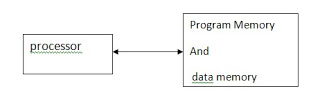
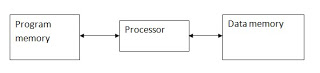
[Difference between Harvard and Von Neumann computer architectures](http://v-codes.blogspot.in/2010/09/difference-between-harward-and-von.html)

**Von Neumann Architecture:**

[](http://4.bp.blogspot.com/_c99lLjgQ8ho/TITToIOFEVI/AAAAAAAAAow/xMOWISGW29I/s1600/1.bmp)

* The computer has single storage system(memory) for storing data as well as program to be executed.
* Processor needs two clock cycles to complete an instruction.Pipelining the instructions is not possible with this architecture.
* In the first clock cycle the processor gets the instruction from memory and decodes it. In the next clock cycle the required data is taken from memory. For each instruction this cycle repeats and hence needs two cycles to complete an instruction.

**Harvard Architecture:**

[](http://1.bp.blogspot.com/_c99lLjgQ8ho/TITT4YRhS-I/AAAAAAAAAo4/q_odEsR3yYo/s1600/2.bmp)

* The computer has two separate memories for storing data and program.
* Processor can complete an instruction in one cycle if appropriate pipelining strategies are implemented.
* In the first stage of pipeline the instruction to be executed can be taken from program memory. In the second stage of pipeline data is taken from the data memory using the decoded instruction or address.

Figure 1-3 gives just a couple of possible high-level diagrams that could be implemented on such a generic embedded system.



Schematic Basics



\

Example Schematic



**2.3.1. Memory Map**

All processors store their programs and data in memory. This memory may reside on the same chip as the processor or in external memory chips. Memory is located in the processor's memory space, and the processor communicates with it by way of two sets of electrical wires called the address bus and the data bus. To read or write a particular location in memory, the processor first writes the desired address onto the address bus. Some logic (either on the processor or in an external circuit), known as an address decoder, interprets the upper address bits on this bus and selects the appropriate memory or peripheral chip. The data is then transferred over the data bus. The address decoder can be an external IC, but today many processors include this logic on-chip. There are also control signals for reading and writing to various devices in a processor's memory space that are commonly referred to as the control bus. These control bus signals include read, write, and chipselect (or chip-enable). Some processors combine the read and write signals into a single read/write signal. On these processors, a read operation is performed by setting the signal to one level and a write operation is performed by setting the signal to the opposite level.

For example, if the signal name is RD/wr (pronounced "read write bar"), the signal is set to a 1 for a read operation and set to 0 for a write operation.

The chip-select signal is set to its active level when the address for a particular device falls within the device's address range. For example, let's say a RAM device occupies the address range from 0x0000 to 0x0FFF. When the software instruction accesses the variable located at address 0x01F2, the chip-select for the RAM is set at its active level.

The read and write signals are set to their active levels by the processor based on the type of memory transaction. Figure 2-5 is an example of a timing diagram, a graphical representation of the timing relationship between the various signals for a given operation. The entire diagram in Figure 2-5 shows one read cycle. In this case, the cycle is reading 16 bits of data from memory. Typically, a table of timing requirements accompanies a timing diagram. The timing requirements detail the minimum and maximum acceptable times for each of the various signals and the timing relationships among the signals.



The clock signal (CLK) is the basis for all operations of the processor and is shown as the top signal in the timing diagram of Figure 2-5. The processor clock is generally a square wave that sequences all operations of the processor.

The next group of signals are the address bus, A[0:20], followed by the data bus, D[0:15]. Such buses are depicted in timing diagrams as shown in Figure 2-5, where a single entry represents the entire range of signals rather than each signal having its own entry. A bus is typically stable (meaning it contains a valid address or data) during the period of time when the single line splits into two lines. In hardware terms, the bus goes from being tristate (single line) to having real information present (dual line), and then back to being tristate again. The next signal is active low chip select (CS/), and after that is the write (WR/) signal, which is also active low. Since this is a timing diagram for a read operation, the write signal stays inactive during the entire cycle. The last signal is the read (RD/). It goes low (active) after the address is set by the processor.

***Figure 2-6. Memory map for the Arcom board***

For



**2.7. Initialize the Hardware**

The hardware initialization should be executed before the startup code described in Chapter 4. The code described there assumes that the hardware has already been initialized and concerns itself only with creating a proper runtime environment for high-level language programs. Figure 2-7 provides an overview of the entire initialization process, from processor reset through hardware initialization and C startup code to main.



The first stage of the initialization process is the reset code. This is a small piece of assembly language

(usually only two or three instructions) that the processor executes immediately after it is powered on or

reset. The sole purpose of this code is to transfer control to the hardware initialization routine. The first

instruction of the reset code must be placed at a specific location in memory, usually called the reset

address or reset vector, which is specified in the processor databook. The reset address for the PXA255

is 0x00000000.

Most of the actual hardware initialization takes place in the second stage. At this point, we need to

inform the processor about its environment. This is also a good place to initialize the interrupt controller

and other critical peripherals. Less critical hardware devices can be initialized when the associated

device driver is started, usually from within main.

