Architecture of MP-SoC and Embedded System
Chapter-3

Architecture of MP-SoC and Embedded System

In today’s scenario in applications necessitate use of multi-processor systems for reasons of recital, scalability and power efficiency, systems becomes ubiquitous and integrated with a single system. Dynamic data management and mapping them on a multi-processor platform becomes a multidimensional problem[12][14]. Each possible set of applications that can be simultaneously active to a different use-case that the system has to be confirmed and experienced for analysing the feasibility and resource utilization of all possible use-cases becomes very challenging and intelligent. In design and development of Multiprocessor System-on-Chips (MP-SoC) is both compound and costly. Many non-trivial results and decisions must be made in order to create systems that are accurate and competent. This is in division due to the fact that anomalies are introduced when separation the single processor field and entering a multiprocessor field. Also, the high degree of density of both hardware and software in MP-SoC design makes propose decisions non-trivial[2][5].

3.1 MP-SoC

A MP-SoC is a combinational system of both hardware and software on a single chip. The hardware (or platform) is complete up of multiple processing elements on which the embedded software (the application) is running. Today, systems resembling these are becoming more and more multifaceted in design, and that requires a much more organized approach to decisions complete in the design process. In the design process of a MP-SoC many criteria have to be considered. These include but are not limited to the following:

Total Number of processing elements creation decisions on how many processing essentials are needed for the MP-SoC to have the desired behaviour is not a simple tasks. Focus on additional devoted or programmable processing elements may provide extra handing out power to the overall system. [6][9].

Processing elements on a MP-SoC can be either devoted to a specific job or operation or programmable and able to perform a wide range of different jobs(tasks-set). Programmable handing out elements facilitated much more flexibility and efficiency to the overall system. however, elements also require direction of the implementation time and any dependencies the system might have. In order to make dynamic decisions on which jobs(tasks set) should be
arranged processing time on the handing out elements, a need for real-time operating-systems (RTOS) arises. A RTOS manages the implementation time on a handing out element by scheduling the jobs by various scheduling policies to be executed how dependencies among jobs are determined [11][17].

**Dynamic mapping the application onto the platform** selecting which jobs should be executed or run on which processing elements. Tasks peremption, resource peremption are much required and moving the running of one tasks onto a different processing resources (element) could gain the overall system in terms of faster running or execution or less peremption. A dynamic mapping of the embedded application onto the selected platform therefore becomes a compound job(tasks) when creating a high-quality MPSoC in terms of accuracy by reaching timely deadlines, efficiency and cost[10][15].

**Selection of operating system** As mentioned earlier, a need for RTOS is described when several jobs(tasks) are mapped onto the same handing out(processing) element. This selection greatly affects the behaviour of the MP-SoC. three issues that describe the behaviour of an operating system are identified: synchronization, allocation and scheduling.[13][15].

(I) **Synchronization** different jobs(tasks) of the system share variables, and execution of assured jobs(tasks) required before others can executed. A mechanism which provide the information regarding dependencies and keep record of which dependencies are determined and which are not. This mechanism is the synchronizer. Each time a tasks finishes implementation, it checks whether dependencies have been determined and resets the dependencies when needed[13][16].

(II) **Allocation** different jobs(tasks) need access to the same resources like processors, CPU’s, communication channels, memories, etc. This admittance is administered by the synchronizer, keeps track of and grants the different tasks access to shared resources.[16].

(III) **Scheduling** In order to make a dynamic decision of which job(tasks) is the mainly appropriate for implementation(execution) on a given processing element, a scheduling purpose(principal) must be followed . Scheduling purpose(principle) can either be static or dynamic. A static principle uses a set criterion. This could either be user defined or based on a characteristic of the given job(tasks) e.g. its time period, interval. A dynamic purpose (principle) uses dynamically updated criteria for making scheduling decisions[16].
3.2 Architecture and Challenges Of MP-SoC

Multi-Processor System on-Chip (MP-SoC) architectures correspond to an emerging paradigm for implementing application which is a desired and specific solutions for various user constraints like time-to-market, performance and power consumption, cost, speed. Application-specific MP-SoCs are usually implemented and designed by using a platform-based approach, where a ample array of customizable parameters must be work to find the best trade-offs such as energy, delay. Multi-Processor Systems-on-Chip (MP-SoC) and Chip- Multi-Processors (CMPs) consider as the de facto standard for embedded and general-purpose architectures. The Design Space Exploration (DSE) phase is used to work the configurable system parameters and it usually consists of a multi-objective optimization research problems. The DSE research problem consists of exploring a large design space consisting of several parameters at system and micro-architectural levels. A heterogeneous MP-SoC is a set of different PEs (Processors) interface and interacting through a communication network [15]. PEs may support either hardware or software jobs(tasks) implementation and execution. Software jobs(tasks) execute in Instruction Set Processors (ISPs), while hardware tasks execute in reconfigurable logic (RL) or dedicated IPs. Figure 3.1 describes the heterogeneous MP-SoC Architecture. the available resources are one processor, named Manager Processor (MP), is associated and responsible for resource control, job(tasks) scheduling, job(tasks) binding, job(tasks) mapping, Job(tasks) migration, and configuration control. The MP starts the initial job(tasks) of each application. New tasks are Loaded into the MPSoc from the tasks memory when a communication to them is required, if they are not already mapped[12][14][24].

