Работа программы-калькулятора, представленная через конечный автомат.
Состояния калькулятора описываются тремя переменными:

- **Rf** - "регистр переднего плана" (число, которое видит пользователь калькулятора на экране)
- **Rb** - "регистр заднего плана" (второй операнд операции)
- **Op** - переменная, в которой сохраняется знак операции.

Первый операнд обозначается x, второй - y, а результат - r.

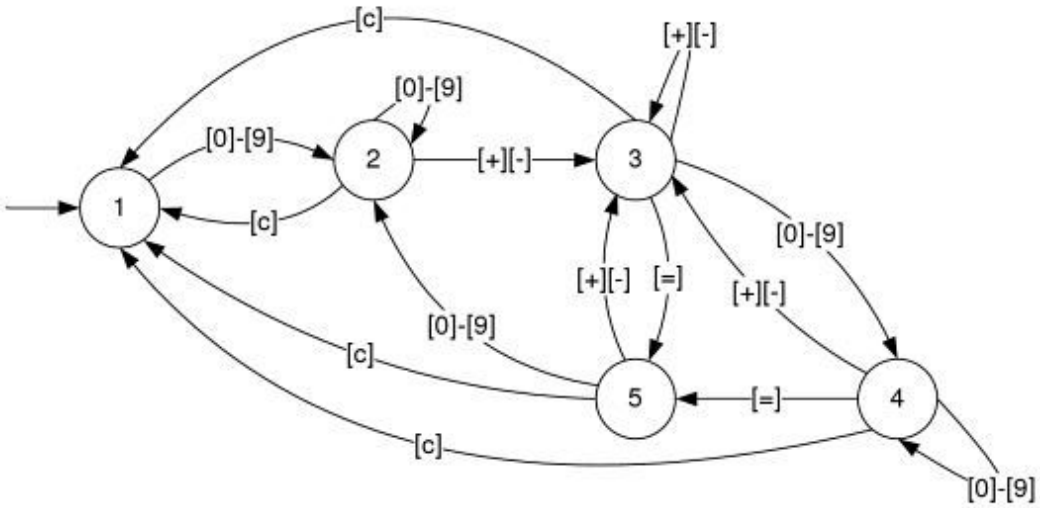


Рис. 11.1. Диаграмма конечного автомата класса `Calculator`

Состояние	Rf	Rb	Op
1	0	0	NULL
2	x	0	NULL
3	x	Rf	[+ | -]
4	y	x	[+ | -]
5	r	0	NULL

- При выполнении *перехода 1-2* регистр Rf заполняется числом, вводимым пользователем. Остальные остаются неизменными.
- При выполнении *перехода 2-3* содержимое Rf копируется в Rb, а в переменную Op записывается знак операции.
- В процессе *перехода 3-4* в регистр Rf записывается второй операнд, вводимый пользователем, а содержимое остальных регистров не меняется.

- В процессе *перехода 4-5* выполняется операция над операндами Rb и Rf (соответственно, либо +, либо -) и её результат записывается в регистр Rf, а регистры Rb и Op становятся пустыми.
- *Переход 5-2* эквивалентен 1-2 (регистры Rb и Op при этом очищаются).
- *Переход 5-3* эквивалентен переходу 2-3.
- В результате *перехода 4-3* выполняется операция над текущими значениями Rb и Rf, её результат записывается в Rf, в регистр Rb записывается содержимое Rf, а в регистр Op - новый символ операции.
- В *результате 3-3* над операндами Rb и Rf выполняется операция, символ которой в данный момент хранится в регистре Op, результат записывается в регистры Rf и Rb, а в регистр Op записывается символ новой операции.
- *Переходы 2-1, 3-1, 4-1 и 5-1* опустошают все регистры калькулятора.

Для реализации конечного автомата понадобятся объекты трех классов: QStateMachine, QState и QSignalTransition.

1. класс **QStateMachine** представляет автомат в целом.
2. класс **QState** представляет различные состояния автомата,
3. класс **QSignalTransition** — переходы между этими состояниям, инициируемые сигналами.

