Real-Time Micro Kernel for CLA

Victor Ramamoorthy

Version 1.0 March 1998
Version 2.0 May 1, 1998

Contents

Real-Time Micro Kernel for CLA....1
Contents....2
0.0 Abstract....3
1.0 Introduction....3
2.0 Rate-Monotonic Scheduling....4
3.0 Overall View of the System....4
4.0 Design of Schedule Cycle....6
4.1 Resource Requirements:....6
4.2 Resource Table....7
4.3 Task Model....8
4.4 Fixed Priority Preemptive Scheduling....9
4.5 Quantized Lehoczky-Sha-Ding Test for Schedulability....9
4.6 Schedule Cycle Length....10
4.7 Period Transformation....11
4.8 Schedule Cycle Generation....11
4. 9 Actual Processor Execution Time in a Time-Slice....12
4.10 The Idea of Interrupt Focus and Interrupt Filtering....12
4.11 Joint Scheduling of PE and MTE/HMTE....13
4.12 Complex Task Models....14
5.0 Implementation Architecture....14
5.1 Types of Tasks....15
5.2 Key Assumptions made of Application Programs and Interrupt Services....16
5.3 Task Data Structures....18
5.4 Time-Slice Clock and Its Interrupt Mechanism....22
5.5 Level 0 Tasks....24
5.6 Level 1 Tasks....28
5.7 Kernel Programs....29
5.8 Kernel *Init* Program....30
5.9 Kernel *Run* Program....30
5.10 Kernel Organization in DRAM and CLA Local Memory....31
5.11 List of Interrupt Services....31
5.12 Model of Interrupt Service Routine (ISR)....32
5.13 Interactive Multitasking....33
5.14 Estimation of DRAM Storage Requirements....33
5.15 List of Kernel Functions....34
5.16 Kernel Area Definition....35
6.0 Schedule Design Tool....35
6.1 Schedule Design Tool Code....35
6.2 Prime Number Table....48
7.0 Guide for Application Programmers....49
7.1 Extracting Task Parameters: Method 1....49
7.2 Extracting Task Parameters: Method 2....50
7.3 Generating Appsets....51
7.4 Merging ISRs....53
7.5 MTE Transfers: Polling....53
7.6 MTE Transfers: Interrupts....55
7.7 Error Handling....55
7.8 Memory Protection....55
7.9 Style Guide for Application Programmers....55

0.0 Abstract

A high performance micro kernel with a static or dynamic Rate-Monotonic Scheduling, smallest code size, simple and flexible construction is presented for a single PE sharing many simultaneous real-time applications. Conventional approach to building kernels do not fit very well into the need for ultra compact design required for CLA. New innovations like Application Registry, Schedule Testing, Interrupt focusing and filtering, and dynamic table driven task switching are introduced. After reviewing the theory of scheduling, high level architecture is described along with a design tool for scheduling.

1.0 Introduction

The aim of this work is to define a simple and high performance micro kernel for the CLA processor. The intended application of CLA is multimedia processing such as video/audio decoding, graphics, and related telecommunication devices. Each of these application areas demand real-time performance and it is intended that a few of these applications run on CLA at the same time. This means that CLA must handle multiple processes and offer *guaranteed real-time service* to each of the clients. Here we study the problem of one PE sharing many tasks.

In general, processors are built to satisfy the familiar value domain requirements. In real-time systems, an additional requirements of time-domain constraints are introduced. This means that a computational process should not only yield accurate values of the computation, but also should produce the results in the stipulated time interval. The time-domain specifications are often expressed in terms of deadlines for event responses. Some of these deadlines may be *hard*, others may be *soft*. Missing a hard deadline results in unacceptable loss of system performance. Missing a soft deadline may or may not be acceptable depending on the application. A major issue in meeting these deadlines is the competition for shared resources by multiple tasks. A resource could be a processor, a data path, a storage facility, or an input/output device. The competition for shared resources must be handled in such a way that no critical deadlines are missed.

The design of such a real-time system mostly belongs the *art* of computer design rather than the *science* of construction. Traditionally, engineers have relied on intuition, rules of thumb, simulation and extensive testing to determine the timing behavior of the systems under construction. One conventional way of solving the timing requirement is to design an *over-dimensioned system* that has plenty of resources available to simplify the resource sharing problem. This approach may not be always acceptable as an over-dimensioned solution is invariably an expensive solution. The crux of the issue is our inability to model a computational system as a *predictable deterministic system*. The complexity of hardware and software interaction precludes one from making precise *temporal* statements about the entire system. The event sequences occurring inside a system are too complex to justify a deterministic view of the processes happening at any time.

However, the situation can be simplified by designing simple task models and simple control software. This is indeed possible because multimedia tasks and data streams are periodic in nature. Secondly the perceptual aspects of the multimedia information help in the design by requiring mostly soft deadlines instead of the hard deadlines. There is a degree of flexibility in the timing precision of the audio and video information presented to the human end user. Third reason is that the CLA chip is designed to be an auxiliary processor to another processor such as a Pentium chip with AGP port. Significant complexity reductions are possible in the control software which is called as the *Real-Time Micro Kernel*. Very conscious efforts are made in the design of the kernel such that it can be fit in the memory space available with the CLA. Orthodox kernel designers may be offended by the approach described here: the main structure of the kernel stands on the principle of removing to a large extent the uncertainty in the timing behavior of the operating system.

2.0 Rate-Monotonic Scheduling

The problem of constructing a real-time system can be traced to the problem of resource scheduling of multiple tasks. The term 'Rate-Monotonic' refers to a particular *priority* assignment in which priorities decrease monotonically with the rates (i.e. the frequencies) of the tasks to be scheduled. This assignment leads to having higher priorities for tasks with shorter periods. Again the *Rate-Monotonic Scheduling* implicitly assumes periodic tasks. Aperiodic tasks need slightly different treatment which is beyond the scope of this report.

The original concept of Rate-Monotonic Scheduling was proposed by Liu and Layland in 1973[1]. In the classic paper, the rate monotonic priority assignment was shown to be the optimal fixed priority strategy under certain restrictions and *schedulability* was related to processor utilization in a straightforward manner. Unfortunately, the restrictions were quite severe. Schedulability was defined as the ability to meet *all* the deadlines of *all* periodic tasks under *all* task *phasings*. This definition is actually a "Strong-Sense" definition because a task set that does not meet this strict condition may still be scheduled under a particular task phasing or the task set may be schedulable but may fail only under overload conditions. This strong-sense definition was useful for systems that are required to guarantee real-time performance without any regard to the temporal relationship of tasks within a task set. Since the aim was to simplify the design of the task scheduler without any regard to the complications arising out of task phasing, Liu and Layland defined the notion of a *critical instant* when all the tasks are released at the same time for scheduling. Then they proceeded to test the task set for schedulability. Liu and Layland provided a simple *sufficiency test* for schedulability which was very pessimistic. Some of the task sets designed by ad-hoc methods had superior utilization than what the theory promised.

