EE-712: Embedded Systems Software

Dinesh Sharma, <dinesh@ee.iitb.ac.in>

EE Department
IIT Bombay, Mumbai

Spring Semester, 2022

Course Contents

- Embedded system basics:
  Embedded microcontroller cores, embedded memories.
  Examples of embedded systems.
- Technological aspects of embedded systems:
  Interfacing between analog and digital blocks: signal conditioning, digital signal processing.
  Sub-system interfacing, interfacing with external systems, user interfacing.
  Design trade-offs due to process compatibility, thermal considerations, etc.
- Software aspects of embedded systems: real time programming, operating systems for embedded systems.

Prerequisites: Signals & Systems; Analog Electronics; Digital Electronics; Microprocessors

Suggested books

- J.W. Valvano, “Embedded Microcomputor System: Real Time Interfacing”, Brooks/Cole, 2000.
- Jack Ganssle, “The Art of Designing Embedded Systems”, Newnes, 1999.
- David Simon, “An Embedded Software Primer”, Addison Wesley, 2000.
- P Horowitz and W Hill, “The Art of Electronics”, Cambridge,1995.
- HW Ott, “Noise Reduction Techniques in Electronic Systems”, Wiley, 1989.
- WC Bosshart, “Printed Circuit Boards: Design and Technology”, Tata McGraw Hill, 1983.
- GL Ginsberg, “Printed Circuit Design”, McGraw Hill, 1991.

Evaluation

- Mid-Semester: 30%
- End-Semester: 40%;
- Assignments, Mini Project Report & Presentation: 30%

The evaluation scheme may need review and modification depending on the pandemic situation, which will influence access to lab for miniproject etc.

Software for Embedded Systems

- Processors for Embedded Systems:
  ARM (Cortex 8) architecture and Programming
- Embedded software: application definition to implementation:
  - Compilation: Source code to Object code
  - Linking: Object code to Executable code
  - Loading: Executable code to running process
- Operating System basics: processes, threads, scheduling etc.
- Multi-processing, hazards in multi-processing, avoidance.
- Real time processes, scheduling Schemes for real time processes
- Techniques for application software development:
  Finite State Machines in software, examples, Lexical analysis (parsing) etc.

Pre-requisites

Essential:

- Familiarity with C programming
- Exposure to Micro-processors, assembly programming
- Access to a C compiler, preferably on a Linux system.

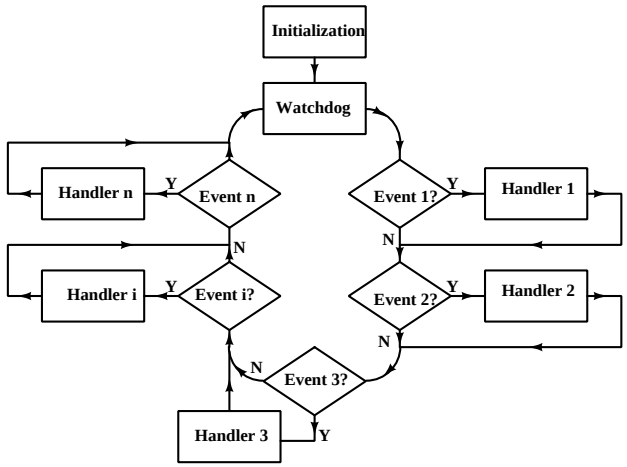
Desirable:

- Familiarity with Operating Systems

How is Embedded Software different?

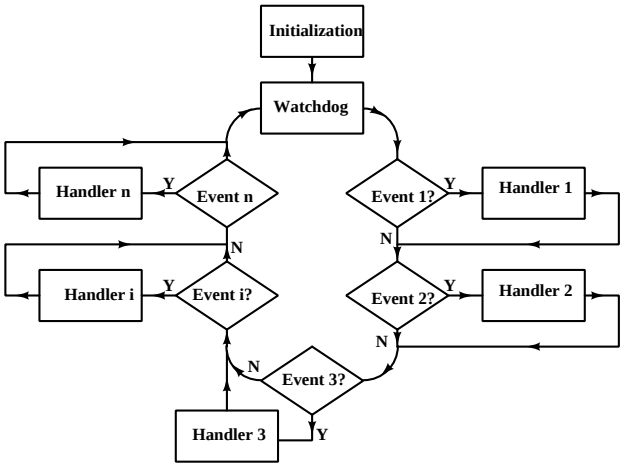
Embedded software implements the functionality of some equipment – say a camera or a washing machine. So it never terminates! (What will your camera do if its software terminated?)

