CHAPITRE 8

USART dans AVR ATmega16 / ATmega32

Introduction

L'AVR ATmega dispose d'un système USART flexible, qui peut être utilisé pour la communication série avec d'autres périphériques tels qu'un ordinateur, des modules série GSM, des modules GPS, etc.

La communication série :

 Lors de l’envoi / réception de données, certains bits sont ajoutés afin de connaître le début / la fin des données, etc. La structure couramment utilisée est la suivante : 8 bits de données, 1 bit de départ (0 logique) et 1 bit d’arrêt (logique 1), comme indiqué :



Il existe également d'autres formats de trame pris en charge disponibles dans UART, tels que le bit de parité, les bits de données variables (5 à 9 bits de données).

 Vitesse (débit en bauds) :

Le débit est « Nombre de bits par seconde (bps) », également appelé Débit en bauds dans un système binaire. Normalement, cela définit la vitesse de la ligne série. Certaines vitesses de transmission standard sont définies, par exemple 1200, 2400, 4800, 19200, 115200 bps, etc. On utilise normalement 9600 bps lorsque la vitesse n'est pas un problème critique.

 Connexion matérielle :

Normalement, dans USART, nous n’avons besoin que des fils Tx (transmission), Rx (réception) et GND.

* AVR ATmega USART a un niveau de tension TTL qui est 0 v pour la logique 0 et 5 v pour la logique 1.
* Dans les ordinateurs et la plupart des anciens périphériques, le protocole RS232 est utilisé pour la communication série, où un connecteur 9 broches est normalement utilisé. La communication série RS232 a des niveaux de tension différents de ceux de la communication série ATmega, à savoir +3 v à +25 v pour le zéro logique et -3 v à -25 v pour la logique 1.
* Donc, pour communiquer avec le protocole RS232, nous devons utiliser un convertisseur de niveau de tension comme le MAX232 IC.

Comme indiqué dans la fiche technique ATmega 16 USART présente les caractéristiques suivantes

* Fonctionnement en duplex intégral (registres de réception et de transmission série indépendants)
* Opération Asynchrone ou Synchrone
* Fonctionnement synchrone synchronisé maître ou esclave
* Générateur de débit en bauds haute résolution
* Prend en charge les trames série avec 5, 6, 7, 8 ou 9 bits de données et 1 ou 2 bits d'arrêt
* Génération et vérification de parité paires ou impaires prises en charge par le matériel
* Détection de dépassement de données
* Détection d'erreur de cadrage
* Le filtrage du bruit inclut la détection des bits à démarrage erroné et le filtre passe-bas numérique
* Trois interruptions distinctes sur TX Complete, Registre de données TX vide et Réception complète
* Mode de communication multiprocesseur
* Mode de communication asynchrone à double vitesse

Registres de base AVR

1. **UDR: registre de données USART**

Il a essentiellement deux registres, l’un est Tx. Byte et autre est Rx Byte. Les deux partage le même registre UDR. N'oubliez pas que lorsque nous écrivons au registre UDR. Le tampon Tx sera écrit et lorsque nous lirons à partir de ce registre, le tampon Rx sera lu. Le tampon utilise un registre à décalage FIFO pour transmettre les données.

       2.  **UCSRA** : **Registre de** contrôle et d'état USART A. Comme son nom l'indique, est utilisé pour les indicateurs de contrôle et d'état. De manière similaire, il existe deux autres registres de contrôle et de statut USART, à savoir UCSRB et UCSRC.

       3. **UBRR** : USART Baud Rate Register, il s’agit d’un registre 16 bits utilisé pour définir le débit en bauds.

Nous verrons ce registre en détail:

**UCSRA: Registre de contrôle et d'état USART A**

****

* **Bit 7 - RXC:** USART Receive Complete

Ce bit de drapeau est défini l’orsqu'il y a des données non lues dans UDR. Le drapeau RXC peut être utilisé pour générer une interruption de réception complète.

* **Bit 6 - TXC:** Transmission USART terminée

Ce bit indicateur est défini lorsque la trame entière de la mémoire tampon Tx est décalée et qu’aucune nouvelle donnée n’est actuellement présente dans la mémoire tampon de transmission (UDR). Le bit de drapeau TXC est automatiquement effacé lorsqu’une interruption de transmission complète est exécutée ou peut être effacé en écrivant un 1 dans son emplacement de bit. L'indicateur TXC peut générer une interruption de transmission terminée.

* **Bit 5 - UDRE:** registre de données USART vide

Si UDRE est 1, le tampon est vide, ce qui indique que le tampon de transmission (UDR) est prêt à recevoir de nouvelles données. L'indicateur UDRE peut générer une interruption vide du registre de données. UDRE est défini après une réinitialisation pour indiquer que le transmetteur est prêt.

* **Bit 4 - FE:** erreur de trame
* **Bit 3 - DOR:** Data OverRun

Ce bit est activé si une condition de dépassement de données est détectée. Une surcharge de données se produit lorsque le tampon de réception est plein (deux caractères) et qu'un nouveau caractère est en attente dans le registre à décalage de réception.

* **Bit 2 - PE:** erreur de parité
* **Bit 1 - U2X:** doublez la vitesse de transmission USART
* **Bit 0 - MPCM:** mode de communication multiprocesseur

**UCSRB: Registre de contrôle et d'état USART B**

****

* **Bit 7 - RXCIE: activation d'** interruption complète RX

                   L'écriture de 1 sur ce bit active l'interruption sur l'indicateur RXC.

* **Bit 6 - TXCIE: activation d'** interruption complète de TX

                   L'écriture de 1 sur ce bit active l'interruption sur le drapeau TXC.

* **Bit 5 - UDRIE:** Activation d'interruption vide du registre de données USART

                   L'écriture de 1 sur ce bit active l'interruption sur le drapeau UDRE.

* **Bit 4 - RXEN:** Activer le récepteur

                  L'écriture de 1 sur ce bit active le récepteur USART.

* **Bit 3 - TXEN:** Activer l'émetteur

                  L'écriture de 1 sur ce bit active le transmetteur USART.

* **Bit 2 - UCSZ2:** Taille du caractère

Les bits UCSZ2 combinés avec le bit UCSZ 1: 0 dans UCSRC définissent le nombre de bits de données (taille de caractère) dans une trame utilisée par le récepteur et l'émetteur.

* **Bit 1 - RXB8:** Réception de données, bit 8
* **Bit 0 - TXB8:** Transmission de données, bit 8

**UCSRC: Registre de contrôle et d'état USART C**

****

* **Bit 7 - URSEL:** sélection du registre

Ce bit sélectionne l'accès au **registre UCSRC** ou au registre **UBRRH** , car les deux registres partagent la même adresse. L'URSEL doit être un lors de l'écriture de l'UCSRC, sinon les données seront écrites dans le registre UBRRH.

* **Bit 6 - UMSEL:** Sélection du mode USART

                  Ce bit sélectionne le mode de fonctionnement asynchrone ou synchrone.

**0 = fonctionnement** asynchrone

**1 = fonctionnement** synchrone

* **Bit 5: 4 - UPM1: 0** : Mode parité

Ces bits permettent d'activer et de définir le type de génération et de contrôle de parité. Si une non-concordance est détectée, l'indicateur PE dans UCSRA sera défini.

| **UPM1** | **UPM0** | **Mode Parité** |
| --- | --- | --- |
| 0 | 0 | désactivé |
| 0 | 1 | Réservé |
| 1 | 0 | Activé, même parité |
| 1 | 1 | Activé, parité impaire |

* **Bit 3 - USBS** : sélection du bit d'arrêt

Ce bit sélectionne le nombre de bits d'arrêt à insérer par le transmetteur. Le récepteur ignore ce paramètre.**0 =** 1 bit ; **1 =** 2 bits

* **Bit 2: 1 - UCSZ1: 0** : Taille du caractère

Les bits UCSZ1: 0 combinés avec le bit UCSZ2 dans UCSRB définissent le nombre de bits de données (taille de caractère) dans une trame utilisée par le récepteur et l'émetteur.

| **UCSZ2** | **UCSZ1** | **UCSZ0** | **Taille du personnage** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 5 bits |
| 0 | 0 | 1 | 6 bits |
| 0 | 1 | 0 | 7 bits |
| **0** | **1** | **1** | **8 bits** |
| 1 | 0 | 0 | Réservé |
| 1 | 0 | 1 | Réservé |
| 1 | 1 | 0 | Réservé |
| 1 | 1 | 1 | 9 bits |

* **Bit 0 - UCPOL:** Polarité d'horloge

Ce bit est utilisé uniquement pour le mode synchrone. Ecrivez ce bit à zéro lorsque le mode asynchrone est utilisé.