In early eighties, a joint research initiative between IBM and Carnegie Mellon University began to revisit the scheduling theory started by Liu and Layland. Many of the restrictions are currently being removed. In ten years time, a solid body of analytical tools are being developed to study the "Weak-Sense" schedulability problem for periodic and aperiodic tasks. Much of the results are mathematical in nature and implementation oriented engineers are slowly getting the hang of the *Rate- Monotonic Analysis*. The main ideas of this approach are to be found in the celebrated paper by Lehoczky, Sha and Ding[2] which provides a systematic way of testing weak-sense schedulability of a task set with different task phasings. Scheduling theory is so vast, a primer such as the paper by Stankovic, Spuri, Natale and Buttazzo[3] provides only skin-deep penetration into the subject. A more readable tutorial is the paper by Sha, Rajkumar and Sathaye[4] is recommended for general reading. A comprehensive view of the methodology is presented in the Ph.D thesis by Daniel Katcher[5] and the references therein.

3.0 Overall View of the System

The main assumptions for the Micro Kernel design are as follows:

---

[1] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment", *Journal of the ACM*, Vol.20, No.1, January 1973, pp. 40-61.
[2] John Lehoczky, Lui Sha and Ye Ding, "The Rate Monotonic Schedling Algorithm: Exact Characterization and Average Case Behvior", *Proceedings of the IEEE Real-Time Systems Symposium 1989*, IEEE Computer Society Press, pp 201-209
[3] John A. Stankovic, Marco Spuri, Marco Di Natale and Georgio C. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems", IEEE Computer, June 1995, pp. 16-25
[4] Lui Sha, Ragunathan Rajkumar, and Shirish S. Sathaye, "Generalized Rate-Monotonic Scheduling Theory: A Framework for developing Real-Time Systems", Proceedings of the IEEE, Vol.82, No.1, January 1994, pp. 68-82
[5] Daniel I. Katcher, "Engineering and Analysis of Real-Time Operating Systems", Ph.D Theis, August 1994, Carnegie Mellon University

1. The kernel must be small enough to fit into the CLA local memory space.
2. The kernel is application specific. For a given set of applications, a matched kernel is created either off-line or at the boot time and loaded into the CLA DRAM. Upon initialization this kernel is loaded into CLA. For a different set of applications, a different kernel schedule is required.
3. The applications are defined by the following general periodic *processing* categories:
    a. Input data inflow from an input device into the system
    b. Processing of input data by a processor or a set of processors and associated DSEs
    c. Data Transfers needed for the step b).
    d. Output data flow of the processed data to an output device.

The system overview is shown in fig.1. In the DRAM, at the starting time, the object code, data, status information are loaded separately along with the kernel. The kernel contains an important information called "*Schedule Cycle*" which is determined *a priori* with the knowledge of the application programs[6].

4. Each application processing intervals are broken into small *time-slices* and in each time-slice a *quanta* of processing is executed. In each quanta, the processing may involve the actual *number-crunching* by the processor or bus transfers of data required for number-crunching or input/output data flows. In each time-slice, the quanta of work done is for only one *application.*
5. Each time-slice is triggered by an *interrupt* from a clock source. It is assumed at this time that this interrupt is triggered by a *hardware device* so that precise timing can be maintained.
6. The kernel exists *three main modes* of operation. In the *sleep* mode, the kernel is not executing any real-time applications and hence is free to do any *Initialization* tasks a kernel is intended to serve. It also responds to several of the system interrupts. In this mode the kernel is mainly checking several of the I/O devices, checking queues, interpreting different commands that are placed in the command queue etc. There is no real restrictions about how long the kernel stays in the sleep mode. Most of the kernel settings are performed when in the kernel is in this mode.
7. When the initializations are performed correctly, applications are loaded and are ready to go, the kernel enters into the *active* mode. During the active mode, the kernel executes all its functions bound by the *time-slice clock*. Here all the real-time applications are executed as dictated by a predetermined *N quanta Schedule Cycle*. The schedule cycle has exactly N time-slices and performs exactly N quanta of work. What needs to be done at any instant is predetermined by the schedule cycle. Some applications get, say *i* quanta of work done, some others get *k* quanta of work done during the schedule cycle. How much of work done for each application (i.e., number of *quantas* of work done), when they should be done (i.e., the *phasing* of the work), and the length of the schedule cycle (i.e., N *quanta cycle*) are the design parameters. The active mode lasts for exactly for N-1 quanta time intervals (N-1 time slices) and enters into the *house-keeping* mode[7].
8. House-Keeping mode lasts exactly for one time-slice (or several if needed) where all important decisions such as responding to an external stimulus, decision regarding returning to active mode, error code interpretation, etc., are done. In house-keeping mode, the kernel does a quick reality check of its environment regarding all the input stimuli that might be waiting for a response from the kernel. During the active mode, the kernel is oblivious to the external world and just keeps the tasks being executed. During the house-keeping mode, it wakes up to the reality and makes some intelligent decisions. In order to maintain real-time performance such a tight control over timing is necessary. From the house-keeping mode the kernel can enter the active mode or the sleep mode. Though this mode is designed take only one time-slice duration, it can be increased to two or three depending on the amount of house-keeping work needed. Another possibility is to alter the schedule cycle slightly to adapt to the actual amount of work done excluding the bus transfer intervals for each task. Separate

---

[6] Only parametric description of the application programs are required. This information is stored in the App Registry.

[7] There is nothing holy about just having one time-slice duration for the house-keeping task. If frequent attendence to house-keeping is required, this can be modeled as a low priority task with a large period and can be included in the schedule design. Here we just assume that only infrequent visit to house-keeping chores is required to make the discussion simple.

algorithms can be run during this mode to take of the adaptation necessary to balance the processes executed.

9. The key intelligence required to meet the real-time requirement is frozen in the form of *Schedule Cycle* for the set of applications to be run[8]. Once this is done, the structure of the kernel reduces to that of a programmable switch that loads and unloads tasks at appropriate times. An illustration of the Schedule cycle is available in fig. 4. The schedule cycle is divided into N *phases*, each phase is tied to an interrupt clock. At phase N-1[9], only house-keeping task is performed as the kernel happens to be in the house keeping mode. The active modes start at phase 1 and ends at phase N-1. Fig. 4 further shows the division of the house-keeping task into sub-activities. Each of the activities are related to essential services that the kernel is expected to perform.

