Q&A – What is CAN?

What is a CAN bus?

A Controller Area Network (CAN) refers to a network of independent controllers. It is a serial communications protocol that efficiently supports distributed real-time control with a very high level of security. The CAN bus standard was developed by Bosch and Intel and the version of the current standard has been in use since 1990.

What is meant when referring to a physical layer, or an application layer?

The International Organization for Standardization (ISO) developed the Open System Interconnect (OSI) model in 1984 as a model of computer communication architecture. There are seven layers to the OSI model: Physical, Data Link, Network, Transport, Session, Presentation, and Application. The intent is that protocols be developed to perform the functions of each layer as needed.

[Source: SAE J1939 Revised Aug 2003]

What is CAN 2.0B?

CAN was divided into different layers according to the ISO/OSI model in order to achieve design transparency and implementation flexibility. CAN 2.0B refers to the specification that defines the Physical Layer and the Data Link Layer for all CAN buses.

How do CAN bus modules communicate?

CAN bus uses two dedicated wires for communication. The wires are called CAN high and CAN low. When the CAN bus is in idle mode, both lines carry 2.5V. When data bits are being transmitted, the CAN high line goes to 3.75V and the CAN low drops to 1.25V, thereby generating a 2.5V differential between the lines. Since communication relies on a voltage differential between the two bus lines, the CAN bus is NOT sensitive to inductive spikes, electrical fields or other noise. This makes CAN bus a reliable choice for networked communications on mobile equipment.

CAN power can be supplied through CAN bus. Or a power supply for the CAN bus modules can be arranged separately. The power supply wiring can be either totally separate from the CAN bus lines (using suitable gauge wiring for each module) resulting in two 2-wire cables being utilized for the network, or it can be integrated into the same cable as the CAN bus lines resulting in a single 4-wire cable. CAN bus cabling is available from multiple vendors.

What is the difference between CANopen and SAE J1939?

These protocols are higher level CAN bus protocols. They both use the CAN 2.0B standard for the physical and data link layer. Each protocol, however, has different standards for the higher layers in the OSI model, and thus the way in which data is transmitted and interpreted is unique to each.

Are there master and slave modules on the CAN bus?

The nature of CAN bus communications allows all modules to transmit and receive data on the bus. Any module can transmit data, which all the rest of the modules receive permitting both peer-to-peer and broadcast data transmissions. In CANopen, the CAN bus normally includes one module acting as a network master which starts the bus communications, but a dedicated master module is not needed. In J1939, the master and slave concept is not used.

What is the practical limit of data throughput on the CAN bus?

CAN bus can use multiple baud rates up to 1 Mbit/s. The most common baud rates are 125 kbit/s (default CANopen) and 250 kbit/s (J1939). The CAN bus communication enables bus loads of up to 100% (data being transmitted all the time and all nodes can transmit), allowing full usage of the nominal bit rate.

Are there distance limitations?

Can bus is a synchronous network, where all receiving modules synchronize to the data coming from a transmitting module. The electrical characteristics of the CAN bus cable restrict the cable length according to the selected bit rate. You can use cabling up to 250 meters with the baud rate of 250 kbit/s. The maximum bus length with a bit rate of 10 kbit/s is 1 km, and the shortest with 1 Mbit/s is 40 meters.

Do I need expensive cabling?

In standard industrial environments, the CAN bus can use standard cabling without shielding or twistedpair
wiring. If very low EMI is required, a twisted-pair cable is recommended. However, this will normally not be required in most applications.

Is the number of nodes (modules on the bus) limited?

In CANopen, there are unique addresses available for up to 127 nodes on the bus. However the practical physical limit of nodes is about 110 units per bus. In J1939, there are 253 unique addresses available for the bus.

Can I use units from different vendors in the same system?

Yes. If the same higher-level protocol such as CANopen or J1939 is used by all the modules on the bus, you can mix components from different vendors in the same control system without software changes.

