УДК 004.057.4:004.716

А. О. Синюхин, С. В. Валов, К. О. Фокина, А. С. Ревенко

ПРОГРАММНО-АППАРАТНАЯ РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА 1-WIRE НА БАЗЕ МИКРОКОНТРОЛЛЕРА STM32

Представлена реализация интерфейса 1-Wire на микроконтроллере STM32 с применением аппаратного таймера. Реализованная низкоуровневая библиотека использует минимум ресурсов микроконтроллера, имеет функционал поиска адресов 1-Wire устройств и может применяться для опроса датчиков в проводных сенсорных сетях, построенных на базе данного интерфейса.

This paper presents an implementation of 1-Wire communication interface based on STM32 microcontroller with the usage of hardware timer. Implemented low-level library uses minimum of MCU resources, has address search functionality and can be applied for polling sensors in wired networks based on this interface.

Ключевые слова: интерфейс 1-Wire, микроконтроллер, конечный автомат.

Keywords: 1-Wire interface, MCU, finite-state machine.

Протокол 1-Wire, разработанный компанией *Dallas Semiconductor*, получил достаточно широкое распространение в охранных системах, системах безопасности, а также датчиках различного типа. Основной особенностью стандарта шины 1-Wire является возможность использования всего двух проводов для коммуникации, один из которых — одновременно сигнальный и обеспечивает питание ведомому устройству. Скорость передачи данных составляет порядка 15,4 Кбит/с, что достаточно для передачи небольшого объема информации, например уникального идентификатора или результата физических измерений. Немаловажным преимуществом является возможность создания сетей из устройств на базе общей шины 1-Wire [3].

Поскольку 1-Wire является зарегистрированной торговой маркой компании *Dallas Semiconductor*, аппаратная поддержка реализована в основном микросхемами компании производителя. В то же время для работы с датчиками или же реализации собственных устройств 1-Wire применение таких микросхем не всегда целесообразно. В данной работе рассматриваются особенности программно-аппаратной поддержки 1-Wire микроконтроллерами семейства STM32 уровня ведущего устройства (master).

Принцип работы интерфейса 1-Wire

Шина 1-Wire основана на модели взаимодействия типа «ведущий — ведомый» [3]. Ведущее устройство (в дальнейшем master) всегда является инициатором передачи данных.

В неактивном режиме шина данных имеет высокий логический уровень — напряжение питания (обычно 3,3 или 5 В), активное состояние соответствует низкому уровню напряжения.

В общем виде процесс передачи данных выглядит следующим образом: устройство-master сбрасывает состояние шины, отправляя импульс Reset, длительностью порядка 480—640us, в ответ на который ведомые устройства (slave) должны ответить импульсом Presence длительностью 60—240μs, тем самым подтвердив свою работоспособность и готовность принимать и отправлять данные. Временные диаграммы сигналов на шине приведены на рисунке.

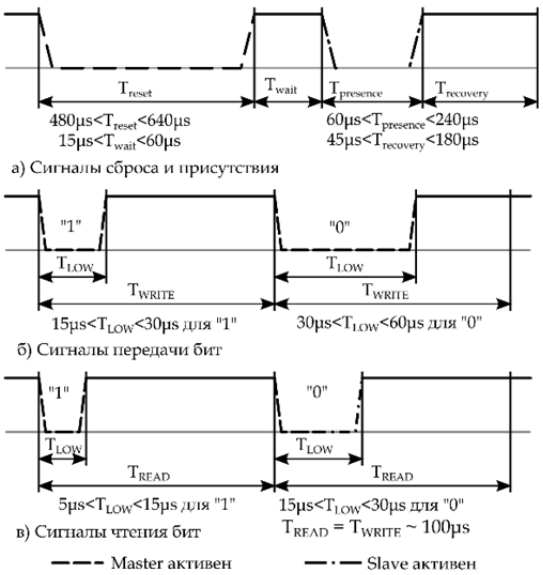

Рис. Временные диаграммы

Получив отклик от slave-устройств, master может производить с ними различные операции. Протокол 1-Wire ориентирован на использование команд, большая часть которых стандартизирована и отведена под специальные нужны для определенных классов устройств [1]. Общими для всех типов устройств являются следующие команды: Match Rom — адресация slave-устройства, Skip Rom — пропуск адресации, Read Rom — чтение уникального идентификатора из памяти, а также Search Rom — поиск адресов устройств.