![Figure 3.1 MP-SoC Architecture](image)
Tasks scheduling is based on a queue policy. Figures shows three queues, one for each tasks type (i.e. hardware, software or initial). A job(tasks) enters on a queue if there are no free resources, and it waits until this condition changes. MPSoC is developing towards processor-pool (PP)-based architectures, which utilize hierarchical on-chip network for inter- and intra-PP communication. application-specific optimization of on-chip communication is a nontrivial job. The approach allows periodic and independent configurations of PPs. Heterogeneous Multi-Processor System-on-Chip (MP-SoC) and media processing are systematically applied in M-Commerce (mobile electronic commerce). heterogeneous MPSoCs give the more direction for parallelization accelerating of sequential media algorithms. Recent years have proved that MP-SoC is the most capable, efficient and feasible method to reach high incorporation and integration provided by the semiconductor technology under the user and market constraints of performance and power consumption [13,16]. due to the high computational and power consumption demands of modern embedded applications, especially for embedded visual media processing, From some view points, heterogeneous MPSoC lays the physical groundwork for mobile electronic commerce, but it highly associated with the difference of processing elements makes the parallel processing of sequential algorithms more challenging in dynamic data management and mapping at run-time . In Run-Time MP-SoC Parallel processing of heterogeneous MPSoC has been a critical and significant problem and one of the research points in embedded system. The multiprocessor System-on-Chip (MPSoC) is a system-on-a-chip (SoC) which uses multiple processors (see multi-core), usually targeted for embedded applications. It is used by platforms that contain multiple, usually heterogeneous, processing elements with specific functionalities reflecting the need of the expected application domain, a memory hierarchy (often using scratchpad RAM and DMA) and I/O components. All these components are linked to each other by an on-chip interconnect. These architectures meet the performance needs of multimedia applications, telecommunication architectures, network security and other application domains while limiting the power consumption through the use of specialized processing elements and architecture. A new heterogeneous architectural design paradigm is emerging usually called a 'platform', including one or more programmable components, either general-purpose or DSP processors, cores or ASIPs (application-specific instruction-set processor), augmented with some specialized data paths or co-processors (accelerators). Through this evolution, embedded processors become ubiquitous and a new role for embedded software in contemporary and future Multiple-Processor Systems-on-Chip (MP-SoC) is reserved. Next to these programmable components, they contain a large number of memories (DRAM, SRAM, FIFOs, ...) organized in many different ways. MP-SoC
Multiprocessor System-On-Chip. Most SoCs today use multiple processing cores. MP-SoCs are characterized by heterogeneous multiprocessors. CPUs, IPs (Intellectual Properties), DSP cores, Memory, Communication Handler (USB, UART, etc). All MP-SoC designs have the following requirements:

a. Speed  
b. Power  
c. Area  
d. Application Performance  
e. Time to market

(a) Components: The different Components of MPSoC as under:

- Hardware
  - Multiple processors
  - Non-programmable IPs
  - Memory
  - Communication Interface
  - Interface heterogeneous components to Comm. Network
  - Communication Network
    - Hierarchical (Busses)
    - NoC

(b) Application in various area:

- Cell phones
- Network Processors
  (Used by Telecomm. and networking to handle high data rates)

- Digital Television and set-top boxes
- High Definition Television
- Video games (PS emotion engine)

3.3 MP-SoC File Format

File is a collection of data-fields, records and domains. The structure of the MP-SoC file format, which is description of the way to store, developed and implemented applications for MP-SoC platforms. The MPSoc file format combines individual implementations for jobs(tasks) into a single binary file. MP-SoC binaries use the .mpsoc extension in their filenames. An MP-SoC
binary categorized into three segments, as shown in Figure 3.2. The first segment describes the file header; the second segment describes the application structure and an index for the implementation data files, which are located somewhere in the third segment. In an MPSoC binary, precisely one mpsocheader and at least one mpsoctask with at least one mpsocimpl should be defined. Other blocks are optional, and the number of blocks per type are bounded by the uint32t data type. Regardless of the implementation data files this binary contains, all its fields are stored in small byte order [5][8][13].

<table>
<thead>
<tr>
<th>Application Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASKS Graph</td>
</tr>
<tr>
<td>Header</td>
</tr>
<tr>
<td>Ports</td>
</tr>
<tr>
<td>Implementation</td>
</tr>
<tr>
<td>Channels</td>
</tr>
<tr>
<td>Constraints</td>
</tr>
<tr>
<td>Binaries</td>
</tr>
<tr>
<td>Header</td>
</tr>
<tr>
<td>Bootstrap Command</td>
</tr>
<tr>
<td>Data</td>
</tr>
</tbody>
</table>

Figure: 3.2 Structure of MP-SoC File Format

(I) **mpsocheader**

The file Command **mpsocheader** associated with data structure and it is placed at the start of a binary that describes an application for an MPSoC platform. This header defines the basic layout of the binary, and is defined in linux/mpsoc.h.

```
a. struct mpsocheader
b. {
c.  uint32t magic1;
d.  uint32t magic2;
e.  char name[16];
f.  uint32t notasks;
g.  uint32t noports;
h.  uint32t noimpls;
i.  uint32t nochannels;
j.  uint32t noconstraints;
k.  }
```
2. **Fields**

1. **magic1**
   A value that describes and identifies this file as an MPSoC file. Use the constant MPSOCMAGIC1 if the file is intended to execute on an MPSoC platform.

2. **magic2**
   Second file identification field which use the constant MPSOCMAGIC2 distinct and defined in linux/mpsoc.h to introduce this binary of type MP-SoC.

3. **name**
   A human readable illustration and representation of the application embedded in this binary. This field must be null-terminated, so the maximum value of length of the name is 15 characters.

4. **noports**
   An integer indicating the total number of ports attached to the tasks in the application.

5. **notasks**
   An integer describing and specifying the number of mpsoctask data structures that follow. This is the number of tasks within the application.

6. **nochannels**
   An integer describing and specifying the number of mpsocchan data structures that follow. This is the number of channels in the application.

7. **noconstraints**
   An integer describing and specifying the number of mpsocconstraint data structures that follow. This is the number of constraints posed on the application.

8. **noimpls**
   An integer representing the total number of implementations for the tasks in the application.

(II) **mpsoctask**

```
struct mpsoctask
```
Fields

1.id

An integer that uniquely identifies and represents this task within an application. A valid range for this field is [0..notasks-1].

