I²C: A Quick Primer

By JIM ROWE

Developed by Philips over 25 years ago, the I²C bus is now a well-established standard for low to medium-speed data communication between ICs. However, its basic operation still isn't well understood, except by people in the chip design business. Here's a quick primer to bring you up to speed.

BACK ABOUT 1980, when digital ICs were really starting to be used in consumer gear such as car radios, stereo systems and TV sets, Philips developed a low-cost interfacing technique for allowing the various ICs in a system to exchange data at a low to medium speed. They called it the "Inter-Integrated Circuit" or "I-squared-C" (I²C) bus and it rapidly became very widely used for data communication between ICs in all kinds of equipment.

These days, I²C is found not just in consumer audio and video gear but in computer and industrial equipment as well. In fact, it's now used in over 1000 ICs, made by more than 50 different companies worldwide. So it's well worth knowing how this very versatile bus works.

I²C is usually described as a 2-wire synchronous serial bus, although strictly speaking it's a 3-wire bus because the PC board's ground line is also an essential part of the communications link. The two nominal wires or "lines" are the data or SDA line and the clock or SCL line. Both of these lines are regarded as bidirectional and to allow this they are both used in wired-OR or "open drain/open collector" mode.

This means that both lines are connected to the PC board's positive supply rail only via pull-up resistors and can only be pulled down to logic low (or ground potential) by the circuitry inside the chips connected to the bus lines. So strictly speaking, any chip connected to the SDA or SCL lines can pull them down but if none of the chips does so, both lines are pulled high by the external resistors. This is known as the "idle bus" condition, incidentally.

In its most basic form, I²C is used for "master-slave" communication – where one device takes control of the bus as the master, to send data to or request data from another device acting as the slave. It's true that because both the SDA and SCL lines are bidirectional, any device on the bus can initiate a data transfer as the master and similarly, any device can play the role of the slave.

So the I²C bus can operate as a multi-master bus and in fact its official specification provides for bus contention and resolution situations, where two devices try to take control of the bus at the same time. However, in most common applications, there is a single master device (usually a microcontroller) and the rest of the devices are used as slaves. These slave devices may be digital tuning chips, controlled-gain amplifier or filter chips, video switching or processing chips, EEPROM memories and LCD panels, etc.

Fig.1 shows the basic master-slave I²C interface circuit. As shown, there are really three lines between the master and slave chips: SCL, SDA and ground. The external pull-up resistors (Rp) provide the only connections between the SCL and SDA lines and the positive supply rail (+V).

In operation, the SDA line can be pulled down to ground by either the master or slave devices, via the open-drain Mosfets connected internally to their SDA pins. Similarly, both chips can monitor the logic levels on the SDA line, via the Schmitt trigger inverters which are also connected to their SDA pins.

The circuitry connected to the SCL

pin inside each chip can be identical to that connected to the SDA pins. However, if only one device is to act as the master, it only needs an open-drain Mosfet connected to the SCL pin (as shown in Fig.1) because it will always be providing the SCL clock pulses. Conversely, when other devices are only being used as slaves, they only need the Schmitt trigger inverter to "receive" the SCL clock pulses.

The next thing to note is that although Fig.1 only shows a master device with a single slave device, the I²C bus can be used to connect one or more masters to many slave devices – as many as 112 different devices in fact. These slave devices are simply connected to the SCL and SDA lines in "daisy chain" fashion, as shown in Fig.2.