- Embedded software is typically event driven. The software does not dictate the program evolution – external events do.
- The software keeps checking for external events.
- When an event occurs, it runs a piece of code designed to handle this event. Otherwise, it checks for the next event . . .



Event loop in Embedded Software

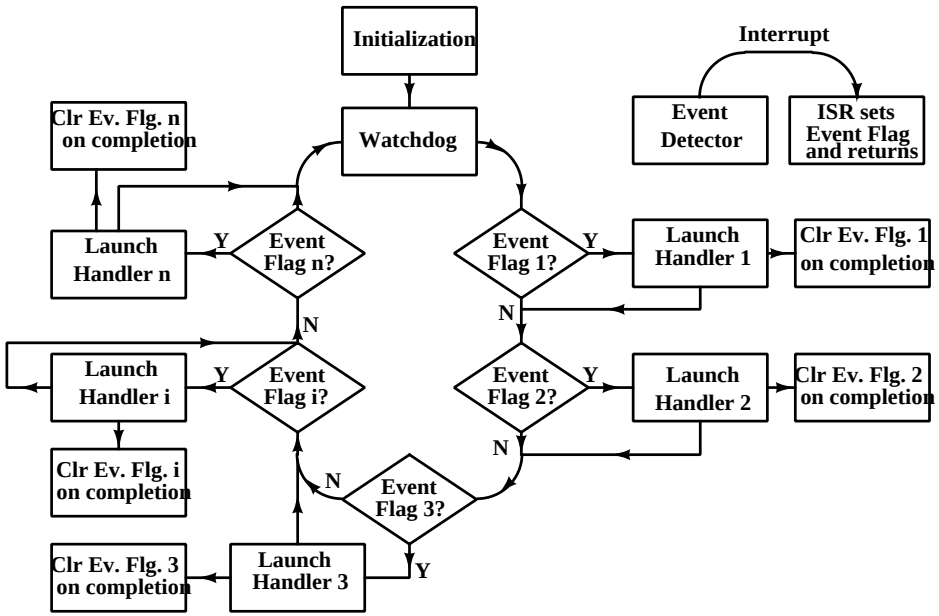
What if an event handler takes long to run?
The next event will not be noticed till the handler has finished!



- Event detectors should be able to interrupt a running handler.
- The interrupt service routine for the event detector just sets a flag and returns.
- Event loop will check for this flag to determine if an event has occurred.

Event loop in Embedded Software

- Since we cannot afford to wait for an event handler to run to completion before checking for the next event, we may launch the handler as a process
- More than one event handler may need to run concurrently!



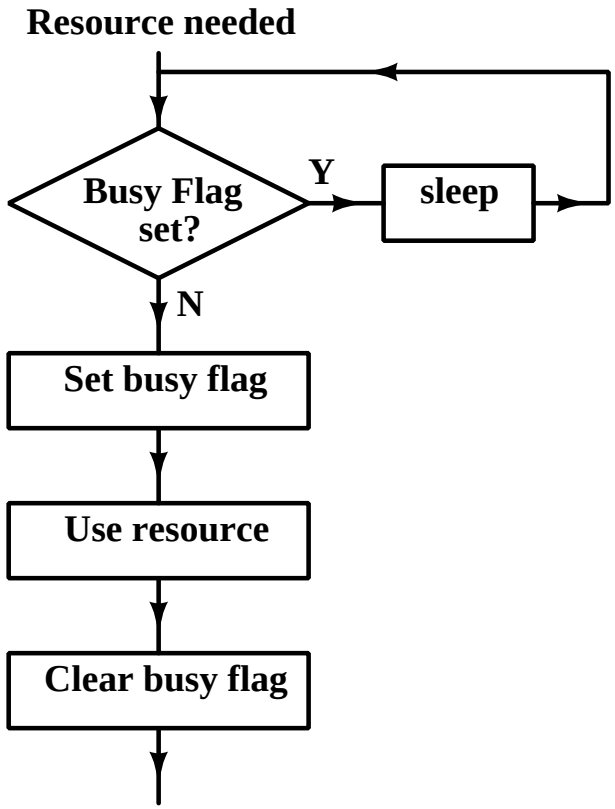
We need a multi-processing system.