2.name

This field must be null terminated, so the maximum length of the name is 15 characters.

3.noports

An integer describing and specifying the number of mpsocport data structures that follow. This should correspond with the number of incoming and outgoing channels to this task.

4.noimpls

An integer specifying the number of mpsocimpl data structures that follow. This is the number of implementations available for this task.

(III)mpsocport

An mpsocport block has three required attributes.

struct mpsocport {
    uint32t id;
}
kairosportt type;

uint32t rate;

};

Fields

1. id

An integer that uniquely specifies and identifies this port within an application. This id is used when connecting the port to a channel. A valid range for this field is [0..noports-1].

2. type

An listing value of type kairosportt, declared linux/kairos/application/port.h. This field describes and specifies whether this port is an input or output port for the tasks it is attached to. When a port is of type PORTIN, the actor will during a firing read tokens from the channel to which the port is connected. When a port is of type PORTOUT, the actor will during a firing write tokens to the channel to which the port is connected.

3. rate

An integer that indicates the communication rate of the tasks.

(III) mpsocimpl

It represents the characteristics of an implementation and execution for a specific architecture, and its location within the MPSoC binary. If an implementation can be used for multiple tasks, its data file has to be provided only once, but for each tasks it implements an mpsocimpl structure should be generated.

struct mpsocimpl

{
    kairosarcht arch;
    uint32t mips;
    uint32t memorySize;
    uint32t numSlots;
uint32t execTime;
uint32t dynamicCost;
uint32t offset;
uint32t size;
uint32t align;
};

**Fields**

1. **arch**

An list value of type archtypet, declared in linux/kairos/platform/arch.h. This specifies the architecture required to execute this implementation.

2. **memSize**

An integer representing the maximum size (in bits) of the state of the implementation’s tasks on the specified architecture.

3. **mips**

An integer representing the amount of processing capacity required to achieve some functionality within execTime time. memSize An integer indicating the maximum size (in bits) of the state of the implementation’s tasks on the specified architecture.

4. **execTime**

An integer describing and indicating the (worst-case) execution time (in time-units) of the implementation’s tasks on the specified architecture.

5. **numSlots**

An integer indicating the number of processes this implementation will spawn when executed. Note that some architectures or platforms are not capable of multi-tasking.
6. **dynamicCost**

An integer indicating and representing the increased energy consumption of the specified architecture during execution of this implementation, as opposed to the static energy consumption of the specified architecture.

7. **Offset**

It returns Offset to the beginning of the data file for this implementation.

8. **Size**

Size of the implementation data file.

9. **Align**

The power of 2 alignment for the offset of the implementation data file for the architecture specified in arch within the binary. This is required to ensure that, if this binary is changed, the contents it retains is correctly aligned for virtual memory paging and other uses.

(IV) **mpsocchan**

It defines a communication channel between two ports. The ports of a channel can be attached to the same tasks (self-loop) or they can connect two distinct tasks.

```c
struct mpsocchan
{
    uint32t id;
    uint32t src;
    uint32t dst;
    uint32t initialTokens;
    uint32t tokenSize;
    uint32t minBandwidth;
};
```
Fields

1. id

An integer that uniquely identifies this channel within an application. A valid range for this field is [0..nochannels-1]

2. src

The source port of this channel. Should be specified with an id of an mpsocport. A valid range for this field is [0..noports-1].

3. Initial Tokens

The number of tokens that reside on this channel when the application is not yet started.

4. tokenSize

The size of the tokens that are communicated over this channel.

5. minBandwidth

The minimal amount of bandwidth that should be reserved for this channel.

6. dst

The destination port of this channel. Should be specified with an id of an mpsocport. A valid range for this field is [0..noports-1].

(V) mpsocconstraint

Defines constraints posed on the mapping of the application in this binary.

typedef enum {
    CONSTRAINTTHROUGHPUT = 0,
    CONSTRAINTLATENCY = 1
} kairosconstraintt;

struct mpsocconstraint {

kairosconstraintt type;
uint32t numerator;
uint32t denominator;
uint32t src;
uint32t snk;
};

**Fields type**
An list of type constraint, declared in linux/kairos/application/constraint.h. Two types of constraints can be specified. A throughput constraint specifies the minimal throughput that should be achieved by the application. A latency constraint (CONSTRAINTLATENCY) is given the maximum allowed latency between two tasks.

1. **numerator**
The number of fractional units that determines the value of this constraint.

2. **denominator**
The divisor of the fractional value of this constraint.

3. **src**
The source tasks as one end of a latency constraint.

4. **snk**
The sink tasks as the other end of a latency constraint.

**3.4 MP-SoC Requirements**
In proposed work, the MPSoC needs are described and represented in the situation of embedded platforms. The general purpose multiprocessor platform, also denoted as a Chip Multi-Processor (CMP), is determined and associated with combining multiple processors on a single chip to expand overall compute performance. In contrast, the embedded platform also amplified the user and process constraints. The input needs for embedded MP-SoC platforms are flexibility,
scalability, predictability and real-time behaviour. MP-SoC platforms require low power consumption, while providing a large amount of processing power i.e. performance. Finally, these MP-SoC solutions should be manage all trade-off between user constraints, process constraints and resource utilization[12][10][8].

**Flexibility and Programmability**

Flexibility can be defined as the capacity and ability to use the same hardware resources like processors, CPU’s, Memories for different purposes. Flexibility is needed to handle dynamic application standards, carry new applications. The outlook or view on timing constraints like deadline, interval between an embedded platform and a general purpose platform is essentially different. Major difference for timing constraint in case of an embedded platform, while for a general purpose platform, minimized the cost of component. Programming effort can be implemented and defined as the design effort necessary to get a wide variety of applications running within specifications on the MP-SoC platform. Custom design for a single product or application is not much feasible and available for longer time.