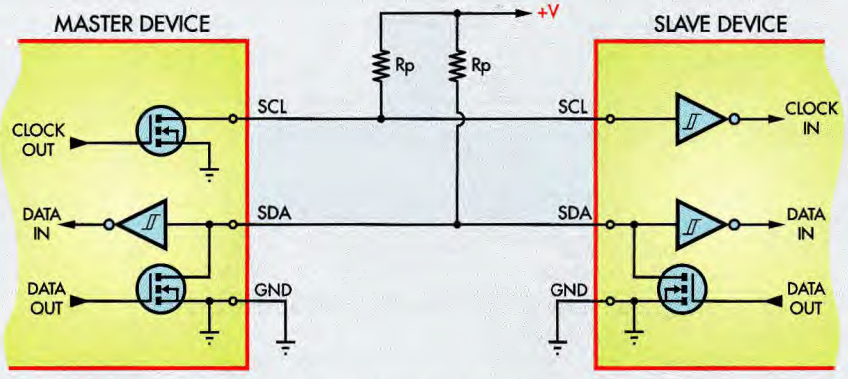

FIG.1: BASIC MASTER-SLAVE I²C INTERFACE

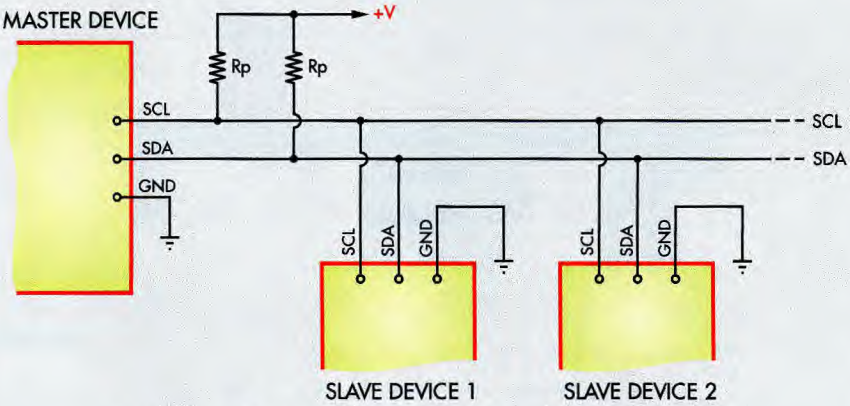

FIG.2: I²C BUS WITH ONE MASTER, MANY SLAVES

Starting and stopping

Before an I²C data transfer operation or sequence can take place, the master device must check to make sure that the bus is idle; ie, both the SCL and SDA lines must be at logic high. If so, it then takes control of the bus by pulling down the SDA line while leaving the SCL line high.

This is described as setting up a "Start condition" and announces to all other devices on the bus that a data transfer is about to take place. The Start condition can be seen on the left in the SCL and SDA waveforms shown in Fig.3(a).

We'll look at the actual data transfer operations in more detail shortly. For the moment, let's look at the way the master device signals the end of a data transfer sequence, by setting up a "Stop condition".

As shown on the right of Fig.3(a), this is simply the reverse of the Start condition. The master device releases control of the SCL line first, so that it goes high, then it releases the SDA line so this also goes high (ie, after the SCL line goes high). Both lines are then high, thus returning the bus to the idle condition.

So this is the basic format of an I²C data transfer sequence: Start condition, the data transfer itself and then the Stop condition – all under the control of the master device.

Now let's look more closely at the fine details.

Data transfer

The master device controls all data transfer on the I²C bus, as this is the device that toggles the SCL line to produce clock pulses – one positive pulse for each data bit, as shown in Fig.3(b). The data bits themselves are conveyed on the SDA line, which must be stable when the SCL line is high. All logic level changes on the SDA line must occur between clock pulses, when the SCL line is low.

Basically, all data is sent over the I²C bus this way, as serial eight-bit bytes with the most significant bit (MSB) first. This is shown in Fig.3(c), which also shows the next important thing you need to know about I²C operation: after each data byte is sent, the receiving device *must* "Acknowledge" that it has been received.

This is normally done by the receiver pulling down the SDA line while the master device provides a ninth clock pulse on the SCL line. Which device pulls down the SDA line to acknowledge reception depends on the direction of data transfer: if the master is sending data to a slave, the slave device must acknowledge. On the other hand, if a slave is sending data to the master, the master itself must acknowledge.

In other words not only is acknowledging mandatory after each byte but it is the receiving device which must do the acknowledging. As noted above, the acknowledge is usually done by pulling the SDA line low during the ninth clock pulse, known as "ACK", but there are some situations where the receiving device acknowledges by leaving the SDA line high. This is known as "ACK-bar" and we'll look at it more closely soon.