4.0 Design of Schedule Cycle

Each application program is assumed to be compiled, linked and created an executable file which is ready to execute. This executable file is assumed to contain the following parts: (1) *Text* or Object code, (2) *Data* that the executable needs to execute the program, (3) *Stack Space* where the temporary and intermediate results are stored and (4) *Status* Information which contains the status of execution of the program. These parts are defined by the following addresses:

```
TEXT_BEGIN_ADDRESS        // Text Address in the global address space
DATA_BEGIN_ADDRESS        // Data Address in the global address space
STACK_BEGIN_ADDRESS       // Stack Address in the global address space
STATUS_BEGIN_ADDRESS      // Status Address in the global  address space
```

It is assumed at this point that the applications programs reside in the DRAM and will be transferred to CLA local memory as and when they are required. Since there are many applications requiring time-shared processing, we need a better arrangement of data structures. This will be illustrated in the following.

The application programmer is expected to provide the following *Resource Requirements* by analyzing the application program:

4.1 Resource Requirements:

1. *Fundamental Frequency* of the Application in Hz. For example, video application may have the fundamental frequency of 60 Hz for 60 fields per second or 30 Hz for 30 frames/sec. Similarly an audio application may have a fundamental frequency of 8000 Hz relating to the audio sampling frequency. Fundamental frequency relates to the basic operational mode of an application and its reciprocal is called the *Fundamental Period* in seconds.
2. *Maximum Inflow Data Transfer Rate* of the application in Bytes/sec. This relates to the amount of data that the application must accept as input during every fundamental period.
3. *Origin of Inflow* refers to the source of the data for the application. For example, this could be a CD-ROM, DVD-ROM, video capture card, microphone etc.
4. *Inflow Path* refers to the transportation path of the inflow data required for the application. This may include devices, buses and other resources encountered by the inflow data before it reaches the application.

---

[8] Frozen schedule does not mean static fixed scheduling. Since the scheduling computations are simple, as shown in sections 4.3 to 4.6, scheduling can be done on the fly. That is the reason why the App Registry is kept in the DRAM.

[9] Strictly speaking, house-keeping task is also a periodic task. It can be included in the design along with other active tasks. The important criterion is how often the house-keeping task needs to be done. If house-keeping task is done infrequently then the system may appear to be sluggish to commands.

5. *Inflow Resource Requirements* are the specifications relating the availability of various resources required by the inflow data. This may include MTE, HMTE, ISA/PCI interrupt services, Semaphores etc.
6. *Maximum Inflow Transport Duration* is the interval of time spent in shipping the input data into the system
7. *Maximum Outflow Data Transfer Rate* of the application in Bytes/sec is the amount of data the application must deliver as output during each fundamental period.
8. *Destination of Outflow* specifies the destination where the outflow is headed.
9. *Outflow path* defines how the application delivers the processed data to the destination and the path taken by the outflow.
10. *Outflow Resource Requirements* specify the resources (interrupts, semaphores etc.) required for completing the outflow transfer within the fundamental period.
11. *Maximum Outflow Transport Duration* is the time it takes to ship the processed data out to the destination.
12. *Required Number of PE* to complete the processing within the fundamental period
13. *Maximum Duration of Processing Interval* as a percentage of fundamental period where actual processing work is done
14. *Tolerance Values* of processing and data transfers.
15. *Number of Semaphore* registers required.
16. *Number of Interrupt Service Routines* required and their *service latencies*.
17. *The storage* required at the local memory for instructions and data

The above 16 parameters capture the real-time issues relevant to an application. This parametric description needs to be condensed to the following *Resource Table* by scaling the fundamental period to a smaller interval called *Period-Slice*. A period-slice is a subdivision of a fundamental period usually from 5 milliseconds to about 500 milliseconds. A smaller period-slice is likely to be inefficient because of the task-switching overload. A larger value, on the other hand may lead to delayed response to other *unplanned* task inputs that may happen during the processing. The value of period-slice depends on the application.

4.2 Resource Table

A resource table is a condensed form of resource requirements of an application set. By describing the application set in terms of parameters described below, offers the following simplified view of the application:

1. *Period-Slice* in milliseconds. It is a subdivision of a fundamental period.
2. *PE time* in milliseconds required to achieve the real-time performance
3. *DSE Time* in milliseconds required to achieve the real-time performance
4. *SMTE-time* in milliseconds required to achieve inflow/outflow requirements
5. *AMTE-time* in milliseconds required to achieve inflow/outflow requirements
6. *HMTE-time* in milliseconds required to achieve inflow/outflow requirements
7. *Number of Semaphores* required.
8. *Number of Interrupts* expected by the application
9. *Average Latency* of Interrupt Service Routine
10. *Local Memory address space* to be saved when task is switched.

	Resources	Description
1	Period-Slice (ms)	Period of the task; Whole or a subdivision of fundamental period; This can be scaled up or down to adjust for priority;
2	PE time(ms)	The amount of computation done by PE in a period; Used for scheduling; Can also be used for checking power dissipation;
3	DSE time (ms)	This value is useful for checking against the power dissipation allowed in the chip;
4	SMTE-time (ms)	This tells the amount of bandwidth used up by SMTE.

5	AMTE-time (ms)	This indicates the bandwidth used by AMTE.
6	HMTE-time (ms)	This indicates the host bandwidth required. All the three values, SMTE-time, AMTE-time and HMTE-time can be used for checking the bandwidth assignment;
7	Number of Semaphores required	Since semaphore is a limited resource, we need to keep an eye on them too.
8	Number of Interrupts	The task has to manage the interrupts as well as perform the desired functionality.
9	Average Latency of Interrupt Service Routine	The model assumed uses a single ISR with many "if-then" filters to find the actual source of interrupt. The average latency of ISR is an important quantity in the design of the kernel.
10	Local Memory low and high addresses	Local memory space needed to be saved when the task is switched.

The construction of the resource table is aimed at helping the design of the kernel. In this first version, semaphores, SMTE, AMTE, HMTE interrupts are not considered in the design[10]. The reason is that the complete CLA system has interdependencies between memory transfer engines, interrupts, and semaphores by the way of resource blocking and delays introduced in waiting. In general, it is very difficult model these interactions and are left for future versions. Since what is needed is a simple kernel with minimum storage requirements, our decision to leave the interdependency parameters seems to be prudent.