**UBRRL et UBRRH: Registres de vitesse de transmission USART**

****

* **Bit 15 - URSEL:** Sélection du registre

Ce bit sélectionne l'accès au **registre UCSRC** ou au registre **UBRRH** , car les deux registres partagent la même adresse. L'URSEL doit être un lors de l'écriture de l'UCSRC, sinon les données seront écrites dans le registre UBRRH.

* **Bit 11: 0 - UBRR11: 0:** registre de débit en **baud** USART.

            Utilisé pour définir le débit en bauds

                   

Exemple: supposons que Fosc = 8 MHz et le débit en bauds requis = 9600 bps.

Alors valeur de **UBRR = 51.088 soit 51.**

 Nous pouvons également définir cette valeur avec du code c en utilisant une macro pré-processeur comme suit.

#define F\_CPU 8000000UL /\* Define frequency here its 8MHz \*/

#define USART\_BAUDRATE 9600

#define BAUD\_PRESCALE (((F\_CPU / (USART\_BAUDRATE \* 16UL))) - 1)

BAUD\_PRESCALE est la valeur que nous devons charger dans le registre UBRR pour définir le débit en bauds défini.

Étapes de programmation

**Initialisation de USART**

1. Activer la transmission et la réception en utilisant le registre UCSRB.
2. Définissez la taille du bit de données sur 8 bits à l'aide du registre UCSRC.
3. Définissez le débit en bauds à l'aide du registre UBRR.

void UART\_init(long USART\_BAUDRATE)

{

 UCSRB |= (1 << RXEN) | (1 << TXEN); /\* Turn on transmission and reception \*/

 UCSRC |= (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1);/\* Use 8-bit char size \*/

 UBRRL = BAUD\_PRESCALE; /\* Load lower 8-bits of the baud rate \*/

 UBRRH = (BAUD\_PRESCALE >> 8); /\* Load upper 8-bits\*/

}

 Arduino - Série

Arduino prend en charge la communication série (UART). Avec cela, on peut communiquer avec un PC ou avec d'autres modules Arduino prenant en charge l'interface de communication série.

Avec la carte Arduino, vous pouvez envoyer toutes les données du PC à la carte Arduino ou de la carte Arduino au PC en utilisant la communication série. Mais vous n'avez pas besoin de port serila séparé (par exemple, port RS 232) pour cette communication. Vous pouvez utiliser le port USB que vous utilisez pour programmer Arduino afin d’effectuer une communication série comme indiqué ci-dessous.

En termes de logiciel, la communication série va comme ci-dessous. Comme vous le savez, si vous téléchargez le programme, celui-ci est exécuté sur un chipset Arduino. Lorsque le programme exécute la bibliothèque Serial, il communique avec le programme Arudio Serial Monitor comme indiqué ci-dessous.

Lorsque la fonction Serial.read () est exécutée, elle reçoit les données envoyées par Arduino Serial Monitor. Lorsque la fonction Serial.pring () est exécutée, la carte Arduino envoie les données au port série du PC et est éventuellement acheminée au moniteur série Arduino.

|  |
| --- |
| **Arduino Development Environment built-in USART commands [**[**www.Arduino.cc**](http://www.Arduino.cc/)**]****Notes:*** **Arduino UNO R3:** omit the “n” from the command
* **Arduino Mega 2560:** insert the US ART channel in use (1, 2, or 3) for “n”
 |
| **Command** | **Description** |
| **Serialn.begin() Serialn.end()****Serialn.available()****Serialn.read() Serialn.flush() Serialn.print()****Serialn.println() Serialn.write()** | Sets Baud rateDisables serial communication. Allows Digital 1(TX) and Digital (0) RX to be used for digital input and output.Determines how many bytes have already been received and stored in the 128 byte buffer.Reads incoming serial data.Flushes the serial receive buffer of data.Prints data to the serial port as ASCII text. An optional second parameter specifies the format for printing (BYTE, BIN, OCT, DEC, HEX).Prints data to the serial port as ASCII text followed by a carriage return.Writes binary data to the serial port. A single byte, a series of bytes, or an array of bytes may be sent. |

SPI dans AVR ATmega16 / ATmega32

**Introduction**

L'interface de périphérique série (SPI) est un protocole de connexion d'interface de bus lancé à l'origine par Motorola Corp. Il utilise quatre broches pour la communication.

* **SDI** (entrée de données série)
* **SDO** (sortie de données série)
* **SCLK** (horloge série)
* **CS** (Chip Select)

Il possède deux broches pour le transfert de données appelées SDI (entrée de données série) et SDO (sortie de données série). La broche SCLK (horloge série) est utilisée pour synchroniser le transfert de données et le maître fournit cette horloge. La broche CS (Chip Select) est utilisée par le maître pour sélectionner un périphérique esclave.



Les périphériques SPI ont des registres à décalage de 8 bits pour envoyer et recevoir des données. Chaque fois que le maître doit envoyer des données, il place les données dans un registre à décalage et génère l'horloge requise. Chaque fois que le maître veut lire des données, l'esclave place les données dans un registre à décalage et le maître génère l'horloge requise. Notez que SPI est un protocole de communication en duplex intégral, c’est-à-dire que les données des registres à décalage maître et esclave sont échangées en même temps.

 **ATmega32 SPI Communication**

ATmega32 a un module SPI intégré. Il peut agir en tant que périphérique SPI maître et esclave. Les broches de communication SPI dans AVR ATmega sont les suivantes :

* **MISO** (Master In Slave Out)

                    Le maître reçoit les données et l'esclave transmet les données via cette broche.

* **MOSI** (Master Out Slave In)

                    Le maître transmet les données et l'esclave reçoit les données via cette broche.

* **SCK** (horloge de décalage)

                    Le maître génère cette horloge pour la communication, qui est utilisée par l'esclave.

                    Seul le maître peut initier une horloge série.

* **SS** (Slave Select)

                    Le maître peut sélectionner esclave via cette broche.

L'interconnexion entre maître et esclave à l'aide de SPI est illustrée dans la figure ci-dessous.



AVR ATmega32 utilise trois registres pour configurer la communication SPI, à savoir Registre de contrôle SPI, Registre d'état SPI et Registre de données SPI.

 **SPCR: Registre de contrôle SPI**

****

**Bit 7 - SPIE:** bit d'activation d'interruption SPI

            **1** = Activer l'interruption SPI.

**0** = Désactiver l'interruption SPI.

**Bit 6 - SPE:** bit d'activation SPI

            **1** = Activer SPI.

**0** = Désactiver SPI.

**Bit 5 - DORD:** bit d'ordre de données

            **1** = LSB transmis en premier.

**0** = MSB transmis en premier.

**Bit 4 - MSTR:** bit de sélection maître / esclave

            **1** = mode maître.

**0** = mode esclave.

**Bit 3 - CPOL:** bit de sélection de polarité d'horloge

            **1** = l'horloge commence à partir de la logique.

**0** = l'horloge démarre à partir du zéro logique.

**Bit 2 - CPHA:** bit de sélection de phase d'horloge

            **1** = échantillon de données sur le bord d'horloge de fuite.

**0** = échantillon de données sur le front d'horloge.

**Bit 1: 0 - SPR1:** bits de sélection de fréquence d'horloge **SPR0**

Le tableau ci-dessous montre le réglage du bit de sélection de fréquence d'horloge SCK.

| **SPI2X** | **SPR1** | **SPR0** | **Fréquence SCK** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | Fosc / 4 |
| 0 | 0 | 1 | Fosc / 16 |
| 0 | 1 | 0 | Fosc / 64 |
| 0 | 1 | 1 | Fosc / 128 |
| 1 | 0 | 0 | Fosc / 2 |
| 1 | 0 | 1 | Fosc / 8 |
| 1 | 1 | 0 | Fosc / 32 |
| 1 | 1 | 1 | Fosc / 64 |

**SPSR: Registre d'état SPI**

****

**Bit 7 - SPIF:** bit de drapeau d'interruption SPI

* Cet indicateur est défini lorsque le transfert en série est terminé.
* Est également activé lorsque la broche SS est basse en mode maître.
* Il peut générer une interruption lorsque le bit SPIE dans SPCR et une interruption globale est activé.

**Bit 6 - WCOL:** Écrire un bit d' **indicateur de** collision

* Ce bit est activé lors de l'écriture dans le registre de données SPI lors du transfert de données précédent.

**Bit 5: 1 - Bits réservés**

**Bit 0 - SPI2X:** bit de vitesse SPI double

* Lorsqu'il est défini, la vitesse SPI (fréquence SCK) est doublée.