**Scalability**

Platform scalability is much required and basic important, both from a design-time MP-SoC of view as from a run-time MP-SoC point of view. Design-time MP-SoC scalability implies the option of adding more components to the platform without the requirement or need for a global platform redesign. Run-time MP-SoC scalability can be described and implemented as the ability to execute the same application without redesign on both a high-end, resource-rich platform and a low-end platform with minimal resources[12][18].

**Predictability and Real-time behavior**

Predictability refers to the degree that one can make a correct result of the application behavior. Real-time performance indicates to the degree at which the expected application or system output is produced within the predicted time-frame. Predictability and real-time behavior are important to ensure the always functioning user requirement of an embedded system[12][18].

**Performance**

Multiprocessor compute performance is important because existing multimedia and wireless applications require more handing out power than a single flexible, programmable processing element can deliver. Furthermore, the user often wants to run multiple applications at the same
time or some applications like e.g. video conferencing will require multiple platform services simultaneously[12][18].

**Power consumption**

Power consumption is definitively an issue for battery-operated devices. Limiting the power consumption is better for the lifetime of the chip, potentially avoids the need for a more expensive package and, reduces the power leakage. With respect to single processor platforms, multiprocessor platforms enable a abridged and reduce power consumption. MP-SoC can reduce the power consumption by executing an application on multiple processing elements operating at a lower voltage and frequency[12][18].

**Reliability**

Reliability can be defined as the capability of a system or component to perform its required functions under stated conditions for a specified period of time. In other words, a reliable MP-SoC hardware or MP-SoC software component time after time performs according to its specifications and, in theory, is totally free of technical errors. The main components of the System-on-Chip (SoC) are: the processing elements, responsible for communicating, responsible for running the applications, responsible for storing both the application data and instructions, the on-chip memories, with the outside world and, finally, the on-chip interconnect structure, I/O components, responsible for linking the processing elements with the memories and the I/O components[12][18].

**3.5 MP-SoC Run-Time Management**

From an generalization viewpoint, the run-time manager is always situated and located between the design time MP-SoC platform and the application. It is calm of a system manager and a run-time library[5][13]. The system manager, associated and defining the quality manager and the resource manager, is responsible for decision making and implementation, while the Run-Time Library (RTLib) provides platform abstraction services. The run-time library associates and collaborates with the system manager for executing the decisions, for run-time MP-SoC interaction with the applications and for monitoring both platform resources[18].
3.6 Run-Time MP-SoC Constraints

The run-time manager provides a glue between the platform hardware layer and the application layer. This means that the run-time manager has to facilitate the services required by the application by building on top of the available platform services. The run-time manager has to associate and operate within the boundary constraints of the MP-SoC environment. The MP-SoC run-time manager role to computing fields when it comes to applications, platform hardware and non-functional constraints[5,13,18].

Platform hardware. The MP-SoC platform hardware is in fact the on-chip equivalent of traditional parallel and distributed systems. It describes how many resources are available at design-time. It provides the heterogeneous architecture and self-possessed of multiple processing elements with a compound memory hierarchy.

Applications functionality the actual user value of the MP-SoC device has efficiency of handling a dynamic set of applications. This requirement is typically found in the desktop computing field. In this field, to maximize the overall throughput and to provide fairness between applications.

Non-functional application constraints. The discussed MP-SoC platforms will often end up in embedded and/or mobile devices. This means the runtime manager will have to consider the
constraints that deal with resource lacking. This includes, for example, a limited amount of battery power.

3.7 Embedded System

Embedded systems associated with general-purpose computer systems. Embedded system is most of the use and justification of computer system products, the nearest to the field. In the past, embedded research and development was classified, identified and implemented with real-time systems and industrial environment, but things have been changing since. With wider deployment of technology and computers, the need for embedded systems has increased. embedded systems are most of house appliances, cars, microwave, air condition, and industrial devices and tools. [5,13]

3.7.1 Types of Embedded System

(I) Microcontroller based embedded system

Microcontrollers can be considered as independent systems which associated with a processor, memory and peripherals which is the industry needed environment to use them within an embedded system are to add software. The processors are usually based on 8 bit stack based architectures such as the MC6800 family. These are controlled in their functionality but their low cost has meant that they are used in many uncertain applications. Microcontrollers are usually available in several forms: • Devices for prototyping or low volume production runs These devices use non-volatile memory to allow the software to be downloaded and returned in the device. UV erasable EPROM used to be the most dedicated but EEPROM is also gaining dedicated. These devices use EPROM instead of the ROM but instead of using the ceramic package with a window to allow the device to be erased, it was packaged in a cheaper plastic pack and thus was only capable of programming a single time — hence the name. These devices are cheaper than the prototype versions but still have the programming disadvantage. However, their lower cost has made them a suitable alternative to producing a ROM device. For low to medium production quantities, they are cost effective and offer the ability to customise software as necessary[18][13].

(II) Microprocessor based Embedded System

Microprocessor-based embedded systems initially took existing general-purpose processors such as the MC6800 and 8080 devices and implemented systems around them using external peripherals and memory. The use of processors in the PC market continued to provide a series of
faster and faster processors such as the MC68020, MC68030 and MC68040 devices from Motorola and the 80286, 80386, 80486 and Pentium devices from Intel[5][13].