4.3 Task Model

Let there be an application set, called AppSet , consisting of n real-time applications $\{a_1, a_2, a_3, \cdots a_n\}$. Each application a_i is described by its micro structure called a *Periodic Task* τ_i which is described by the triplet where is the maximum computational load that a PE has to perform within a *Task Period* with a *Priority* of . An appset α_n and its task set are equivalent from the scheduling point of view. All the tasks in the task set are periodic and have a *deadline* requirement of finishing the computation within their respective task periods. If all the tasks in α_n meet their deadlines, then α_n is said to be *schedulable*.

Let be the utilization of PE for an appset α_n is given by . By definition, this quantity must be less than or equal to unity. The periods T_i and computation C_i are given in milliseconds. If α_n is schedulable, then a derived set from α_n consisting of only *n-k* applications obtained by removing *k* applications of α_n, called is also schedulable for all . This also means that .

[10] In fact, MTE and HMTE are also system resources just as a PE. By considering the total resources in the system, an optimum schedule can be arrived to control MTE, HMTE and PE. This means some hardware changes in the MTE to include external control to transfer data. This aspect is further covered in a later section.

If there are two schedulable appsets α_n and β_m, and if a third appset is constructed by selecting applications from both α_n and β_m, then δ_k may or may not be schedulable. Note that there are possible ways of creating the new appset δ_k and each combination needs separate testing for schedulability.

Note that the condition may appear to be sufficient to imply that an appset is schedulable[11].

However this is not true. An appset may satisfy the condition and yet may not be schedulable. Liu and Layland[12] derived a least upper bound on processor utilization equal to for a task set with *n* periodic tasks. For a large number of tasks, this bound approaches the value of 0.69. The implication is that only task sets with 69% of processor utilization are schedulable. Any task set with total utilization less than this bound could be scheduled by the Rate Monotonic algorithm defined below. However some task sets that are *above* this bound may also happen to be schedulable.

4.4 Fixed Priority Preemptive Scheduling

The idea of priorities leads the concept of preemptive scheduling. This is illustrated in fig. 2b. The highest priority task preempts the low priority one and gets executed. This means the low priority task has to wait until higher priority tasks are executed. Rate-Monotonic Scheduling theory shows that if the priorities are arranged such that is valid for the condition then a task set could be scheduled if it meets its schedulability requirements. Such an assignment is known as Rate Monotonic priority assignment. Any schedulable task set could be scheduled with Rate Monotonic assignment.

4.5 Quantized Lehoczky-Sha-Ding Test for Schedulability

The real crux of schedulability problem is finding when a task set is schedulable. It took more than decade to answer the above question since the original formulation by Liu and Layland. Here we present a discrete version of the Lehoczky-Sha-Ding[13] test that is of use in the present context. Though the periods and computation intervals of a task is given in milliseconds, it is necessary to quantize them with a time interval called Time-Slice Duration, $T_{Time-Slice}$. This is because, during the time-slice interval, a PE does a quanta of work. At the end of this interval, a decision to switch task or not is made.

---

[11] In fact, considerable effort was focused on relating scheduling and utilization. Strictly speaking, utilization bound depends on the task model. For multiframe task models, the Liu-Layland utilization can be more than unity and still the task set could be schedulable.

[12] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment", *Journal of the ACM*, Vol.20, No.1, January 1973, pp. 40-61.

[13] John Lehoczky, Lui Sha and Ye Ding, "The Rate Monotonic Schedling Algorithm: Exact Characterization and Average Case Behvior", *Proceedings of the IEEE Real-Time Systems Symposium 1989*, IEEE Computer Society Press, pp 201-209.

Given a task set $\Gamma_n = \left\{ \tau_1, \tau_2, \cdots \tau_n \right\}$, find a normalized task set where

, where the notation means the

greatest integer closest to *x*.[14] The algorithm for testing the schedulability is then an iterative one:

1. Compute $t(0) = \sum_{i=1}^{n} Q[C_i]$
2. Compute $t(i) = \sum_{i=1}^{n} Q[C_i] \cdot \left\lceil \frac{t(i-1)}{Q[T_i]} \right\rceil$ for $i = 1,2,3\cdots$
3. if $t(i+1) = t(i)$ and $t(i+1) \leq Q[T_n]$, then the task $\Gamma_n = \left\{ \tau_1, \tau_2, \cdots \tau_n \right\}$ is schedulable.

The convergence in step 3 above is necessary to guarantee schedulability. The main idea of the tesing is to release all the tasks simultaneously (that is, *the critical instant*) and check if the task with the lowest priority still meets its deadline equal to its period. At the critical instant, the task with the lowest priority has the longest waiting time before it starts executing. If the task with lowest priority can meet its deadline, then all the other tasks with higher priorities are bound to meet their deadlines, and hence the task set is schedulable. Note that the testing is fast even for a large number of tasks. This test can also be implemented as a part of boot time processing for the kernel.

4.6 Schedule Cycle Length

Since the normalized task set has integers as periods, it is possible to store the schedule - which is also cyclical - as a Schedule Cycle Table. The length of the schedule table is of importance, because it needs storage space in DRAM[15]. The schedule cycle length is given by the least common multiple of

. This is so because tasks are all periodic. They tend to have harmonic consonance and dissonance intervals. The smallest interval separating two consonance of all tasks is the schedule cycle length. At the consonance, all the tasks are appear simultaneously, that is creating a critical instant. If the tasks all meet their deadlines at consonance, they are bound to meet their deadlines at all other times. That is why the critical instant plays an important role in the scheduing theory. Schedule Cycle length can be found in the following manner:

1. Express $Q[T_i] = p_1^{e_1(i)} \cdot p_2^{e_2(i)} \cdot p_3^{e_3(i)} \cdots p_M^{e_M(i)}$ for $i = 1,2,3,\cdots n$, where $p_1, p_2, p_3, \cdots p_M$ are the first *M* prime numbers[16]. From the numerical range point of view, *M* is chosen to be 170. Any integer can be expressed in the above manner by Prime Number factorization theorem. the exponents can be zero or positive numbers.

---

[14] That is, if *x* consists of an integer part and a fractional part , then

[15] It is conceivable that the schedule cycle table can be constructed on the fly without any storage. But computing the table first helps in reducing the complexity of implementation.

[16] This can be also thought of as taking a prime-spectrum of any integer.

2. The Schedule Cycle length is given by

$$L = p_1^{\max_{i=1,2,3,\cdots n}\{e_1(i)\}} \cdot p_2^{\max_{i=1,2,3,\cdots n}\{e_2(i)\}} \cdot p_3^{\max_{i=1,2,3,\cdots n}\{e_3(i)\}} \cdots p_M^{\max_{i=1,2,3,\cdots n}\{e_M(i)\}}$$

The prime number factorization required in step 1 can be done by a modified Eratosthenes Sieve algorithm[17] and an implementation[18] is given in a later section.