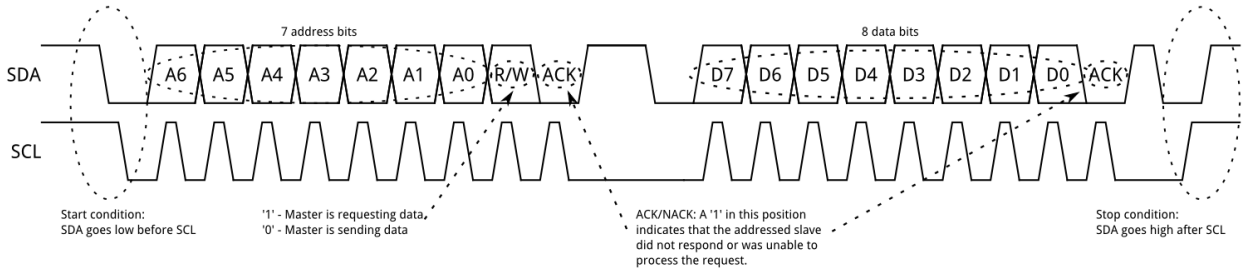
How popular is CAN?

Currently there are hundreds of millions of CAN nodes in use in the world. CAN is being widely used in passenger cars, buses, factory automation, work machines, agriculture, forestry and mining applications. The applications for CAN are increasing all the time.

Where can I get more information about CAN?

The higher-level protocol support organizations provide information about CAN bus, protocols and keep a database of CAN bus products. The CANopen protocol is developed by CiA (CAN in Automation) at www.can-cia.org. SAE (Society of Automotive Engineers) can provide information about the J1939 protocol at www.sae.org.

I2C



Messages are broken up into two types of frame: an address frame, where the master indicates the slave to which the message is being sent, and one or more data frames, which are 8-bit data messages passed from master to slave or vice versa. Data is placed on the SDA line after SCL goes low, and is sampled after the SCL line goes high. The time between clock edge and data read/write is defined by the devices on the bus and will vary from chip to chip.

Start Condition

To initiate the address frame, the master device leaves SCL high and pulls SDA low. This puts all slave devices on notice that a transmission is about to start. If two master devices wish to take ownership of the bus at one time, whichever device pulls SDA low first wins the race and gains control of the bus. It is possible to issue repeated starts, initiating a new communication sequence without relinquishing control of the bus to other masters; we'll talk about that later.

Address Frame

The address frame is always first in any new communication sequence. For a 7-bit address, the address is clocked out most significant bit (MSB) first, followed by a R/W bit indicating whether this is a read (1) or write (0) operation.

The 9th bit of the frame is the NACK/ACK bit. This is the case for all frames (data or address). Once the first 8 bits of the frame are sent, the receiving device is given control over SDA. If the receiving device does not pull the SDA line low before the 9th clock pulse, it can be inferred that the receiving device either did not receive the data or did not know how to parse the message. In that case, the exchange halts, and it's up to the master of the system to decide how to proceed.

Data Frames

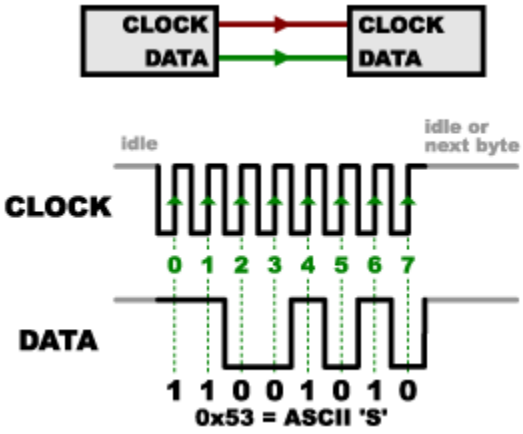
After the address frame has been sent, data can begin being transmitted. The master will simply continue generating clock pulses at a regular interval, and the data will be placed on SDA by either the master or the slave, depending on whether the R/W bit indicated a read or write operation. The number of data frames is arbitrary, and most slave devices will auto-increment the internal register, meaning that subsequent reads or writes will come from the next register in line.

Stop condition

Once all the data frames have been sent, the master will generate a stop condition. Stop conditions are defined by a 0->1 (low to high) transition on SDA *after* a 0->1 transition on SCL, with SCL remaining high. During normal data writing operation, the value on SDA should **not** change when SCL is high, to avoid false stop conditions.