После успешного обнаружения устройств на шине master начинает отправку данных, передав сначала байт команды, а затем аргументы, если они имеются. Обычно количество передаваемых байт для каждой команды известно всем устройствам сети заранее.

На физическом уровне биты данных кодируются следующим способом: низкий уровень шины продолжительностью 30—60μs воспринимается slave-устройством как нулевой бит, а продолжительностью 15—30μs — как единица. По окончании передачи master может переключиться в режим приема данных, осуществляя при этом тактирова-

ние шины. Тактирование представляет собой передачу коротких импульсов длительностью порядка 5μs с интервалами порядка 100μs, количество импульсов должно соответствовать количеству бит, которые необходимо принять. Устройство slave, обнаруживая начало каждого такого импульса, реагирует следующим образом: если необходимо передать нулевой бит, то шина удерживается в низком уровне на протяжении 15—30us, если единицу — то никаких действий не производится. Подробнее этот процесс изложен, например, в [6].

Реализация интерфейса на стороне master-устройства

Перейдем к вопросу реализации протокола со стороны master-устройства. В основе программной библиотеки лежит принцип конечного автомата, имеющего следующий набор состояний:

а) MasterOff — выключено;

б) MasterReset — осуществляется отправка импульса сброса Reset, ожидается появление отклика Presence;

в) MasterWrite — передача данных, данные передаются побитно до тех пор, пока все биты не будут переданы. По окончании передачи возможны два варианта:

— переход в состояние MasterOff с установкой флага успешного завершения операции (Complete);

— переход в состояние MasterRead с последующим чтением;

г) MasterRead — чтение данных: происходит отправка импульсов тактирования, по окончании чтения осуществляется переход в состояние MasterOff и устанавливается флаг Complete.

Генерация импульсов нужной длительности, а также чтение с последующим декодированием реализованы с использованием аппаратного таймера микроконтроллера. Таймер должен иметь как минимум два канала: один из них должен работать в режиме генерации импульсов (PWM mode), второй — в режиме захвата (Capture mode). Согласно [5], некоторые из таймеров общего назначения микроконтроллеров семейства STM32F10x позволяют использовать вышеупомянутые режимы одновременно, коммутируя к входам и выходам таймера одну и ту же линию GPIO. Такой подход позволяет задействовать всего одну линию GPIO как для передачи, так для приема, тем самым сокращается количество используемой периферии и упрощается трассировка печатной платы устройства в целом.

Для выбранного таймера разрешаются следующие источники прерываний: Update interrupt — прерывание по переполнению счетчика, Compare interrupt — прерывание по изменению уровня сигнала на линии-входе. Update interrupt срабатывает по окончании тайм-слота, Compare interrupt позволяет реагировать на изменение уровня сигнала на шине (по фронту / спаду).

Частоту счета таймера целесообразно выбрать равной 1 МГц, что позволит генерировать и принимать импульсы на шине с разрешением до 1μs — достаточным для их корректной интерпретации.

Обработка состояний конечного автомата происходит в функции-обработчике прерываний таймера, представленной в следующем листинге:

```
void TIMER_IRQ_HANDLER() {
 switch (State) {
  case MasterOff:
   ClearAllIrqFlags();
  break;
  case MasterReset:
   if (Timer->SR & FLAG_CAPTURE) { /* захват presence-импульса */
    EnableIRQOnUpdate();
    Timer->CCR_OUT ← 0;
    ResetLen ← Timer->REG_CAPTURE;
    ClearIRQFlag(FLAG_CAPTURE);
   }
   else if (Timer->SR & FLAG_UIF) { /* таймаут: отклик отсутствует */
    ClearIRQFlag(FLAG_UPDATE);
    DisableTimer();
    Complete ← true;
   }
  break;
  case MasterWrite:
   if (Timer->SR & FLAG_UPDATE) { /* начало следующего тайм-слота */
    ClearAllIrqFlags();
    if (TxBitIndex >= BitsToTx) {
     if (RxAfterTx && BitsToRx > 0)
      SwitchStateToRx();
     else
      SwitchStateToOff();
    }
    else {
     Timer->CCR_OUT ← TxImpulses[TxBitIndex];
     TxBitIndex ← TxBitIndex + 1;
    }
   }
   ClearAllIrqFlags();
  break;
  case MasterRead:
   if (Timer->SR & FLAG_UDATE) { /* начало следующего тайм-слота */
    ClearIRQFlag(FLAG_UPDATE);
    if (Complete) {
     Timer->CR1 &= ~TIM_CR1_CEN;
     ClearAllIrqFlags();
     return;
    }
    BitCaptured ← false;
   }
   if (Timer->SR & FLAG_CAPTURE) { /* захват импульса по фронту */
    if (!BitCaptured) {
     /* сохраняем длительность принятого импульса */
     RxImpulses[RxBitIndex] = Timer->CCR_CAPTURE;
     RxBitIndex ← RxBitIndex + 1;
```

```
        /* Прочитали очередной импульс */
        BitCaptured ← true;
        if (RxBitIndex >= BitsToRx) {
          /* Закончили чтение импульсов */
          Timer->CCR_OUT ← 0;
          Complete ← true;
        }
    }
        ClearIRQFlag(FLAG_CAPTURE);
      }
    break;
  }
}
```