(III)Board based embedded system

An alternative is to use hardware that has already been built and tested such as board-based systems as provided by PCs and through international board standards such as VMEbus. The main advantage is the reduced work Load and the availability of ported software that can simply be utilised with very little effort.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Au1xxx</td>
<td>Advanced Micro Devices</td>
</tr>
<tr>
<td>ARM7, ARM9</td>
<td>ARM</td>
</tr>
<tr>
<td>C167CS, C165H, C164Cl</td>
<td>Infineon</td>
</tr>
<tr>
<td>5282, 5272, 5307, 5407</td>
<td>Motorola/Freescale</td>
</tr>
<tr>
<td>I960</td>
<td>Vmetro</td>
</tr>
<tr>
<td>32170, 32180, 32182, 32192</td>
<td>Renesas/Mitsubishi</td>
</tr>
<tr>
<td>MMC2113, MMC2114</td>
<td>Motorola/Freescale</td>
</tr>
<tr>
<td>R3K, R4K, 5K, 16</td>
<td>MTI4Kx, IDT, MIPS Technologies,</td>
</tr>
<tr>
<td>82xx, 74xx,8xx,7xx,6xx,5xx,4xx</td>
<td>IBM, Motorola/Freescale</td>
</tr>
<tr>
<td>680x0 (68K, 68030, 68040, 68060, ...), 683xx</td>
<td>Motorola/Freescale,…</td>
</tr>
<tr>
<td>SH3 (7702,7707, 7708,7709), SH4 (7750)</td>
<td>Hitachi,…</td>
</tr>
</tbody>
</table>

3.7.2 Real Time Embedded system

In ways virtually incredible just a few decades ago, embedded systems are new innovation and reshaping the way people live, work, and play. For example, in most vehicles driven today embed intelligent computer chips that perform value-added tasks, which make the vehicles easier, cleaner, and more facility using microcontroller based system. Telephone systems is the one example in which system rely on multiple associated and integrated hardware and software systems to communicate or connect people around the world. Many of these embedded systems are reliable, feasible, predictable. The devices that embed them are convenient, user-friendly, and dependable. One special class of embedded systems is illustrious from the take a break by its requirement to take action to external events in real time. This is classified as the real-time embedded system.“As shown in Figure 3.4 embedded systems in the home assume many forms, including cable and satellite boxes for televisions, security systems, home theater systems, and telephone answering machines. As advances in microprocessors continue to improve the functionality of ordinary products, embedded systems are helping drive the development of additional home based innovations[5][13].
3.7.3 Issues in Real Time Embedded system

The expected output and benefits from a change to component technology are its price and reduced development times scorching from the accessibility of reusable components that can be bought as off-the-shelf items. In market hardware components are available but lacking in development of software applications like RTOS and failure detection and finding out arriving at applications that described and consist mainly of assemblies of separately procured components[18].

3.8 Scheduling Policies for Uni-Processors

In a multiprogramming system, multiple processes stay alive simultaneously in main memory. Each process alias and alternates between using a processor and waiting for some event to take place, such as the end of an I/O operation. Either the processor or processors are kept hectic by running and executing one process while the others are in queue for waiting. The objective of processor scheduling is to allocate processes to be executed by either the processor or processors in deadlock condition, In real time embedded systems scheduling policy and activity categorized into three separate functions: long term, medium term, and short term scheduling.
3.8.1 Types of Scheduling

Table 3.2: Types of Scheduling

<table>
<thead>
<tr>
<th>Long Term Scheduling</th>
<th>The action to insert to the pool of tasks, jobs, processes to be executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium Term Scheduling</td>
<td>The action to insert to the number of tasks, jobs, processes that are partially or fully in main memory</td>
</tr>
<tr>
<td>Short Term Scheduling</td>
<td>The action as to which accessible process will be executed by the Processor</td>
</tr>
<tr>
<td>I/O Scheduling</td>
<td>The action as to which process’s pending I/O request shall be handled by an accessible I/O device</td>
</tr>
</tbody>
</table>

**Long-Term Scheduling**

The long-term scheduler determines which programs are described and admitted to the system for processing. It controls the degree of multiprogramming. Once occurrence a job or user program becomes a process and is inserted to the queue for the short-term scheduler. In some systems, a newly created or occur process starts in a various swapped-out condition, in which case it is inserted to a pool of queue for the medium-term scheduler. when batch type system implemented, or for the batch section of a operating system, recently submitted jobs are routed to secondary memory storage (disk) and stored into a batch queue. There are two actions created and generated here. First, the scheduler take have to take action or make a decision when the operating system can take on more than one additional processes. Second, the scheduler must make a decision which job or jobs to accept and turn into processes. Let us briefly consider these two actions. The action as to when to create a new process is generally determined by the desired degree of multi programming. More jobs are competing for the same amount of processor time. Thus, the long-term scheduler have some limitation like the degree of multiprogramming to give acceptable service to the existing set of jobs. Each time a job terminates, the scheduler may take a decision to add one or more new jobs. For active programs in a time-sharing system, a process design request can be generated by the act of a user attempting to connect to the system[18].
Medium-Term Scheduling

Medium term scheduler action is based on the require to handle the degree of multiprogramming. it cannot use virtual memory, memory management is also an challenge. So, the swapping-in action will judge the memory requirements of the swapped-out processes[13].

Short-Term Scheduling

During the frequency of execution, the long-term scheduler run and executes quite occasionally and makes the action of whether or not to give on a generated or new process. The medium-term scheduler is executed somewhat more commonly to take a swapping action. It is also known as the dispatcher, run by most commonly and makes the fine-grained action of which process to execute next. The short-term scheduler is invoked whenever an event occurs that may lead to the blocking of the current process or that may provide an opportunity to pre-emption a currently running process in support of another. Examples of such events include

• Clock interrupts

• I/O interrupts

• Operating system calls

• Signals (e.g., semaphores)

3.8.2 Scheduling Criteria for Uni-Processors

Turnaround time This is the period of time among the capitulation of a process and its completion. turnaround time associated with actual execution time and time spent waiting for resources like memory CPUs, processor. This is an appropriate calculate or estimation for a batch job[18].

Response time for an interactive process, this is the time from the capitulation of a request until the response begins to be received. A job can begin to implement some output to the user while progressing to job the request. So, this is an enhanced calculation than turnaround time from the user’s requirements[18].

Deadlines When job or process or tasks ended deadlines can be precise, the scheduling discipline should secondary other achievement to that of maximizing the percentage of deadlines met.
**Predictability** A given job should execute the same quantity of time and at about the same cost despite of the Load on the system. A wide disparity in response time or turnaround time is off-putting to users.