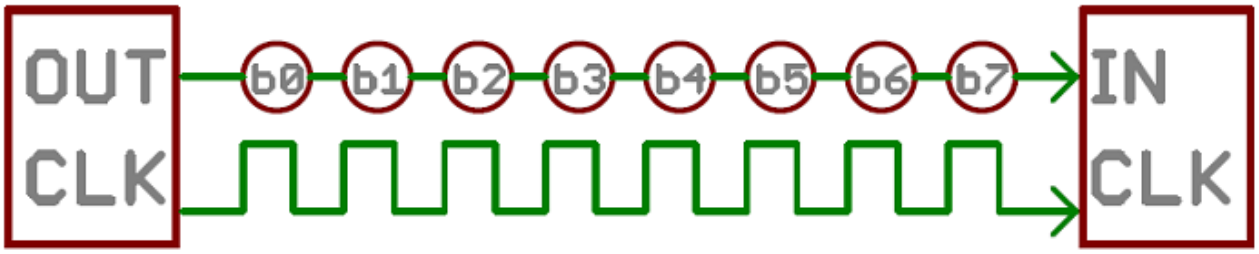
A Synchronous Solution

SPI works in a slightly different manner. It's a "synchronous" data bus, which means that it uses separate lines for data and a "clock" that keeps both sides in perfect sync. The clock is an oscillating signal that tells the receiver exactly when to sample the bits on the data line. This could be the rising (low to high) or falling (high to low) edge of the clock signal; the datasheet will specify which one to use. When the receiver detects that edge, it will immediately look at the data line to read the next bit (see the arrows in the below diagram). Because the clock is sent along with the data, specifying the speed isn't important, although devices will have a top speed at which they can operate (We'll discuss choosing the proper clock edge and speed in a bit).

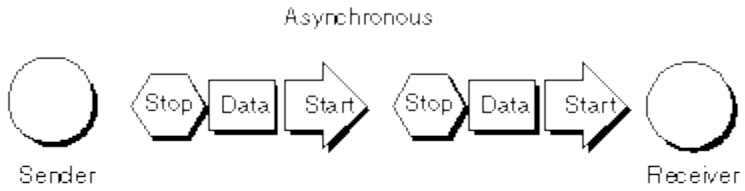


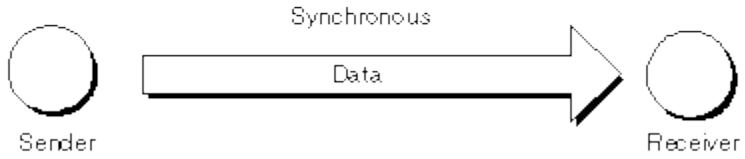
UART

- Data bits,
- Synchronization bits,
- Parity bits,
- and Baud rate.



- *Example of a serial interface, transmitting one bit every clock pulse. Just 2 wires required!*





Boot sequence of a computer

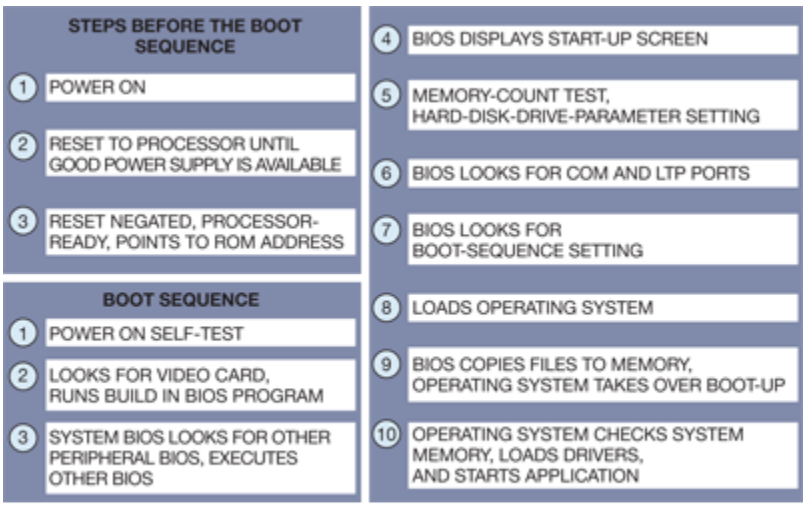
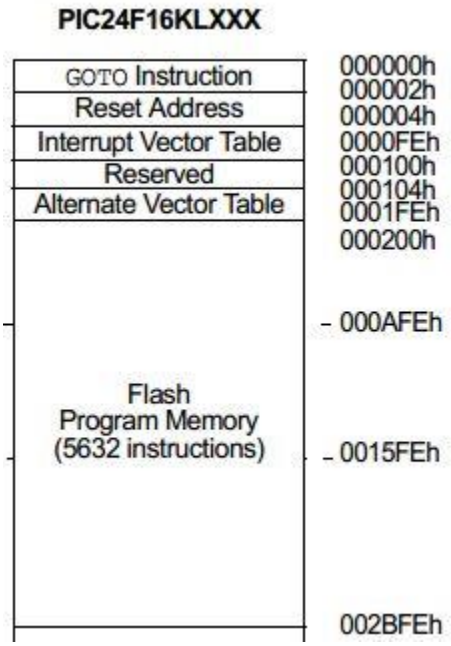