 **SPDR: registre de données SPI**

****

* Registre de données SPI utilisé pour transférer des données entre le fichier de registre et le registre à décalage SPI.
* L'écriture dans le SPDR initie la transmission de données.

 **Quand l'appareil est en mode maître**

* Le maître écrit un octet de données dans SPDR. L'écriture sur SPDR démarre la transmission de données.
* Les données 8 bits commencent à être décalées vers l'esclave et, après le décalage de l'octet complet, le générateur d'horloge SPI s'arrête et le bit SPIF est activé.

 **Quand l'appareil est en mode esclave**

* L'interface SPI esclave reste en veille tant que la broche SS est maintenue au niveau haut par le maître.
* Il s'active uniquement lorsque la broche SS est enfoncée bas. Les données sont décalées avec l'horloge SCK entrante depuis le maître pendant l'opération d'écriture.
* SPIF est défini après le décalage complet d'un octet.

 **Fonctionnalité de broche SS Mode maître**

* En mode maître, la broche SS est utilisée comme broche GPIO.
* Définissez la direction de la broche SS en sortie pour sélectionner le périphérique esclave utilisant cette broche.
* Notez que si la broche SS est configurée en entrée, elle doit être réglée sur High pour le fonctionnement principal.
* S'il est défini comme entrée en mode maître et qu'il est piloté par un circuit externe, le système SPI le reconnaît comme un autre maître sélectionnant SPI comme esclave en raison de son comportement actif à l'état bas.
* Cela effacera le bit MSTR dans le registre SPCR et SPI passera en mode esclave.

 **Fonctionnalité de broche SS Mode esclave**

* En mode esclave, la broche SS est toujours configurée comme entrée.
* Quand il est bas, le SPI s'active.
* Et lorsque la logique SPI est élevée, elle est réinitialisée et ne reçoit aucune donnée entrante.

Exemple

Faisons la communication SPI en utilisant la famille AVR ATmega16 (Master) et ATmega32 (Slave). Le Maître enverra un compte continu à Slave. Affichez les données envoyées et reçues sur un écran LCD 16x2.

 Schéma d'interface



Commençons par programmer le périphérique maître, à savoir ATmega16

**Étapes d'initialisation du maître SPI**

Pour initialiser ATmega16 en tant que maître, procédez comme suit.

* Définissez les directions des broches MOSI, SCK et SS en sortie.
* Définissez la direction de la broche MISO en entrée.
* Faites broche SS haute.
* Activez SPI en mode maître en définissant les bits SPE et MSTR dans le registre SPCR.
* Définissez la combinaison SPI Clock Rate Bits pour définir la fréquence SCK.

**Fonction SPI\_Init**

void SPI\_Init() /\* SPI Initialize function \*/

{ DDRB |= (1<<MOSI)|(1<<SCK)|(1<<SS); /\* Make MOSI, SCK, SS as Output pin \*/

 DDRB &= ~(1<<MISO); /\* Make MISO pin as input pin \*/

 PORTB |= (1<<SS); /\* Make high on SS pin \*/

 SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0); /\* Enable SPI in master mode with Fosc/16 \*/

 SPSR &= ~(1<<SPI2X); /\* Disable speed doubler \*/

}

 **Étapes d'écriture du maître SPI**

* Copier les données à transmettre dans le registre SPDR.
* Attendez la fin de la transmission, c.-à-d. Que le drapeau SPIF de poll est élevé.
* Alors que l'indicateur SPIF est défini, lisez SPDR à l'aide du tampon de vidage.
* Le bit SPIF est effacé par H / W lors de l'exécution du programme ISR correspondant.
* Notez que pour effacer le bit SPIF, vous devez lire les registres SPIF et SPDR en alternance.

**Fonction SPI\_Write** Argument d'entrée: Il a l'argument d'entrée des données à transmettre.

void SPI\_Write(char data) /\* SPI write data function \*/

{ char flush\_buffer;

 SPDR = data; /\* Write data to SPI data register \*/

 while(!(SPSR & (1<<SPIF))); /\* Wait till transmission complete \*/

 flush\_buffer = SPDR; /\* Flush received data \*/

/\* Note: SPIF flag is cleared by first reading SPSR (with SPIF set) and then accessing SPDR hence flush buffer used here to access SPDR after SPSR read \*/

}

**Étapes de lecture du maître SPI**

* Étant donné que l'écriture dans SPDR génère un SCK pour la transmission, écrivez des données factices dans le registre SPDR.
* Attendez la fin de la transmission, c’est-à-dire le drapeau SPIF d’enquête, jusqu’à ce qu’il devienne élevé.
* Lorsque l'indicateur SPIF est défini, lisez les données reçues demandées dans SPDR.

**Fonction SPI\_Read**

Valeur renvoyée: Renvoie le type de données de caractères reçu.

char SPI\_Read() /\* SPI read data function \*/

{ SPDR = 0xFF;

 while(!(SPSR & (1<<SPIF))); /\* Wait till reception complete \*/

 return(SPDR); /\* Return received data \*/

}

Programme pour périphérique maître SPI

/\*

 \* ATmega16\_Master.c

 \* http://www.electronicwings.com

 \*/

#define F\_CPU 8000000UL /\* Define CPU Frequency 8MHz \*/

#include <avr/io.h> /\* Include AVR std. library file \*/

#include <util/delay.h> /\* Include Delay header file \*/

#include <stdio.h> /\* Include Std. i/p o/p file \*/

#include <string.h> /\* Include String header file \*/

#include "LCD\_16x2\_H\_file.h" /\* Include LCD header file \*/

#include "SPI\_Master\_H\_file.h" /\* Include SPI master header file \*/

int main(void)

{

 uint8\_t count;

 char buffer[5];

 LCD\_Init();

 SPI\_Init();

 LCD\_String\_xy(1, 0, "Master Device");

 LCD\_String\_xy(2, 0, "Sending Data: ");

 SS\_Enable;

 count = 0;

 while (1) /\* Send Continuous count \*/

 {

 SPI\_Write(count);

 sprintf(buffer, "%d ", count);

 LCD\_String\_xy(2, 13, buffer);

 count++;

 \_delay\_ms(500);

 }

}

Programme maintenant pour un appareil esclave, par exemple ATmega32

**Étapes d'initialisation de l'esclave SPI**

* Définissez la direction des broches MOSI, SCK et SS du périphérique en tant qu’entrée.
* Définissez la direction des broches MISO du périphérique en sortie.
* Activez SPI en mode esclave en définissant le bit SPE et en effaçant le bit MSTR.

**Fonction SPI\_Init**

void SPI\_Init() /\* SPI Initialize function \*/

{

 DDRB &= ~((1<<MOSI)|(1<<SCK)|(1<<SS)); /\* Make MOSI, SCK, SS as

 input pins \*/

 DDRB |= (1<<MISO); /\* Make MISO pin as

 output pin \*/

 SPCR = (1<<SPE); /\* Enable SPI in slave mode \*/

}

**Étapes de transmission esclave SPI**

Il a la même fonction et les mêmes étapes que nous utilisons SPI Write en mode Master.

**Étapes de réception esclave SPI**

* Attendez que SPIF devienne élevé.
* Lire les données reçues du registre SPDR.

**Fonction SPI\_Receive**

Valeur de retour: retourne le type de données de caractères reçu.

char SPI\_Receive() /\* SPI Receive data function \*/

{

 while(!(SPSR & (1<<SPIF))); /\* Wait till reception complete \*/

 return(SPDR); /\* Return received data \*/

}

 Programme pour périphérique esclave SPI

/\*

 \* ATmega32\_SPI\_Slave.c

 \* http://www.electronicwings.com

 \*/

#define F\_CPU 8000000UL /\* Define CPU Frequency 8MHz \*/

#include <avr/io.h> /\* Include AVR std. library file \*/

#include <util/delay.h> /\* Include Delay header file \*/

#include <stdio.h> /\* Include std. i/p o/p file \*/

#include <string.h> /\* Include string header file \*/

#include "LCD\_16x2\_H\_file.h" /\* Include LCD header file \*/

#include "SPI\_Slave\_H\_file.h" /\* Include SPI slave header file \*/

int main(void)

{

 uint8\_t count;

 char buffer[5];

 LCD\_Init();

 SPI\_Init();

 LCD\_String\_xy(1, 0, "Slave Device");

 LCD\_String\_xy(2, 0, "Receive Data: ");

 while (1) /\* Receive count continuous \*/

 {

 count = SPI\_Receive();

 sprintf(buffer, "%d ", count);

 LCD\_String\_xy(2, 13, buffer);

 }

}

Arduino SPI

L'interface série périphérique (SPI) est un protocole de données série synchrone utilisé par les microcontrôleurs pour communiquer rapidement avec un ou plusieurs périphériques sur de courtes distances. Il peut également être utilisé pour la communication entre deux microcontrôleurs.