**Throughput** The scheduling policy should attempt to maximize the number of processes ended per unit of time. This is a calculation of how much work is being performed. This clearly depends on the average length of a process but is also partial by the scheduling policy, which may affect resource utilization.

**Processor utilization** This is the percentage of time that the processor is full of activity or busy. For an expensive shared system, this is a important condition. In single-user systems and in some other systems, such as real-time systems, this condition less important than some of the others.

**Enforcing priorities** When processes are allocated or assigned priorities, the scheduling policy should supports higher-priority processes.

**Balancing resources** The scheduling policy follow to keep and track the resources of the system is full of activity or busy. jobs which will underutilize for overLoaded resources should be supported. This condition also associated with medium-term and long-term scheduling.

### 3.8.3 Comparison of Various Scheduling Policies

- **First-come-first-served**: Select the process that has been waiting the longest for service.

- **Round robin**: Use time quantum (slicing) to limit any running process to a short burst of processor time, and rotate among all ready processes.

- **Shortest process next**: It identifies the process with the shortest expected processing time, and do not preempt the process.

- **Shortest remaining time**: It identifies the process with the shortest expected enduring process time. A process may be preempted when another process becomes ready.

- **Highest response ratio next**: it is a foundation the scheduling action on an calculate approximately of normalized turnaround time.

\[ R = W + S/s \]

R  response ratio
w  time spent waiting for the processor
s  expected service time
Find out the Finishing Time, Turnaround Time by Various Scheduling Policies

Table 3.3: Process Table

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.4: Finishing time & Turnaround Time of Scheduling Policies

<table>
<thead>
<tr>
<th>Process</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finish Time</td>
<td>3</td>
<td>9</td>
<td>13</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>Turnaround Time</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>RR q = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finish Time</td>
<td>4</td>
<td>18</td>
<td>17</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>Turnaround Time</td>
<td>4</td>
<td>16</td>
<td>13</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>RR q = 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finish Time</td>
<td>3</td>
<td>17</td>
<td>11</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>Turnaround Time</td>
<td>3</td>
<td>15</td>
<td>7</td>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>SPN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finish Time</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td>20</td>
<td>11</td>
</tr>
<tr>
<td>Turnaround Time</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>SRT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finish Time</td>
<td>3</td>
<td>15</td>
<td>8</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>Turnaround Time</td>
<td>3</td>
<td>13</td>
<td>4</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>HRRN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finish Time</td>
<td>3</td>
<td>9</td>
<td>13</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>Turnaround Time</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>14</td>
<td>7</td>
</tr>
</tbody>
</table>
The operating system must follow and make three types of scheduling policies and methods with respect to the execution of processes. Long-term scheduling determines when new processes are admitted to the system. Medium-term scheduling is part of the swapping function and determines when a program is brought partially or fully into main memory so that it may be executed. Short term scheduling determines which ready process will be executed next by the processor[18].

3.9 Scheduling Policies for multiprocessors

In Real time Embedded system and RTS (Real Time System) design issues is explored and defined. This is process followed and running by a various scheduling of processes and jobs on a multiprocessor system then after altered design data for multiprocessor thread scheduling are examined and check for run-time. In Multiprocessors system supprts and dedicated work with real-time scheduling, deadline scheduling and rate monotonic scheduling When a computer system contains multiprocessors, some new issues are introduced into the invent of the scheduling policy. Types of multiprocessor as under :

• **Loosely coupled or distributed multiprocessor,** is a collection and coloration and collection of relatively independent systems, each processor having its own main memory.

• **Functionally specialized processors:** some of the common example is an Input / output processor. For the processor In this case one processor work as master and second is general-
purpose processor. Specialized processors are managed and controlled by the master processor and supports the service oriented environment.

- **Tightly coupled multiprocessing:** it associated and work together on a single chip and a set in this case all the processors shared a common memory and managed by operating system.

### 3.9.1 Design issues in multiprocessors scheduling policies

Scheduling on a multiprocessors and MP-SoC involves three interrelated issues:

**Allocation of Processes to Processors** when data theory or assumptions is widely use and the architecture of the multiprocessor and MP-SoC is standardized, it indicates not a single processor has a particular benefit with esteem to access to either input-output device or main memory then the simplest scheduling follows to handle the processors as a shared common resource and assign processes to processors on order time. A query produce as to whether the assignment should be either static or dynamic. If a process is directly assigned to one processor for long time from activation and execution until its completion, then a committed short-term queue is managed and maintained for each processor. Application of this approach is that there may be less complexity in the scheduling function, because the processor assigned is made a signle occurrence and for all. A policy which use of dedicated processors and it known as group scheduling, as discussed later. A drawback of static assignment is one processor can not be utilized for tasks completion, with a null or empty queue, while another processor has successfully executed. To discard this situation, a shared pool of queue can be used. All processes want to execute into a global queue and are scheduled to any available processor. Until the process is not terminated or discarded, the process may be run and executed on ubiquities processors at different times. In a tightly coupled common memory architecture, the process id, process name, process arrival time for every processes will be available to all processors, so the average time of scheduling a process is not depended to identity of the processor on which it is scheduled. Another feature is dynamic Load balancing, in which threads simultaneously moved from one queue of processor to a queue for another processor. Linux follow this approach and application of Load balancing. In spite of whether processes are assigned and dedicated to processors, it means that required to assign processes to resources like processors. Main two approaches have been followed (I) used: master/slave and (II) peer. Kernel is a heart of operating system and master/slave architecture focused on key kernel functions of the OS always execute on a dedicated processor. The other processors is able to execute user programs. The master is integrate all scheduling job and key application and
responsible for scheduling jobs. Once a process is occur and work active at that time the slave needs service, so it is necessary to dispatch a request to the master and wait for the response to achieve some service. This application is simple but requires some augmentation to a uniprocessor multiprogramming operating system[18].