Figure 1 Microsoft's Windows XP moves through a suite of steps as it boots a system.

BOOT PROCESS IN MICROCONTROLLER

The microcontroller boot process starts by simply applying power to the system. Once the voltage rails stabilize, the microcontroller looks to the reset vector for the location in flash where the start-up instruction can be found. The reset vector is a special location within the flash memory map. For example, taking a look at Figure 1, it can be seen that at address 0x0002 is where the reset vector is located within the memory map.



ResetVector

Figure 1 – Reset Vector Location

The address that is stored at the reset vector is loaded by the microcontroller and the instructions that are contained there are then loaded and executed by the CPU. Now these first instructions aren't the start of main that the developer created. Instead, these are instructions on how to start-up the microcontroller.

The first thing that usually occurs is that the vector tables that are stored in flash are copied to RAM. They are copied from and to the location that is specified in the linker file at the time the executable program is created. One reason for copying the vector tables to RAM is that it is faster to execute from RAM than flash. This helps to decrease the latency of any interrupt calls within the system. Depending on the particular architecture of the microcontroller there may then be an instruction to update a vector table register so that the microcontroller knows where the start of the RAM table is.

Next the initialized data sections are copied into RAM. This is usually variables that are stored in the .data section of the linker. Examples of initialized data would be static, global and static local variables that have been provided with an initialization value during compile time. These are explicit definitions such as int Var = 0x32;.

Following the copy of the data section, the .bss section is also copied. The .bss section contains variables that are not initialized explicitly or that have been initialized to a value of zero. A simple example is that the variable static int Var; would be contained within this section.

Finally, the microcontroller will copy any RAM functions from flash to RAM. Once again it is sometimes worthwhile to execute certain functions out of RAM rather than flash due to the execution speed being slightly faster. These are functions usually decided upon by the developer and purposely placed there in the linker file prior to compiling the program.

This entire process is often known as the “C Copy Down”. Without performing this copy down the C environment would not be properly setup in order to execute the program. Usually once the copy down has been completed the microcontroller then jumps to the start of main where the developers application then begins.

The microcontroller boot process is actually relatively straight forward. Often times though it is written in assembly language or some other obscure and sparsely documented way so that it is difficult to get a clear understanding of how the microcontroller actually gets to main. Instead it looks like a very complex and nearly unknowable process which then makes custom development of boot code a potentially painful process.

Jacob Beningo is a Certified Software Development Professional (CSDP) whose expertise is in embedded software. He works with companies to decrease costs and time to market while maintaining a quality and robust product. He is an avid tweeter, a tip and trick guru, a homebrew connoisseur and a fan of pineapple! Feel free to contact him at jacob@beningo.com at his website www.beningo.com, and sign-up for his monthly embedded newsletter here

```
/*
 * string operations without using String_fun.c
*  Created on: May 17, 2015
 *      Author: naga
 */
#include <stdio.h>
#include <malloc.h>
#include <stddef.h>
void strlen_def(char *);
void memchr_def(char *, char);
void memcmp_def(char *, char *);
void memcpy_def(char *);
void memset_def(char *);
```

```
void memcat_def(char *, char *);
void strncmp_def(char *, char *, char);
void strncpy_def(char *, char);
void strncat_def(char *, char *, char);
int main(void) {
        char *str1;
        char *str2;
        str1 = (char *) malloc(20 * sizeof(char));
        str2 = (char *) malloc(20 * sizeof(char));
        printf("\n str1 ");
        scanf("%s", str1);
        printf("\n str2 ");
        scanf("%s", str2);
        printf("\n str1 = %s, str2 = %s", str1, str2);
        strlen_def(str2);
        memchr_def(str1, 'a');
        memcmp_def(str1, str2);
        memcpy_def(str2);
        memcat_def(str1, str2);
        strncmp_def(str1, str2, 4);
        strncpy_def(str2, 4);
        strncat_def(str1, str2, 4);
        memset_def(str1);
}
void strncat_def(char *data_1, char *data_2, char ch) {
        int i = 0, j = 0;
        char *str_temp = (char *) malloc(20 * sizeof(char));
        while (*data_1) {
```