Managing multiple handlers

- Multiple event handlers may be active at the same time.
- We need to manage these through a multi-processing system.
- The system should schedule these processes to run concurrently.
- Multiple running processes may require common resources – such as memory buffers.
- This can lead to hazards if these are not implemented carefully.

Example of a hazard in multi-processing

- Consider a resource R which can be used by two different processes P1 and P2.
- To ensure orderly use of this shared resource, we associate a busy flag with the resource.



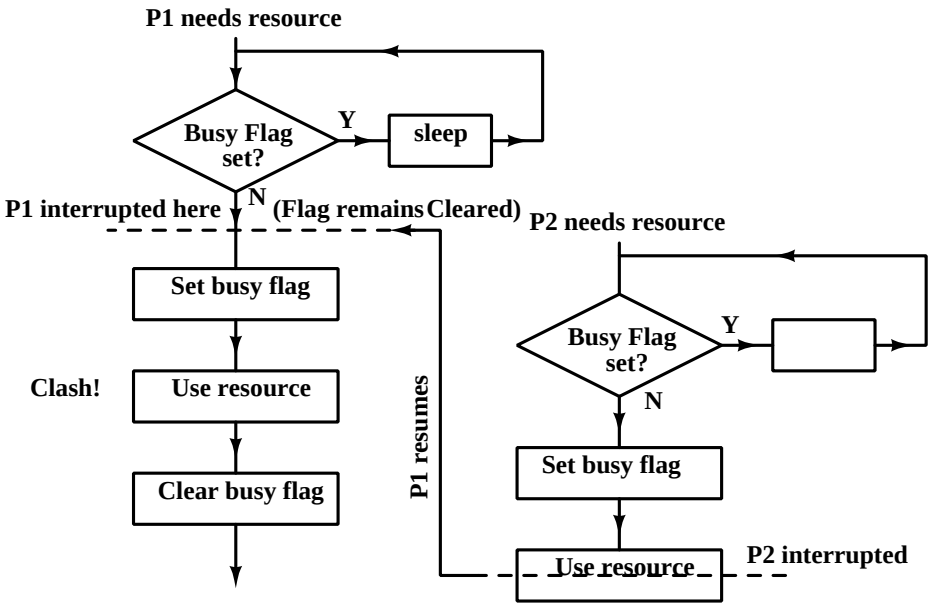
- Whenever a process wants to use this resource, it first checks the busy flag. If this is set, the process sleeps for some time and then re-tries.
- If the busy flag is cleared, it sets the busy flag and starts using the resource.
- When it is through with the use of R, it clears the busy flag.

Surely this would ensure orderly use of R?

Shared Resource Hazard

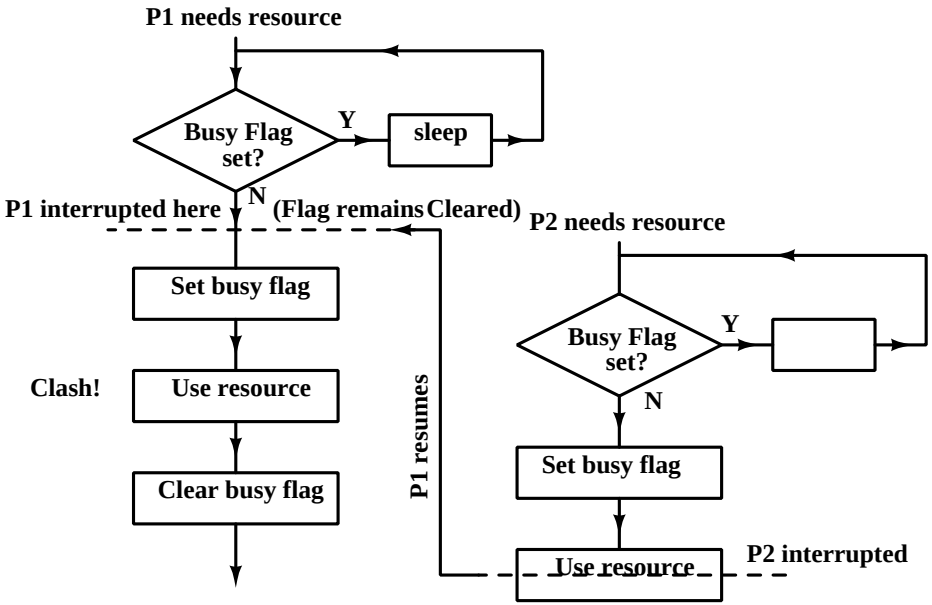
Consider the following scenario:

- P1 needs R, checks for busy flag and finds it cleared.
- After checking the flag, but before setting it, P1 is interrupted (by an interrupt or by the process scheduler).
- P2 starts running and it also needs R.
- P2 checks for busy flag and finds it cleared!



Shared Resource Hazard

- Finding the busy flag cleared, P2 starts using R.
- Before completing its use of R, P2 is interrupted and P1 resumes.
- P1 will resume from where it was interrupted – after checking and before setting the busy flag.
- P1 sets busy flag and starts using R!!
- CLASH!



Critical Code

- In spite of being careful about the use of a busy flag, a clash has occurred where P1 and P2 are both using the resource, clobbering each other's work.
- This has happened because P1 was interrupted at a critical juncture – after checking but before setting the busy flag.
- Such code is called "critical code". Critical code is any code which must run "atomically" (without being interrupted) for ensuring correct operation.
- Such bugs are very hard to detect, since these occur only once in a while.

Critical Code: solutions

- We can solve the problem of critical code through a hardware/software combination (using a processor which provides for checking and setting/clearing a flag in a single instruction)
- A software solution is through the use of semaphores from an operating system.
- A process needing a resource requests a semaphore from the OS.
- If the resource is available, the OS gives the semaphore, enabling the process to use the resource.
- If the resource is not available, the OS swaps the process out and runs other processes until the resource becomes available.
- The critical code for checking the availability of the resource and marking it busy is now moved to the OS, which can disable interrupts for a short while, running the critical code atomically.

Other requirements of Embedded Software

- A single binary code is loaded as the embedded software. This binary includes the OS (if used) as well as the application program.
- The binary code should resolve all addresses in absolute form (ROM-able code).
- Since the program serves a fixed application, unused capabilities of the hardware will be wasted. Therefore, the hardware as well as the software configuration should be optimized for the application.
- For example, the amount of memory, number of port pins, buffers allocated in software for I-O etc. should all be optimum for the application for cost competitive embedded system design.
- In practice, most embedded systems are resource constrained and careful optimization is required to deliver the required functionality using these limited resources.

Software: From source to run image

- The programmer writes the software in a programming language, using a simplified view of the processor (called the programmers' view). This is called the "source code".
- The source code is then translated to "object code" by a compiler (for programs written in high level languages like C) or an assembler (for programs written in assembly language).
- Assembly language programs provide close access to the processor hardware and can potentially generate final executable code which runs much faster and consumes less power.
- However, programs written in assembly language are tied to a given processor and are not easy to port to other processors.
- Optimizing compilers can produce run time code which comes close to hand optimized assembler code in code size and performance.