Addressing

By now you're probably wondering how the I²C master device can selectively communicate with one particular slave device when there may be a number of slaves on the bus. That's easy: the first data byte sent out by the master after it grabs the bus and sets up the Start condition is an address byte, specifying which slave device

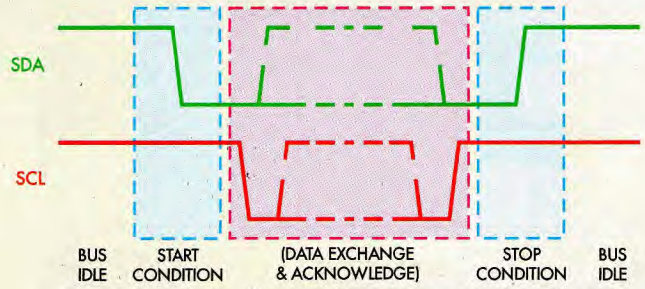

Fig.3(a): BASIC I²C DATA EXCHANGE SEQUENCE

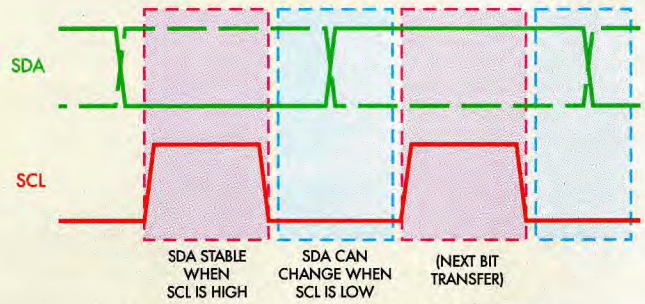

Fig.3(b): DATA (ADDRESS) BIT TRANSFER

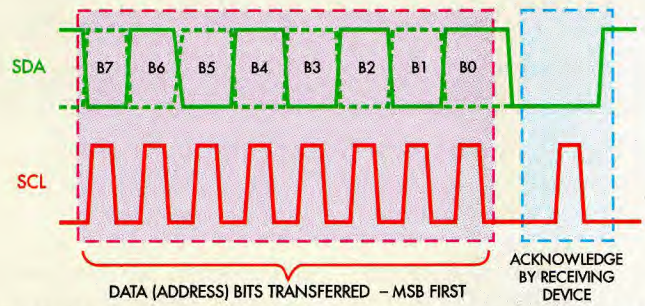

Fig.3(c): DATA (ADDRESS) BYTE TRANSFER WITH ACKNOWLEDGE

it wants to communicate with. It also specifies whether it wants to write data to the device or read back data from it. Basically, the first seven bits of this first byte form the actual slave address, while the eighth bit specifies a read (1) or a write (0) operation.

This addressing scheme is part of the I²C bus specification and devices designed to communicate via the I²C bus are given unique addresses (licensed to the chip maker by NXP, the current name for Philips Semiconductors). In some cases the address is built right inside the chip and can't be changed, while in others it can be set to one of a number of addresses in a range allocated to that device, by tying one or more of its other pins to logic high or low.

The latter arrangement is especially useful for devices like EEPROMs and other memories, where you might want to have a number of them on the same bus. Each device can be given its own unique address to prevent confusion.

Because the I²C address code uses seven bits, this means that in theory you should be able to have a maximum of 128 (2^7) devices connected to the same bus. However, as part of the I²C specification, 16 of the possible address codes are reserved for either special or future purposes, so in practice you can only have a maximum of 112 different chips on the same bus. That's still more than enough to handle the vast majority of situations.

By the way, since the read/write bit effectively forms the eighth bit of the address byte, it's quite valid to regard each device as having two addresses on the I²C bus – one address for writing data to it and the other for reading from it. For example, a device might have an effective write address of say 42h (01000010 binary) and a read address of 43h (01000011). This is really the same device address, with only the final read/write bit changing in value.

By the way, the current I²C specification also provides for 10-bit extended device addressing as an alternative to the 7-bit addressing scheme. However 10-bit device addressing is apparently not used much yet.

Single byte transfers

To make basic I²C operation a little clearer, let's now take a look at Fig.4(a). This shows the sequence of operations involved in a master device taking control of the bus, addressing a particular slave device and then writing a single byte of data to that device.

As you can see, the sequence begins with the master setting up the Start condition (S) and then sending the 7-bit address of the slave it wants to receive the data. It then follows with zero in the eighth bit (R/W-bar) position to indicate that it wants to "write" or send the data.

