## PERFORMANCE METRICS AND SCHEDULING ALGORITHMS

**Performance Metrics of RTOS- Task Specifications-Task state - Real Time Scheduling algorithms: Cyclic executive, Rate monotonic, IRIS and Least laxity scheduling- Schedulability Analysis**

### Performance Metrics of RTOS

**An embedded system typically has enough CPU power to do the job, but typically only just enough — there is no excess. Memory size is usually limited. It is not unreasonably small, but there isn't likely to be any possibility of adding more. Power consumption is usually an issue and the software — its size and efficiency – can have a significant bearing on the number of watts burned by the embedded device. It is clear that it is vital in an embedded system that the real time operating system (RTOS) has the smallest possible impact on memory footprint and makes very efficient use of the CPU.**

### RTOS metrics

**There are three areas of interest when looking at the performance and usage characteristics of an RTOS:**

- **Memory – how much ROM and RAM does the kernel need and how is this affected by options and configuration**
- **Latency - which is broadly the delay between something happening and the response to that occurrence. This is a particular minefield of terminology and misinformation, but there are two essential latencies to consider: interrupt response and task scheduling.**
- **Performance of kernel services - How long does it take to perform specific actions. A key problem is that there is no real standardization. One possibility would be the Embedded Microprocessor Benchmark Consortium, but that does not cover all MCUs and, anyway, it is more oriented towards CPU, rather than MPU, benchmarking.**

### RTOS Metrics – Memory footprint

**As all embedded systems have some limitations on available memory, the requirements of an RTOS, on a given CPU, need to be understood. An OS will use both ROM and RAM. ROM, which is normally flash memory, is used for the kernel, along with code for the runtime library and any middleware components. This code – or parts of it – may be copied to RAM on boot up, as this can offer improved performance. There is also likely to be some read only data. If the kernel is statically configured, this data will include extensive**

**information about kernel objects. However, nowadays, most kernels are dynamically configured.**

**RAM space will be used for kernel data structures, including some or all of the kernel object information, again depending upon whether the kernel is statically or dynamically configured. There will also be some global variables. If code is copied from flash to RAM, that space must also be accounted for. There are a number of factors that affect the memory footprint of an RTOS. The CPU architecture is key. The number of instructions can vary drastically from one processor to another, so looking at size figures for, say, PowerPC gives no indication of what the ARM version might be like.**

**Embedded compilers generally have a large number of optimization settings. These can be used to reduce code size, but that will most likely affect performance. Optimizations affect ROM footprint, and also RAM. Data size can also be affected by optimization, as data structures can be packed or unpacked. Again both ROM and RAM can be affected. Packing data has an adverse effect on performance. Most RTOS products have a number of optional components. Obviously, the choice of those components will have a very significant effect upon memory footprint. Most RTOS kernels are scalable, which means that, all being well, only the code to support required functionality is included in the memory image. For some RTOS, scalability only applies to the kernel. For others, scalability is extended to the rest of the middleware.**

**Although an RTOS vendor may provide or publish memory usage information, you may wish to make measurements yourself in order to ensure that they are representative of the type of application that you are designing. These measurements are not difficult. Normally the map file, generated by the linker, gives the necessary memory utilization data. Different linkers produce different kinds of map files with varying amounts of information included in a variety of formats. Possibilities extend from a mass of hex numbers through to an interactive HTML document and everything in between. There are some specialized tools that extract memory usage information from executable files. An example is obj dump .**

## Interrupt latency

**The time related performance measurements are probably of most concern to developers using an RTOS. A key characteristic of a real time system is its timely response to external events and an embedded system is typically notified of an event by means of an interrupt, so interrupt latency is critical.**

- **System: the total delay between the interrupt signal being asserted and the start of the interrupt service routine execution.**
- **OS: the time between the CPU interrupt sequence starting and the**

**initiation of the ISR. This is really the operating system overhead, but many people refer to it as the latency. This means that some vendors claim zero interrupt latency.**

