certainty that there was at least one transmission error. Parity checks will always detect a single bit error. However, burst bit errors may or may not be detected by parity checking — an even number of errors, for example, will not be detected.

As an example of parity-based error checking, consider wanting to transmit the following 7-bit word: 0101010. Assuming even parity, we would actually transmit the 8-bit word: 01010101, where the least-significant bit is the parity bit. Now, suppose during transmission, a bit gets flipped, so that a receiver receives the following 8-bit word: 11010101. The receiver detects that this word has odd parity; knowing that the word was supposed to have even parity, the receiver determines that this word has an error. Instead, suppose the receiver receives: 11110101. This word has even parity, and so the receiver thinks the word is correct, even though it contains two errors.

*Checksum* is a stronger form of error checking that is applied to a packet of data. A packet of data will contain multiple words of data. Using parity checking, we used one extra bit per word to help us detect errors. Using checksum, we use an extra word per packet for the same purpose. For example, we may compute the XOR sum of all the data words in a packet and send this value along with the data packet. Upon receiving the data packet words and the checksum word, the receiver will compute the XOR sum of all the data words it received. If the computed checksum word equals the received checksum word, the data packet is assumed to be correct. Otherwise, it is assumed to be incorrect. Again, not all error combinations can be detected. We can of course use both parity and checksum for stronger error checking.

As an example, suppose a packet consists of four words: 0000000, 0101010, 1010101, and 0000000. The XOR checksum of these four words is 1111111. A transmitter can thus send that checksum word at the end of the packet. Now, suppose the receiver receives 1000001, 0101010, 1010101, and 0000000. Note that two bits have switched in the first word, and that parity-based error checking would not detect this error. The receiver computes the checksum of this packet and obtains 0111110. This differs from the received checksum of 1111111, and thus the receiver determines that an error has occurred.

Note that errors can also occur in the parity bit or the checksum word itself.

When using parity or checksum error detection, error correction is typically done by a retransmission and acknowledgment protocol. Here, the transmitter sends a data packet and expects to receive an acknowledgment from the receiver indicating that the data packet was received correctly. If an acknowledgment is not received, the transmitter retransmits the data packet and waits for a second acknowledgment.

## 6.9 Serial Protocols

In this section, we describe four popular serial protocols, namely the $I^2C$ protocol, the CAN protocol, the FireWire protocol, and the USB protocol.

### $I^2C$

Philips Semiconductors developed the *Inter-IC*, or $I^2C$, bus nearly 20 years ago. $I^2C$ is a two-wire serial bus protocol. This protocol enables peripheral ICs in electronic systems to

