STLinux application note

I²C on STLinux

Introduction

This document aims to resolve some of the common misconceptions regarding the support of I²C on Linux. It provides information on the Linux kernel I²C subsystem and how to use it effectively on the 2.6.x series kernels.

- *Section 1* provides a brief description of the I²C protocol.
- *Section 2* describes the software API in detail.
- *Section 3* describes the efficiency of each type of message.

STLinux only supports the host running as a single I²C master, so multi-master and slave operation (although supported by the hardware) is only described where it causes complications to running as a single master.

Other hardware (for example, Marvell PXA) under Linux does support slave mode. If there is a strong customer requirement then it may be added to the ST support in future.

Contents

1 The I²C protocol

This section describes I²C transactions at a communications protocol level. The physical level is not described, although there is a decription of a simple mapping between the protocol components and the bus-level signaling.

For more details see *The I²C Bus Specification Version 2.1*, NXP document (9398 393 40011), available for download from *http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf*.

1.1 I²C bus

An I²C bus consists of a number of interconnected devices. There can be one or more controlling master devices and a number of slave devices. All communication is controlled by a master device. If more than one master device is present on the bus, only one can be active at a time. Master devices may also be slaves if the hardware supports it.

Each slave device on the bus is identified by a unique device address.

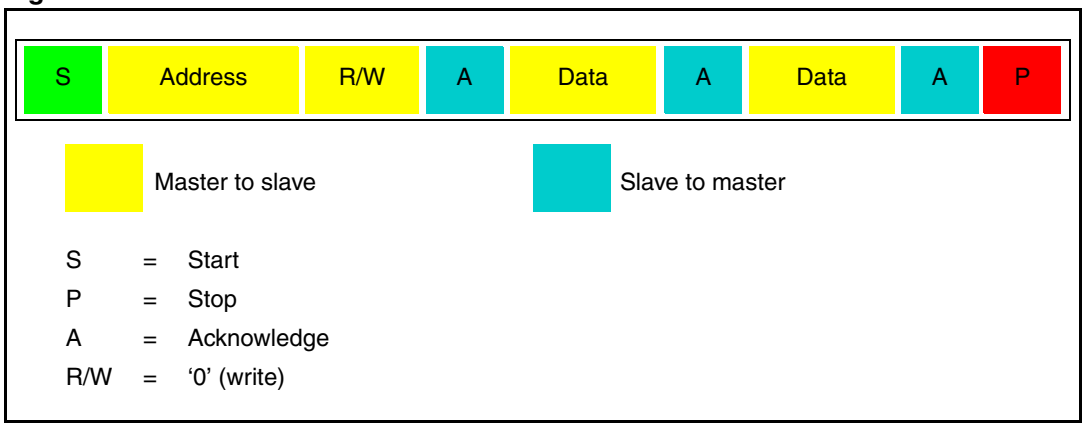
1.2 I²C transactions

I²C messages consist of one or more transactions. I²C transactions consist of a start, a slave address selection, a read/write selection, a data transfer and then a stop. All data transfer is acknowledged by the receiver back to the transmitter.

A data transfer can consist of one or more byte orientated read or write operations, but reads and writes cannot be mixed in a single transaction. Transactions can be combined to provide combined read and write messages (this is called a “data transfer combined format” in the I²C specification). This is described in more detail in the *Section 1.3: Repeated start transactions*.

A write transaction from the master to the slave has the format displayed in *Figure 1*.

Figure 1. Format of a write transaction from master to slave



A read transaction has a similar format, as shown in *Figure 2*.

Figure 2. Format of a read transaction from master to slave

S = Start
P = Stop
A = Acknowledge
Ā = Not acknowledge
R/W = '1' (read)

1.3 Repeated start transactions

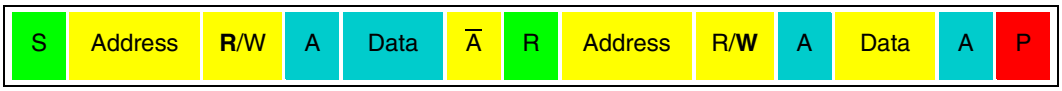
When transactions are combined, the STOP of the first transaction and the START of the second transaction can be replaced by a "repeated START" instead. This gives a small performance increase over multiple transactions as it removes the requirement for the STOP setup time.

In fast mode only, the setup time for a repeated start is half the STOP to START setup time. Therefore in standard mode (100 kHz), 4 µs per transaction is saved. In fast mode (400 kHz), 1.3 µs per transaction is saved. This represents about half a clock cycle or about 2% per single-byte transaction. For multi-byte transactions, there is no significant performance increase.

The main reason for using repeated start transactions is to lock the bus to a particular master in multi-master systems. Another less common reason is to hold an I²C repeater open.

A combined format message consisting of a single byte read and write is shown in *Figure 3*.

Figure 3. Format of a repeated start transaction



S = Start
R = Repeated start
P = Stop
A = Acknowledge
Ā = Not acknowledge

*Note: The slave address is always reissued so it is possible to address different slaves (each slave resets its state on detecting a START condition). For a multiple read or write (but not both) to the **same** address it is **considerably less efficient** than using a single transaction with multiple byte transfers due to the repeated address and the setup time for the repeated start.*

2 STLinux I²C support

In Linux, the I²C bus can be accessed both through user-space and the kernel. This section describes the kernel interface, but the user-space interface follows similar conventions.