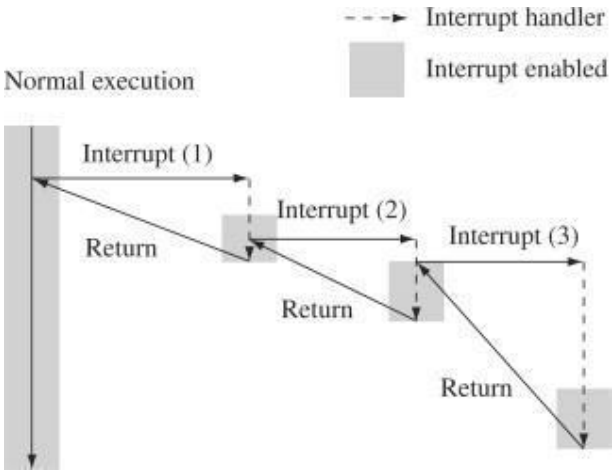



**Fig:2.1: Interrupt latency**

**Measurement**

**Interrupt response is the sum of two distinct times:**

$$T_{IL} = T_H + T_{OS}$$

**where:**

- $T_H$ **is the hardware dependent time, which depends on the interrupt controller on the board as well as the type of the interrupt**
- $T_{OS}$ **is the OS induced overhead**

**Ideally, quoted figures should include the best and worst case scenarios. The worst case is when the kernel disables interrupts. To measure a time interval, like interrupt latency, with any accuracy, requires a suitable instrument. The best tool to use is an oscilloscope. One approach is to use one pin on a GPIO interface to generate the interrupt. This pin can be monitored on the 'scope. At the start of the interrupt service routine, another pin, which is also being monitored, is toggled. The interval between the two signals may be easily read from the instrument.**

**Importance**

**Many embedded systems are real time and it is those applications, along with fault tolerant systems, where knowledge of interrupt latency is important. If the requirement is to maximize bandwidth on a particular interface, the latency on that specific interrupt needs to be measured. To give an idea of numbers, the majority of systems exhibit no problems, even if they are subjected to interrupt latencies of tens of microseconds**

**Interrupt latency is the sum of the hardware dependent time, which depends on the interrupt controller as well as the type of the interrupt, and the OS induced overhead. Ideally, quoted figures should include the best and worst case scenarios. The worst case is when the kernel disables interrupts. To measure a time interval, like interrupt latency, with any accuracy, requires a suitable instrument and the best tool to use is an oscilloscope. One approach is to use one pin on a GPIO interface to generate the interrupt and monitor it on the 'scope. At the start of the interrupt service routine, another pin, which is also being monitored, is toggled and the interval between the two signals may be easily read.**

**Scheduling latency**

**A key part of the functionality of an RTOS is its ability to support a multi-threading execution environment. Being real time, the efficiency at which threads or tasks are scheduled is of some importance and the scheduler is at the core of an RTOS. so it is reasonable that a user might be interested in its performance. It is hard to get a clear picture of performance, as there is a wide variation in the techniques employed to make measurements and in the interpretation of the results.**

**There are really two separate measurements to consider:**

- **The context switch time**
- **The time overhead that the RTOS introduces when scheduling a task**

**Timing kernel services**

**An RTOS is likely to have a great many system service API (application program interface) calls, probably numbering into the hundreds. To assess timing, it is not useful to try to analyze the timing of every single call. It makes more sense to focus on the frequently used services.**

**For most RTOS there are four key categories of service call:**

- **Threading services**
- **Synchronization services**

- **Inter-process communication services**
- **Memory services**

**All RTOS vendors provide performance data for their products, some of which is more comprehensive than others. This information may be very useful, but can also be misleading if interpreted incorrectly. It is important to understand the techniques used to make measurements and the terminology used to describe the results. There are also trade-offs – generally size against speed – and these, too, need to be thoroughly understood. Without this understanding, a fair comparison is not possible. If timing is critical to your application, it is strongly recommend that you perform your own measurements. This enables you to be sure that the hardware and software environment is correct and that the figures are directly relevant to your application.**