If that particular slave device is present on the bus and ready to accept the data, it must then respond by pulling the SDA line low (A) before the master sends out the ninth (acknowledge) clock pulse. None of the devices on the bus with other addresses will respond.

Following this acknowledgement, the master then sends out the eight bits of the data byte itself, after which the slave device must respond again with an acknowledge bit. Once the master detects this second acknowledge, it sets the Stop condition (P) to signify the end of the transfer and release the bus lines.

Reading a single data byte from a slave device is very similar. Fig.4(b) shows the details. Here the master device again sets up the Start condition (S) and sends out the 7-bit address for the slave device it wants to read the byte from. Now comes the first difference, because in this case it sends out a '1' for the eighth R/W-bar bit, to indicate that it wants to read back a data byte from the slave.

If the addressed slave is present and ready to send back the data byte, it then

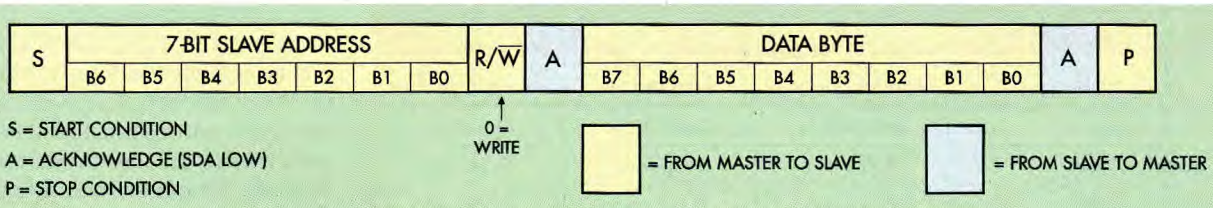

Fig.4(a): MASTER WRITING A SINGLE DATA BYTE TO A SLAVE DEVICE

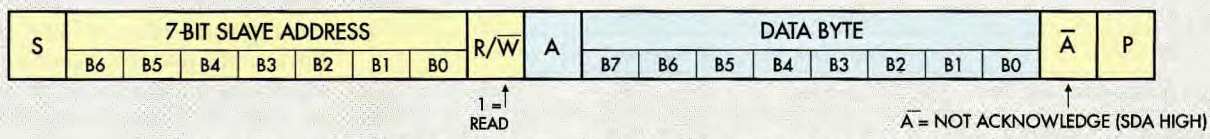

Fig.4(b): MASTER READING A SINGLE DATA BYTE FROM A SLAVE DEVICE

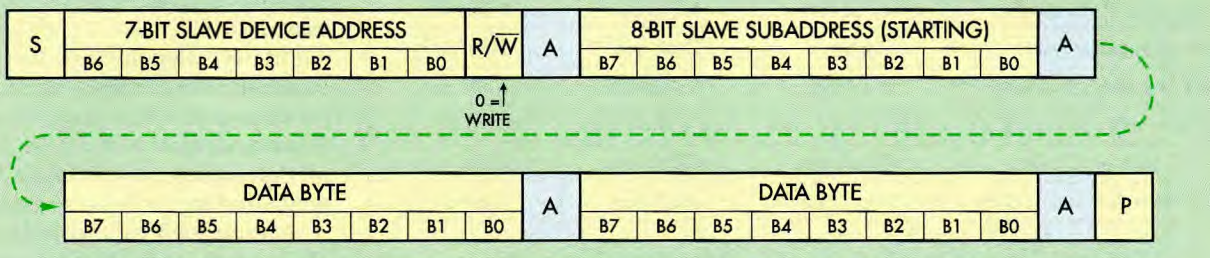

Fig.4(c): MASTER WRITING MULTIPLE DATA BYTES TO SLAVE DEVICE SUBADDRESSES (IN SEQUENCE)