Avec une connexion SPI, il y a toujours un périphérique maître (généralement un microcontrôleur) qui contrôle les périphériques. En règle générale, il existe trois lignes communes à tous les périphériques:

* MISO (Master In Slave Out) - La ligne Slave pour l’envoi de données au maître,
* MOSI (Master Out Slave In) - La ligne Master d’envoi de données aux périphériques,
* SCK (Serial Clock) - Les impulsions d'horloge qui synchronisent la transmission de données générée par le maître

et une ligne spécifique à chaque appareil:

* SS (Slave Select) - la broche sur chaque périphérique que le maître peut utiliser pour activer et désactiver des périphériques spécifiques.

Lorsque la broche Slave Select d’un périphérique est faible, il communique avec le maître. Quand il est haut, il ignore le maître. Cela vous permet d'avoir plusieurs périphériques SPI partageant les mêmes lignes MISO, MOSI et CLK.

Pour écrire du code pour un nouveau périphérique SPI, vous devez noter quelques points:

* Arduino utilisera automatiquement la meilleure vitesse, qui est égale ou inférieure au nombre que vous utilisez avec SPISettings .
* Les données sont-elles décalées en premier (MSB) ou au bit le moins significatif (LSB)? Ceci est contrôlé par le deuxième paramètre SPISettings , MSBFIRST ou LSBFIRST. La plupart des puces SPI utilisent le premier ordre de données MSB.
* L'horloge de données est-elle inactive à l'état haut ou bas? Les échantillons sont-ils sur le front montant ou descendant des impulsions d'horloge? Ces modes sont contrôlés par le troisième paramètre de SPISettings .

De manière générale, il existe quatre modes de transmission. Ces modes contrôlent si les données entrent et sortent sur le front montant ou descendant du signal d'horloge de données (appelé phase d' horloge ) et si l'horloge est inactive à l'état haut ou bas (appelée polarité de l'horloge ). Les quatre modes combinent la polarité et la phase en fonction de ce tableau:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mode | Clock Polarity (CPOL) | Clock Phase (CPHA) | Output Edge | Data Capture |
| SPI\_MODE0 | 0 | 0 | Falling | Rising |
| SPI\_MODE1 | 0 | 1 | Rising | Falling |
| SPI\_MODE2 | 1 | 0 | Rising | Falling |
| SPI\_MODE3 | 1 | 1 | Falling | Rising |

Une fois que vous avez défini vos paramètres SPI, utilisez SPI.beginTransaction () pour commencer à utiliser le port SPI. Le port SPI sera configuré avec tous vos paramètres. Le moyen le plus simple et le plus efficace d'utiliser SPISettings se trouve directement dans SPI.beginTransaction ().

Par exemple:

SPI.beginTransaction(SPISettings(14000000, MSBFIRST, SPI\_MODE0));

Si d'autres bibliothèques utilisent SPI à partir d'interruptions, elles ne pourront pas accéder à SPI jusqu'à ce que vous appeliez SPI.endTransaction(). Les paramètres SPI sont appliqués au début de la transaction et SPI.endTransaction() ne modifient pas lesparamètres SPI. Sauf si vous ou une bibliothèque appelez beginTransaction une seconde fois, les paramètres sont conservés . Vous devriez essayer de minimiser le temps qui s'écoule avant d'appeler SPI.endTransaction(), pour une meilleure compatibilité si votre programme est utilisé avec d'autres bibliothèques qui utilisent SPI.

Avec la plupart des périphériques SPI SPI.beginTransaction(), vous écrivez ensuite la broche LOW de sélection d’esclave, appelez SPI.transfer()un nombre quelconque de fois pour transférer des données, puis écrivez la broche SS HAUT et, enfin, appelez SPI.endTransaction().

Exemples

* [Capteur de pression barométrique](https://www.arduino.cc/en/Tutorial/BarometricPressureSensor) : lisez la pression atmosphérique et la température d'un capteur à l'aide du protocole SPI.
* [Contrôle numérique du pot](https://www.arduino.cc/en/Tutorial/DigitalPotControl) : Contrôlez un potentiomètre numérique AD5206 en utilisant le protocole SPI.

Les connexions à la carte Arduino sont très simples à faire. Considérez le brochage du MCP4162 :



Vdd est connecté au 5 V, Vss à la masse GND, /CS au connecteur D10, SCK au connecteur D13, SDI au connecteur D11 et enfin SDO au connecteur D12 de l'Arduino Uno. Maintenant, on fait varier la sortie du MCP4162 avec le programme suivant :

*/\**

 *Example 34.1 - SPI bus demo using a Microchip MCP4162 digital potentiometer*

 *http://tronixstuff.com/tutorials > chapter 34 | CC by-sa-nc | John Boxall*

*\*/*

#include "SPI.h" // necessary library

**int** ss = 10; // using digital pin 10 for SPI slave select

**int** del = 200; // used for various delays

**void** setup()

{

 pinMode(ss, OUTPUT); // we use this for SS pin

 SPI.begin(); // wake up the SPI bus.

 SPI.setBitOrder(MSBFIRST);

 // our MCP4162 requires data to be sent MSB (most significant byte) first

}

**void** setValue(**int** value)

{

 digitalWrite(ss, LOW);

 SPI.transfer(0); // send command byte

 SPI.transfer(value); // send value (0~255)

 digitalWrite(ss, HIGH);

}

**void** loop()

{

 **for** (**int** a = 0; a < 256; a++)

 {

 setValue(a);

 delay(del);

 }

 **for** (**int** a = 255; a >= 0; --a)

 {

 setValue(a);

 delay(del);

 }

}

I2C dans AVR ATmega16 / ATmega32

Introduction

I2C (Inter Integrated Circuit) est un protocole de connexion d'interface de bus série. Il est également appelé TWI (interface à deux fils) car il utilise seulement deux fils pour la communication, deux fils appelés SDA (données série) et SCL (horloge série). L’ATmega16 / ATmega32 à base d’AVR possède un module TWI composé de plusieurs sous-modules, comme indiqué sur la figure.

I2Cworks dans deux modes à savoir,

* Mode maître
* Mode esclave



**Module ATmega16 / 32 I2C**

Broches SDA et SCL

* Ces broches sont utilisées pour interfacer les périphériques externes et le microcontrôleur basés sur TWI.
* Les pilotes de sortie contiennent un limiteur de vitesse de balayage. Les étages d’entrée contiennent une unité de suppression de pointes qui supprime les pointes inférieures à 50 ns.

Unité d'interface de bus

* L'unité d'interface de bus contient la commande marche / arrêt responsable de la génération et de la détection des conditions START, REPEATED START et STOP.
* Le registre à décalage TWDR add / data contient des données à transmettre et à recevoir.
* Le bit ACK reçoit un accusé de réception / transfert en mode émetteur et est généré par le logiciel en mode de réception.
* L'unité de suppression de pointes prend en charge les pointes tandis que l'état du bus de surveillance surveille en continu la détection d'arbitrage et en informe l'unité de contrôle.

Unité de correspondance d'adresse

En mode esclave, l’unité de correspondance d’adresses reçoit l’adresse 7 bits entrante et la compare à l’adresse du registre TWAR (registre d’adresses à deux fils) pour vérifier si elle correspond ou non. Lorsqu’une correspondance est détectée, il est intime que la centrale prenne les mesures nécessaires. Il considère également l'adresse d'appel générale si le bit TWGCE dans TWAR est activé.

Unité de génération de débit

Le générateur de débit binaire contrôle la période SCL en mode maître pour générer la fréquence SCL. Il est calculé par,

Fréquence SCL = (fréquence CLK de la CPU) / (16 + 2 (TWBR) \* 4 ^ TWPS)

Où TWPS est la valeur d'un bit de pré-détartreur dans TWSR.

Unité de contrôle

* L'unité de contrôle contient TWSR (registre d'état TWI), TWCR (registre de contrôle TWI).
* Il contrôle le processus global d'attention pour les événements nécessaires, identifiant les événements qui se produisent, l'assertion d'interruption TWINT et la mise à jour de TWSR.
* Tant que le drapeau TWINT est réglé, SCL est maintenu bas. TWINT défini chaque fois que TWI termine la tâche en cours.