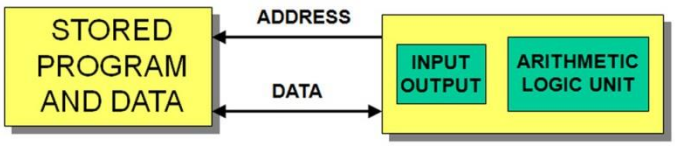
```
                str_temp[i++] = *data_1++;
        }
        while (j < ch) {
                str_temp[i++] = data_2[j++];
        }
        printf("\n ncat = %s", str_temp);
}
void strncpy_def(char *data_2, char ch) {
        char *str_temp = (char *) malloc(20 * sizeof(char));
        int i = 0;
        while (i < ch) {
                str_temp[i++] = *data_2++;
        }
        printf("\n cp_data_4 = %s", str_temp);
}
void strncmp_def(char *data_1, char *data_2, char ch) {
        int count = 0;
        while (*data_1++ == *data_2++) {
                ++count;

                if (count == ch) {
                        printf("\n 4 compare");
                        break;
                }
        }
        if (count == 4) {
                printf("\n 4 equal");
        } else {
```

```
                printf("\n 4 not equal");
        }
}
void memcat_def(char *data1, char *data2) {
        char *str_temp = (char *) calloc(20, sizeof(char));
        char i = 0;
        while (*data1) {
                str_temp[i++] = *data1++;
        }
        while (*data2) {
                str_temp[i++] = *data2++;
        }
        printf("\n cat data1 = %s", str_temp);
}
void memset_def(char *data) {
        printf("\n before = %s", data);
        *data = 0;
        printf("\n After = %d", *data);
}
void memcpy_def(char *data) {
        char *str_temp = (char *) calloc(20, sizeof(char));
        char i = 0;
        while (*data) {
                str_temp[i++] = *data++;
        }
        printf("\n cpy = %s", str_temp);
}
```

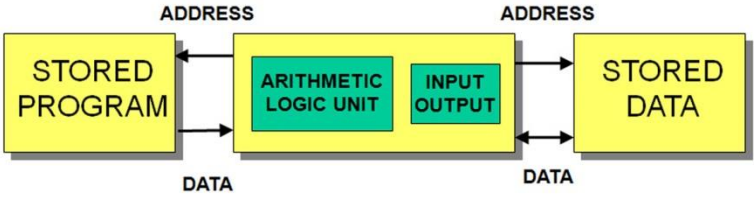
```
void memcmp_def(char *data_1, char *data_2) {
	while (*data_1 == *data_2) {
		if ((*data_1 == '\0') || (*data_2 == '\0')) {
			break;
		}
		data_1++;
		data_2++;
	}
	if ((*data_1 == '\0') && (*data_2 == '\0')) {
		printf("\n same");
	} else {
		printf("\n not same");
	}
}
void memchr_def(char *data, char ch) {
	char *str_temp;
	int i = 0;
	str_temp = (char *) calloc(20, sizeof(char));
	while (*data++ == ch)
		;
	while (*data) {
		str_temp[i++] = *data++;
	}
	printf("\n str_tmp = %s", str_temp);
	free(str_temp);
	str_temp = NULL;
}
void strlen_def(char *data) {
```

```
    char count = 0;
    while (data[count++] != '\0');
    printf("\n strlen = %d", count);
}
```