**Real Time Scheduling algorithms**

**In real time operating systems(RTOS) most of the tasks are periodic in nature. Periodic data mostly comes from sensors, servo control, and real-time monitoring systems. In real time operating systems, these periodic tasks utilize most of the processor computation power. A real-time control system consists of many concurrent periodic tasks having individual timing constraints. These timing constraints include release time (ri), worst case execution time(Ci), period (ti) and deadline(Di) for each individual task Ti. Real time embedded systems have time constraints linked with the output of the system. The scheduling algorithms are used to determine which task is going to execute when more than one task is available in the ready queue.**

**The operating system must guarantee that each task is activated at its proper rate and meets its deadline. To ensure this, some periodic scheduling algorithms are used. There are basic two types of scheduling algorithms**

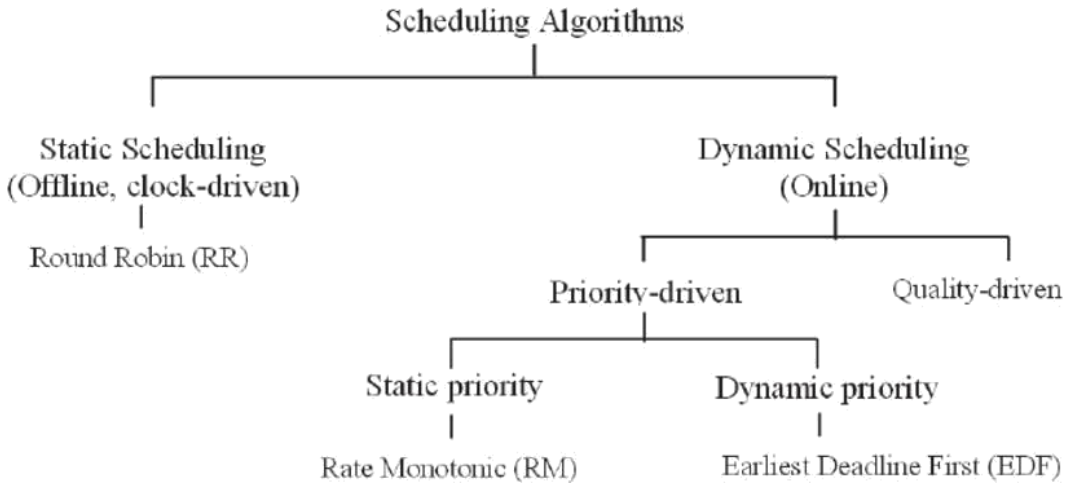



**Fig:2.2:Classification of scheduling algorithm**

**Offline Scheduling Algorithm**

**Offline scheduling algorithm selects a task to execute with reference to a predetermined schedule, which repeats itself after specific interval of time. For example, if we have three tasks $T_a$, $T_b$ and $T_c$ then $T_a$ will always execute first, then $T_b$ and after that $T_c$ respectively.**

**Online Scheduling Algorithm**

**In Online scheduling a task executes with respect to its priority, which is determined in real time according to specific rule and priorities of tasks may change during execution. The online line scheduling algorithm has two types. They are more flexible because they can change the priority of tasks on run time according to the utilization factor of tasks.**

**Fixed priority algorithms**

**In fixed priority if the $k^{th}$ job of a task T1 has higher priority than the $k^{th}$ job of task T2 according to some specified scheduling event, then every job of T1 will always execute first then the job of T2 i.e. on next occurrence priority does not change. More formally, if job J(1,K) of task T1 has higher priority than J(2,K) of task T2 then J(1,K+1) will always has higher priority than of J(2,K+1)**
**. One of best example of fixed priority algorithm is rate monotonic scheduling algorithm.**

**Dynamic priority algorithms**

**In dynamic priority algorithm, different jobs of a task may have different priority on next occurrence, it may be higher or it may be lower than**