С помощью перечисленных классов можно построить диаграмму, показанную на рисунке. Но для работы программы этого недостаточно. Автомат должен не только переходить из одного состоянии в другое, он также должен выполнить различные операции.

```
class Calculator : public QObject
{
   Q_OBJECT
public:
   explicit Calculator(QObject *parent = 0);
signals:
   void valueChanged(int value);    // сигнал сообщает вовне, что состояние
класса изменилось
   void digitButtonPressed();       // была нажата цифровая клавиша
   void operationButtonPressed();   // была нажата клавиша операции
   void cancelButtonPressed();      // была нажата клавиша отмены
   void equalButtonPressed();       // была нажата клавиша "равно"

public slots:
   void digitButtonPressed(int button);
   void operationButtonPressed(int button);
```

```
private slots:
   // Содержимое этих методов соответствует описанию переходов автомата
   void s1Entered();
   void s1Exited();

   void s2Entered();
   void s2Exited();

   void s3Entered();
   void s3Exited();

   void s4Entered();
   void s4Exited();

   void s5Entered();
   void s5Exited();
private:
   int Rf;                      // Регистр переднего плана (отображается на
экране в данный момент)
   int Rb;                      // Регистр заднего плана (второй апперанд)
   Buttons transitionButton;    // Переменная, содержащая код клавиши, нажатие
которой вызвало переход
   Buttons Op;                  // Переменная, хранящая знак операции
   void doOp(Buttons op);       // Выполняет арифметическую операцию, символ
которой передан ему в качестве аргумента
   QStateMachine * machine;     // Конечный автомат
   // Переменные, описывающие состояния 1-5
   QState* s1;
   QState* s2;
   QState* s3;
   QState* s4;
   QState* s5;

};
```

Переменные `s1-s5` представляют собой состояния автомата. Для того чтобы установить переход между двумя состояниями, вызывается метод `addTransition()` объекта класса `QState`. Переходы инициируются сигналами (есть еще вариант, в котором переходы инициируются событиями Qt), поэтому первым аргументом метода `addTransition()` должен быть указатель на объект-источник сигнала, вторым аргументом — имя сигнала, а третьим аргументом — целевое состояние перехода. Таким образом, строка `s1->addTransition(this, SIGNAL(digitButtonPressed()), s2);` устанавливает переход между состояниями `s1` и `s2`, инициируемый сигналом `digitButtonPressed()` объекта, определяемого указателем `this`. Между двумя состояниями можно установить несколько

переходов, инициируемых разными событиями. Сигналов, которые могут инициировать переходы, в данном примере четыре:

1. `digitButtonPressed();`
2. `operationButtonPressed();`
3. `cancelButtonPressed();`
4. `equalButtonPressed().`

Каждый вызов метода `addTransition()` создает новый объект класса `QSignalTransition` (или `QEventTransition`, в зависимости от того, какой из перегруженных вариантов метода был выбран). Указатели на этот объект возвращаются методами `addTransition()`, но поскольку сейчас нечего добавить к объекту, описывающему переход, они не используются.
Каждый раз, когда автомат переходит из одного состояния в другое, генерируются два сигнала. Состояние, из которого осуществляется переход, эмитирует сигнал `exited()`, тогда как целевое состояние эмитирует сигнал `entered()`. Благодаря этим сигналам можно связать с переходами определенные действия.
Этот важный момент необходимо четко понимать: действия выполняются, когда автомат переходит из одного состояния в другое. Когда автомат находится в некотором состоянии, никакие действия не выполняются. Состояние просто указывает на то, что произойдет, когда на входе автомата появится очередное событие.
Теперь, когда все состояния связаны между собой переходами и назначены слоты для сигналов `entered()` и `exited()` состояний, создаем объект класса `QStateMachine`. Для того чтобы создать конечный автомат, состояния добавляются в объект класса `QStateMachine` с помощью метода `addState()` этого класса. Необходимо назначить начальное состояние, что делается с помощью метода `setInitialState()`. Согласно теории, у автомата должны быть (или, по крайней мере, желательно, чтобы были) допускающие (конечные) состояния. Для описания такого состояния в системе конечных автоматов Qt предусмотрен специальный класс `QFinalState`. Когда автомат переходит в это состояние, он эмитирует сигнал `finished()`. В то время как начальное состояние у автомата может быть только одно, конечных состояний (в полном соответствии с теорией) может быть несколько. У автомата, описывающего работу класса `Calculator`, нет конечных состояний, их нет у многих других классов, для которых предназначены конечные автоматы Qt. После того как автомат создан, вызывают метод `start()` объекта `QStateMachine` и автомат начинает выполнять свою работу.