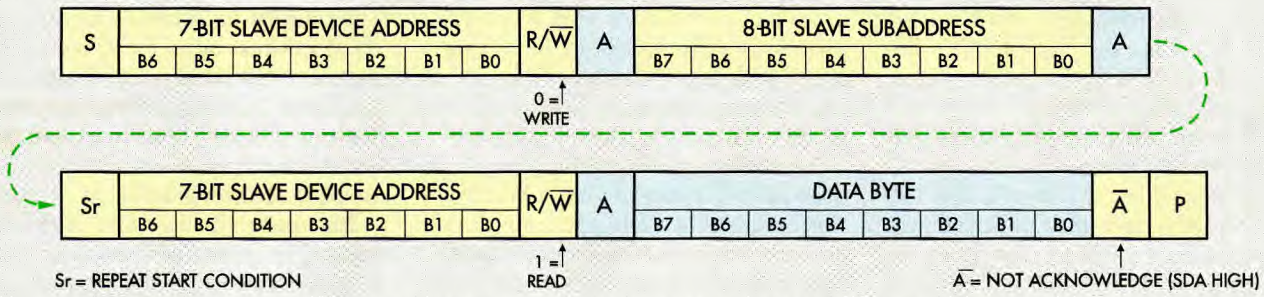

Fig.4(d): MASTER READING A SINGLE DATA BYTE FROM A SLAVE DEVICE SUBADDRESS

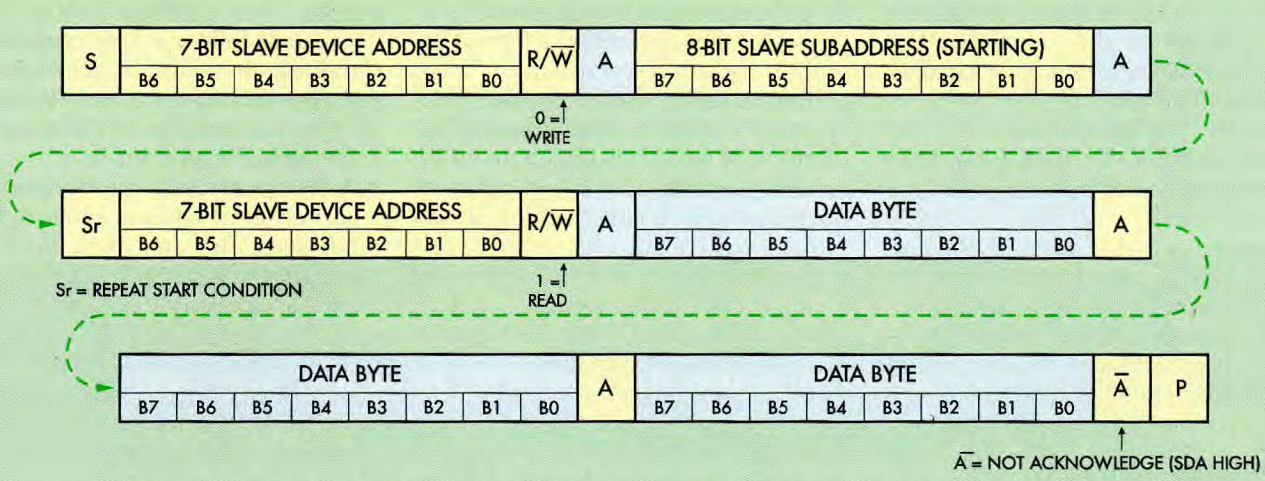

Fig.4(e): MASTER READING MULTIPLE DATA BYTES FROM SLAVE DEVICE SUBADDRESSES (IN SEQUENCE)

Table 1: I²C Timing and Electrical Characteristics

Parameter	Standard-mode	Fast-mode	Fast-mode Plus
SCL clock frequency	0–100kb/s	0–400kb/s	0–1Mb/s
SCL clock low time	4.7µs min	1.3µs min	0.5µs min
SCL clock high time	4.0µs min	0.6µs min	0.26µs min
Setup time, S or Sr condition	4.7µs min	0.6µs min	0.26µs min
Hold time, S or Sr condition	4.0µs min	0.6µs min	0.26µs min
Data setup time	250ns min	100ns min	50ns min
Data valid time	3.45µs max	0.9µs max	0.45µs max
Acknowledge data valid time	3.45µs max	0.9µs max	0.45µs max
Rise time, SCL or SDA sigs	1µs max	300ns max	120ns max
Fall time, SCL or SDA sigs	300ns max	300ns max	120ns max
Setup time, stop (P) condition	4.0µs min	0.6µs min	0.26µs min
Bus free time, P – S cond's	4.7µs min	1.3µs min	0.5µs min
Low level output current	3mA min	3mA min	20mA min
Output low volts (3mA sink)	0.4V max	0.4V max	0.4V max
High level volts, SDA or SCL	Vdd + 0.5V max	Vdd + 0.5V max	Vdd + 0.5V max
Shunt C, SDA or SCL lines	400pF max	400pF max	550pF max
Capacitance for each I/O pin	10pF max	10pF max	10pF max