**the other tasks. One example of a dynamic priority algorithm is the earliest deadline first algorithm.**

## Processor utilization factor (U)

**For a given task set of n periodic tasks, processor utilization factor U is the fraction of time that is spent for the execution of the task set. If Si is a task from task set then Ci/Ti is the time spent by the processor for the execution of Si . Processor utilization factor is denoted as**

$$U=\sum_{i=1}^{n} \frac{C_i}{T_i}$$

**Similarly, for the task set of n periodic tasks processor utilization is greater than one then that task set will not be schedulable by any algorithm. Processor utilization factor tells about the processor load on a single processor. U=1 means 100% processor utilization. Following scheduling algorithms will be discussed in details**

### Rate Monotonic (RM) Scheduling Algorithm

**The Rate Monotonic scheduling algorithm is a simple rule that assigns priorities to different tasks according to their time period. That is task with smallest time period will have highest priority and a task with longest time period will have lowest priority for execution. As the time period of a task does not change so not its priority changes over time, therefore Rate monotonic is fixed priority algorithm. The priorities are decided before the start of execution and they does not change overtime.**

**Rate monotonic scheduling Algorithm works on the principle of preemption. Preemption occurs on a given processor when higher priority task blocked lower priority task from execution. This blocking occurs due to priority level of different tasks in a given task set. rate monotonic is a preemptive algorithm which means if a task with shorter period comes during execution it will gain a higher priority and can block or preemptive currently running tasks. In RM priorities are assigned according to time period. Priority of a task is inversely proportional to its timer period.**

**Task with lowest time period has highest priority and the task with highest period will have lowest priority. A given task is scheduled under rate monotonic scheduling Algorithm, if its satisfies the following equation:**

## Example of Rate Monotonic (RM) Scheduling Algorithm

**For example, we have a task set that consists of three tasks as follows**

**Table 2.1 Rate Monotonic (RM) Scheduling Algorithm**

| Tasks | Release time(ri) | Execution time(Ci) | Deadline (Di) | Time period(Ti) |
|---|---|---|---|---|
| **T1** | 0 | 0.5 | 3 | 3 |
| **T2** | 0 | 1 | 4 | 4 |
| **T3** | 0 | 2 | 6 | 6 |

**Task set U= 0.5/3 +1/4 +2/6 = 0.167+ 0.25 + 0.333 = 0.75**

**As processor utilization is less than 1 or 100% so task set is schedulable and it also satisfies the above equation of rate monotonic scheduling algorithm.**

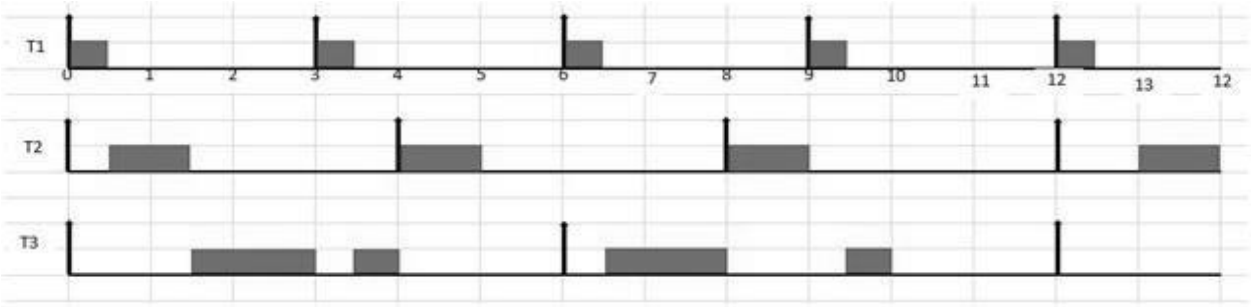



**Fig:2.3: RM scheduling of Task set in Table 2.1**

**A task set given in the above table is RM scheduling in the given figure. The explanation of above is as follows**

1. **According to RM scheduling algorithm task with shorter period has higher priority so T1 has high priority, T2 has intermediate priority and T3 has lowest priority. At t=0 all the tasks are released. Now T1 has highest priority so it executes first till t=0.5.**
2. **At t=0.5 task T2 has higher priority than T3 so it executes first for one-time units till t=1.5. After its completion only one task is remained in the system that is T3, so it starts its execution and executes till t=3.**
3. **At t=3 T1 releases, as it has higher priority than T3 so it preempts or blocks T3 and starts it execution till t=3.5. After that the remaining part of T3 executes.**
4. **At t=4 T2 releases and completes it execution as there is no task running in the system at this time.**
5. **At t=6 both T1 and T3 are released at the same time but T1 has higher priority due to shorter period so it preempts T3 and executes till t=6.5, after that T3 starts running and executes till t=8.**