Как уже отмечалось, полезные действия выполняются в слотах, связанных с сигналами `entered()` и `exited()`.

```
Calculator::Calculator(QObject *parent) :
    QObject(parent)
{
    Rf = 0;
    Rb = 0;
    Op = opNone;

    emit valueChanged(Rf);

    s1 = new QState();
    s2 = new QState();
    s3 = new QState();
    s4 = new QState();
    s5 = new QState();

    s1->addTransition(this, SIGNAL(digitButtonPressed()), s2);

    s2->addTransition(this, SIGNAL(cancelButtonPressed()), s1);
    s2->addTransition(this, SIGNAL(digitButtonPressed()), s2);
    s2->addTransition(this, SIGNAL(operationButtonPressed()), s3);

    s3->addTransition(this, SIGNAL(cancelButtonPressed()), s1);
    s3->addTransition(this, SIGNAL(operationButtonPressed()), s3);
    s3->addTransition(this, SIGNAL(digitButtonPressed()), s4);
    s3->addTransition(this, SIGNAL(equalButtonPressed()), s5);

    s4->addTransition(this, SIGNAL(cancelButtonPressed()), s1);
    s4->addTransition(this, SIGNAL(digitButtonPressed()), s4);
    s4->addTransition(this, SIGNAL(operationButtonPressed()), s3);
    s4->addTransition(this, SIGNAL(equalButtonPressed()), s5);

    s5->addTransition(this, SIGNAL(cancelButtonPressed()), s1);
    s5->addTransition(this, SIGNAL(digitButtonPressed()), s2);
    s5->addTransition(this, SIGNAL(operationButtonPressed()), s3);

    connect (s1, SIGNAL(entered()), this, SLOT(s1Entered()));
    connect (s1, SIGNAL(exited()), this, SLOT(s1Exited()));

    connect (s2, SIGNAL(entered()), this, SLOT(s2Entered()));
    connect (s2, SIGNAL(exited()), this, SLOT(s2Exited()));

    connect (s3, SIGNAL(entered()), this, SLOT(s3Entered()));
    connect (s3, SIGNAL(exited()), this, SLOT(s3Exited()));

    connect (s4, SIGNAL(entered()), this, SLOT(s4Entered()));
    connect (s4, SIGNAL(exited()), this, SLOT(s4Exited()));

    connect (s5, SIGNAL(entered()), this, SLOT(s5Entered()));
```

```
    connect (s5, SIGNAL(exited()), this, SLOT(s5Exited()));

    machine = new QStateMachine(0);
    machine->addState(s1);
    machine->addState(s2);
    machine->addState(s3);
    machine->addState(s4);
    machine->addState(s5);
    machine->setInitialState(s1);
    machine->start();
}

void Calculator::digitButtonPressed(int button)
{
    transitionButton = (Buttons) button;
    emit digitButtonPressed();
}

void Calculator::operationButtonPressed(int button)
{
    transitionButton = (Buttons) button;
    if (button == opCancel)
        emit cancelButtonPressed();
    else
        if (button == opEqual)
            emit equalButtonPressed();
        else
            emit operationButtonPressed();
}

void Calculator::s1Entered()
{
    Rf = 0;
    Rb = 0;
    Op =  opNone;
    /* Строка, содержащая emit, заставляет объект испустить сигнал
valueChanged() с новым значением,переданным в аргументе.*/
    emit valueChanged(Rf);
}

void Calculator::s1Exited()
{

}

void Calculator::s2Entered()
{
    if (Rf < 9999999) {
```

```
        Rf = Rf*10 + transitionButton;
        emit valueChanged(Rf);
    }
}

void Calculator::s2Exited()
{

}

void Calculator::s3Entered()
{
    if (Rb != 0) {
        doOp(Op);
        emit valueChanged(Rf);
    }
    Rb = Rf;
    Op = transitionButton;
}

void Calculator::s3Exited()
{
    if (transitionButton > 9) {
        doOp(Op);
        Rb = 0;
        Op = transitionButton;
        emit valueChanged(Rf);
    } else  {
        Rf = 0;
    }
}

void Calculator::s4Entered()
{
    s2Entered();
}

void Calculator::s4Exited()
{

}

void Calculator::s5Entered()
{
    doOp(Op);
    Op = opNone;
    emit valueChanged(Rf);
}
```

```
void Calculator::s5Exited()
{
    if (transitionButton <= 9) {
        Rb = 0;
        Rf = 0;
    }
}

void Calculator::doOp(Buttons op)
{
    switch (op) {
    case opPlus:
        Rf = Rf + Rb;
        break;
    case opMinus:
        Rf = Rb - Rf;
        break;
    default:
        break;
    }

}
```

