



Transferts I2C

obsolete voir www.didel.com/I2CWire.pdf juin 2017

Autres nouveaux documents:

www.didel.com/I2Cbb.pdf

www.didel.com/OledI2C.pdf

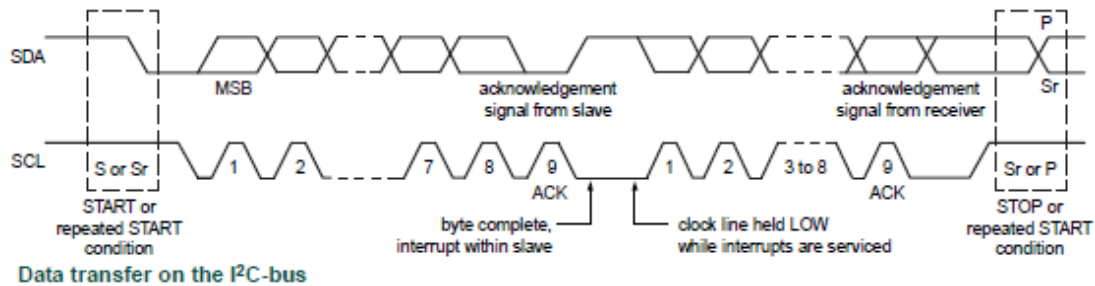
www.didel.com/I2CTwi.pdf

www.didel.com/I2CTwiSlave.pdf

La documentation complète sur le bus I2C est en

http://www.nxp.com/documents/user_manual/UM10204.pdf

La documentation SMBus précise les formats à utiliser avec la librairie Python utilisée en particulier sous Raspberry. http://smbus.org/specs/SMBus_3_0_20141220.pdf



Des librairies existent sur chaque environnement pour commander des périphérique I2C.

Arduino utilise la librairie Wire, qui charge la mémoire de 2000 bytes.

Les librairies plus légères proposées par Didel n'ont pas de tampon intermédiaire et sont 10 fois plus petite.

Librairie wire.h

La librairie Arduino <wire.h>, cache plusieurs caractéristiques du circuit par rapport à la documentation. Il faut comprendre que, avec cette librairie, le circuit a une adresse globale 7 bits. Le 8e bit qui définit la lecture ou l'écriture est géré par des fonctions de nom différent.

De même l'acknowledge est géré implicitement. Les fonctions `wire` sont les suivantes

Wire.begin(); Appelé dans le set-up pour charger la librairie (maître par défaut)

Wire.beginTransmission(Adr1307); Envoie l'adresse

Wire.write(value); Ecrit une adresse ou un byte – peut se répéter

Wire.endTransmission(); Termine l'envoi en écriture simple

Wire.endTransmission(false); Sépare l'envoi de la commande et la lecture du data

Wire.requestFrom(address, quantity); demande des bytes et les mets dans le tampon

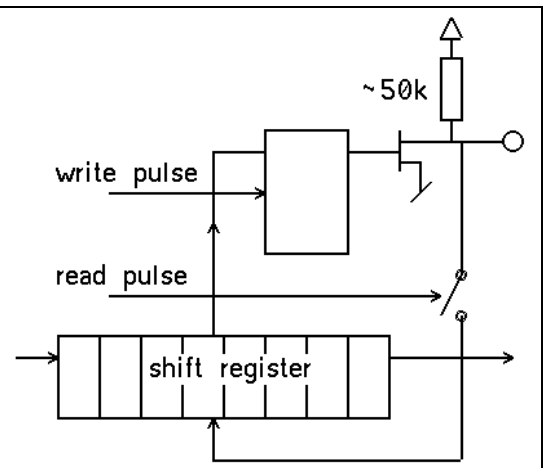
while (Wire.available()) { . . si on attend plusieurs mots, vide le tampon

byte x = Wire.read(); lit dans le tampon

Comme exemple, commandons le circuit 8574 qui offre 8 lignes d'entrée sortie.

Le 8574 répond à l'adresse 7 bits 0x20 si ses lignes A2A1A0 sont à zéro. Le 8e bit vaut 0 (écriture) ou 1 (lecture). donc certains disent que l'adresse est 0x40/0x41.

Les 8 sorties sont en collecteur ouvert: En écrivant un zéro, on peut "tirer" un courant de 20 mA. En écrivant un un, on permet à un signal extérieurs d'imposer un 1 ou un zéro, et de le lire.



Par exemple, si on câble 4 sorties (leds active à zéro) sur les lignes de poids faible, et 4 entrées sur les lignes de poids fort (poussoirs), on a le programme suivant pour copier les 4 poussoirs sur le Leds:

Préparation	Lecture
<pre>#include <Wire.h> #define A8574 0x20</pre>	<pre>Wire.requestFrom(A8574, 1); da = Wire.read() ; da << = 4 ;</pre>
Set-up, direction des pins	Copie
<pre>Wire.begin(); Wire.beginTransmission(0x20); Wire.write(0xF0); Wire.endTransmission();</pre>	<pre>Wire.beginTransmission(A8574); Wire.write(da); Wire.endTransmission();</pre>

Si l'écriture de 0xF0 n'est pas faite dans le set-up, la première lecture sera erronée, puisque la direction est donnée dans l'écriture. Il faut s'assurer dans chaque écriture que les 4 bits de poids fort, sur lesquels on a câblé ici des poussoirs, sont à un.