Voyons les registres dans le module ATmega16 / 32 I2C

TWBR: registre de débit binaire TWI

Registre de débit TWI utilisé pour générer la fréquence SCL tout en fonctionnant en mode maître



TWCR: Registre de contrôle TWI

Résistance de contrôle TWI utilisée pour contrôler les événements de toutes les communications I2C.



**Bit 7 - TWINT:** interruption TWI

* Ce bit est activé chaque fois que TWI termine son événement actuel (comme démarrer, arrêter, transmettre, recevoir, etc.).
* Tandis que le bit I dans SREG et le bit TWIE dans TWCR sont activés, le vecteur d'interruption TWI est appelé chaque fois qu'une interruption TWI se produit.
* **L'indicateur d'interruption TWI doit être effacé par le logiciel en y écrivant une logique.**Ce bit n'est pas automatiquement effacé par le matériel.

**Bit 6 - TWEA:** bit d'accusé de réception d'activation TWI

* Il s'agit du bit d'activation de l'accusé de réception TWI. Il est défini en mode récepteur pour générer un accusé de réception et effacé en mode émission.

**Bit 5 - TWSTA:** bit de condition TWI START

* Le dispositif maître a activé ce bit pour générer la condition START en surveillant l’état du bus libre afin de prendre le contrôle sur le bus TWI.

**Bit 4 - TWSTO:** bit de condition TWI STOP

* Le dispositif maître a activé ce bit pour générer une condition STOP afin de laisser le contrôle sur le bus TWI.

**Bit 3 - TWWC:** collision d'écriture TWI

* Ce bit est activé lors de l'écriture dans le registre TWDR avant la fin de la transmission actuelle, c.-à-d. Que TWINT est faible.

**Bit 2 - TWEN:** bit d'activation TWI

* Ce bit est activé pour activer l'interface TWI dans le périphérique et prend le contrôle des broches d'E / S.

**Bit 1** - **Réservé**

**Bit 0 - TWIE:** activation d'interruption TWI

* Ce bit est utilisé pour activer la routine d'interruption TWI alors que le bit I de SREG est défini tant que l'indicateur TWINT est élevé.

**TWSR: Registre d'état TWI**



**Bit 7: Bit 3 - TWS7: TWS3:** bits d'état TWI

* Les bits d'état TWI indiquent l'état de la commande et du bus TWI

**Bit 1: 0 - TWPS1: TWPS0:** bits de pré- **détartrage** TWI

* TWI bits de pré-détartreur utilisés dans la formule de débit pour calculer la fréquence SCL

| **TWPS1** | **TWPS0** | **Exposant** | **Valeur du détartreur** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 4 |
| 1 | 0 | 2 | 16 |
| 1 | 1 | 3 | 64 |

**TWDR: registre de données TWI**

* TWDR contient des données à transmettre ou à recevoir.
* Ce n'est pas accessible en écriture tant que TWI est en train de déplacer un octet.
* Les données restent stables tant que TWINT est défini.



**TWAR: registre d'adresses TWI**

* Le registre TWAR contient l'adresse de l'unité TWI en mode esclave.
* Il est principalement utilisé dans les systèmes multi-maîtres.



**Bit 7: 1 - TWA6: TWA0:** bits d'adresse TWI

* Les bits d'adresse TWI contiennent une adresse TWI de 7 bits avec laquelle il peut être appelé par d'autres maîtres en mode esclave.

**Bit 0 - TWGCE:** bit d'activation d'appel général TWI

* Le bit d'activation d'appel général TWI, lorsqu'il est activé, permet la reconnaissance de l'appel général sur le bus TWI

Il existe quatre modes de transmission dans I2C dans lesquels le périphérique I2C fonctionne.

* Lorsque l'appareil est maître, il fonctionne en modes de transmission MT et MR.
* Et lorsque l'appareil est esclave, il fonctionne en modes de transmission ST et SR.

| **SR No.** | **Mode de transmission** | **Opération** |
| --- | --- | --- |
| 1 | Émetteur maître (MT) | Le dispositif maître écrit des données sur SDA. |
| 2 | Maître Récepteur (MR) | Le dispositif maître lit les données à partir de SDA. |
| 3 | Émetteur Esclave (ST) | Le périphérique esclave écrit des données sur SDA. |
| 4 | Récepteur esclave (SR) | Le périphérique esclave lit les données à partir de SDA. |

 Programmation de l'ATmega32 I2C en mode maître

Pour programmer l'AVR ATmega32 I2C en mode maître, vous devez d'abord initialiser le module I2C avec une fréquence SCL, puis exécuter une opération de lecture et d'écriture à l'aide des événements START, REPEATED START, STOP. Commençons par initialiser le module TWI dans ATmega16.

**Initialisation I2C**

Pour initialiserATmega16 TWI en mode maître, nous devons définir la fréquence d'horloge SCL en définissant les valeurs dans le registre TWBR et les bits TWPS dans le registre TWSR.

La valeur TWBR est définie par la formule de fréquence SCL ci-dessus. par exemple

/\* Define bit rate \*/

#define BITRATE(TWSR) ((F\_CPU/SCL\_CLK)-16)/(2\*pow(4,(TWSR&((1<<TWPS0)|(1<<TWPS1)))))

Maintenant, chargez cette valeur dans le registre TWBR

void I2C\_Init() /\* I2C initialize function \*/

{ TWBR = BITRATE(TWSR=0x00); /\* Get bit rate register value by formula \*/

}

**Événements I2C**

Comme l’AVR I2C est orienté octet et basé sur des interruptions, c’est-à-dire des interruptions émises après chaque événement de bus. Des événements tels que Démarrer, Écrire, Lire, Arrêter. Le statut est également mis à jour après chaque événement de bus. Voyons donc les événements et conditions I2C avec leurs codes de fonction.

Événements I2C en mode Maître

**DÉPARTS)**

* Condition START émise par le maître.
* Il est généré par la transition haut à bas sur SDA alors que SCL est élevé.
* Lorsque le bus est libre, le périphérique maître émet une condition START avec l'adresse du périphérique esclave (SLA) pour lancer la transmission.

**Fonction I2C\_Start**

Cette fonction initie la condition START

Argument d'entrée: - il a l'argument d'entrée de l'adresse d'écriture du périphérique esclave (SLA + W).

Retour: - renvoie l'état de l'événement.

uint8\_t I2C\_Start(charwrite\_address)/\* I2C start function \*/

{

 uint8\_t status; /\* Declare variable \*/

 TWCR=(1<<TWSTA)|(1<<TWEN)|(1<<TWINT); /\* Enable TWI, generate START \*/

 while(!(TWCR&(1<<TWINT))); /\* Wait until TWI finish its current job \*/

 status=TWSR&0xF8; /\* Read TWI status register \*/

 if(status!=0x08) /\* Check weather START transmitted or not? \*/

 return 0; /\* Return 0 to indicate start condition fail \*/

 TWDR=write\_address; /\* Write SLA+W in TWI data register \*/

 TWCR=(1<<TWEN)|(1<<TWINT); /\* Enable TWI & clear interrupt flag \*/

 while(!(TWCR&(1<<TWINT))); /\* Wait until TWI finish its current job \*/

 status=TWSR&0xF8; /\* Read TWI status register \*/

 if(status==0x18) /\* Check for SLA+W transmitted &ack received \*/

 return 1; /\* Return 1 to indicate ack received \*/

 if(status==0x20) /\* Check for SLA+W transmitted &nack received \*/

 return 2; /\* Return 2 to indicate nack received \*/

 else

 return3; /\* Else return 3 to indicate SLA+W failed \*/

}

**DÉPART RÉPÉTÉ (Sr)**

* Condition de départ répété émise par le maître.
* Il est généré en émettant une autre condition START pour lancer une nouvelle transmission.
* Une condition de démarrage répété avec une adresse d'appareil esclave (SLA) est émise entre les conditions de démarrage et d'arrêt

**Fonction I2C\_Repeated\_Start**

Cette fonction génère une condition de démarrage répété pour l'opération de lecture.

Argument d'entrée: - il a l'argument d'entrée de l'adresse de lecture du dispositif esclave (SLA + R).