**Process Scheduling**

In multiprocessor systems are not allocated and assigned to a committed processor. processes process are always dedicated to the processors but either it can be used a single queue for all processors or some sort of priority scheme is applicable for used, there are more than one queues based on priority, it is necessary to all follow and added into the pool for performing their service. Complex or compound resolution is simplified because one processor has integrated and control and maintain all memory and input/output resources. There are two drawback to this application: (1) if any type of failure in the master it effects all the system and it put down the entire system, and (2) the master can restricted to access processes and execution of process and create blockage. Kernel can run and executed on any of the processor from pool in peer architecture. and each processor work strategy is independent scheduling from the pool of available active processes. This application create complex environment for operating system and complicated for the execution process. The operating system have to indicate that two processors do not select the same process from the requested process and as a result process lost from the queue. Techniques or method must be engaged to resolve and synchronize competing claims to resources. [13][15].

**The Use of Multiprogramming on Individual Processors** Individual processor is able to execute concurrent process means one processor can execute multiple process so quickly it looks like that multiple process execute same time but processor execute only one process at one that and other process need to wait. In multiprogramming on individual processors needs synchronization between each process. One process will allocate one processor for particular amount of time for its process execution when it allocated time expired process goes in to waiting state it depends upon process scheduling algorithm. In multiprogramming individuals program execute in individual processor based on scheduling algorithm. The use of multiprogramming on Individual processors can increase use of resource utilization like processor, I/O, Memory. Multiprogramming on individual processor makes processor as much as busy as possible[10,18].
Process Dispatching** The active design issue associated to multiprocessor scheduling is the assignment, allocation and selection of a process to execution and run. The use of priorities or of complicated scheduling algorithms based on past usage may improve performance over a simple-minded first-come-first-served strategy. When Researcher consider multiprocessors, these complexities may be unnecessary or even counterproductive, and a simpler approach may be more effective with less overhead. In the case of thread scheduling, new issues come into play that may be more important than priorities or execution histories[14][16].

3.10 Real-Time Operating System

RTOS classified into two components namely “Real-Time” and “Operating System”.

**Real-Time**

Real-Time shows an expecting response or reaction to an event on the instant of its evolution. The expectant response show the logical accuracy of the result created. The instant of the events’ progress depicts deadline for generating the result[18].

**Operating System**

Operating System (OS) is a system program that provides an interface or middleware between hardware and application programs. OS is commonly able to with features like: Multitasking, multiprogramming, Synchronization, Interrupt and Event Handling, Input/Output, Inter-tasks Communication, timers and clocks and memory management, disk scheduling, thread synchronization to fulfill its primary role of managing the hardware resources to meet the demands of application programs. RTOS is therefore an operating system which supports real-time applications and embedded systems by providing logically correct result within the deadline required. RTOS is required for some component of all real-time application to complete process in some given deadline. An embedded system in a simple air conditioner does not require RTOS, but as the complication of applications expands ahead of simple tasks, today’s Modern embedded systems are designing more complex hardware-wise with all generation. Every time added new features into them. Embedded Application programs execution on the embedded system platforms will become more and more complex to be maintained, managed and run. Embedded Application program struggle to find out the response from the system. An RTOS will be efficient to allow the execution and run real-time applications to be developed and raised more easily [13][16][18].
A real-time operating system (RTOS) is the need and primary feature of many embedded systems today and facilitates a software environment on which to develop applications. To design an RTOS application is more complex from a simple application for a digital stopwatch to a much more complex application for aircraft navigation. [14][16][18]. For example, in some applications, kernel which is a core of RTOS can comprise kernel, which is the centre decision-making software that associated with the features of scheduling, minimal logic, and resource management. RTOS can be a combination of various modules, including the kernel, a file system, networking protocol stacks, and other components required for a particular application, as illustrated at a high level in Figure 3.8.

Figure: 3.8 High-level view of an RTOS, its kernel, and other components found in embedded systems.
3.10.1 Classification of RTOS

RTOS’s are generally classified into three types,

(I) Hard Real Time RTOS,

(II) Firm Real Time RTOS and

(III) Soft Real Time RTOS as described below:

• **Hard real-time**: degree of acceptance for missed deadlines is very small or zero. A missed deadline has tragic results for the system[13][18].

• **Firm real-time**: missing and to complete process in a deadline which produce the result in an unacceptable quality decrease[18].

• **Soft real-time**: processes are not complete in given deadlines and can be improved from system. decline in system quality is acceptable[13].

3.10.2 RTOS Architecture

The architecture of an RTOS is dependent on the complexity of its deployment. Good RTOSs are scalable to meet different sets of requirements for different applications. For simple applications, an RTOS usually comprises only a kernel. For more complex embedded systems, an RTOS can be a combination of various modules, including the kernel, networking protocol stacks, and other components as illustrated in Figure 3.9

![Figure:3.9 Real-Time Operating System](image-url)
3.10.3 Kernel

An operating system generally consists of two parts: kernel space (kernel mode) and user space (user mode). Kernel is the smallest and central component of an operating system. Its services include managing memory and devices and also to provide an interface for software applications to use the resources. Additional services such as managing protection of programs and multitasking may be included depending on architecture of operating system. There are three broad categories of kernel models available, namely:

**Monolithic kernel**

It runs all basic system services (i.e. process and memory management, interrupt handling and I/O communication, file system, etc) in kernel space. As such, monolithic kernels provide rich and powerful abstractions of the underlying hardware. Amount of context switches and messaging involved are greatly reduced which makes it run faster than microkernel. Examples are Linux and Windows[5][13].

![Monolithic Kernel Base Operating System](image)

**Microkernel**

It runs only basic process communication (messaging) and I/O control. The other system services (file system, networking, etc) reside in user space in the form of daemons/servers. Thus, microkernels provide a smaller set of simple hardware abstractions. It is more stable than monolithic as
the kernel is unaffected even if the servers failed (i.e. File System). Examples are AmigaOS and QNX[13][18].