Von Neuman

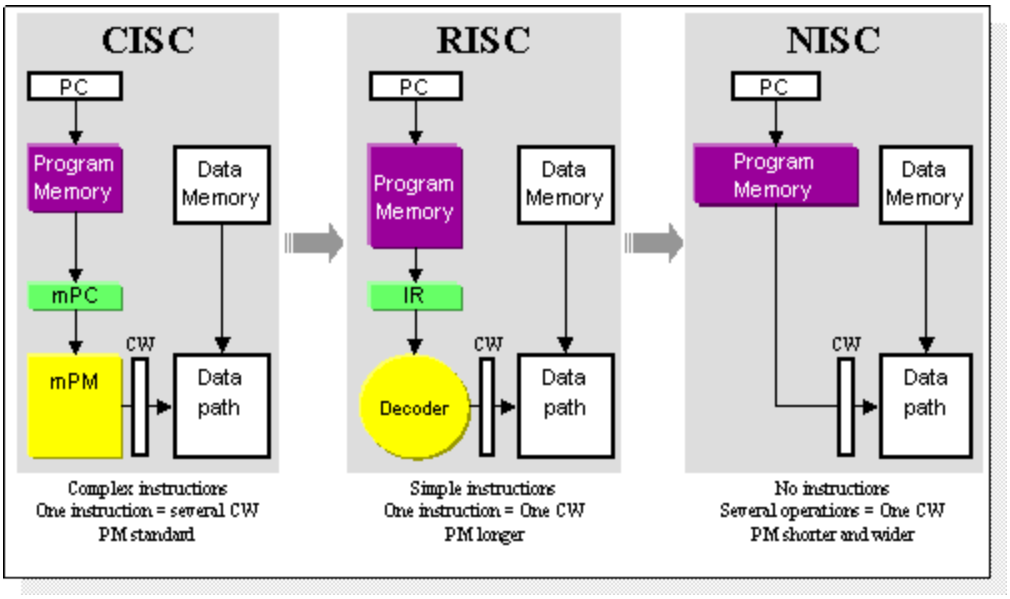


Havard



CISC and RISC architecture Microcontrollers:

CISC Processors	***RISC Processors***
Complex Instruction Set Computer	Reduced Instruction Set Computer
When an MCU supports many addressing modes for arithmetic and logical instructions and for memory accesses and data transfer instructions, the MCU is said to of CISC architecture.	When an MCU has an instruction set that supports one or two addressing modes for arithmetic and logical instructions and few for memory accesses and data transfer instructions, the MCU is said to of RISC architecture
Large number of complex instructions	Small number of instructions
Instructions are of variable number of bytes	Instructions are of fixed number of bytes
Instructions take varying amounts of time for execution	Instructions take fixed amount of time for execution