4.7 Period Transformation

Though the above model fits well into multimedia applications, there may exist need for changing the priority of different tasks independent of the periods. A simple period transformation is useful in this context. Given a task , its priority is determined by the period . Now by applying a period transformation of , where a is a non-negative scale factor, the task priority can be changed within the task set . This facility is also included in the design tool in the last part of this report.

4.8 Schedule Cycle Generation

Rate Monotonic Algorithm simplifies the schedule cycle generation to a great extent. In fact the following "C-like " code fragment can be used for Schedule Cycle generation:

```
int schedule[ 1 to L];                  // schedule is an array which holds the final schedule;
                                        // L is the schedule Cycle Length;
int QT[1 to n], QC[1 to n];             // There are n tasks with periods QT[i] and computation QC[i]

int noofcycles;             //holds number of cycles of a task within a schedule cycle
int address, addressmax,addressmin;         //address pointers
int i,j,k,extra;

for(i=1; i <=N; i++)
{
        schedule[i] = 0;                // Initialize the array to zero
}

for(i =1; i <=N; i++)
{
        noofcycles = L/QT[i];

        if( i==1)
        {
                for(j=1; j <=noofcycles-1; j++)
                {
                        for(k=1; k <=QC[i]; k++)
                        {
                                address = (j-1)*QT[i] + k;  // This is the highest priority task; so let it
                                                            // have its schedule.
                                schedule[address] = i;
                        }
```

---

[17] http://www.utm.edu/research/primes/index.html#primers

[18] Our implementation simply tests for all prime numbers in section 5.8. We need to use the prime number sieve only for square root of the input number for primality.

```
			}
		}
		else
		{
			for(j=1; j <=noofcycles-1; j++)			// This is a low priority task
									// search if there is any slot open
			{
				extra = 0;				// counter for used up time-slices
				addressmin = (j-1)*QT[i] + 1;
				addressmax = j*QT[i];
				for(address=addressmin; address <=addressmax; address++)
				{
					if( schedule[address] = 0 && extra < QC[i]) // look for a free time-slice
					{
						schedule[address] = i;
						extra++;
					}
				}
			}
		}
}
```

The array schedule stores zero to indicate a free slot and task number to indicate occupancy. Hence schedule cycle generation can be done either on the fly or off-line.

4. 9 Actual Processor Execution Time in a Time-Slice

Though the duration $T_{Time-Slice}$ determines the duration at which a task takes control over the processor, the actual amount of work done during a time-slice depends on many events such as occurrence of interrupts. Interrupts can be triggered by a different task that was previously executed in a different time-slice, and by the same task executed before the current one. If is the expected number of interrupts that can occur during a time-slice, then the actual work done during a time-slice is given by:

where is the time required for saving the context and is the average time required for servicing an interrupt.

The above expression is admittedly an approximate one; the time spent by the processor for the task execution may vary stochastically depending on the task loaded and tasks previously executed. This means that if there is no control over the number of interrupts that can happen during a time-slice, the PE may spend more time on servicing the interrupts than performing the desired task - real-time guarantees can not be met.

4.10 The Idea of Interrupt Focus and Interrupt Filtering

Interrupts are both useful and detrimental to real-time performance. In order to decrease the average number of interrupts received during the time-slice interval, the following idea of *Interrupt Focus* is introduced:

(1) During the time-slice, PE should focus only on the interrupts that are *relevant to the task* that will be executed during the time-slice. Servicing other interrupts would cause an unwanted coupling between different tasks which would be very difficult to resolve. Further servicing interrupts unrelated to the current task does not help in achieving an improved real-time performance.

(2) This means that there should be a way to selectively *filter* the interrupts when the tasks are switched by the kernel. Fortunately, CLA architecture provides such a filtering mechanism with DSE DONE bits for DSE interrupts to PE. Similarly SMTC DONE and AMTC DONE bits are used to filter the MTC interrupts.

   (a) The critical assumption made here is that a MTC is *dynamically* allocated to a single task. This means that we switch tasks on a MTC depending on the current time-slice we are in. There are two MTCs attached to a PE and PE can monitor the interrupts from only these two MTCs. When we switch the tasks, we write a different parameter blocks and commands to a MTC. When the MTC is finished with the job, it writes to a semaphore register to indicate which job was finished before it issues an interrupt to PE. Thus PE, on receiving the interrupt, can check with the semaphore registers and find out which task caused the interrupt. Another assumption made here is that semaphore registers are statically bound to the tasks so that tracking an interrupt source is made easy.

   (b) However a DSE is *statically* bound to a single task. We do not switch tasks with DSEs. Since PE knows which DSE caused the interrupt, it can attend to the interrupt signal at the appropriate time. When different tasks are switched on the PE, DSEs just wait for their turn to receive commands from the PE.

4.11 Joint Scheduling of PE and MTE/HMTE

Just as a PE is a valuable resource, the data transferring devices such as MTE and HMTE are also precious resources sharing the global bus bandwidth. We can include MTE/HMTE in the scheduling model as a coupled design of combined PE-MTE/HMTE resource sharing. Following the model described in section 4.3, we assume the following coupled model:

Let there be an application set, called AppSet α_n, consisting of *n* real-time applications $\{a_1, a_2, a_3, \cdots a_n\}$. Each application a_i is described by its micro structure called a *Periodic Task* τ_i which is described by the coupled PE-MTE/HMTE model consisting of set where C_i is the maximum computational load (in milliseconds) that a PE has to perform within a *Task Period* T_i with a *Priority* of P_i; within the same task period, the task may engage in data transfers requiring the services of MTE/HMTE for a duration bounded by the time instants with the same priority P_i of the PE. One crucial difference of the MTE/HMTE devices is portrayed by the random delay bounded by the time instants $\left[\eta_i^{\min}, \eta_i^{\max}\right]$ from the time request is placed on them. The service time can be anywhere within the interval $[d_i^{\min}, d_i^{\max}]$.

Notice that PE scheduling is modeled as a deterministic scheduling problem whereas MTE/HMTE assume a stochastic formulation because MTE/HMTE are shared among other bus devices. PE does not have this restriction and can be engaged in service at the start of the task period. A request made to MTE/HMTE is modeled to have a random delay in responding to the request given by the interval $\left[\eta_i^{\min}, \eta_i^{\max}\right]$. The data transfer is modeled to last for an interval $[d_i^{\min}, d_i^{\max}]$. Since the global bus is fast enough, the service interval of MTE/HMTE devices is likely to the same order as C_i most of the times.