acknowledges the request by pulling the SDA line low (A) for the ninth clock pulse. Then when the master sends out a further eight clock pulses on the SCL line, the slave toggles the SDA line to transmit the data bits back to the master.

Now comes the second change. Although it's the master that now has to acknowledge that it has received the data byte from the slave, it doesn't do this by pulling the SDA line down on the ninth clock pulse as before for a normal acknowledge (A). Instead, it leaves it high for a not-acknowledge (A-bar). Can you guess why? It's because this is the only way the master can indicate to the slave that the transfer is ending and no further data bytes need be sent.

Finally, the master ends the sequence as before by setting the Stop (P) condition and releasing the bus lines.

Remember that in both Fig.4(a) and Fig.4(b), all the clock pulses on the SCL line are provided by the master device. It also sets the Start and Stop conditions, specifies the slave address and specifies whether the data byte is to be written to the slave or read from it.

Device subaddresses

Although each device connected to an I²C bus has a single main address, the I²C specification does allow it to have multiple secondary or subaddresses. This can be very useful where a device such as an audio or video processor chip needs to have many registers or latches to store its various control parameters, or in the case of a memory device, to store the data.

Since a complete second byte can be used to specify the device subaddress, this means that a complex device can have as many as 256 subaddresses (2^8). This may sound like more than enough but some very complex video processing chips do need over 180 different subaddress registers to store their set-up parameters and status bytes.

Fortunately, devices which do have multiple subaddresses usually have an additional handy feature: a subaddress "pointer" register which automatically increments after each data byte write or read operation. This allows a master device to write a string of data bytes into successive subaddress registers, or read data bytes back from them, in a single multi-byte operation. All it needs to do is specify the starting subaddress first and then send or receive the data bytes one after the other.

Multi-byte transfers

Fig.4(c) shows the sequence of operations involved for a master device to write a number of data bytes into successive subaddresses in a slave device. As you can see, the sequence begins as before with the master setting up the Start condition (S), followed by the main address of the slave device which is to receive the data, plus an R/W-bar bit of zero to indicate a data write.

When the addressed slave device acknowledges (A), the master then sends a second byte specifying the starting subaddress. Then, after the slave acknowledges again, the master simply begins sending the data bytes themselves, one after the other. The slave acknowledges the receipt of each data byte and saves them in consecutive subaddress registers, starting with the subaddress specified by the master.

Finally, after the last data byte has been sent and acknowledged, the master sets up the Stop condition (P) and releases control of the bus. As before, all the clock pulses on the SCL line are generated by the master device.

Reading back a data byte from a single slave device subaddress is similar but with a couple of noteworthy differences – see Fig.4(d).

As before, the master sets up the Start condition (S) and sends the main slave address plus an R/W-bar bit of zero to indicate a write. Then, when the slave acknowledges, the master again sends a second byte specifying the slave subaddress and waits for the slave to acknowledge.

But now things take a different course, because the master now has to set what is known as a "repeat start" condition (Sr), to signal that it is still controlling the bus (note: a repeat start condition is virtually the same as a normal Start condition except that it does not follow a Stop condition). It then sends the main slave address again but this time with an R/W-bar bit of "1" to indicate that it wishes to read a byte rather than write one.

After the slave acknowledges this repeated slave address byte, it then sends back the data byte from the specified subaddress, toggling the SDA line as before in synchronism with the SCL clock pulses from the master. Following the last data bit, the master must acknowledge, of course, but if this is the one and only byte to be read back the master does so not by pulling the SDA line low for a normal acknowledge but by leaving it high for a NACK (A-bar). This is to indicate to

the slave that there are no more bytes to be read back.

Finally it sets up the Stop condition (P) as before, to release control of the bus.