Недостатки подхода

Вместе с тем тот факт, что для одного перехода вызываются два слота — один на выходе, другой на входе — может стать источником путаницы и ошибок при реализации конечного автомата. Иногда разумнее иметь один метод, полностью описывающий действия, связанные с конкретным переходом.

Еще одно неудобство, присущее системе сигналов `exited()` и `entered()`, связано с тем, что в слотах, обрабатывающих эти сигналы, трудно определить, куда мы "уходим" и, соответственно, откуда мы "приходим". В данном примере мы определяем это по косвенным при-знакам — состояниям переменных `Rb`, `Rf`, `Op` и специальной переменной `transitionButton`, которая содержит код клавиши, нажатие которой вызвало переход. Все это наталкивает на мысль об альтернативной реализации, в которой сигналы эмитировались бы не объектами, описывающими состояния, а объектами, описывающими переходы.

Готовых объектов для решения этой задачи в Qt нет, но ничто не препятствует написанию своих собственных. Создание собственной системы объектов начнем с класса `QXtTransition`, который является расширенной версией класса, описывающего переходы между состояниями.

```
class QXtTransition : public QSignalTransition
{
   Q_OBJECT
public:
      explicit  QXtTransition(QObject*  sender,  const  char*  signal,  QObject*
reciever, const char* slot,  QState* sourceState = 0);
protected:
   void onTransition ( QEvent * event );
signals:
   void transiting(QState * from, QAbstractState * to, QString label);
};
```

Каждый раз, когда конечный автомат выполняет переход из одного состояния в другое, в объекте, описывающем переход, вызывается метод `onTransition()`, объявленный в разделе `protected:`. В классе `QXtTransition` перекрываем метод `onTransition()` для того, чтобы он эмитировал сигнал `transiting()`.

У этого сигнала три параметра:

1. параметр `from` указывает на объект-состояние, из которого выполняется переход;
2. параметр `to` указывает на целевое состояние перехода;
3. параметр `label` идентифицирует, какой именно переход `QXtTransition` стал источником сигнала (это сделано на тот случай, если один слот связан с несколькими сигналами `transiting()`).

В переменную `label` записывается либо имя сигнала, вызвавшего переход, либо имя объекта `QXtTransition`, если таковое присвоено ему с помощью метода `setObjectName()`.

Реализация конструктора `QXtTransition()`.

```
QXtTransition::QXtTransition(QObject* sender, const char* signal, QObject*
reciever, const char* slot, QState* sourceState):
   QSignalTransition(sender, signal, sourceState)
{
   int offset = (*slot == '0' + QSLOT_CODE) ? 1 : 0;
   const QMetaObject *meta = reciever->metaObject();
   int slotIndex;
   if ((slotIndex = meta->indexOfSlot(slot+offset)) == -1)
      {
      if ((slotIndex =
meta->indexOfSlot(QMetaObject::normalizedSignature(slot+offset))) == -1) {
          qWarning("QXtTransition: no such slot %s::%s", meta->className(),
slot+offset);
          return;
       }
   }
       offset = (*signal == '0'+QSIGNAL_CODE) ? 1 : 0;
```

```
    int signalIndex =
this->metaObject()->indexOfSignal("transiting(QState*,QAbstractState*,QString)"
);
    if (signalIndex == -1) {
        qWarning() << "QXtTransition: failed to find signal";
        return;
    }
    if (!meta->connect(this, signalIndex, reciever, slotIndex))
        qWarning() << "QXtTransition: failed to connect signal and slot";
}
```