The schedule testing for PE is the same as the one given in section 4.5. However this basic deterministic test is repeated many times with randomly chosen stochastic variables representing a delay in responding to a data transfer request and representing the data transfer duration involving MTE/HMTE devices. Since the data transfer is modeled as a coupled process to the PE task periods, data transfers also assume the same priority as the parent PE task. The value of the random variable can be related to the FIFO depth used in the MTE/HMTE devices. Similarly the random variable can be obtained from the object code of an application.

Monte-Carlo simulation with a large sample size can verify the simultaneous scheduling of PE and MTE/HMTE devices. A program to do the stochastic simulation can be made from the code listing given in section 6.1.

4.12 Complex Task Models

Recent interest in multimedia applications has triggered interest in the scheduling theory and new models are being proposed. One such model is the Multiframe Task model[19], which allows for varying computational loads in a periodic manner rather than a fixed computation. This model captures the computational load variations from a task period to the next. Instead of assuming a worst case computational load C_i, the computation is modeled as a periodic sequence with period *k* of task periods. In other words, the computation varies in a periodic manner and repeats its cycle once in *k* periods. It is shown that this model could be suited for MPEG tasks where the frame structure follows a standard pattern such as IBBPBBPBBIBB... This model is also shown o achieve very high utilization while firmly grounded in the Rate Monotonic Theory. We will be looking into this model in our next revision.

5.0 Implementation Architecture

The architectural view of the kernel system is shown in fig. 5. In this model, kernel gets triggered by two events: (1) Time-Slice Interrupt and (2) Global Wakeup Interrupt. All other interrupts are ignored by the kernel. As shown in fig. 5, kernel has a flat architecture with a number of function calls that it supports.

The applications are loaded with partitions for text, data, stack, status and message areas. There are *n* different applications are loaded in the DRAM. Each application is assumed to be have its full set of command parameters available at the running time. The kernel writes into the kernel message area of the applications to inform of error conditions and why it halted the applications. Each application maintains its own user (or application) stack and the kernel does not interfere with this stack.

A set of applications are treated as a composite block known as *appset*. By design, each appset is created to be schedulable and tested for schedulability. Each appset has information regarding its constituent applications, their parameters and other relevant data. Each appset is created by application programmers who have tested their applications for real-time performance. Each appset data resides inside a centralized database called *App Registry*. At any time, the app registry can contain more than one appset. At the start

[19] A.K.Mok and Deji Chen, "A Multiframe Model for Real-Time Tasks", IEEE Transactions on Software Engineering, Vol 23., No.10, October 1997, pp 635-645.

time, which appset has to be scheduled is notified to the kernel by issuing a command to the kernel. The kernel receives its commands by reading its command queue and updating its queue pointers. The kernel reads its command queue only during its house-keeping time-slice interval.

With each appset, there is a *Schedule_Cycle_Table* either created a priori by the application designers and simply loaded into the DRAM or created on-the-fly by the kernel *Init* program by reading the data in the app registry. Since there could be more than one appset in the app registry, it is also possible to assemble a new collection of applications, a few from one appset and others from another appset. In this case, the kernel init program creates a new appset with appropriate data and generates a new Schedule_Cycle_Table for this new appset. Once the Schedule_Cycle_Table is ready, the kernel can start the appset.

In order to switch the tasks without any interruptions, the kernel maintains separate Task Stacks, one for each application. Task Stacks contain all the necessary information pertaining to each task at the time of task switching. Note that only State information regarding PE is stored in the Task Stack. The implicit assumption here is that DSEs are statically assigned to tasks and they just wait for the PE to issue new commands. At each time-slice, PE is loaded with a particular task, say task *j*, and only DSE assigned task *j* can now talk to the PE during the time-slice intended for task *j*. Similarly PE shifts its *Interrupt Focus* to the corresponding DSE during the time-slice interval.

There is a different mechanism suggested for MTCs. Only two MTCs are attached to a PE and PE gets an interrupt signal from these two MTCs. Since there are only two MTCs are available, they will be used to facilitate data tarnsfers for all the tasks. Suppose task *j* wants to move data from DRAM to CLA local memeory, it writes a small control program containing parameter blocks and other relevant data into a MTC. On finishing this data transfer, the MTC writes to a specific semaphore register - which is statically bound to task *j* - and generates an interrupt to the PE. When this interrupt is serviced by the PE, let us say, during the time-slice corresponding to task *k*, the PE will scan the semaphore registers and detect that the interrupt generated was not relevant to task *k*. The PE can ignore this interrupt until the task *j* is switched in.

This means that there should be mechanisms to perform *Interrupt Filtering* while tasks are switched. At any instant in time, the PE can be interrupted by a variety of sources: (1) interrupts from DSE associated with the current task (2) interrupts from a DSE associated with a previous task executed in a previous time-slice (3) MTC interrupts from the current task, (4) MTC interrupts from a previous task, (5) Software interrupts from the current task, (6) Software interrupts from previous tasks etc. This interrupt bombardment is detrimental to the real-time performance of the system. By having interrupt focus and filtering mechanisms, the PE is saved from too much of distraction and forced to perform the task to the fullest possible extent.

The intelligence for multitasking lies in the Schedule_Cycle_Table. We assume a specific organization so that it can be simple and maintained with ease. The creation of this table can be done just before the execution of appset or at the design time. It makes no difference when it is done. The processing requirements are also minimal.

The crucial assumption in building the kernel is that only PE is controlled for multitasking. If we can schedule all the resources of the system - PE, DSEs, MTE, HMTE, Semaphores - a greater efficiency will be achieved. This will be the focus for future work.

5.1 Types of Tasks

Tasks that the kernel manages can be divided into three types:

(a) A *Null Task* is the one where nothing happens and the PE is just waiting. Null tasks are part of the scheduling process because Rate-Monotonic Scheduling can only provide a maximum utilization of

about 88% processor cycles. In many cases, the utilization is likely to be less. This means that there must be periods where PE does not do any useful work but wait for the next time-slice interrupt. A null task can occur anywhere in the schedule cycle.

(b) A *House Keeping Task* where the kernel commands and other system related work are processed. By design, house keeping task occurs at the phase N-1 before a new schedule cycle of N phases begins. That is , the house keeping task is processed once in every N time-slices[20]. House keeping task does not do any application related work, but only kernel related work if activated in the *non-interactive* mode. Any new kernel commands entered during a schedule cycle are interpreted and executed in the time-slice interval when house keeping task appears. A large value of N may introduce sluggish response to kernel commands. In the *interactive mode*, the house keeping task queries the command queues of all applications and makes intelligent decisions regarding scheduling.

