I2C register definition for infrared reciever

Matthieu Vigne

April 2017

Contents

Abstract

The infrared detection system built this year to detect the opponent's robots uses I2C communication to relay information to the master (the Beaglebone). The microcontroller on the infrared receiver circuit (an ATtiny841) is thus an I2C slave, and a list of registers for this slave must be defined. This document is the only reference for the register definition: the microcontroller code MUST correspond to the content of this document. Any change in the register definition must be referenced in this document.

At the end of this document, some information is also provided to help in the understanding of the source code of both the emitter and the receiver.

Unless otherwise specified, any number followed by an 'h' suffix is in hexadecimal, 'b' stands for binary, while no suffix means decimal base is used.

1 Slave address

The I2C slave address has to be defined in the microcontroller code. The address chosen is :

0100 001[R/W], i.e. 42h in write mode

The [R/W] bit defines the operation to be perform: read on logic 1, write on logic 0.

This address was chosen such as to prevent any conflict with all the possible addresses of the other I2C devices currently used: the L3GD20H gyroscope, the LSM303D accelerometer, and the PCA9635 led driver (both reserved addresses as specified in the documentation, and all the addresses possibles given the current constraints created by the PCB routing).

2 Register definition

Table 1: Register definition

Name	**Address**		**Type**	**Default**
	Hex	**Binary**		
WHO_AM_I	00	0000 0000	R	0000 1111 (0Fh)
IR_FREQUENCY	01	0000 0001	R	-
RAW_DATA1	02	0000 0010	R	-
RAW_DATA2	03	0000 0011	R	-
ROBOT1_POS	04	0000 0100	R	-
ROBOT2_POS	05	0000 0101	R	-

3 Register description

3.1 WHO_AM_I (00h)

Table 2: WHO_AM_I register

0	0	0	0	1	1	1	1

Device identification register. Constant value of 0Fh.

3.2 IR_FREQUENCY (01h)

Table 3: IR_FREQUENCY register

0	0	0	0	0	0	0	IR

- IR : type of IR receiver used :
    - 0: 38.4kHz receiver (TSDP34138 and TSDP34338)
    - 1: 57.6kHz receiver (TSDP34156 and TSDP34338)

Currently this register is read only, its value is determined by the switch on the receiver board. It could be made writable, as an additional way to set the sensors to use. Remember that the emitter frequency must match the reciever's frequency, and should be different from the other team's frequency.

3.3 RAW_DATA1 (02h), RAW_DATA2 (03h)

Table 4: RAW_DATA1 register

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

Table 5: RAW_DATA2 register

0	0	0	0	0	0	0	D_8

- D_i : current value of sensor i :
    - 0: detection
    - 1: no detection

Sensors are numbered starting from 0, in the counter-clockwise direction. The first sensor is the one located above the status LED (at the right of the PCB in DesignSpark).

3.4 ROBOT1_POS (04h), ROBOT2_POS (05h)

Table 6: ROBOTx_POS register

S_2	S_1	S_0	P_4	P_3	P_2	P_1	P_0

These registers hold the position and size of the two largest beacon detection zone seen.

- $S_2S_1S_0$b : the size of the detection zone, i.e. the number of receiver that see the beacon. This number can give an idea of how close the other robot is (the closer it is, the more it will be seen by the sensors). A value of 000b means no robot is seen, in which case the data P_i is meaningless. Note that this size cannot go above **6** : if it is the case, the microcontroller **must** consider that both robots are visible by the receiver, and that their detection zone has in fact merged. In that case, the zone will be split in half at its center (rounding up the number of sensors, i.e a zone of 7 means 4 sensors on each side), and each half will correspond to one robot. This should not

happen in regular operation (the beacon should not be close enough to be detected by all sensors).

- $P_4P_3P_2P_1P_0$b : the position of the other robot. The sensors are numbered from 0 to 8: the position of the robot is twice the number of the sensor at the center of the detection zone (this multiplication is to guarantee a whole number when an even number of sensors see the beacon). $P_4P_3P_2P_1P_0$b can thus go from 0 to 17, giving a resolution of 20 degrees / increment. Note that, when programming the ATtiny, special care must be taken to implement correctly detection on both side of the 0th sensor (modulo operation must be applied).

Note : the size of the detection zone stored in ROBOT2_POS register shall always be less than or equal to that of ROBOT1_POS.

Example of register value: imagine sensors 8, 0, 1, 2 and sensors 4, 5 see the emitter's light. There are two detection zones, one of size 4 around 0.5 and one of size 2 around 4.5. Then: ROBOT1_POS = 1000 0001b and ROBOT2_POS = 0100 1001b.

4 Note on emitter and receiver code

4.1 Emitter code

The code of the Attiny85 on the emitter simply consists in generating two signals, a 38.7/57.6kHz square signal (that we will call the carrier even though we are not modulating it), and lower frequency signal, the envelop.

Both signal are generated thanks to hardware timers (and not software, this would not be reliable enough). The carrier is generated on PB0 thanks to TIMER0. The envelop is generated on PB1 thanks to TIMER1. An external AND gate then mixes the signals.

Choosing the envelop was somewhat tricky, because of the automatic gain control (AGC) filters in the TSDP34... See Vishay's application note about this. Because of the AGC, we had to switch to the TSDP341.. series. Remember that the envelop is the signal we will get on the reciever (only inverted).

In order not to trigger the AGC, we had to use a low frequency, low duty cycle signal. Currently, the carrier is a 200Hz signal (5ms period), with a 192us impulsion. This gives a duty cycle of 4%, and a burst length of 11 cycles at 57.6kHz, 7.4 cycles at 58kHz: this signal will thus remain in the passing zone of the filter.

Note that all these values are theoretical : in practice, we see some variations due to errors in clock speed (we are using the internal clock). However, this errors of a few percents has no consequence on the overall working of the device.

4.2 Reciever code

The chip used on the reciever is a Attiny841. For sensor reading, it relies on interrupts: all three interrupts (INT0, PCINT0 and PCINT1) are used, each for reading one multiplexer. The code simply sets the multiplexer, enables interrupts, wait at least 5.2ms, disable interrupts, looks at the result then starts again with the next sensor (we loop through the tree sensors on each multiplexer).

During the interrupt, we time how long the low pulse is, and, if it is close enough to 200us, we consider we have received a valid signal.

In parallel with this code for reading the sensors, I2C communication must be handled. This is done thanks to hardware I2C slave support of the chip, enabled in software thanks to the library WireS, provided by orangkucing on gitlab (https://github.com/orangkucing/WireS). Note that the source code of WireS (WireS.cpp and WireS.h) is already in the folder CodeRecepteur: thus, there is no need to install this library, as it gets compiled with the source code.

During the main loop, an array called I2CRegisters is filled with the correct data. WireS then works with handlers, and return the corresponding array value to the master. Currently all I2C registers are read-only, so no write operation is ever perform, but the write handler is present nonetheless. Note also that sequential reading is supported : as long as the master acknowledges the transaction, the register number is incremented (with loopback at the end), and the next value is returned.