Реализация метода `onTransition()`

```
void QXtTransition::onTransition(QEvent *e)
{
    QString label = objectName() == "" ? QString::fromLatin1(signal().data(),
signal().size()) : objectName();
    emit transiting(sourceState(), targetState(), label);
    QSignalTransition::onTransition(e);
}
```

Самое интересное происходит в конструкторе. Для удобства указывается слот, который будет связан с сигналом `transiting()`, прямо в конструкторе. Разумеется, ничто не мешает связать сигнал и слот традиционным способом, с помощью `connect()`. Итак, первый параметр конструктора — указатель на объект-источник сигнала, вызывающего переход (класс `QXtTransition` основан на классе `QSignalTransition`). Второй параметр — имя сигнала. За ним идет указатель на объект, которому принадлежит слот, с которым нужно связать сигнал `transiting()`. Следующий параметр — имя слота. В последнем параметре конструктора передаётся указатель на объект, описывающий состояние-источник перехода.

Для решения задачи связывания сигнала и слота в конструкторе нам фактически придется сделать то, что делает функция `connect()` класса `QObject`. Нужно, что-бы в конструкторе класса слот, который будет связан с сигналом `transiting()`, передавался так же, как в методе `connect()`, т. е. с помощью макроса `SLOT()`. Этот макрос преобразует переданное ему имя слота в строку символов `char` с нулевым окончанием, имеющую определенный формат.

Фактически конструктор получает указатель на объект и имя метода этого объекта. Чтобы взаимодействовать с этим методом, нужен его индекс, т. е. номер в описании методов объекта. Для каждого объекта Qt, наследующего `QObject`, можно получить метаобъект, который, в свою очередь, позволит получить об объекте данные, которые обычно доступны только во время компиляции программы.

Константный указатель на метаобъект объекта `foo` можно получить, вызвав метод `foo.metaObject()`. Имея метаобъект для объекта `foo`, можно выполнить интроспекцию объекта foo, в том числе узнать, какими свойствами и методами обладает объект `foo`, и вызвать эти методы.

Индекс слота объекта можно найти с помощью метода `indexOfSlot()` соответствующего метаобъекта. Метод `indexOfSlot()` гораздо более строг к формату имени слота, чем макрос `SLOT()` (ибо `indexOfSlot()` предназначен для внутреннего употребления). По этой причине приходится выполнять некоторые дополнительные операции, чтобы быть уверенными, что имя слота приведено к каноническому виду. Иначе метод `indexOfSlot()` не найдет этот слот, даже если он определен в соответствующем классе (в этом случае `indexOfSlot()` возвращает значение –1).

Затем нужно найти индекс сигнала `transiting()` класса `QXtTransition`. Это делается практически так же, как в случае поиска индекса слота, за исключением того, что имя слота, которое задает программист, использующий наш класс, нам неизвестно, а имя сигнала мы знаем. Далее связываем сигнал и слот с помощью метода `connect()` метаобъекта. Этот метод отличается от одноименного метода класса `QObject`. Вместо имен сигнала и слота ему передаются индексы (именно для этого их и искали). Если связывание прошло успешно, метод возвращает значение `true`.