An I²C transaction is described by the simple structure `i2c_msg` defined in `include/linux/i2c.h`:

```
/*
 * I2C Message - used for pure I2C transaction,
 * also from /dev interface
 */

struct i2c_msg {
   __u16 addr;    /* slave address  */
   __u16 flags;
   __u16 len;     /* msg length */
   __u8 *buf;     /* pointer to msg data */
};
```

The `flags` field in `i2c_msg` has the settings given in *Table 1*.

Table 1. Values for the flags field

Flag	Value	Description
I2C_M_RD	0x0001	Read data
I2C_M_TEN	0x0010	Use a 10-bit I²C address
I2C_M_RECV_LEN	0x0400	Length is the first received byte
I2C_M_NO_RD_ACK	0x0800	Ignore read acknowledge
I2C_M_IGNORE_NAK	0x1000	Ignore not acknowledge
I2C_M_REV_DIR_ADDR	0x2000	Send a read flag on the bus for a write transaction or a write flag on the bus when a read is requested (This is a work around for broken I²C slave hardware)
I2C_M_NOSTART	0x4000	Do not issue any more START/address after the initial START/address in a combined message (This is a work around for broken I²C slave hardware)

Transactions are then transmitted across the bus using the kernel API:

```
int i2c_transfer(
   struct i2c_adapter *adap,
   struct i2c_msg *msgs,
   int num)
```

The Linux calls required to reproduce the example transactions given in *Section 1.2 on page 3* are described in C-like pseudo-code in *Section 2.1*, *Section 2.2* and *Section 2.3*. Also shown is the output from performing the operations on a real I²C bus (bus 0 on an MB411 fitted with a Cut 4 STi7109). The bus traffic was captured using a hardware I²C analyzer and

a hardware I²C slave emulator set to `slave_address=0x38` which transmitted 0xAB 0xCD on byte reads.

2.1 Example 1: An I²C transaction writing 2 bytes of data

```
struct i2c_msg msg[1];
unsigned char buffer[2];

msg[0].addr = slave_address;
msg[0].flags = 0;
msg[0].len = 2;
msg[0].buf = buffer;

if (i2c_transfer(client.adapter, msg, 1) < 0){
  pr_err("i2c_test: i2c_transfer failed\n");
}
```

Analyzer output:

```
9.27ms        2        SP       W        38       00       00
```

This means a transaction of duration 9.27 ms, 2 bytes, start and stop detected, write, slave address 0x38, data 0x00 0x00 was detected on the bus.

2.2 Example 2: An I²C transaction reading 2 bytes of data

```
struct i2c_msg msg[1];
unsigned char buffer[2];

msg[0].addr = slave_address;
msg[0].flags = I2C_M_RD;
msg[0].len = 2;
msg[0].buf = buffer;

if (i2c_transfer(client.adapter, msg, 1) < 0){
  pr_err("i2c_test: i2c_transfer failed\n");
}
```

Analyzer output:

```
35.0ms        2        SP       R        38       AB       CD*
```

This means a transaction of duration 35 ms, 2 bytes, start and stop detected, read, slave address 0x38, data 0xAB 0xCD with NACK (the `*`) was detected on the bus.

2.3 Example 3: An I²C combined message read 1 byte then write 1 byte of data

```
struct i2c_msg msg[2];
unsigned char buffer[2];

msg[0].addr = slave_address;
msg[0].flags = I2C_M_RD;
msg[0].len = 1;
msg[0].buf = buffer;

msg[1].addr = slave_address;
msg[1].flags = 0;
msg[1].len = 1;
msg[1].buf = buffer;

if (i2c_transfer(client.adapter, msg, 2) < 0){
  pr_err("i2c_test: i2c_transfer failed\n");
}
```

Analyzer output:

```
19.4ms        1        S        R        38AB*

16.0ms        1        SP       W        38       AB
```

The meaning of this transaction is as described in the previous examples.

*Note: The first transaction is **NOT** followed by a STOP (P) as combined messages automatically use repeated start transactions.*

3 Efficient I²C transfers

The I²C bus is a slow external serial bus (100 kHz in Standard Mode, 400 kHz in Fast Mode). *Table 2* shows the efficiency of each type of message.

Table 2. Message efficiency

Message	Efficiency	Reason
A combined message containing transactions with a large amount of data in each transaction.	Highest	Multiple data requires no additional START/address to be transmitted. Multiple transactions in a combined message use a repeated start.
Single transactions containing large amount of data.	High	Multiple data requires no additional START/address to be transmitted but a single transaction results in a STOP/START time gap.
A combined message with small amount of data in each transaction.	Poor	More instances of START/address need to be transmitted but at least a repeated start is used.
Single transactions containing a single byte of data.	Very Poor	STOP transmitted at each transaction so STOP/START time gap required before next transaction. Many instances of probably redundant START/address need to be transmitted.

These guidelines are true for all transactions on an I²C bus and hence have no reliance on any particular operating system, in other words, it is valid for OS21 as well as Linux.

If a device driver is using I²C accesses of the same type (read or write), a small number of bytes and the same slave address then the available bandwidth on the bus is small due to the high protocol overhead. In fact, only about 42% of bits transmitted on the bus are the actual data in the case of single byte transfers. Taking timing into consideration, the available bits per second bus bandwidth will be <30% compared to single transactions containing large amounts of data.

4 Revision history

Table 3. Document revision history

Date	Revision	Changes
12-Aug-2008	A	Initial release.