Exemples de fonctions avec commande (souvent appelé mode)

Ecriture 2 bytes - commande Speed SetSpeed (vitLeft, vitRight);	Lecture 2 bytes - commande PosL EncoL = GetPos ();
<pre>void SetSpeed (int8_t sl, int8_t sr) { Wire.beginTransmission(AdX); Wire.write (Speed); Wire.write(sl); Wire.write(sr); Wire.endTransmission(); }</pre>	<pre>uint16_t GetPosL () { uint16_t dd; Wire.beginTransmission(AdX); Wire.write(PosL); Wire.endTransmission(false); Wire.requestFrom(AdX, 2); dd = (Wire.read()<<8); dd = dd Wire.read(); return dd; }</pre>

Remarque: l'utilisation de Wire.available(); n'est pas nécessaire quand on sait exactement ce que l'on attend et on sait que l'accès ne sera pas retardé. La librairie Wire met les caractères qui arrivent dans un tampon et on peut aller les chercher dans ce tampon jusqu'à ce qu'il soit vide.

Commande directe (bit-banging)

On peut écrire facilement les routines I2C pour un maître, que l'on peut aussi utiliser avec des processeurs de petites taille puisque le logiciel utilise environ 300 bytes. Les routines sont simples, conviennent pour un Tiny programmé en C et la version pour assembleur Pic 12F/16F n'utilise que 180 bytes.

Voir www.didel.com/diduino/I2Cbb.pdf pour une description "bit banging" qui permet de programmer I2C sur n'importe quelle ligne d'un microcontrôleur.

Librairie Wire (extrait du web)

Wire.beginTransmission(address)

Description

Begin a transmission to the I2C slave device with the given address. Subsequently, queue bytes for transmission with the [write\(\)](#) function and transmit them by calling [endTransmission\(\)](#).

Parameters

address: the 7-bit address of the device to transmit to

Returns

None

write()

Description

Writes data from a slave device in response to a request from a master, or queues bytes for transmission from a master to slave device (in-between calls to beginTransmission() and endTransmission()).

Syntax

Wire.write(value)

Wire.write(string)

Wire.write(data, length)

Parameters

value: a value to send as a single byte
string: a string to send as a series of bytes
data: an array of data to send as bytes
length: the number of bytes to transmit

Returns

byte: write() will return the number of bytes written, though reading that number is optional

Wire.requestFrom()

Description

Used by the master to request bytes from a slave device. The bytes may then be retrieved with the [available\(\)](#) and [read\(\)](#) functions.

As of Arduino 1.0.1, requestFrom() accepts a boolean argument changing its behavior for compatibility with certain I2C devices.

If true, requestFrom() sends a stop message after the request, releasing the I2C bus.

If false, requestFrom() sends a restart message after the request. The bus will not be released, which prevents another master device from requesting between messages. This allows one master device to send multiple requests while in control.

The default value is true.

Syntax

```
Wire.requestFrom(address, quantity)  
Wire.requestFrom(address, quantity, stop)
```

Parameters

address: the 7-bit address of the device to request bytes from

quantity: the number of bytes to request

stop : boolean. true will send a stop message after the request, releasing the bus. false will continually send a restart after the request, keeping the connection active.

Returns

byte : the number of bytes returned from the slave device

read()

Description

Reads a byte that was transmitted from a slave device to a master after a call to [requestFrom\(\)](#) or was transmitted from a master to a slave. read() inherits from the [Stream](#) utility class.

Syntax

```
Wire.read()
```

Parameters

none

Returns

The next byte received

Wire.available()

Description

Returns the number of bytes available for retrieval with [read\(\)](#). This should be called on a master device after a call to [requestFrom\(\)](#) or on a slave inside the [onReceive\(\)](#) handler.

available() inherits from the [Stream](#) utility class.

Parameters

None

Returns

The number of bytes available for reading.

Wire.endTransmission()

Description

Ends a transmission to a slave device that was begun by [beginTransmission\(\)](#) and transmits the bytes that were queued by [write\(\)](#).

As of Arduino 1.0.1, endTransmission() accepts a boolean argument changing its behavior for compatibility with certain I2C devices.

If true, endTransmission() sends a stop message after transmission, releasing the I2C bus.

If false, endTransmission() sends a restart message after transmission. The bus will not be released, which prevents another master device from transmitting between messages. This allows one master device to send multiple transmissions while in control.

The default value is true.

Syntax

```
Wire.endTransmission()  
Wire.endTransmission(stop)
```

Parameters

stop : boolean. true will send a stop message, releasing the bus after transmission. false will send a restart, keeping the connection active.

Returns

byte, which indicates the status of the transmission:

- 0:success
- 1:data too long to fit in transmit buffer
- 2:received NACK on transmit of address
- 3:received NACK on transmit of data
- 4:other error