Retour: - renvoie l'état de l'événement.

uint8\_t I2C\_Repeated\_Start(charread\_address) /\* I2C repeated start function \*/

{

 uint8\_t status; /\* Declare variable \*/

 TWCR=(1<<TWSTA)|(1<<TWEN)|(1<<TWINT);/\* Enable TWI, generate start \*/

 while(!(TWCR&(1<<TWINT))); /\* Wait until TWI finish its current job \*/

 status=TWSR&0xF8; /\* Read TWI status register \*/

 if(status!=0x10) /\* Check for repeated start transmitted \*/

 return 0; /\* Return 0 for repeated start condition fail \*/

 TWDR=read\_address; /\* Write SLA+R in TWI data register \*/

 TWCR=(1<<TWEN)|(1<<TWINT); /\* Enable TWI and clear interrupt flag \*/

 while(!(TWCR&(1<<TWINT))); /\* Wait until TWI finish its current job \*/

 status=TWSR&0xF8; /\* Read TWI status register \*/

 if(status==0x40) /\* Check for SLA+R transmitted &ack received \*/

 return 1; /\* Return 1 to indicate ack received \*/

 if(status==0x48) /\* Check for SLA+R transmitted &nack received \*/

 return 2; /\* Return 2 to indicate nack received \*/

 else

 return 3; /\* Else return 3 to indicate SLA+W failed \*/

}

**ÉCRIRE (W)**

* L'événement de données / adresse WRITE est émis par le maître après l'acquittement réussi de START reçu du périphérique esclave.

**Fonction I2C\_Write**

Cette fonction écrit les données / adresses sur le bus

Argument d'entrée: - il a un argument d'entrée de données / adresse.

Retour: - renvoie l'état de l'événement.

uint8\_t I2C\_Write(chardata) /\* I2C write function \*/

{ uint8\_tstatus; /\* Declare variable \*/

 TWDR=data; /\* Copy data in TWI data register \*/

 TWCR=(1<<TWEN)|(1<<TWINT); /\* Enable TWI and clear interrupt flag \*/

 while(!(TWCR&(1<<TWINT))); /\* Wait until TWI finish its current job \*/

 status=TWSR&0xF8; /\* Read TWI status register \*/

 if(status==0x28) /\* Check for data transmitted &ack received \*/

 return 0; /\* Return 0 to indicate ack received \*/

 if(status==0x30) /\* Check for data transmitted &nack received \*/

 return 1; /\* Return 1 to indicate nack received \*/

 else

 return 2; /\* Else return 2 for data transmission failure \*/

}

**LIRE (R)**

* L'événement de lecture de données est émis par le maître après la réussite de la condition de répétition du démarrage.

**Fonction I2C\_Read\_Ack**

Cette fonction lit les données disponibles sur la ligne SDA et renvoie son accusé de réception au périphérique esclave lorsque les données ont été lues avec succès. Elle demande également à l'esclave de transmettre d'autres données.

Argument d'entrée: - il n'a pas d'argument d'entrée.

Return: - renvoie les données reçues.

char I2C\_Read\_Ack() /\* I2C read ack function \*/

{ TWCR=(1<<TWEN)|(1<<TWINT)|(1<<TWEA); /\* Enable TWI, generation of ack \*/

 while(!(TWCR&(1<<TWINT))); /\* Wait until TWI finish its current job \*/

 returnTWDR; /\* Return received data \*/

}

**Fonction I2C\_Read\_Nack**

Cette fonction lit le dernier octet de données nécessaire disponible sur la ligne SDA mais ne renvoie pas d'accusé de réception. Auparavant, il indiquait à un esclave que son maître ne voulait pas de données à venir et voulait arrêter la communication.

Argument d'entrée: - il n'a pas d'argument d'entrée.

Retour: - renvoie les données reçues.

char I2C\_Read\_Nack() /\* I2C read nack function \*/

{

 TWCR=(1<<TWEN)|(1<<TWINT); /\* Enable TWI and clear interrupt flag \*/

 while(!(TWCR&(1<<TWINT))); /\* Wait until TWI finish its current job \*/

 return TWDR; /\* Return received data \*/

}

**STOP (P)**

* Événement STOP émis par le maître pour indiquer qu’il doit s’arrêter.
* Il est généré par une transition faible à élevée sur SDA alors que SCL est élevé.

**Fonction I2C\_Stop**

Cette fonction initie la condition STOP

Argument d'entrée: - il n'a pas d'argument d'entrée.

Return: - il ne renvoie aucun type de données.

void I2C\_Stop() /\* I2C stop function \*/

{

 TWCR=(1<<TWSTO)|(1<<TWINT)|(1<<TWEN);/\* Enable TWI, generate stop \*/

 while(TWCR&(1<<TWSTO)); /\* Wait until stop condition execution \*/

}

Exemple

Prenons l'exemple du premier transmetteur principal (MT) et des modes de récepteur maître (MR). Considérez ATmega16 en tant que mémoire maître et mémoire EEPROM en tant qu'esclave. Nous pouvons écrire des données sur EEPROM en mode émetteur principal (MT) et les lire à partir de EEPROM en mode récepteur maître (MR).

* Ici, ATmega16 est un appareil maître et l'EEPROM est un appareil esclave.

**Étapes de programmation pour l'opération d'écriture:**

1. Initialiser I2C.
2. Générer une condition START.
3. Envoyez l'adresse d'écriture du périphérique esclave (SLA + W) et vérifiez l'accusé de réception.
4. Écrivez l'adresse de l'emplacement de la mémoire pour les périphériques de mémoire sur lesquels nous voulons écrire.
5. Ecrire les données jusqu'au dernier octet.
6. Générer une condition STOP.

**Étapes de programmation pour l'opération de lecture:**

1. Initialiser I2C.
2. Générer une condition START.
3. Écrire l'adresse du périphérique (SLA + W) et vérifier l'accusé de réception.
4. Écrire l'adresse de l'emplacement de la mémoire pour les périphériques de mémoire.
5. Génère une condition de démarrage répété.
6. Lire les données et renvoyer l'accusé de réception.
7. Retour Pas d'accusé de réception pour le dernier octet.
8. Générer une condition STOP.

Programme

/\*

 \* ATmega16\_Master\_I2C.c

 \* http://www.electronicwings.com

 \*/

#define F\_CPU 8000000UL /\* Define CPU clock Frequency 8MHz \*/

#include <avr/io.h> /\* Include AVR std. library file \*/

#include <util/delay.h> /\* Include Delay header file \*/

#include <string.h> /\* Include string header file \*/

#include "I2C\_Master\_H\_file.h" /\* Include I2C header file \*/

#include "LCD\_16x2\_H\_file.h" /\* Include LCD header file \*/

#define EEPROM\_Write\_Addess 0xA0

#define EEPROM\_Read\_Addess 0xA1

int main(void)

{

 char array[10] = "test"; /\* Declare array to be print \*/

 LCD\_Init(); /\* Initialize LCD \*/

 I2C\_Init(); /\* Initialize I2C \*/

 I2C\_Start(EEPROM\_Write\_Addess);/\* Start I2C with device write address \*/

 I2C\_Write(0x00); /\* Write start memory address for data write \*/

 for (int i = 0; i<strlen(array); i++)/\* Write array data \*/

 {

 I2C\_Write(array[i]);

 }

 I2C\_Stop(); /\* Stop I2C \*/

 \_delay\_ms(10);

 I2C\_Start(EEPROM\_Write\_Addess);/\* Start I2C with device write address \*/

 I2C\_Write(0x00); /\* Write start memory address \*/

 I2C\_Repeated\_Start(EEPROM\_Read\_Addess);/\* Repeat start I2C SLA+R \*/

 for (int i = 0; i<strlen(array); i++)/\* Read data with acknowledgment \*/

 {

 LCD\_Char(I2C\_Read\_Ack());

 }

 I2C\_Read\_Nack(); /\* Read flush data with nack \*/

 I2C\_Stop(); /\* Stop I2C \*/

 return 0;

}

Programmation de l'ATmega32 I2C en mode esclave

Pour programmer l'AVR basé sur ATMéga16 I2C en mode esclave, nous devons initialiser le module TWI. En mode esclave, l’appareil ne peut pas générer les conditions START, REPEATED START ou STOP. Un périphérique esclave doit toujours écouter le bus TWI pour être adressé par un périphérique maître ou un appel général. Laissez initialiser le module TWI en mode esclave.

**Fonction d'initialisation de l'esclave I2C**

Ici, pour initialiser ATmega16 en mode esclave, nous devons attribuer une adresse de périphérique à 7 bits dans le registre TWAR. Après l'attribution de l'adresse, nous devons activer TWI et le bit d'accusé de réception dans TWCR. Et **effacez le drapeau d’interruption TWI en y écrivant la logique 1.**

Argument d'entrée: - il possède un argument d'entrée d'adresse d'esclave.