## Memory Layout of C Program

Before talking of memory layout of a C program and its various segments to store data and code instructions we should first understand that a compiler driver (that invokes the language preprocessor, compiler, assembler, and linker, as needed on behalf of the user) can generate three types of object files depending upon the options supplied to the compiler driver. Technically an object file is a sequence of bytes stored on disk in a file. These object files are as follows:

* *Relocatable object file:* These are static library files. Static linkers such as the Unixld program take collection of relocatable object files and command line arguments as input and generate a fully linked executable object file as output that can be loaded into memory and run. Relocatable object files contain binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file.
* *Executable object file:* These are executable files contain binary code and data in a form that can be copied directly into memory and executed.
* *Shared object file:* These special type of relocatable object files are loaded into memory and linked dynamically, at either load time or run time.

Object files have a specific format, however this format may vary from system to system. Some most prevalent formats are .coff (Common Object File Format), .pe(Portable Executable), and elf (Executable and Linkable Format).

## Code and Data Segments

However, the actual layout of a program's in-memory image is left entirely up to the operating system, and often the program itself as well. This article focus on the concepts of code and data segments of a program and does not take any specific platform into account. For a running program both the machine instructions (program code) and data are stored in the same memory space. The memory is logically divided into text and data segments. Modern systems use a single text segment to store program instructions, but more than one segment for data, depending upon the storage class of the data being stored there. These segments can be described as follows:

1. Text or Code Segment
2. Initialized Data Segments
3. Uninitialized Data Segments
4. Stack Segment
5. Heap Segment

## 

## Text or Code Segment

*Code segment*, also known as *text segment* contains machine code of the compiled program. The text segment of an executable object file is often read-only segment that prevents a program from being accidentally modified.

## Data Segments

*Data segment* stores program data. This data could be in form of initialized or uninitialized variables, and it could be local or global. Data segment is further divided into four sub-data segments (initialized data segment, uninitialized or .bss data segment, stack, and heap) to store variables depending upon if they are local or global, and initialized or uninitialized.

### Initialized Data or Data Segment

*Initialized data* or simply *data segment* stores all global, static, constant, and external variables (declared with extern keyword) that are initialized beforehand.

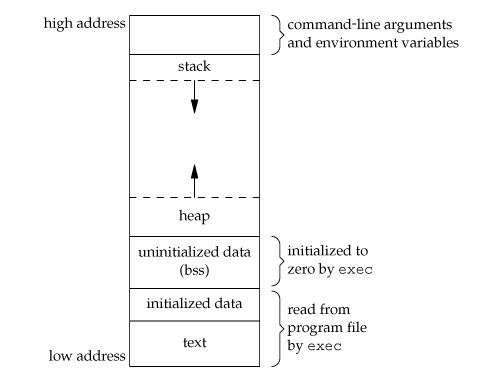
### Uninitialized Data or .bss Segment

Contrary to initialized data segment,*uninitialized data* or *.bss segment*stores all uninitialized global, static, and external variables (declared withextern keyword). Global, external, and static variable are by default initialized to zero. This section occupies no actual space in the object file; it is merely a place holder. Object file formats distinguish between initialized and uninitialized variables for space efficiency; uninitialized variables do not have to occupy any actual disk space in the object file.

Randal E. Bryant explains in his famous book on *Computer Systems: A Programmer's Perspective*, **Why is uninitialized data called .bss?**  
The use of the term .bss to denote uninitialized data is universal. It was originally an acronym for the "Block Storage Start" instruction from the IBM 704 assembly language (circa 1957) and the acronym has stuck. A simple way to remember the difference between the .data and .bss sections is to think of "bss" as an abbreviation for "Better Save Space!"

### Stack Segment

*Stack segment* is used to store all local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over. Local variables have a scope to the block which they are defined in; they are created when control enters into the block. Local variables do not appear in *data* or *bss* segment. Also all recursive function calls are added to stack. Data is added or removed in a last-in-first-out manner to stack. When a new stack frame needs to be added (as a result of a newly called function), the stack grows downward (See the figure 1).

Fig 1. - Memory layout of a C program

### Heap Segment

*Heap segment* is also part of RAM where dynamically allocated variables are stored. In C language dynamic memory allocation is done by using malloc and callocfunctions. When some more memory need to be allocated using malloc and callocfunction, heap grows upward as shown in above diagram.

The stack and heap are traditionally located at opposite ends of the process's virtual address space.

## Check Size of Code, Data, and .BSS Segments

The size command, a GNU utility, reports the sizes (in bytes) of the text, data, .bss segments, and total size for each of the object or archive files in its argument. By default, one line of output is generated for each object file or each module in an archive. The size command has no information for stack and heap sections because the size of those areas is decided on run-time.

For example, see the following C program and the size of its object file.

#include <stdio.h>

int main ()

{

unsigned int x = 0x76543210;

char \*c = (char\*) &x;

if (\*c == 0x10)

{

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html) ("Underlying architecture is little endian. **\n**");

}

else

{

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html) ("Underlying architecture is big endian. **\n**");

}

return 0;

}

For the above mentioned program check-endianness.c (which finds whether the underlying architecture is little endian or big endian) the size of text, data, .bss segments, and the total size is examined as follows with help of the size command. The fourth and fifth columns are the total of the three sizes, displayed in decimal and hexadecimal, respectively. You can read man page of size for more details.

[root@host ~/cprogs]$ gcc check-endianness.c -o check-endianness

[krishaku@adc6140630 ~/cprogs]$ size check-endianness

text data bss dec hex filename

1235 492 16 1743 6cf check-endianness