6. **At t=8 T2 with higher priority than T3 releases so it preempts T3 and starts its execution.**
7. **At t=9 T1 is released again and it preempts T3 and executes first and at t=9.5 T3 executes its remaining part. Similarly, the execution goes on.**

**Advantages**

- **It is easy to implement.**
- **If any static priority assignment algorithm can meet the deadlines then rate monotonic scheduling can also do the same. It is optimal.**
- **It consists of calculated copy of the time periods unlike other time-sharing algorithms as Round robin which neglects the scheduling needs of the processes.**

**Disadvantages**

- **It is very difficult to support a periodic and sporadic tasks under RMA.**
- **RMA is not optimal when tasks period and deadline differ.**

**Least laxity scheduling**

**Least Laxity First (LLF) is a job level dynamic priority scheduling algorithm. It means that every instant is a scheduling event because laxity of each task changes on every instant of time. A task which has least laxity at an instant, it will have higher priority than others at this instant.**

**Introduction to Least Laxity first (LLF) scheduling Algorithm**

**More formally, priority of a task is inversely proportional to its run time laxity. As the laxity of a task is defined as its urgency to execute. Mathematically it is described as**

$$L_i = D_i - C_i$$

$$L_i = D_i - (t_i + C_i^R)$$

**Here $D_i$ is the deadline of a task, $C_i$ is the worst-case execution time(WCET) and Li**
**is laxity of a task. It means laxity is the time remaining after completing the WCET before the deadline arrives. For finding the laxity of a task in run time current instant of time also included in the above formula**.

**Here is the current instant of time and is the remaining WCET of the task. By using the above equation laxity of each task is calculated at every instant of time, then the priority is assigned. One important thing to note is that**

**laxity of a running task does not changes it remains same whereas the laxity all other tasks is decreased by one after every one-time unit.**

**Example of Least Laxity first scheduling Algorithm**

**An example of LLF is given below for a task set**.

**Table 2.2 Least Laxity first scheduling Algorithm**

| Task | Release time(ri) | Execution Time(Ci) | Deadline (Di) | Period(Ti) |
|---|---|---|---|---|
| **T1** | 0 | 2 | 6 | 6 |
| **T2** | 0 | 2 | 8 | 8 |
| **T3** | 0 | 3 | 10 | 10 |