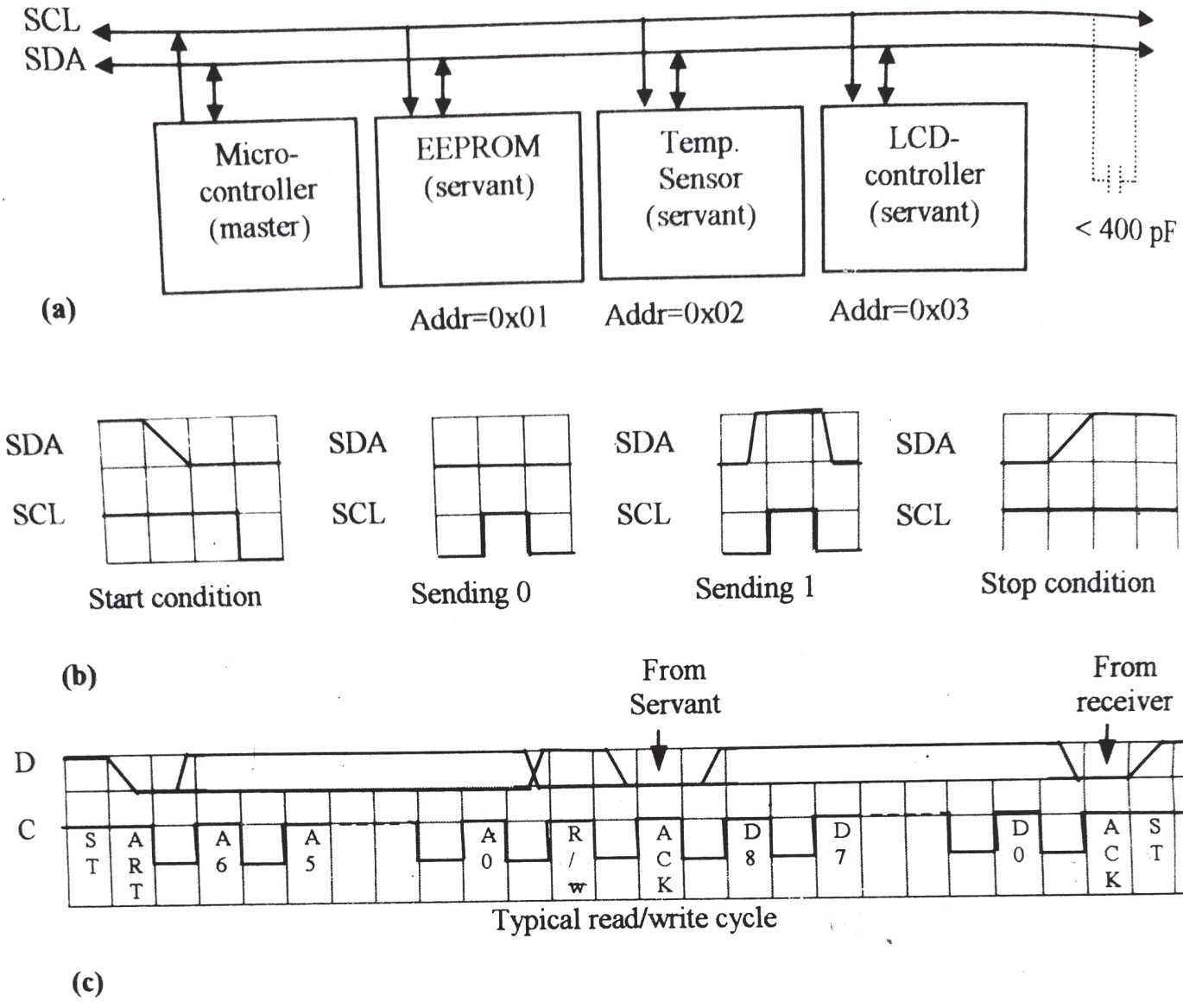


Figure 6.24: $I^2C$ bus structure.

communicate with each other using simple communication hardware. Based on the original specification of the $I^2C$, data transfer rates of up to 100 kbits/s and 7-bit addressing are possible. Seven-bit addressing allows a total of 128 devices to communicate over a shared $I^2C$ bus. With increased data transfer rate requirements, the $I^2C$ specification has been recently enhanced to include fast-mode, 3.4 Mbits/s, with 10-bit addressing. Common devices capable of interfacing to an $I^2C$ bus include EPROMs, Flash and some RAM memory devices, real-time clocks, watchdog timers, and microcontrollers.

A sample $I^2C$ network is depicted in Figure 6.24(a). The bus consists of two wires; a data wire called serial-data-line (*SDA*) and a clock wire called serial-clock-line (*SCL*). The $I^2C$ specification does not limit the length of the bus wires, as long as the total capacitance of the bus remains under 400 pF. In this example, there are four devices attached to the bus. One of these devices, the microcontroller, is a master. The other three devices, a temperature sensor, an EEPROM, and a LCD-controller, are servants. Each of these servant devices is assigned a unique address, as shown in Figure 6.24(a). Only master devices can initiate a data transfer on

an $I^2C$ bus. The protocol does not limit the number of master devices on an $I^2C$ bus, but typically, in a microcontroller-based system, the microcontroller serves as the master. Both master and servant devices can be senders or receivers of data. This will depend on the function of the device. In our example, the microcontroller and EEPROM send and receive data, while the temperature sensor sends data and the LCD-controller receives data. In Figure 6.24(a), arrows connecting the devices to the $I^2C$ bus wires depict the data movement direction. Normally, all servant devices residing on an $I^2C$ assert high-impedance on the bus while the master device maintains logic high, signaling an idle condition.