(c) A *Non-Null Task* is the one where the actual work required for an application is carried out. In order to track the information necessary to switch the tasks and to create the illusion of simultaneous processing of several applications, it is necessary to store the task information to be picked up at a later time-slice when a particular task is selected by the schedule cycle. There are several assumptions made regarding the application programs and the interrupt handling design. These are detailed next.

5.2 Key Assumptions made of Application Programs and Interrupt Services

Application programs are assumed to take care of the following:

(a) Application programs take care of interrupts by providing their own interrupt service routines. Each application may have a different need and different set of interrupts. Important interrupts from DSEs must be handled by the applications themselves. Applications are assumed to use special library functions to deal with SMTE, HMTE and AMTE data transfers. The kernel is oblivious to these interrupts. There could be also software interrupts generated by subroutines. Another source of interrupts is the numerical exceptions such as overflow, underflow, zero flags. To be specific, it is assumed that the **applications take care of the following list** of interrupts by including ISRs as library functions:

1. DSE DONE Interrupts
2. CT Done Interrupts except for the time-slice interrupt and GWI
3. SMTC Done Interrupts
4. AMTC Done Interrupts
5. Global Wake Up Interrupt
6. Hardware Break Point Interrupt
7. Trace Interrupt
8. Trap Interrupt
9. Software Interrupts generated by subroutines
10. Software Interrupts generated by numerical exceptions

The kernel does not care for the above interrupts and takes no effort to process them.

(b) The interrupts are processed *sequentially*. Interrupts are disabled when an interrupt is serviced[21]. The interrupt service mode is indicated by the bit 7 of PE PSW register S1. Interrupts are enabled by a CONT RTI instruction. It is conceivable that nested interrupt services can be handled by suitable interrupt service routines that can take care of nested interrupts. However, nested interrupts create havoc in real-time operations. The waiting time to come out of a set of nested interrupts is very hard to

[20] This does not have to so. We can definitely have more than one house-keeping time-slices to improve the response of the kernel system to commands.

[21] David C. Wyland, " Cirrus Leapfrog Architecture: Processing and Memory Transfer Engines (CLAE) - Hardware Architecture Specification ", Revision 6.0, page 20.

predict. To circumvent the timing jitters created by nested interrupts, the interrupts are serviced sequentially. This means that at any time, only one interrupt can be serviced. The interrupts that happened at the same time, are serviced according to their priority.

(c) The time-slice clock interrupt is given the highest priority in the interrupt service. Since there is no nesting of interrupts, the time-slice clock interrupt service has a waiting time of at most a single interrupt service. It is easier to compile from the list of all ISRs, the maximum service duration ($t_{ISR-MAX}$) and an average service duration ( ) of all ISRs . These two numbers can be used to correct time-slice clocks to minimize the processing time jitter that the tasks may experience.

(d) The fundamental assumption made here is that the quantity $t_{ISR-MAX}$ is very small compared to the time-slice duration. With this assumption, we can approximate the scheduling of the tasks close to the ideal. If this assumption is not true, we need to devise strategies to minimize time-slice jitter caused by large ISR latency.

(e) The Counter Timer CT0 is chosen to be the source for triggering time-slice interrupts. This timer will be exclusively used for generating time-slice clock.

(f) Though it is not absolutely necessary, Semaphore register 0 is used for avoiding race conditions in reading and writing data structures used in the kernel. We use the semaphore lock so that the possibility of another PE writing commands to the kernel can still be entertained with the model described here.

(g) Each task (or application) owns its own DSE or two. DSEs are NOT switched. They just wait in a dormant state until PE comes back in the next schedule cycle to feed some commands and work. Only PE gets loaded different tasks at different time-slices. Hence an interrupt issued by a DSE belonging to a previous task gets masked by disabling DSE DONE bits. PE notices only the interrupt from a DSE belonging to the current task and is oblivious to others. This idea of interrupt focus is crucial to the successful operation of the kernel.

(h) There are two possibilities for handling MTE/HMTE transfers:

(1) MTE and HMTE do not use interrupts to signal data transfer end. Instead they pass messages to the task that sent the request in the first place. MTE and HMTE maintain input and output queues where the tasks leave their data transfer requests. Each request contains Task-ID, Task-Request number and a parameter block containing data transfer details. The parameter block details are specified in the hardware specification[22]. Each request is placed in the input queue for MTE or HMTE. When the transfer is finished, these engines write the Task-ID, Task-Request number and task completion message in their output queues. The tasks can query the output queue and pop out output response. The details of this mechanism needs to be specified.

(2) MTCs are *dynamically* bound to specific tasks and the interrupts generated from a MTC can be uniquely identified by the PE by monitoring AMTC DONE and SMTC DONE bits along with corresponding semaphore registers. This provides an easy way of implementing interrupt filtering by the applications.

[22] David C. Wyland, " Cirrus Leapfrog Architecture: Processing and Memory Transfer Engines (CLAE) - Hardware Architecture Specification ", Revision 6.0, page 48. See also page 52 for details on programming MTE.

(3) In either way, it is the application that takes care of MTE/HMTE transfers. The kernel does not deal with interrupts generated from data transfers.

5.3 Task Data Structures

Because of the key assumptions made in the earlier section, it is possible to arrive at a task data structure that is both simple and efficient. For each of the n tasks that the kernel deals with there is an associated *Task Stack* where critical information regarding the task is stored. The complete information regarding the task is stored in the DRAM. A small part of this data structure called *Temp Stack* can be stored in CLA memory for quick manipulation and time critical decision making. Temp Stack makes sense only if there is a speed advantage in storing it in the CLA. Otherwise, it can be eliminated. What should really go in the Temp Stack depends on the way Time-Slice Clock ISR is written. This part of the specification is to be decided later.