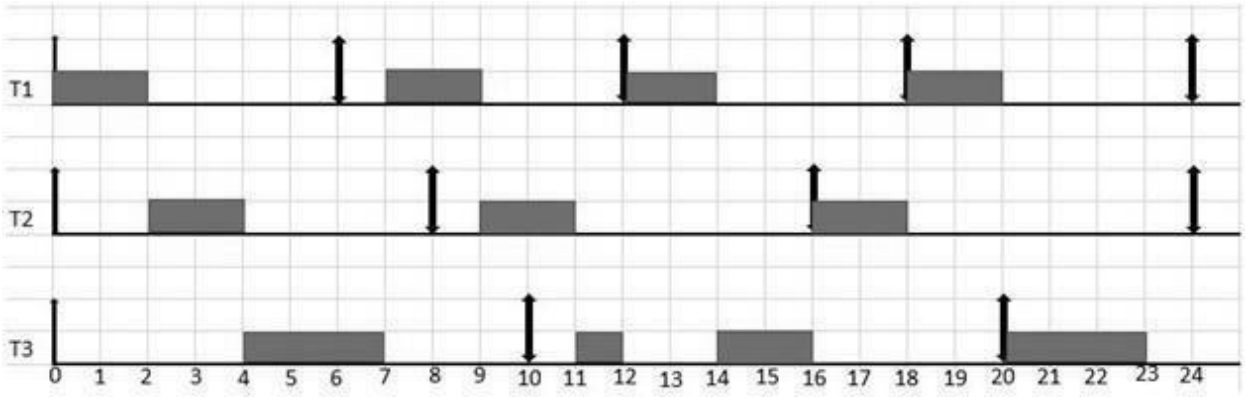



**Fig:2.4: LLF scheduling algorithm**

1. **At t=0 laxities of each task are calculated by using the equation**

**L1 = 6-(0+2) =4**

**L2 = 8-(0+2) =6**

**L3= 10-(0+3) =7**

**As task T1 has least laxity so it will execute with higher priority. Similarly, At t=1 its priority is calculated it is 4 and T2 has 5 and T3 has 6, so again due to least laxity T1 continue to execute.**

2. **At t=2 T1is out of the system so Now we compare the laxities of T2 and T3 as following**

$$L2= 8-(2+2) =4$$

$$L3= 10-(2+3) =5$$

**Clearly T2 starts execution due to less laxity than T3. At t=3 T2 has laxity 4 and T3 also has laxity 4, so ties are broken randomly so we continue to execute T2. At t=4 no task is remained in the system except T3 so it executes till t=6. At t=6 T1 comes in the system so laxities are calculated again**

$$L1 = 6-(0+2) =4$$

$$L3= 10-(6+1) =3$$

**So T3 continues its execution.**

3. **At t=8 T2 comes in the system where as T1 is running task. So at this instant laxities are calculated**

$$L1 = 12-(8+1) =3$$
$$L2= 16-(8+2) =6$$

**So T1 completes its execution. After that T2 starts running and at t=10 due to laxity comparison T2 has higher priority than T3 so it completes it execution.**

$$L2= 16-(10+1) =5$$

$$L3= 20-(10+3) =7$$

**t=11 only T3 in the system so it starts its execution.**

4. **At t=12 T1 comes in the system and due to laxity comparison at t=12 T1 wins the priority and starts its execution by preempting T3. T1 completes it execution and after that at t=14 due to alone task T3 starts running its remaining part. So, in this way this task set executes under LLF algorithm.**

**LLF is an optimal algorithm because if a task set will pass utilization test then it is surely schedulable by LLF. Another advantage of LLF is that it some advance knowledge about which task going to miss its deadline. On other hand it also has some disadvantages as well one is its enormous computation demand as each time instant is a scheduling event. It gives poor performance when more than one task has least laxity.**

**Cyclic executives:**

- **Scheduling tables**
- **Frames**
- **Frame size constraints**
- **Generating schedules**
- **Non-independent tasks**
- **Pros and cons**

**Cyclic Scheduling**

**This is an important way to sequence tasks in a real time system. Cyclic scheduling is static – computed offline and stored in a table. Task scheduling is non-preemptive. Non-periodic work can be run during time slots not used by periodic tasks. Implicit low priority for non-periodic work. Usually non-periodic work must be scheduled preemptively. Scheduling table executes completely in one hyper period H. Then repeats H is least common multiple of all task periods N quanta per hyper period. Multiple tables can support multiple system modes E.g., an aircraft might support takeoff, cruising, landing, and taxiing modes. Mode switches permitted only at hyper period boundaries. Otherwise,**
**hard to meet deadlines**.

$$T(t_k) = \begin{cases} T_i & \text{if } T_i \text{ is to be scheduled at time } t_k \\ I & \text{if no periodic task is scheduled at time } t_k \end{cases}$$