All data transfers on an $I^2C$ bus are initiated by a start condition. A start condition is shown in Figure 6.24(b). A high to low transition of the *SDA* line while the *SCL* signal is held high signals a start condition. All data transfers on an $I^2C$ bus are terminated by a stop condition. A stop condition is shown in Figure 6.24(b). A low to high transition of the *SDA* line while the *SCL* signal is held high signals a stop condition. Actual data is transferred in between start and stop conditions. A typical $I^2C$ byte write cycle works as follows. The master device initiates the transfer by a start condition. Then, the address of the device that the byte is being written to is sent starting with the most significant down to the least significant bit. Ones and zeros are sent as shown in Figure 6.24(b). Here, the bit value is placed on the *SDA* line by the master device while the *SCL* line is low and maintained stable until after a clock pulse on *SCL*. If performing a write, right after sending the address of the receiving device, the master sends a zero. The receiving device in return acknowledges the transmission by holding the *SDA* line low during the first *ACK* clock cycle. Following the acknowledgment, the master device transmits a byte of data starting with the most significant down to the least significant bit. The receiving device, in this case the servant, acknowledges the reception of data by holding the *SDA* line low during the second *ACK* clock cycle. If performing a read operation, the master initiates the transfer by a start condition, sends the address of the device that is being read, sends a one (logic high on *SDA* line) requesting a read and waits to receive an acknowledgment. Then, the sender sends a byte of data. The receiver, master device in this case, acknowledges the reception of data and terminates the transfer by generating a stop condition. The timing diagram of a typical read/write cycle is depicted in Figure 6.24(c).

## CAN

The *controller area network* (CAN) bus is a serial communication protocol for real-time applications, possibly carried over a twisted pair of wires. This protocol was developed by Robert Bosch GmbH to enable communication among various electronic components of cars as an alternative to expensive and cumbersome wiring harnesses. The robustness of the protocol has expanded its use to many other automation and industrial applications. Some characteristics of the CAN protocol include high-integrity serial data communications, real-time support, data rates of up to 1 Mbit/s, 11-bit addressing, error detection, and confinement capabilities. The CAN protocol is documented in ISO 11898 (for high-speed applications) and ISO 11519-2 (for lower speed applications). Common applications, other than automobiles, using CAN include elevator controllers, copiers, telescopes, production-line control systems, and medical instruments. Among devices that incorporate a CAN interface

are the 8051-compatible 8592 processor, and a variety of standalone CAN controllers, such the 80C200, from Philips.

The CAN specification does not specify the actual layout and structure of the physic bus itself. Instead, it requires that a device connected to the CAN bus is able to transmit detect, on the physical bus, one of two signals called dominant or recessive. For example dominant signal may be represented as logic '0' and recessive as logic '1' on a single da wire. Furthermore, the physical CAN bus must guarantee that if one of two devices asserts dominant signal and another device simultaneously a recessive signal, the dominant signa prevails. Given a physical CAN bus with the above-mentioned properties, the protocol defin data packet format and transmission rules to prioritize messages, guarantee latency time allow for multiple masters, handles transmission errors, retransmit corrupted messages, an distinguish between a permanent failure of a node versus temporary errors.

## FireWire

The *FireWire* (a.k.a. *I-Link* or *Lynx*) is a high-performance serial bus developed by Appl Computer Inc. Because the specification of the FireWire protocol is given by the 1394 IEE designation, many refer to it as the IEEE 1394, or simply 1394. The need for FireWire i driven by the rapidly growing need for mass information transfer. Typical local or wide are networks (LANs/WANs) are incapable of providing cost-effective connection capabilities and do not guarantee bandwidth for real-time applications. Some characteristics of the FireWir protocol include transfer rates of 12.5 to 400 Mbits/s, 64-bit addressing, real-time connection/disconnect and address assignment (a.k.a., plug-and-play capabilities), and packet-based layered design structure.