В приведенном примере были создавали объекты `QSignalTransition` неявно, при вызове метода `addTransition()`. Было бы неплохо создавать объекты `QXtTransition` таким же неявным способом. Но класс `QState` этого сделать не может. Понадобится собственный класс-потомок класса `QState` с методом `addTransition()`, умеющим добавлять переходы `QXtTransition`.

```
class QXtState : public QState
{
   Q_OBJECT
public:
   explicit QXtState(QState* parent = 0);
   void addTransition(QAbstractTransition* transition);
   QSignalTransition*    addTransition(QObject * sender, const char * signal,
QAbstractState* target);
   QAbstractTransition* addTransition(QAbstractState* target);
   QXtTransition * addTransition(QObject* sender, const char* signal,
QObject* reciever, const char* slot, QState* target);
public slots:
protected:
};
```

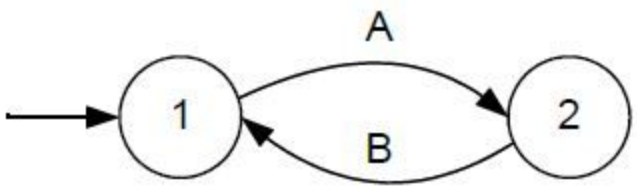
Поскольку метод, объявленный в классе-потомке, делает невидимыми все перегруженные одноименные методы, мы не только добавляем новый метод `addTransition()`, но и

перекрываем старые, чтобы наш класс, в случае необходимости, можно было использовать как обычный `QState`. С точки зрения реализации этот класс проще, чем предыдущий.

```
QXtState::QXtState(QState *parent) :
   QState(parent)
{
}

void QXtState::addTransition(QAbstractTransition * transition)
{
   QState::addTransition(transition);
}

QSignalTransition * QXtState::addTransition(QObject * sender, const char *
signal, QAbstractState * target)
{
   return QState::addTransition(sender, signal, target);
}

QAbstractTransition * QXtState::addTransition(QAbstractState * target)
{
   return QState::addTransition(target);
}

QXtTransition * QXtState::addTransition(QObject * sender, const char * signal,
QObject * reciever, const char * slot, QState * target)
{
   QXtTransition * transition = new QXtTransition(sender, signal, reciever,
slot, this);
   transition->setTargetState(target);
   addTransition(transition);
   return transition;
}
```

В методах `addTransition()`, повторяющих методы базового класса, передаются параметры методам базового класса. В методе `addTransition()` создается объект класса `QXtTransition`. Поскольку в конструкторе класса необходим указатель на объект-приемник и слот, который будет связан с сигналом `transiting()`, их тоже необходимо передать в параметрах метода `addTransition()`. В конструкторе передается только указатель на объект, описывающий исходное состояние автомата (точно так же устроены конструкторы и других классов-потомков `QAbstractTransition`). Объект, описывающий переход, должен знать и целевое состояние, которое мы за-даем с помощью

метода `setTargetState()`. Далее вызывается унаследованный метод `addTransition()`, которому передаётся указатель на созданный объект.
Рассмотрим работу описанных классов на примере простейшего конечного автомата. Автомат имеет два состояния и два перехода между ними.




```
QStateMachine * machine1 = new QStateMachine(0);
QXtState * xs1 = new QXtState();
xs1->setObjectName("1");
QXtState * xs2 = new QXtState();
xs2->setObjectName("2");
QXtTransition* t;
xt = xs1->addTransition(ui->pushButton, SIGNAL(clicked()), this,
SLOT(doTransiting(QState*,QAbstractState*,QString)), xs2);
xt->setObjectName("A");
xt = xs2->addTransition(ui->pushButton, SIGNAL(clicked()), this,
SLOT(doTransiting(QState*,QAbstractState*,QString)),xs1);
xt->setObjectName("B");
machine1->addState(xs1);
machine1->addState(xs2);
machine1->setInitialState(xs1);
machine1->start();
```

Переход из состояния 1 в состояние 2, так же как и обратный переход, инициируется сигналом `clicked()` объекта `pushButton`. С сигналами `transiting()` обоих объектов, описывающих переходы между состояниями, связывается один и тот же слот `doTransiting()`.

```
void Dialog::DoTransiting(QState *from, QAbstractState *to,
QString label)
{
      qWarning() << "transit" << from->objectName() << to->objectName()
      << label;
}
```

В результате, щелкая по кнопке `pushButton`, мы получим на консоли примерно следующие строчки:

```
transit "1" "2" "A"
transit "2" "1" "B"
```

```
transit "1" "2" "A"
...
```