Переменная State хранит состояние конечного автомата. Флаг Complete устанавливается в true по окончании очередной операции с шиной (сброс, отправка или прием). Переменная RxAfterTx позволяет переключиться в режим приема сразу после окончания передачи, что бывает необходимо, поскольку многие устройства 1-Wire работают по схеме «запрос — ответ» и требуют чтения после отправки данных.

Функция переключения в состояние приема данных выглядит следующим образом:

```
void SwitchStateToRx() {
  /* устанавливаем длину импульса для режима чтения */
  Timer->CCR_OUT ← READ_IMPULSE_LEN;
  /* перезагрузка регистра таймера */
  Timer->EGR |= TIM_EGR_UG;
  /* переключаем состояние конечного автомата */
  state ← MasterRead;
}
```

Кроме изменения значения переменной состояния в этой функции происходит установка длины тайм-слота для режима чтения.

Функция переключения в состояние «выключено» производит выключение таймера, устанавливает флаг окончания операции и меняет состояние конечного автомата в MasterOff:

```
void SwitchStateToOff() {
  Timer->CR1 &= ~TIM_CR1_CEN; /* остановка таймера */
  Complete ← true;
  state ← MasterOff;
}
```

Функция ClearIRQFlag(flag) очищает соответствующий флаг (флаги) прерывания в регистре состояний SR таймера:

```
void ClearIRQFlag(flags) {
   Timer->SR &= ~flags;
}
```

Функция ClearAllIrqFlags() аналогична ей, но очищает все флаги прерываний посредством записи в регистр нуля:

```
void ClearAllIrqFlags() {
  Timer->SR ← 0;
}
```

В реальной микропрограмме эти функции целесообразно реализовать в виде макросов, что сократит временные расходы на вызов функции.

Массив TxImpulses должен содержать последовательность длительностей импульсов, подготовленных для отправки. Их количество определяется переменной BitsToTx. Аналогичный массив RxImpulses применяется для приема импульсов в процессе чтения. Количество бит, которые необходимо принять, определяется переменной BitsToRx.

Перевод автомата в состояние сброса производится внешней функцией, которая кроме изменения значения переменной состояния State производит также конфигурацию таймера в режим одиночного импульса (One Pulse Mode) с целью генерации импульса Reset:

```
void SendReset() {
 Complete ← false;
 State ← MasterReset;
 ClearAllIrqFlags();
 /* устанавливаем тайм-слот в 1ms */
 Timer->ARR ← RESET_PERIOD;
 /* устанавливаем длительность низкого уровня (500us) */
 Timer->CCR_OUT ← RESET_PULSE;
 /* обновляем значения в регистрах ARR и CCR */
 Timer->EGR ← TIM_EGR_UG;
 /* разрешаем необходимые прерывания */
 Timer->DIER ← TIM_DIER_UIE | TIM_DIER_CAPTURE;
 ClearAllIrqFlags();
 /* Запускаем таймер в режиме одиночного импульса */
 Timer->CR1 ← TIM_CR1_OPM | TIM_CR1_CEN;
}
```

После того как Reset успешно отправлен, целесообразно проверить, откликаются ли устройства на шине, посредством функции

```
bool HasPresence() {
 return Complete && (ResetLen >= RESET_WITH_PRESENCE);
}
```

где RESET_WITH_PRESENCE — минимально допустимая длительность от начала импульса Reset до окончания импульса Presence.

