx.Init(base,18);
    x.Init(base,26);
    Fx.Init(base,34);
    n.Init(base,42);
};

// Проверка существования динамических переменных
BOOL rdsbcppDynVarsOk(void)
{ return TRUE; };

// Функции реакции на события
void rdsbcppModel(void);

// Конструктор
rdsbcppBlockClass(RDS_PBLOCKDATA data);
// Деструктор
~rdsbcppBlockClass();
}; // class rdsbcppBlockClass
//-----
/* В тексте программы размещается экспортированная функция модели с именем
rdsbcppBlockEntryPoint (общая структура любой функции модели блока описана в
п.2.3 руководства программиста). Эта функция формируется полностью
автоматически и, фактически, представляет собой один большой оператор switch, в
метки case которого модуль автокомпиляции вставляет вызовы функций реакций на
события, сформированных для каждого введенного пользователем фрагмента
программы. Кроме того, реакции на некоторые события добавляются автоматически.
*/
//-----
#pragma argsused
extern "C" __declspec(dllexport) int RDSCALL rdsbcppBlockEntryPoint(
    int CallMode,           // Режим вызова (сообщение RDS_BFM_*)
    RDS_PBLOCKDATA BlockData, // Данные блока
    LPVOID ExtParam         // Дополнительные данные (зависят от CallMode)
)
{ int result=RDS_BFR_DONE; // Код возврата
  // Объект класса блока (хранится в личной области данных)
  rdsbcppBlockClass *data=(rdsbcppBlockClass*)(BlockData->BlockData);

  // Перехват ошибок математики - начало
  volatile unsigned int FPMask=_control87(0,0);
  _control87(MCW_EM,MCW_EM);

  switch(CallMode)
  { case RDS_BFM_INIT: // Инициализация блока
    BlockData->BlockData=(data=new rdsbcppBlockClass(BlockData));
    break;

    case RDS_BFM_CLEANUP: // Очистка (вызывается перед удалением)
    delete data;
    break;

    case RDS_BFM_VARCHHECK: // Проверка допустимости структуры переменных
    if(strcmp((char*)ExtParam,"{SSDDDDDD}")!=0)
        return RDS_BFR_BADVARSMSG;
    break;

```

```

        case RDS_BFM_MODEL: // Один такт моделирования
            data->rdsbcppInitVars(BlockData->VarData);
            data->rdsbcppModel();
            BlockData->BlockData=data;
            break;

    } // switch(CallMode)

    // Перехват ошибок математики - конец
    _clear87();
    _control87(FPMask,MCW_EM);

    return result;
}
//-----

// Конструктор класса блока
rdsbcppBlockClass::rdsbcppBlockClass(RDS_PBLOCKDATA data)
{ // Сохранение адреса структуры данных блока
    rdsbcppBlockData=data;
}
//-----

// Деструктор класса блока
rdsbcppBlockClass::~rdsbcppBlockClass()
{
}
//-----

/* Далее размещаются «тела» всех функций, автоматически построенных вокруг
кусков программы, введенных пользователем на вкладках редактора модели. */
//-----
// Один такт моделирования
void rdsbcppBlockClass::rdsbcppModel(void)
{
    #line 1 "0"
/* Начиная с этой точки, вставляется текст программы, введенной пользователем на
вкладке редактора модели – в данном случае, на вкладке «модель». */
    //Формирование аргументов функции
    x=a+n*dx;
    n=n+1;