3.10.4 Tasks Management

Tasks management allows programmers to design their software as a number of separate “chunks” of codes with each handling a distinct goal and deadline. This service encompasses mechanism such as scheduler and dispatcher that creates and maintain tasks objects.

Tasks Object

To achieve concurrency in real-time application program, the application is decompose into small, schedulable, and sequential program units known as “Tasks”. In real-time context, tasks is the basic unit of execution and is governed by three time-critical properties; release time, deadline and execution time. Release time refers to the point in time from which the tasks can be executed. Deadline is the point in time by which the tasks must complete. Execution time denotes the time the tasks takes to execute.

A tasks object is defined by the following set of components:

- Tasks Control block (Tasks data structures residing in RAM and only accessible by RTOS)
- Tasks Stack (Data defined in program residing in RAM and accessible by stack pointer)
• Tasks Routine (Program code residing in ROM)

<table>
<thead>
<tr>
<th>Tasks ID</th>
<th>Tasks name</th>
<th>Arrival time</th>
<th>Processing time</th>
<th>Completion time</th>
<th>Execution time</th>
</tr>
</thead>
</table>

Figure 3.12 Typical Tasks Control Block (TCB)

Each task may exist in any of the four states, including running, ready, or blocked and dormant as shown in Figure 5.9. During the execution of an application program, individual tasks are continuously changing from one state to another. However, only one task is in the running mode (i.e., given CPU control) at any point of the execution. In the process where CPU control is change from one task to another, context of the to-be-suspended tasks will be saved while context of the to-be-executed tasks will be retrieved. This process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching[3][8].

A State denote microprocessor is executing the tasks

A state denote tasks is waiting for external events

A state denote tasks waiting for other higher priority tasks to be

Figure 3.13 State Transition of Processes
Scheduler

The scheduler keeps record of the state of each task and selects from among them that are ready to locate the CPU to one of them. A scheduler helps to maximize CPU utilization among different tasks in a multi-tasking program and to minimize waiting time. There are generally two types of schedulers: non-preemptive and priority-based preemptive. Non-preemptive scheduling or cooperative multitasking requires the tasks to cooperate with each other to explicitly give up control of the processor. When a task releases the control of the processor, the next most important task that is ready to run will be executed. A task that is newly assigned with a higher priority will only gain control of the processor when the current executing tasks voluntarily gives up the control. Figure 3.14 gives an example of a non-preemptive scheduling[3][5][18].

Tasks Priority

![Non-preemptive Scheduling Diagram](image)

Priority-based preemptive scheduling requires control of the processor be given to the tasks of the highest priority at all time. In the event that makes a higher priority tasks ready to run, the current tasks is immediately suspended and the control of the processor is given to the higher priority tasks. Figure 3.15 shows an example of a preemptive scheduling.
**Dispatcher**

The dispatcher gives responsibility and control of the CPU to the tasks assigned or allocated by the scheduler for performing switching and find out the best path of execution. When RTOS is executed, the path of execution passes through one of three areas: through the tasks program code, through an interrupt service routine, or through the kernel[5][18].

### 3.10.5 Selection of RTOS

RTOS tends to be a selection for many embedded projects. But is an RTOS always necessary? The answer lies on careful analysis in understanding what an application needs to deliver to determine whether implementing RTOS is a requirement or an extravagance. Most programmers are not familiar with RTOS constraints and requirements. An RTOS is usually chosen based on its performance or one’s comfort and familiarity with the product. However, such a selection criteria is insufficient. To make matter worse, there is a wide variety of RTOS ranging from commercial RTOS, open-source RTOS to internally developed RTOS to choose from. Therefore, it is incumbent upon the programmers to exercise extra caution in the selection process. The selection criteria of RTOS can be broadly classified into two main areas; technical features of RTOS and commercial aspect of the implementation[18][19].
3.10.6 Tasks Synchronization & Inter tasks Communication

Tasks synchronization and inter tasks communications serves to enable information to be transmitted safely from one tasks to another. The service also makes it possible for tasks to coordinate and cooperate with one another[3][18].

Tasks Synchronization

Synchronization is needed and essential for tasks to share concurrent resources and allow multiple simultaneous tasks to be executed. Operating system Tasks synchronization is through two important methods; 1) Event Objects and 2) Semaphores. Event objects have necessary to use tasks synchronization is required without sharing a common resource. They allow one or more tasks to keep waiting for a specified event to occur. An event object can exist in either of two states: triggered and non-triggered. An event object in a triggered state indicates that a waiting tasks may resume. In contrast, if the event object is in a nontriggered state, a waiting tasks will need to stay suspended[5][18].

when a tasks releases the semaphore. Generally. There are three types of semaphore:

- Binary Semaphores (semaphore value of either 0 or 1 to indicate unavailability and availability respectively)
- Counting Semaphores (semaphore value of 0 or greater indicating it can be acquired/released multiple times)
- Mutually Exclusion Semaphores (semaphore value of 0 or 1 but lock count can be 0 or greater for recursive locking)

3.10.7 Inter tasks Communication

Inter tasks communication and interpretation involves sharing of common data among tasks along with sharing of common memory space, sharing of data. few of methods available for running inter task communications includes:

- Message queues
- Pipes
- Remote procedural calls (RPC)
A message queue is an entity used for inter task communication stored into a shared memory through which tasks send or receive messages placed in a shared memory. Tasks and ISRs send and receive messages to the queue through services provided by the kernel. A task seeking for a message from an empty queue is blocked either for a duration or until a message is received. The sending and receiving of messages to and from the queue may follow 1) First In First Out (FIFO), 2) Last in First Out (LIFO) or 3) Priority (PRI) sequence[16][18][19].