Return: - il ne renvoie aucun type de données.

void I2C\_Slave\_Init(uint8\_tslave\_address)

{ TWAR=slave\_address; /\* Assign Address in TWI address register \*/

 TWCR=(1<<TWEN)|(1<<TWEA)|(1<<TWINT);/\* Enable TWI, Enable ack generation \*/

}

Voyons maintenant que des événements se sont produits en mode esclave,

**Écoutez le bus**

* Le périphérique esclave doit toujours écouter le bus TWI pour vérifier s’il est adressé par un maître quelconque.
* Quand il est adressé, le bit TWINT est activé. Donc besoin de surveiller le bit TWINT.

**Fonction I2C\_Slave\_Listen**

Argument d'entrée: - il n'a pas d'argument d'entrée.

Retour: - renvoie l'état de l'événement.

int8\_t I2C\_Slave\_Listen()

{ while(1)

 { uint8\_t status; /\* Declare variable \*/

 while(!(TWCR&(1<<TWINT))); /\* Wait to be addressed \*/

 status=TWSR&0xF8; /\* Read TWI status register \*/

 if(status==0x60||status==0x68) /\* Own SLA+W received &ack returned \*/

 return 0; /\* Return 0 to indicate ack returned \*/

 if(status==0xA8||status==0xB0) /\* Own SLA+R received &ack returned \*/

 return 1; /\* Return 0 to indicate ack returned \*/

 if(status==0x70||status==0x78) /\* General call received &ack returned \*/

 return 2; /\* Return 1 to indicate ack returned \*/

 else

 continue; /\* Else continue \*/

 }}

**Transmettre des données**

* Après avoir été adressé par le maître avec l'adresse SLA + R, l'esclave doit envoyer les données demandées.
* Les données à envoyer doivent être écrites dans le registre TWDR.
* Après l’écriture des données, activez TWI avec accusé de réception et désactivez le drapeau d’interruption.
* Si l'accusé de réception n'a pas été reçu du maître, l'appareil esclave effacera l'indicateur TWINT et écoutera à nouveau le bus.
* De même, si REPEATED START / STOP est reçu, le dispositif esclave effacera le drapeau TWINT et écoutera à nouveau le bus.

**Fonction I2C\_Slave\_Transmit**

Argument d'entrée: - il a l'argument d'entrée des données de caractère à envoyer.

Retour: - renvoie l'état de l'événement.

int8\_t I2C\_Slave\_Transmit(chardata)

{ uint8\_t status;

 TWDR=data; /\* Write data to TWDR to be transmitted \*/

 TWCR=(1<<TWEN)|(1<<TWINT)|(1<<TWEA);/\* Enable TWI & clear interrupt flag \*/

 while(!(TWCR&(1<<TWINT))); /\* Wait until TWI finish its current job \*/

 status=TWSR&0xF8; /\* Read TWI status register \*/

 if(status==0xA0) /\* Check for STOP/REPEATED START received \*/

 {

 TWCR|=(1<<TWINT); /\* Clear interrupt flag & return -1 \*/

 return -1;

 }

 if(status==0xB8) /\* Check for data transmitted &ack received \*/

 return0; /\* If yes then return 0 \*/

 if(status==0xC0) /\* Check for data transmitted &nack received \*/

 {

 TWCR|=(1<<TWINT); /\* Clear interrupt flag & return -2 \*/

 return -2; }

 if(status==0xC8) /\* Last byte transmitted with ack received \*/

 return -3; /\* If yes then return -3 \*/

 else /\* else return -4 \*/

 return -4;

}

**Recevoir des données**

* Une fois adressé par le maître avec l'adresse SLA + W, l'esclave doit recevoir les données envoyées par le maître.
* Après chaque octet reçu, l’esclave doit renvoyer un accusé de réception à son sujet.
* Si REPEATED START / STOP est reçu, l’appareil esclave effacera le drapeau TWINT et écoutera à nouveau le bus.

**Fonction I2C\_Slave\_Recieve**

Argument d'entrée: - il n'a pas d'argument d'entrée.

Retour: - renvoie les données reçues ou l'état de l'événement.

char I2C\_Slave\_Receive()

{

 uint8\_tstatus; /\* Declare variable \*/

 TWCR=(1<<TWEN)|(1<<TWEA)|(1<<TWINT);/\* Enable TWI & generation of ack \*/

 while(!(TWCR&(1<<TWINT))); /\* Wait until TWI finish its current job \*/

 status=TWSR&0xF8; /\* Read TWI status register \*/

 if(status==0x80||status==0x90)/\* Check for data received &ack returned \*/

 return TWDR; /\* If yes then return received data \*/

/\* Check for data received, nack returned & switched to not addressed slave mode \*/

 if(status==0x88||status==0x98)

 return TWDR; /\* If yes then return received data \*/

 if(status==0xA0) /\* Check wether STOP/REPEATED START \*/

 { TWCR|=(1<<TWINT); /\* Clear interrupt flag & return -1 \*/

 return -1; }

 else

 return -2; /\* Else return -2 \*/

}

Exemple

Prenons un exemple, ici ATmega16 en tant que périphérique maître et ATmega32 en tant que périphérique esclave. Le premier appareil maître envoie le compte à l’appareil esclave, puis le même maître lit l’appareil esclave.

**Schéma d'interface**



**Communication maître esclave ATmega I2C**

**Étapes de programmation dans l'appareil maître**

1. Initialiser I2C.
2. Générer une condition START.
3. Écrire l'adresse du périphérique (SLA + W) et vérifier l'accusé de réception.
4. Après l'accusé de réception, écrivez les données sur l'esclave.
5. Générez une condition de démarrage répété avec SLA + R.
6. Recevoir des données de l'appareil esclave.

**Étapes de programmation dans un appareil esclave**

1. Initialise I2C avec l'adresse du périphérique esclave.
2. Écoutez le bus pour que le maître vous adresse.
3. Lorsque le périphérique maître adresse SLA + W, recevez les données de ce dernier.
4. Renvoyer l'accusé de réception après chaque octet reçu.
5. Efface le drapeau d'interruption après réception de REPEATED START / STOP.
6. Imprimer les données reçues sur l’écran LCD.
7. Écoutez à nouveau le bus pour que le maître vous adresse.
8. Une fois adressé avec SLA + R par le périphérique maître, transmettez les données au périphérique maître.
9. Transmettez les données jusqu'à ce que NACK / REPEATED START / STOP soit reçu du maître.
10. Efface le drapeau d'interruption après réception de NACK / REPEATED START / STOP.

Programme pour appareil maître

/\*

 \* ATmega16\_Master.c

 \* http://www.electronicwings.com

 \*

 \*/

#define F\_CPU 8000000UL /\* Define CPU clock Frequency 8MHz \*/

#include <avr/io.h> /\* Include AVR std. library file \*/

#include <util/delay.h> /\* Include inbuilt defined Delay header file \*/

#include <stdio.h> /\* Include standard I/O header file \*/

#include <string.h> /\* Include string header file \*/

#include "I2C\_Master\_H\_file.h" /\* Include I2C header file \*/

#include "LCD\_16x2\_H\_file.h" /\* Include LCD header file \*/

#define Slave\_Write\_Address 0x20

#define Slave\_Read\_Address 0x21

#define count 10

int main()

{ char buffer[10];

 LCD\_Init(); /\* Initialize LCD \*/

 I2C\_Init(); /\* Initialize I2C \*/

 LCD\_String\_xy(1, 0, "Master Device");

 while (1)

 { LCD\_String\_xy(2, 0, "Sending : ");

 I2C\_Start\_Wait(Slave\_Write\_Address);/\* Start I2C with SLA+W \*/

 \_delay\_ms(5);

 for (uint8\_t i = 0; i < count ; i++)

 { sprintf(buffer, "%d ",i);

 LCD\_String\_xy(2, 13, buffer);

 I2C\_Write(i); /\* Send Incrementing count \*/

 \_delay\_ms(500);

 }

 LCD\_String\_xy(2, 0, "Receiving : ");

 I2C\_Repeated\_Start(Slave\_Read\_Address);/\* Repeated Start with SLA+R \*/

 \_delay\_ms(5);

 for (uint8\_t i = 0; i < count; i++)

 { if(i < count - 1)

 sprintf(buffer, "%d ", I2C\_Read\_Ack());/\* Read & Ack of data \*/

 else

 sprintf(buffer, "%d ", I2C\_Read\_Nack());/\* Read & Nack to data \*/

 LCD\_String\_xy(2, 13, buffer);

 \_delay\_ms(500);