    //====Вид функции=====
    Fx=(sin(x)+0.5*sin(3*x))*x;
    //=====

    if(x>b)rdsStopCalc();// Остановка

    /* Служебный комментарий, предохраняющий от незавершенного комментария в
тексте пользователя */
    #line 195 "-1" // Со следующей строки - генерируемый текст
} // rdsbcppBlockClass::rdsbcppModel

```

Глядя на достаточно сложный текст файла программы, возникает законный вопрос: нужно ли вообще иметь представление об этом тексте и о структуре программы, если она создается автоматически? Действительно, для создания простых программ, типа рассмотренных в §2, это не обязательно.

Однако, следует иметь в виду, что примеры §2 не дают полного представления о возможностях разработанного модуля автокомпиляции. Эти возможности гораздо шире и дают инструмент для построения сложных блоков, в которых используются такие понятия как «события», «функции», «настройки» и т.п. Использование всех возможностей модуля

автокомпиляции позволяет пользователям, не имеющим достаточных навыков программирования, или желающих упростить свою работу, создавать на почти профессиональном уровне достаточно сложные программы. Представление о структуре файла в этом случае помогает разработчику блока правильно распределить создаваемые фрагменты текста по вкладкам, рассмотрению которых посвящены многие параграфы описания пользователя.

Некоторые разделы описания поначалу могут показаться трудными для понимания, тогда стоит начать с создания простых автокомпилируемых блоков, постепенно, по мере необходимости, переходя к более сложным задачам. Труд, потраченный на освоение более сложных конструкций, окупится сторицей в будущем.

§3.3 Некоторые проблемы, возникающие в связи с созданием и применением автокомпилируемых блоков.

Коснемся некоторых проблем, возникающих в связи с созданием и применением автокомпилируемых блоков, на которые следует обратить особое внимание.

1. Способ хранения файлов моделей. Копирование и перенос их на другие компьютеры.

Открытость модуля автокомпиляции, которая является большим преимуществом таких модулей по сравнению с теми, в которых алгоритм не просматривается, создает одновременно ряд неудобств. Ведь преднамеренно или, чаще всего по ошибке, любой программист может внести в программу изменения. Эти изменения могут нарушить работу программ, даже созданных ранее, если файл модели (с расширением «.mdl») у них был общим.

При копировании автокомпилируемого блока в случаях, если ожидается его изменение, следует менять название его файла, чтобы он потеряли связь с ранее созданными блоками. В комплексе RDS предусмотрены специальные средства предупреждения и автоматического изменения названий, чтобы по возможности избежать этих ошибок. Эти средства подробно описаны в §3.4 описания пользователя. Их следует внимательно изучить и даже экспериментально испробовать, чтобы не столкнуться с неприятностями.

Для полностью отработанных модулей блока, которые не должны подвергаться изменению имеется специальная метка (см. §3.5.7 описания пользователя), запрещающая введение изменений в программу модуля.

2. Организация автокомпилируемых блоков и некоторые трудности разработки и отладки программ.

Несмотря на то, что программа, первоначально создаваемая пользователем, может быть достаточно простой, ее необходимо правильно написать на некотором стандартном языке высокого уровня (в реализованном варианте комплекса RDS был применен язык C++). Для этого требуется опыт программирования. Обратим внимание на несколько моментов.

1. Отладочные средства при разработке автокомпилируемых блоков ограничены по сравнению с отладочными средствами в средах разработки программ на языках высокого уровня. Поэтому целесообразно предусмотреть, например, ряд дополнительных переменных, которые могут быть встроены в различные фрагменты программы модели или

дополнительно использовать свои оригинальные способы для контроля правильности работы алгоритма.

2. Следует внимательно относиться к заданию статических и динамических переменных и придания им атрибутов типа: вход, выход и т.п. Необходимо понять стоит ли считать блок каждый такт или только при изменении входов, а для этого надо хорошо представлять, как организован динамический процесс в системе RDS (он описан в §1.3 и §3.6.4 описания пользователя). Напомним, что понятия «такт расчета» и «шаг расчета» отличаются. Величина шага по времени задается в блоке «планировщик». В каждом такте расчета вызываются все блоки и по входам рассчитываются выходы, которые передаются на следующие блоки. При наличии нескольких последовательно соединенных блоков важно, чтобы значение входа достигло конечного блока в цепочке, поэтому задается количество тактов расчета, проводимых за один шаг. Не стоит удивляться, если значение некоторого входа не дойдет до выхода за один шаг расчета при наличии длинных, последовательно соединенных цепочек блоков.

3. Использование и понимание вкладок «события», «функции», «настройки», предусмотренных в редакторе автокомпилируемых блоков, требует определенного опыта. При разработке сложных блоков они очень полезны, но при этом важно хорошо структурировать алгоритм для правильного размещения отдельных фрагментов программы на нужных вкладках, предусмотренных в редакторе. На первых порах рекомендуем пользоваться ими с осторожностью и ограничиться построением простых автокомпилируемых программ.