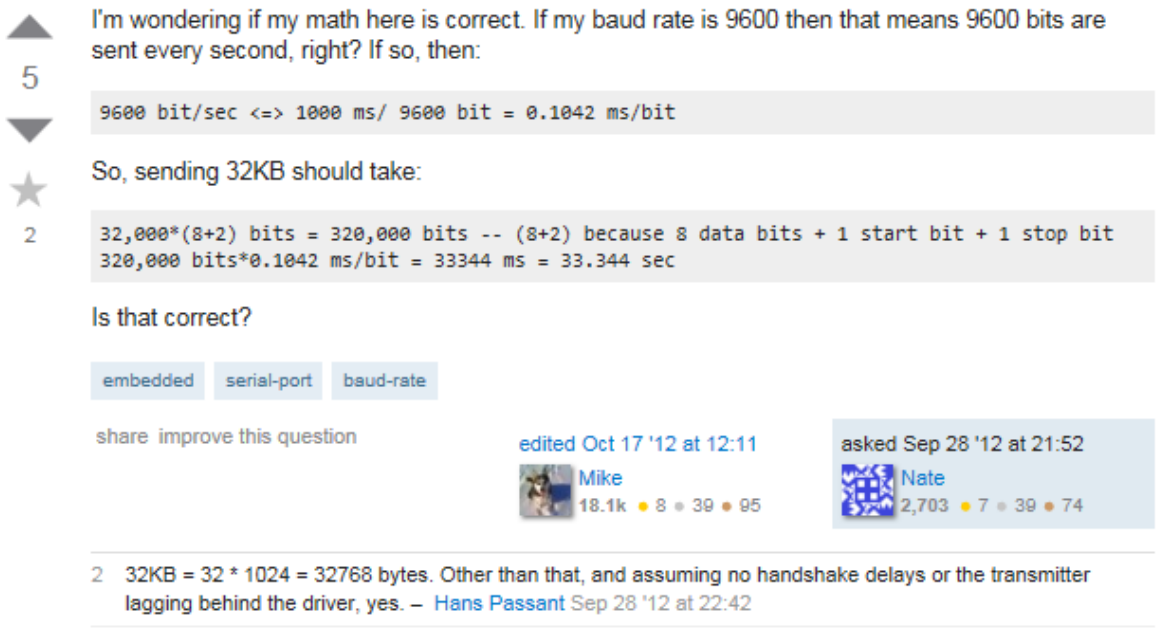


	CISC	RISC
1	Complex instructions taking multiple cycles	Simple instructions taking 1 cycle
2	Any instruction may reference memory	Only LOADS/STORES reference memory
3	Not pipelined or less pipelined	Highly piplined
4	Instructions interpreted by the microprogram	Instructions executed by the hardware
5	Variable format instructions	Fixed format instructions
6	Many instructions and modes	Few instructions and modes
7	Complexity in the microprogram	Complexity is in the compiler
8	Single register set	Multiple register sets

Source:
www.egr.msu.edu/classes/ece482/Teams/97fall/xdesign2/arm/andy.doc

RISC vs. CISC

CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy



5

2

I'm wondering if my math here is correct. If my baud rate is 9600 then that means 9600 bits are sent every second, right? If so, then:

```
9600 bit/sec <=> 1000 ms/ 9600 bit = 0.1042 ms/bit
```

So, sending 32KB should take:

```
32,000*(8+2) bits = 320,000 bits -- (8+2) because 8 data bits + 1 start bit + 1 stop bit
320,000 bits*0.1042 ms/bit = 33344 ms = 33.344 sec
```

Is that correct?

embedded serial-port baud-rate

share improve this question

edited Oct 17 '12 at 12:11
Mike
18.1k ● 8 ● 39 ● 95

asked Sep 28 '12 at 21:52
Nate
2,703 ● 7 ● 39 ● 74

2 32KB = 32 * 1024 = 32768 bytes. Other than that, and assuming no handshake delays or the transmitter lagging behind the driver, yes. – Hans Passant Sep 28 '12 at 22:42

OSI Model				
Layer		Data unit	Function[3]	Examples
Host layers	7. Application	Data	High-level APIs, including resource sharing, remote file access, directory services and virtual terminals	HTTP, FTP, SMTP
	6. Presentation		Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption	ASCII, EBCDIC, JPEG
	5. Session		Managing communication sessions, i.e. continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes	RPC, PAP
	4. Transport	Segments	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing	TCP, UDP
Media layers	3. Network	Packet/Datagram	Structuring and managing a multi-node network, including addressing, routing and traffic control	IPv4, IPv6, IPsec, AppleTalk
	2. Data link	Bit/Frame	Reliable transmission of data frames between two nodes connected by a physical layer	PPP, IEEE 802.2, L2TP
	1. Physical	Bit	Transmission and reception of raw bit streams over a physical medium	DSL, USB