The sequence of events when the master device wants to read back a number of data bytes from consecutive slave subaddresses is very similar. This is shown in Fig.4(e).

Here the master sets the Start condition (S) and sends the slave device address first as before, followed by an R/W-bar bit of zero for writing. Then, when the slave acknowledges, it sends the starting subaddress and waits for the slave to acknowledge again. It then sets a repeat start (Sr) condition and again sends the slave device address, followed by an R/W-bar bit of one to indicate that data will now be read from the slave rather than written to it. After acknowledging (A), the slave then sends back the first byte of data from the specified starting subaddress.

When the master acknowledges that it has received this first data byte by pulling the SDA line down (A), the slave continues to increment its internal subaddress pointer and send back data bytes from consecutive subaddresses. This continues until the master acknowledges the last byte it wishes to receive in the current transfer, by leaving the SDA line high during the ninth clock pulse (ie, A-bar) rather than by pulling it down as for the earlier bytes. This again signals the slave that no more bytes are to be sent back.

Finally, as before, the last step is for the master to set up the Stop condition (P) and release control of the bus.

Other bus events

We have now looked at virtually all of the events that take place on a basic "single master/multiple slaves" I²C interface bus. As most common applications are of this type, you shouldn't have much trouble with them if you've kept up so far.

There are also other kinds of I²C bus events like "clock synchronisation", "arbitration" and "clock stretching" but these mainly come into the picture with more complex multi-master systems – like 10-bit addressing. For the most part, you won't need to worry about them, so I'm not going to try explaining them here.

If you do need to find out more about them, they're covered quite well in NXP's I²C Bus Specification and User Manual, which can be downloaded from the NXP website (**www.nxp.com/acrobat_download/usermanuals/UM10204-3.pdf**).

I²C bus speeds

When Philips first developed the I²C bus, it was only intended for low-speed operation – up to 100kb/s (kilobits per second). However, over the years, the I²C specification has been revised and expanded and nowadays there are a total of four different allowable bus speed modes.

The original 0-100kb/s mode is now known as "Standard Mode", while the other three modes are designated "Fast Mode" (0-400kb/s), "Fast Mode Plus" (0-1Mb/s) and "High Speed Mode" (0-3.4Mb/s).

Table 1 shows the most important electrical characteristics of the three speed modes in common use.

If devices are to be used on an I²C bus running in one of the higher speed modes, they must be given SDA and SCL driver and buffer stages capable of working at those higher speeds. On the other hand, a device which does have higher speed drivers and buffers can always be used on an I²C bus operating in Standard Mode. In fact, this backwards compatibility is part of the I²C specification.

Debugging & troubleshooting

Debugging and troubleshooting of I²C bus circuits can often be done by monitoring bus activity using a dual-trace oscilloscope, with one trace watching the SCL line and the other the SDA line. If the scope is arranged to trigger on a negative-going edge on the SDA line, it will trigger for each Start condition.

However, for tracking down more subtle problems, it may be necessary to use an I²C debugging program running on a PC, linked into the bus via a suitable hardware interface adaptor. This type of program usually allows you to do things like reading data from slave device subaddress registers, or writing data into them to program device operation "on the run".

There are various I²C debugging programs currently available, many of them designed to work with their own USB-I²C hardware interface. These combined software and hardware packages can be fairly pricey though – up to hundreds of dollars in some cases.

Fortunately, for quite a few years now, Philips/NXP has made available a freeware debugging program of their own, called "URD" – short for "Universal Register Debugger". The current version of this is v3.12, which can run on any version of Windows up to Windows XP. It comes as a self-installing file called URD312.EXE and when it installs, it also unpacks two PDF files.

These PDF files provide a User Manual for the program (URDUser.pdf) and a reference manual (URDLang.pdf) for its programming language, which seems to be derived from Microsoft's Visual Basic for Applications.

Hardware interface

URD v3.12 is compatible with a very simple hardware interface which connects to one of the PC's parallel printer ports and comes with a driver for this type of interface. Elsewhere in this issue, you'll find a simple LPT/I²C interface of this type described, so you can build one up yourself to use with the URD program. Together they make a much cheaper alternative to commercial I²C debugging packages. **SC**