 }

 I2C\_Stop(); /\* Stop I2C \*/

 }

}

Programme pour appareil esclave

/\*

 \* ATmega32\_Slave.c

 \* http://www.electronicwings.com

 \*

 \*/

#define F\_CPU 8000000UL /\* Define CPU clock Frequency 8MHz \*/

#include <avr/io.h> /\* Include AVR std. library file \*/

#include <util/delay.h> /\* Include inbuilt defined Delay header file \*/

#include <stdio.h> /\* Include standard I/O header file \*/

#include <string.h> /\* Include string header file \*/

#include "LCD\_16x2\_H\_file.h" /\* Include LCD header file \*/

#include "I2C\_Slave\_H\_File.h" /\* Include I2C slave header file \*/

#define Slave\_Address 0x20

int main(void)

{

 char buffer[10];

 int8\_t count = 0;

 LCD\_Init();

 I2C\_Slave\_Init(Slave\_Address);

 LCD\_String\_xy(1, 0, "Slave Device");

 while (1)

 { switch(I2C\_Slave\_Listen()) /\* Check for SLA+W or SLA+R \*/

 { case 0:{

 LCD\_String\_xy(2, 0, "Receiving : ");

 do

 { sprintf(buffer, "%d ", count);

 LCD\_String\_xy(2, 13, buffer);

 count = I2C\_Slave\_Receive();/\* Receive data byte\*/

 } while (count != -1);/\* Receive until STOP/REPEATED START \*/

 count = 0;

 break;

 }

 case 1:

 { int8\_t Ack\_status;

 LCD\_String\_xy(2, 0, "Sending : ");

 do

 {

 Ack\_status = I2C\_Slave\_Transmit(count);/\* Send data byte \*/

 sprintf(buffer, "%d ",count);

 LCD\_String\_xy(2, 13, buffer);

 count++;

 } while (Ack\_status == 0);/\* Send until Ack is receive \*/

 break;

 }

 default:

 break;

 }

 }

}

Arduino I2C

Intoduction

I2C (Inter-Integrated Circuit) est un protocole de connexion d'interface de bus série. Il est également appelé TWI (interface à deux fils) car il utilise seulement deux fils pour la communication. Ces deux fils sont SDA (données série) et SCL (horloge série).

I2C est un protocole de communication basé sur les accusés de réception, c’est-à-dire que l’émetteur recherche un accusé de réception du récepteur après la transmission des données pour savoir si les données ont été reçues correctement par le récepteur.

I2C a deux modes à savoir,

* Mode maître
* Mode esclave

**Le** fil **SDA** (données série) est utilisé pour l’échange de données entre les appareils maître et esclave.

**SCL** (horloge série) est utilisée pour l’horloge synchrone entre le maître et l’esclave.

L'appareil maître établit la communication avec un appareil esclave. L'adresse d'un appareil esclave est nécessaire pour engager la conversation. Le périphérique esclave répond à un périphérique maître lorsqu'il est adressé par un périphérique maître.

Le périphérique I2C a une adresse unique 7 bits ou 10 bits. Ainsi, pour accéder à ces périphériques, un maître doit les adresser par l’adresse unique 7 bits ou 10 bits.

I2C est utilisé dans de nombreuses applications telles que la lecture de l'horloge temps réel (RTC), l'accès à la mémoire externe EEPROM. Il est également utilisé dans des modules de capteurs tels qu'un gyroscope, un magnétomètre, etc.

Le protocole I2C utilise 2 lignes pour la communication:

1. **Horloge série (SCL):** C'est un signal d'horloge. Les données seront envoyées à d'autres appareils lors d'un événement d'horloge. Seul le périphérique maître a le contrôle de cette ligne SCL.
2. **Données série (SDA):** Il s'agit d'une ligne de données série utilisée pour échanger des données entre un appareil maître et un appareil esclave.



**Communication maître-esclave I2C**

Le bus I2C est une configuration à drain ouvert, ce qui signifie qu’ils peuvent tirer la ligne de signal correspondante vers le bas mais ne peuvent pas l’alourdir. Par conséquent, la ligne passera à l'état inconnu. Pour éviter cela, les résistances de rappel doivent être connectées aux broches SCL et SDA.

Arduino I2C Pins

La carte Arduino Uno ne comporte qu'un seul module I2C, mais elle fournit ces lignes SDA et SCL à deux emplacements différents.
**Remarque: lors de la** communication avec des périphériques utilisant le protocole de communication I2C, des résistances de rappel doivent être utilisées. La valeur des résistances de rappel peut varier en fonction des dispositifs utilisés.

Fonctions I2C pour Arduino

**Wire.write (data)**

Il est utilisé pour écrire (transmettre) des données sur le périphérique maître ou esclave.

**Paramètre**

datail peut s'agir d'une valeur à un octet, d'une chaîne de caractères, d'un tableau de données.

**Résultats :            Nombre d'octets écrits.**

**par exemple,**Wire.write (7); // envoie un octet de données

Wire.write («i2c»); // envoie une chaîne à un périphérique esclave

Wire.write (a, 6); // ici a est un tableau

**Wire.available ()**

Cette fonction est utilisée par un maître ou un esclave pour vérifier si les données demandées sont disponibles ou non. Il retourne le no. d'octets disponibles.

**Wire.read ()**

Il est utilisé pour lire les données demandées par le maître à partir de l'esclave ou pour lire les données transmises d'un maître à un esclave.

Fonctions pour Arduino I2C Master

**Remarque:** chaque périphérique esclave I2C a une adresse unique. Lors de la communication via le protocole I2C, cette adresse esclave doit être utilisée par le maître.

**Remarque:** chaque périphérique esclave I2C a une adresse unique. Lors de la communication via le protocole I2C, cette adresse esclave doit être utilisée par le maître.

Arduino a **Wire Library** qui nous permet de communiquer avec des périphériques I2C.

**Wire.begin ()** Il initie la bibliothèque Wire et rejoint le bus en tant que maître.

**Wire.beginTransmission (**  slave address **)**

Cette fonction commence une transmission avec le périphérique esclave I2C ayant une adresse esclave spécifiée.

            slave address Adresse 7 bits du périphérique avec lequel nous souhaitons communiquer.

Par exemple, Wire.beginTransmission (50) // commence la transmission avec l'esclave ayant l'adresse 50

**Câble. requestDe (**Address**,** No. of byte **)** OU

**Câble. requestFrom (adresse, no d'octet, stop)**

Cette fonction est utilisée par le maître pour demander ou recevoir des données d'un périphérique esclave. Les données demandées peuvent être lues à l’aide de Wire.read ().

**Paramètres**

                        Address adresse de l'appareil avec lequel nous voulons communiquer

                        No. of byte non. d'octet doit demander

                        Stop C'est un booléen.

***true*** - envoie un message d'arrêt après la demande, libérant le bus,

***false*** - envoie continuellement un redémarrage après la demande, en maintenant la connexion active

**Résultats :**       Nombre d'octets renvoyés par un périphérique esclave.

**par exemple,** Wire.requestFrom (50, 4) // request 4 no. d'octets d'esclave ayant l'adresse 50

        Wire.requestFrom (50, 4, true) // cessera de recevoir des données après 4 octets, libérant le bus

**Wire.endTransmission ()**

Il met fin à une transmission vers un périphérique esclave commencée par beginTransmission () et transmet les octets mis en file d'attente par write ().

**Résultats :**                         Retour octet indiquant l'état de la transmission

Fonctions pour Arduino I2C Slave

**Wire.begin (adresse)**

Il initie la bibliothèque Wire et rejoint le bus I2C en tant qu'esclave avec l'adresse spécifiée.

**Paramètre :**       Address Adresse d'esclave 7 bits, si elle n'est pas spécifiée, rejoignez le bus en tant que maître

**Wire.onReceive (gestionnaire)**

La fonction de gestionnaire à appeler lorsqu'un périphérique esclave reçoit une donnée transmise d'un maître.

**par exemple**

void setup() {

 Wire.begin(8); // join i2c bus with address #8

 Wire.onReceive(receiveEvent); // register event

 Serial.begin(9600); // start serial for output

}

void receiveEvent (int howmany){

 while (1 < Wire.available()) { // loop through all but the last

 char c = Wire.read(); // receive byte as a character

 Serial.print(c); // print the character

 }

}

**Wire.onRequest (gestionnaire)**

La fonction de gestionnaire à appeler lorsque le maître demande à un périphérique esclave, ne prend aucun paramètre et ne renvoie rien.

**par exemple**

void setup() {

 Wire.begin(8); // join i2c bus with address #8

 Wire.onRequest(requestEvent); // register event

}

void loop() {

 delay(100);

}

// function that executes whenever data is requested by master

// this function is registered as an event, see setup ()

void requestEvent() {

 Wire.write("hello "); // respond with message of 6 bytes

 // as expected by master

}

 Référencées :

1/ Fiche technique ATmega16 et ATmega32

2/ www.arduino.cc

3/ http://www.electronicwings.com