Для начала отправки или чтения данных необходимо вызвать функцию Start(), передав в нее один из режимов работы: WriteRead — передача с последующим приемом, Write — только передача данных, Read — только прием данных.

```
void Start(mode) {
 Complete ← false; BitCaptured ← false;
 Timer->ARR = BIT_PPERIOD;
 if (mode == Write || mode == WriteRead) {
  State ← MasterWrite;
  Timer->CCR_OUT ← TxImpulses[TxBitIndex];
  ONEWIRE_MASTER_TIMER->EGR ← TIM_EGR_UG;
  if (mode == WriteRead)
   RxAfterTx ← true;
  else
   RxAfterTx ← false;
 }
 else if (mode == Read) {
  State ← MasterRead;
  Timer->CCR_OUT ← BIT_READ_PULSE;
 }
 Timer->CR1 |= TIM_CR1_CEN; /* Запуск таймера */
}
```


Константа BIT_PERIOD соответствует тайм-слоту длительностью 100μs, в течение которого происходит отправка или прием одного бита данных, BIT_READ_PULSE — длительность импульса чтения.

Процедура поиска адресов устройств

Каждое slave-устройство имеет уникальный 64-битный адрес, состоящий из 8-битного идентификатора семейства, 48-битного серийного номера и 8-битной контрольной суммы.

В случае множества устройств на одной шине перед началом обмена данными необходимо выяснить адреса устройств, выполнив процедуру поиска посредством команды Search Rom. Поиск адресов представляет собой обход бинарного дерева, каждый узел которого соответствует биту адреса в текущей позиции (равной расстоянию от корня): 1 — если узел является правым потомком родителя, 0 — если левым. Листовые узлы соответствуют последним битам адресов устройств. Двигаясь от корня дерева к листу, можно получить полный адрес устройства. На физическом уровне шина 1-Wire подчиняется правилу монтажного «и», откуда следует, что в случае одновременной передачи бита каждым устройством получим 1 тогда и только тогда, когда 1 передали все устройства. После отправки Search Rom все узлы отправляют одновременно 2 бита: исходный и инверсный, что дает информацию о том, какой бит может присутствовать в текущей позиции всех адресов (табл.).

Адресация устройств

Исходный бит	Инверсный бит	Информация
0	0	В текущем бите всех адресов есть как 0, так и 1
0	1	В текущем бите всех адресов присутствует только 0
1	0	В текущем бите всех адресов присутствует только 1
1	1	Устройства на шине отсутствуют

Далее master выполняет адресацию, отправляя следующий бит, в результате чего задействованными остаются только устройства, имеющие этот бит в данной позиции своего адреса. Оставшиеся устройства вновь присылают 2 бита, и процесс повторяется до тех пор, пока не будет получен последний бит одного из адресов. В случае ветвления (биты 00) необходимо выполнить сброс и повторить процесс, уходя в другую, еще не пройденную ветвь дерева.


```
int SearchRom(OnewireAddr_T *addr, int nAddr) {
 int8_t lastBranch ← -1, currentBranch;
 uint32_t foundDevices ← 0;
 OnewireAddr_T lastAddr ← {0}; /* последний найденный адрес */
 do {
  currentBranch ← -1;
  BitToRx ← 0; BitsToTx ← 0;
  uint32_t retries ← 0;
  do {
   SendReset();
   WaitUntil(!Complete);
  } while (!HasPresence() && (++retries< MAX_RETRIES));
  if (!HasPresence())
   return NO_RESPONSE;
  PushTxByte(SEARCH_ROM);
  for (uint8_t nbit ← 0; nbit < BITS_IN_ADDR; nbit++) {
   BitsToRx ← 2;
   Start(WriteRead);
   WaitUntil(!Complete, SEND_RECV_TIMEOUT);
   if (!Complete)
    return TX_ERROR;
   uint32_t bitPair ← BitPair(GetRxBit(0), GetRxBit(1));
   BitToRx ← 0; BitsToTx ← 0;
   bool sendBit ← false;
   switch (bitPair) {
    case 0x00: /* биты "00", случай ветвления */
     if (nbit < lastBranch) { /* не вернулись к последнему ветвлению */
      if (GetBit(&lastAddr, nbit)) {
       sendBit ← true;
       if (currentBranch < nbit)
        currentBranch ← nbit;
      }
      else
       sendBit ← false;
     }
     else if (nbit == lastBranch) /* вернулись к последней точке ветвления */
      sendBit ← false;
     else {
      sendBit ← true;
      if (currentBranch < nbit)
       currentBranch ← nbit; /* номер бита, где случилось ветвление */
     }
     break;
    case 0x1: /* биты "01", необходимо отправить "0" */
```

```
        sendBit ← false;
        break;
      case 0x2: /* биты "10", необходимо отправить "1" */
        sendBit ← true;
        break;
      case 0x3: /* биты "11", отсутствие устройств на шине */
        return NO_DEVICES;
     }
     SetBit(&lastAddr, nbit, sendBit);
     PushTxBit(sendBit);
     DelayMs(10); /* программная задержка на 10ms */
    }
    lastBranch ← currentBranch;
    addr[foundDevices++] ← lastAddr;
  } while ((lastBranch >= 0) && (foundDevices < nAddr));
  return foundDevices;
}
```