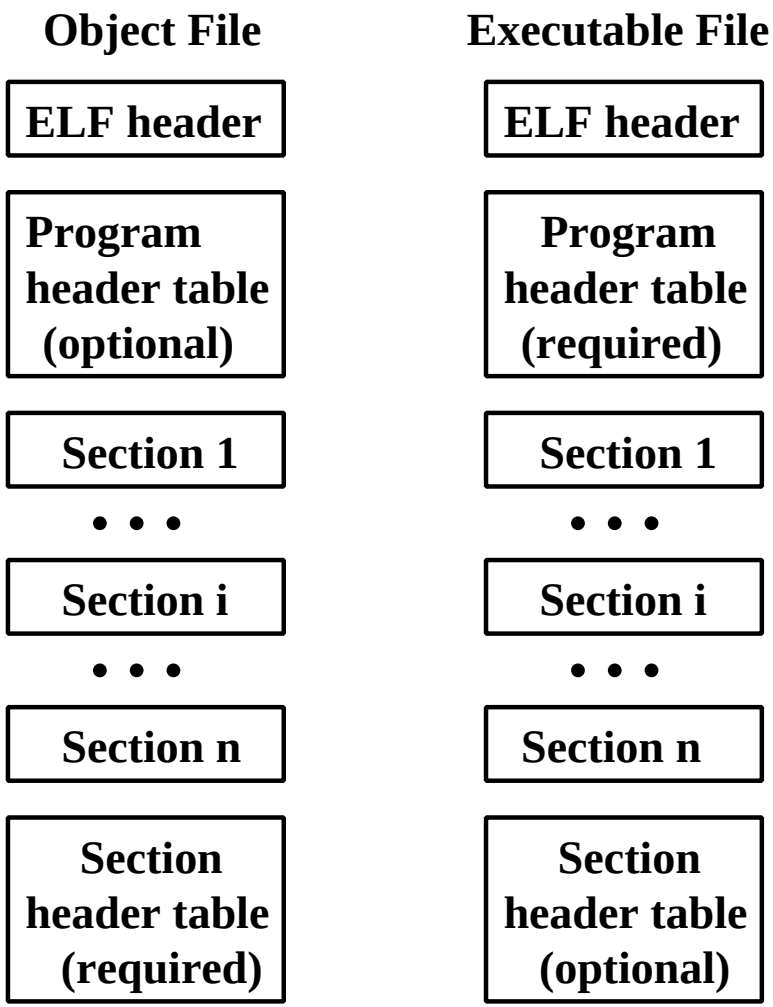
What is object code?

- An object code file (typically .o or .obj) is created for every file in the source code (.c or .s or .asm etc.).
- Object code contains a translation of the source code to the binary instructions of a particular processor.
- However, at this stage, the binary code is not ready to be executed by a processor. This is because:
  - The translated code is (potentially) distributed over several files. Code in one file is unaware of functions and variables in other files.
  - Even for a single file object code, some library function like printf may be used, which must be brought in from a library.
  - Information about where a function or a variable will be placed in memory is not yet available.
  - As a result, during translation, it is not possible to fill in the addresses required in many instructions.

Linking object code files

- A dedicated program has to combine the object files into an "executable file". This program is called a "Linker".
- Object files are inputs to the linker. These should provide the necessary information for linking and relocation of code.
- The format of these files is operating system dependent.
- In Unix, the a.out format was used earlier. It has now evolved into the ELF format, which stands for "Executable and Linkable File" format. As the name implies, it is used for object files as well as executable files.
- For historical reasons, the name a.out is still used as the default file name for the executable file created by the compiler/assembler and linker.

Object File Format: ELF



- Depending on the kind of file it is describing, there are optional and essential sections in an ELF file.
- An object file must have a “section header table” to be used by the linker.
- An executable file must have a “program header table” to be used by a loader program which creates a run image in the memory from the executable file.
- ELF header provides a road map of what information is where.

Segments in an executable file

The linker reads one or more object files and combines all instructions, data, stack information etc. collected from sections of the object files into “segments”.

Code Segment:

Instructions collected from all object file are placed in one segment. This segment is called the ‘code’ or the ‘text’ segment.

Data Segment: Initialized data is grouped into a ‘data’ segment. Since the program defines the starting value for these data objects, these values must be stored in the executable program. These go into the data segment.

Apart from traditional data items, this segment carries things like the text of a prompt or headers included in a format statement.

Segments in an executable file

BSS Segment: A program will typically use uninitialized data. These are data items which have been declared but no specific value is to be loaded into them at the start of the program.

- For example, when we declare: int a[100]; in a c program, the executable file need not use up 100 int locations for ‘a’.
- The file only needs to carry information about reserving 100*sizeof(int) locations in the memory at run time for ‘a’.
- We do need to have a pointer to the start of these locations, so a label must be placed in the executable file.
- The location counter must be advanced by 100* sizeof(integer) for the next declared uninitialized data item.