**Frames:**

**Divide hyper periods into frames .Timing is enforced only at frame boundaries.**

**Consider a system with four task**

- $T_1 = (4,1)$
- $T_2 = (5, 1.8)$
- $T_3 = (20, 1)$
- $T_4 = (20, 2)$

**Possible schedule:**

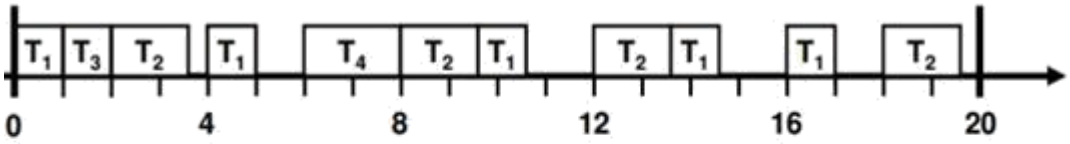



**Table starts out with:**

- $(0, T_1), (1, T_3), (2, T_2), (3.8, I), (4, T_1), \ldots$

**task is executed as a function call and must fit within a single frame. Multiple tasks may be executed in a frame size is f Number of frames per hyper period is F = H/f.**

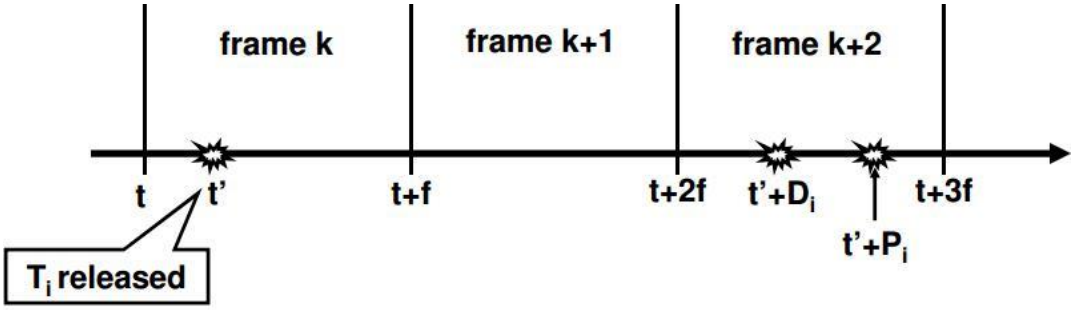
**Frame Size Constraints:**

**1. Tasks must fit into frames so, f ≥ Ci for all tasks Justification: Non-preemptive tasks should finish executing within a single frame**

**2. f must evenly divide H Equivalently, f must evenly divide P for some task i Justification: Keep table size small**

**3. There should be a complete frame between the release and deadline of every task**

**Justification: Want to detect missed deadlines by the time the deadline arrives**




- **Therefore: $2f - \gcd(P_i, f) \leq D_i$ for each task i**

**Cyclic executive is one of the major software architectures for embedded systems. Historically, cyclic executives dominate safety-critical systems Simplicity and predictability wins. However, there are significant drawbacks. Finding a schedule might require significant offline computation.**

**Drawbacks:**

- **Difficult to incorporate sporadic processes.**
- **Incorporate processes with long periods**
- **Problematic for tasks with dependencies – Think about an OS without synchronization**
- **Time consuming to construct the cyclic executive – Or adding a new task to the task set**
- **"Manual" scheduler construction**
- **Cannot deal with any runtime changes**
- **Denies the advantages of concurrent programming**

**Summary:**

- **Off-line scheduling**
- **Doesn't use the process abstraction of the OS**
- **Manually created table of procedures to be called – Waits for a periodic interrupt for synchronization**
- **Minor cycle – Loops the execution of the procedures in the table**

**TEXT / REFERENCE BOOKS**

**1. Jane W. S Liu, "Real Time Systems" Pearson Higher Education, 3rd Edition, 2000.**

**2. Raj Kamal, "Embedded Systems- Architecture, Programming and Design" Tata McGraw Hill, 2nd Edition, 2014.**

**3. Jean J. Labrosse, "Micro C/OS-I : The real time kernel" CMP Books, 2nd Edition,2015.**

**5. Richard Barry, "Mastering the Free RTOS: Real Time Kernel", Real Time Engineers Ltd, 1st Edition, 2016.**

**6. David E. Simon, " An Embedded Software Primer", Pearson Education Asia Publication**