Функции GetBit() и SetBit() получают и устанавливают бит в нужной позиции адреса. Функция PushTxByte() добавляет байт к отправке посредством преобразования бит в длительности и записи их в конец массива TxImpulses. Функция PushTxBit() добавляет к отправке 1 бит. Значение константы SEND_RECV_TIMEOUT равно максимально допустимому времени отправки 1 бита и приема 2 бит.

Выводы

Реализованная низкоуровневая библиотека для работы с шиной 1-Wire позволяет взаимодействовать с различными типами устройств, в отличие от реализаций, использующих для формирования тайм-слотов программные задержки [4], использует минимум аппаратной периферии микроконтроллера, позволяет работать в многозадачной среде, например в рамках операционной системы реального времени. Тестирование программного кода показало достаточно высокую устойчивость такой реализации к ошибкам на шине данных, а также повышенную отказоустойчивость.

Список литературы

1. *Complete* 1-Wire Command Codes // OWFS Website. URL: http://owfs.sourceforge.net/commands.html (дата обращения: 05.08.2019).

2. *Using* a UART to Implement a 1-Wire Bus Master // Maxim Integrated. URL: https://www.maximintegrated.com/en/app-notes/index.mvp/id/214 (дата обращения: 05.08.2019).

3. *Елисеев Н.* Интерфейс 1-Wire: устройство и применение // ЭЛЕКТРОНИКА: Наука, Технология, Бизнес. 2007. Вып. 8. URL: http://www.electronics.ru/files/article_pdf/0/article_657_119.pdf (дата обращения: 05.08.2019).

4. *Программная* реализация протокола 1-Wire (iButton, MicroLan) на микроконтроллерах AVR. URL: https://aterlux.ru/article/1wire (дата обращения: 05.08.2019).

5. *RM0008* Rev 20 Reference manual. STMicroelectronics. December 2018. URL: https://www.st.com/resource/en/reference_manual/cd00171190.pdf (дата обращения: 25.06.2019).

6. *1-Wire* Online Tutorial // Maxim Integrated. URL: https://www.maximintegrated.com/en/products/1-wire/flash/overview/index.cfm (дата обращения: 25.06.2019).

Об авторах

Александр Олегович Синюхин — ассист., Балтийский федеральный университет им. И. Канта, Россия.
E-mail: asinyukhin@inbox.ru

Сергей Владимирович Валов — технический директор, ООО «Роуттех», Россия.
E-mail: valov@moarstack.net

Ксения Олеговна Фокина — магистрант, Балтийский федеральный университет им. И. Канта, Россия.
E-mail: fk.ksenia@gmail.com

Андрей Сергеевич Ревенко — канд. физ.-мат. наук, доц., Балтийский федеральный университет им. И. Канта, Россия.
E-mail: andy.revenko@gmail.com

The authors

Alexander O. Sinyukhin, Assistant, I. Kant Baltic Federal University, Russia.
E-mail: asinyukhin@inbox.ru

Sergey V. Valov, Chief Technology Officer, «RoutTech» Ltd, Russia.
E-mail: valov@moarstack.net

Ksenya O. Fokina, Master's Student, I. Kant Baltic Federal University, Russia.
E-mail: fk.ksenia@gmail.com

Dr Andrey S. Revenko, Associate Professor, I. Kant Baltic Federal University, Russia.
E-mail: andy.revenko@gmail.com