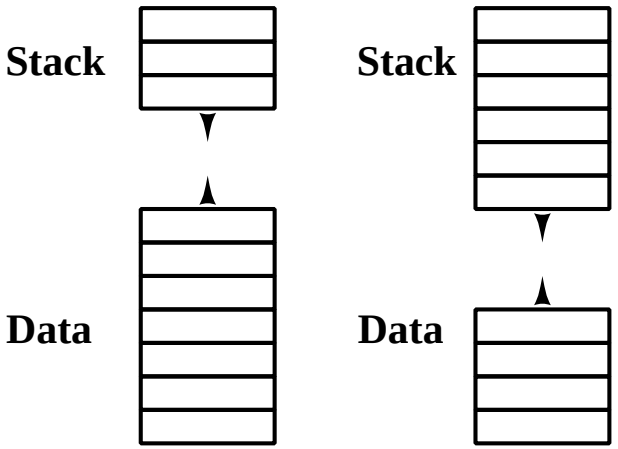
This information constitutes the BSS segment. BSS stands for Block Started by Symbol.

Segments in an executable file

Stack Segment: The stack pointer needs to be initialized to an appropriate value in the running program so that enough storage space is available for the stack. This is provided by the stack segment.

- Typically, the data and BSS segments are placed at one end of the usable memory along with code, while the the stack segment is placed at the other end.
- Storage in the stack segment changes dynamically during the execution of the program as functions are called and ‘auto’ type variables are declared by the called functions.
- Dynamic memory allocation can change the storage requirement for data during run time.
- Placing BSS/data and Stack segments at the opposite ends of the available memory provides flexibility in dividing the memory between the two.

Data and Stack



- When data and stack are grouped at opposite ends and grow towards each other, the only constraint is that the size of the two segments should not exceed the available memory.
- We can have high memory usage for data and less for stack or low memory usage for data and high for stack.

If Stack had been placed at the end of the data segment, with both growing in the same direction, the size of data segment would be fixed. In that case, we would run out of memory if data segment goes past its end, even though memory is available at the end of stack segment.

Linking Object Code Files

- The linker Keeps adding all instructions from the object files into the code segment.
- It does not know at what address these instructions will be loaded in the processor memory. It keeps track of the offset within the code segment for each instruction and associates each label in the code with an offset in the code segment.
- Similarly, It collects all pre-defined data in the data segment, keeping track of the offset within the data segment for each pre-initialized data object.
- The linker creates a symbol table, associating each instruction label and each data item with its offset in the corresponding segment.

Linking Object Code Files

- For the BSS segment, The linker makes entries in the symbol table for all uninitialized data items, again associating offsets in BSS segment with the corresponding symbol names.
- The linker includes object files from libraries for all symbol references which were not defined in the user program. It issues an error and quits if all symbols have not been resolved in terms of offsets within segments.
- After resolving all references to data symbols and code labels, the liker writes out the executable file.
- Translation of the offsets within segments to actual addresses is done at the time of loading the executable file into memory. This job is done by the “loader”.
- So the software goes through the following transformations: Source code $\rightarrow$ Object code $\rightarrow$ executable file $\rightarrow$ Running process in memory.

Software evolution from source code to run image

- The loader obtains memory from the operating systems for the program to be run. It then reads the executable file into memory, patching segment offsets to actual addresses.
- Both the linker and loader have to track symbol addresses and patch the code where these symbols are accessed. Both operate on files in ELF format.
- (To add to the confusion, the linker on Linux is called ld . . . !)
- After loading the program and patching symbol references to actual memory address, a ”process page” is created with a process identification number (pid), user id for the owner of the program and other data pertaining to run time information for the program.
- The operating system then sets the proper environment for the user program and performs a call to the entry point address of the program. The user program now starts running.