While $I^2C$ and CAN bus protocols are designed mostly for interfacing ICs, FireWire is designed for interfacing among independent electronic devices (e.g., a desktop computer and a digital scanner). Moreover, FireWire is capable of supporting an entire local-area network similar to one based on Ethernet. The 64-bit wide address space of FireWire is partitioned as 10 bits for network identifiers, 6 bits for node identifiers, and 48 bits for memory address. A local-area network based on FireWire can consist of 1,023 subnetworks, each consisting of 63 nodes, with each node, in turn, addressable by 281 terabytes of distinct locations! FireWire is feasible for applications such as disk drives, printers, scanners, cameras, and many other consumer electronics devices.

## USB

The *Universal Serial Bus* (USB) protocol is designed to make it easier for PC users to connect monitors, printers, digital speakers, modems and input devices like scanners, digital cameras, joysticks, and multimedia game equipment. USB has two data rates, 12 Mbps for devices requiring increased bandwidth, and 1.5 Mbps for lower-speed devices like joysticks and game pads. USB uses a tiered star topology, which means that some USB devices, called USB hubs, can serve as connection ports for other USB peripherals. Only one device needs to be plugged into the PC. Other devices can then be plugged into the hub. USB hubs may be embedded in such devices as monitors, printers and keyboards. Standalone hubs could also be made

available, providing a handful of convenient USB ports right on the desktop. Hubs feature an upstream connection (pointed toward the PC) as well as multiple downstream ports to allow the connection of additional peripheral devices. Up to 127 USB devices can be connected together in this way.

USB host controllers manage and control the driver software and bandwidth required by each peripheral connected to the bus. Users don't need to do a thing, because all the configuration steps happen automatically. The USB host controller even allocates electrical power to the USB devices. Like USB host controllers, USB hubs can detect attachments and detachments of peripherals occurring downstream and supply appropriate levels of power to downstream devices. Since power is distributed through USB cables, with a maximum length of 5 meters, you no longer need a clunky AC power supply box for many devices.

## 6.10 Parallel Protocols

In this section, we briefly describe two popular parallel protocols, namely the PCI bus protocol and the ARM Bus protocol.

### PCI Bus

The *Peripheral Component Interconnect* (PCI) bus is a high-performance bus for interconnecting chips, expansion boards (e.g., a video card that plugs into a main board like a Pentium mother board), and processor memory subsystems. The PCI bus originated at Intel in the early 1990s, was then adopted by the industry as a standard and administered by the PCI Special Interest Group (PCISIG), and was first used in personal computers in 1994 along with Intel 486 processors. The PCI bus has since largely replaced the earlier bus architectures such as the ISA/EISA bus described earlier, and Micro Channel bus protocols. Some characteristics of the PCI bus protocol include transfer rates of 127.2 to 508.6 Mbits/s, 32-bit addressing, synchronous bus architecture (i.e., all transfers take place with respect to a clock signal), and multiplexed 32-bit data/address lines. It must be noted that later additions to the specification of the PCI bus extend the protocol to allow 64-bit data and addressing while maintaining compatibility with the 32-bit schemes.

### ARM Bus

While PCI is a widely used industry standards, many other bus protocols are predominantly designed and used internally by various IC design companies. One such bus is the ARM bus designed by the ARM Corporation and documented in ARM's application note 18v. This bus is designed to interface with the ARM line of processors. The ARM bus supports 32-bit data transfer and 32-bit addressing and, similar to PCI, is implemented using synchronous data transfer architecture. The transfer rate on an ARM bus is not specified and instead is a function of the clock speed used in a particular application. More specifically, if the clock speed of the ARM bus is denoted as $X$, then the transfer rate is $16 \times X$ bits/s.