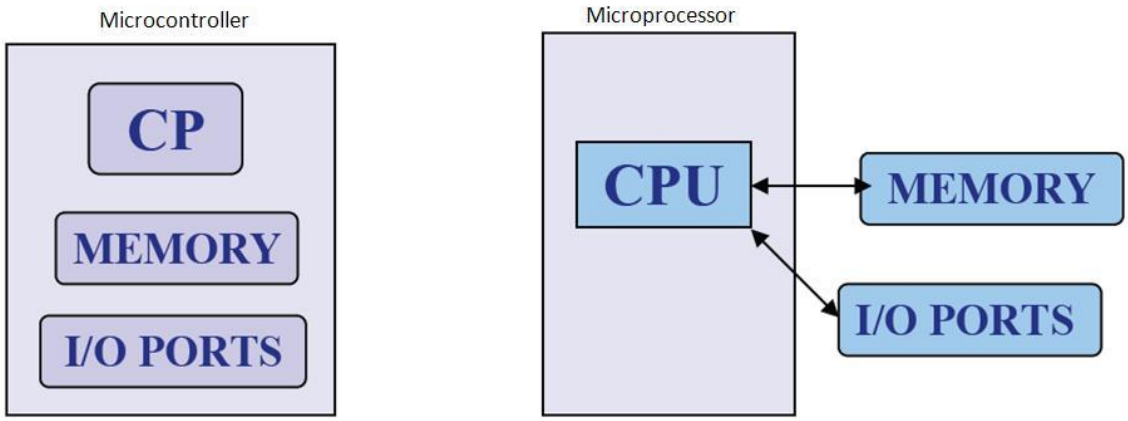
Microprocessor vs. Microcontroller

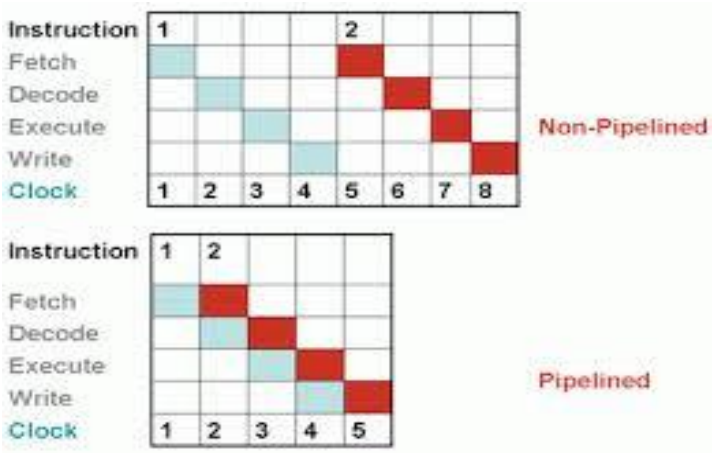
Microcontroller

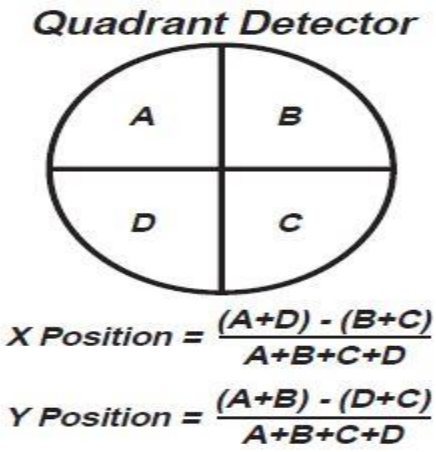
- CPU, RAM, ROM, I/O and timer are all on a single chip
- Fix amount of on-chip ROM, RAM, I/O ports
- For applications in which cost, power and space are critical
- Not Expansive
- Single-purpose

Microprocessor

- CPU is stand-alone, RAM, ROM, I/O, timer are separate
- Designer can decide the amount of ROM, RAM and I/O ports
- Expansive
- Versatility
- General-purpose


Microcontroller
CP
MEMORY
I/O PORTS
Microprocessor
CPU
MEMORY
I/O PORTS


Instruction
Fetch
Decode
Execute
Write
Clock
1
2
1 2 3 4 5 6 7 8
Non-Pipelined
Instruction
Fetch
Decode
Execute
Write
Clock
1 2
1 2 3 4 5
Pipelined


Quadrant Detector
A
B
D
C
X Position = (A+D) - (B+C) / A+B+C+D
Y Position = (A+B) - (D+C) / A+B+C+D

Check Sum Calculation

```
/*****************************************************************************
*/
/*  C Program to implement Check Sum.*/
/*  Download more programs at http://sourcecode4u.com/ */
/*****************************************************************************
*/
#include<stdio.h>
#include<math.h>
 int sender(int b[10],int k)
 {
   int checksum,sum=0,i;
      printf("\n****SENDER****\n");
for(i=0;i<k;i++)
     sum+=b[i];
     printf("SUM IS: %d",sum);
   checksum=~sum;
   printf("\nSENDER's CHECKSUM IS:%d",checksum);
   return checksum;
 }
 int receiver(int c[10],int k,int scheck)
 {
    int checksum,sum=0,i;
    printf("\n\n****RECEIVER****\n");
    for(i=0;i<k;i++)
     sum+=c[i];
     printf(" RECEIVER SUM IS:%d",sum);
     sum=sum+scheck;
```

```
    checksum=~sum;
    printf("\nRECEIVER's CHECKSUM IS:%d",checksum);
    return checksum;
}
main()
 {
   int a[10],i,m,scheck,rcheck;
   printf("\nENTER SIZE OF THE STRING:");
   scanf("%d",&m);
   printf("\nENTER THE ELEMENTS OF THE ARRAY:");
   for(i=0;i<m;i++)
  scanf("%d",&a[i]);
  scheck=sender(a,m);
  rcheck=receiver(a,m,scheck);
  if(rcheck==0)
    printf("\n\nNO ERROR IN TRANSMISSION\n\n");
  else
    printf("\n\nERROR DETECTED");
 }
```