Start Up for a User Program

- A user program begins as a called function from the operating system.
- This function is typically called “main”.
- The main program receives some parameters from the calling code in the operating system. These are:
  1. argc – an integer, is the number of “arguments” by which a function was called.
  2. argv – This is actually an array of strings. Each string is an array of characters terminated by a null. (Recall that an array is represented as a pointer to the start of the array). So each element of the argv array is of type char *. Therefore the type for argv itself is char * *.
  3. envp – like argv is an array of strings describing the current run time environment. Therefore envp is also a pointer to pointers to character.

argc, argv data structure

Suppose we invoke a program with two arguments:
myprog data-in data-out

- The name by which the program has been invoked is “myprog”.
- The program has been called with two arguments:
  the first one is “data-in”, the second is “data-out”.
- Each argument is a C string, *i.e.* a null terminated array of characters. argv is an array of such strings.
- argv[0] is the name by which the program is invoked. In this example, argv[0] is a pointer pointing to the collection of characters: ‘m’, ‘y’, ‘p’, ‘r’, ‘o’, ‘g’ \0 in data memory.
- Similarly argv[1] points to characters ‘d’,‘a’,‘t’,‘a’,‘-’,‘i’,‘n’, \0 and argv[2] points to ‘d’,‘a’,‘t’,‘a’,‘-’,‘o’,‘u’, ‘t’ \0 in data memory.
- argv itself is an array of these three pointers, ending with a NULL pointer. Thus argv is of type char * *.
- envp is a similar data structure.

Example for argc, argv

You can compile and run the following c program:

```
#include <stdio.h>
#include <stdlib.h>
int main (argc, argv)
     int argc;
     char *argv[];
{
  int i;
  i = argc;
  printf("Function called as: %s ", argv[0]);
  printf("with %d arguments\n", argc-1);
  for (i = 1; i <argc; i++)
    printf("Argument %d is %s\n", i, argv[i]);
  exit(0);
}
```

Example for argc, argv

Save the c file as arg-test.c, then compile and link using
cc arg-test.c
Now run a.out with a number of arguments.
./a.out this that other

One can symbolically link a.out with other names.
ln -s a.out newname
ln -s a.out othername
Now run the executable with names a.out, newname and othername.

- Many program use the fact that it is possible to find by what name the program has been invoked.
- For example, programs compress and uncompress use the same binary code.
- When invoked as compress, the argument file is compressed.
- When invoked as uncompress, the argument file is uncompressed.

Right left rule for declarations in C

- In the given C program for examining arguments, we declared argv as: char *argv[ ]; – How do we interpret this?
- For interpreting declarations in C, we use the right left right rule.

We read the declarative keywords as:
* $\Rightarrow$ pointer to, [ ] $\Rightarrow$ array of
( ) $\Rightarrow$ Function returning

First find the identifier being declared. This is the name (which is not a key word) in the declaration.

From the identifier, we scan right till we hit the end or an unmatched right paranthesis.
Then we scan left till we hit the end of an unmatched left paranthesis. (These parantheses should not be a matched pair as used to denote a function.)

Continue this till the whole declaration is parsed. We shall illustrate the

Right left rule for declarations in C

Let us take the argv declaration:

char *argv[ ];
Identifier is argv. Going right, we find [ ].
So argv is an array of . . .
There is nothing more on the right, so we must go left. Here we find *
So argv is an array of pointers to . . .
As we continue on the left, we find char and that is where it ends.
So argv is an array of pointers to char.

That was easy!
Let us take something more complicated –
int *(*func( ))( );

Right left rule for declarations in C

int *(*func( ))( );

Identifier is func. going right, we find ( ).
So func is a function returning . . .
Going further right, we find the unmatched).
So we must stop here and go left.

On the left, we find *. So func is a function returning a pointer to . . .
Going further left we meet the unmatched (.
So we continue right from where we were stopped by a ‘)’.

Here We find ( ). So func is:
a function returning a pointer to a function returning . . .

Right left rule for declarations in C

int *(*func( ))( );

So far we had parsed the declaration as:
a function returning a pointer to a a function returning . . .
and We had reached the end on the right.
continuing left, we find a *. So, func is:

a function returning a pointer to a function returning a pointer to . . .

On continuing further left, we find int. Thus,

func is:
a function returning a pointer to a function returning a pointer to int.
Figure out that “int (*(*ptable)[])();” declares ptable as a pointer to an array of pointers to functions returning int.