First let us look into a data structure needed for App Registry which stores appsets containing a number of applications, say n. Let there be k appsets stored in the app registry. Then App Registry looks like:

```
Struct APP_REGISTRY
{
        Integer Registry-ID;          // Registry ID: a possibly a simple version of GUID
        Integer Number of Appsets;
        Integer   *Schedule_Cycle_Pointer;          // Where Schedule_Cycle_Table is kept
        APP_SET appsets[k];
}
```

An appset contains n different applications. They are organized in the following way:

```
Struct APP_SET
{
        Char [ 32] Appset Name;
        Char [ 32] Description;
        Char [ 8] Version;
        Char [ 8] Date of Release;
        Char [ 32] Author;
        Integer APPSET_ID;
        Integer Number_of_apps;
        Integer Schedule_Cycle_Length;
        Integer Number_Of_House_Keeping_Tasks;
        Integer House_Keeping_ID;                   // Is it -1 or something else?
        Integer Utilization;
        Integer   *Schedule_Cycle_Pointer;          // Where Schedule_Cycle_Table is kept
        Integer Time-Slice;           // Time-Slice in milliseconds
        APP   *apps[n];               // Pointers to apps structure
}
```

Finally the APP structure actually knows where applications are residing:

```
Struct APP
{
        Char [ 32] Application Name;
        Char [ 32] Author;
        Char [ 8] Release Date;
        Char [ 8] Version;
        Integer APP-ID;               // Same as Task-ID
        Integer Task_Period;
        Integer Computation;
```

```
        Integer Priority;

        Integer   * App_Text_Address;       // Address Pointer to Code Text
        Integer   App_Text_Length;          // Size of Text
        Integer   * App_Data_Address;       // Address Pointer to
                                            //Data required by
                                            //the Application

        Integer   App_Data_Length;          // Size of Data
        Integer   * App_Stack_Address;      // Pointer to Application
                                            //Stack maintained by the
                                            //   Application

        Integer   App_Stack_Size; // Size of Application
                                  //Stack required by
                                  //the Application

        Integer   * App_Status_Address;     // Pointer to Status section

        Integer   App_Status_Size;// Size of Status
        Integer   *App_Message_Area;
        Integer   App_Message_Area_Size;
}
```

An example of the Task Stack in a " C-like" pseudo code is given below:

```
Struct  TaskData
{
        Integer  Task_number;     // Task_number takes values from
                                            //0 to n-1
        Integer   PE-ID;          // PE number from 0 to 3 which runs
                                            // the task

        Integer   DSE-ID[8];      // DSEs used by the Application
        Integer   * Cluster_Address;        // Base Cluster Address
        Integer   * App_Text_Address;       // Address Pointer to Code Text
        Integer   App_Text_Length;          // Size of Text
        Integer   * App_Data_Address;       // Address Pointer to
                                            //Data required by
                                            //the Application

        Integer   App_Data_Length;          // Size of Data
        Integer   * App_Stack_Address;      // Pointer to Application
                                            //Stack maintained by the
                                            //   Application

        Integer   App_Stack_Size; // Size of Application
                                  //Stack required by
                                  //the Application

        Integer   * App_Status_Address;     // Pointer to Status section

        Integer   App_Status_Size;// Size of Status

        Integer   * Temp_Stack;   // Address Pointer to temporary
                                  //stack maintained by the
                                  // kernel for each task; this
                                  //can be in DRAM or CLA local
                                  // memory; Temp_Stack is a scratch-pad
                                  //and can be put CLA
                                  //if there are speed advantages
```

```
            Integer Temp_Stack_Size; // Size of Temp_Stack

            Integer   * Current_Text_Address;   // Address where execution
                                                        // must begin

            Integer   * Message_Queue_ Address;       // Pointer to Message
                                                        // Queue where (error)
                                                        //   messages can be left
                                                        // regarding the
                                                        //   execution

            Integer   DSE_Interrupt_Enable;     // This word tells how the application deals with
                                                //    the DSE interrupts; This word will be loaded
                                                //    into PE Cluster Hardware Register C1 before
                                                //    the application is run in every time-slice

            Integer   CT_Interrupt_Enable;      //  This word will be loaded into PE Cluster Hardware
                                                //     Register C3 before the application is run in
                                                //     every time-slice

            Integer Semaphore_Interrupt_Enable;         // This word will be loaded into PE cluster
                                                        //    hardware register C5

            Integer Interrupt_Enable;           // This word will be stored in Special Hardware
                                                   Register S4

            Integer General_Purpose_Register[32];       // 32 registers (R0-R31)are stored here

            Integer Special_Purpose_Register[32];       // 32 registers (S0-S31) are stored here
                                                        // It may not be necessary to store all
                                                        //   these registers, but done to avoid
                                                        //   possible trouble

            Long Total_Num _Jobs                // Total number of jobs done so far = Number of
                                                // quanta of work done so far = Current job number

            Long Total_PE_Time                  // Accumulated time spent by PE in processing

            Long  Time_Slice_Jitter             // Jitter experienced by the task because of
                                                // interrupt services

            Long  Start_Time                    // Time at which task was started

      }Task_Stack[n];
```

A CLA Task Stack is a simplified version of a process task. It does not have facilities for extended queue structures needed for handling nested interrupts. Since our interrupt processing is sequential, and the task switching is deterministic, much of machinery needed for a general purpose kernel can be avoided.

There are N phases to a schedule cycle which is determined a priori from the resource table information provided by the application developers; N= Number of phases of a Schedule Cycle; each phase corresponds to a clock timer interrupt; We also have n different applications to be multitasked by the kernel. This calls for a schedule data structure as described below:

```
      Struct SCHEDULE
      {
            Integer   Phase;            // Phase takes values from 0 to N-1
            Integer   Task-ID; // Task-ID takes values from 0 to n-1
                                    // 0 means a NULL_TASK
                                    // N-1 means a House_Keeping_Task, but it is in general -1 for
```

```
                                        // multiple house-keeping tasks in a schedule_cycle

            Boolean ENABLE_TASK;  // This is used to turn on/off a task
                                        // True means that task is enabled
        }
```

The struct SCHEDULE schedule[N] below is a table of N rows and 3 colums; the first column is the sequential index of the phases. The second column gives the task numbers which are turned on. If this column has a zero entry, then it means that the corresponding phase contains a Null Task. The third column contains the binary information whether the task is enabled or not. Initially all the tasks are enabled. Note that this structure contains the entire schedule cycle and hence consumes more storage space.

We can also use a little compression in storage space with the following assumptions: (1) Number of tasks is less than 15 so that we use a nibble to represent the task number. Then the house-keeping task can be represented by 0xFF and the rest of the tasks by their binary numbers. (2) The schedule cycle length can be represented by a 16 bit number. This is possible because we can use period transformations to reduce the schedule cycle length. (3) We can combine the turn on/off signal in the high nibble so that a compact 3 byte representation can be used for each entry in the schedule cycle:

```
            Struct SCHEDULE
        {
            16 bit-Integer     Phase;           // Phase takes values from 0 to N-1
            Byte               Task-ID; //       Task-ID takes values from 0 to 14
                                        // 0 means a NULL_TASK
                                        // 15 means a House_Keeping_Task
                                        // (Task-ID & 0x0F) gives the task number
                                        // (Task-ID & 0xF0) gives the task enable information
                                        // If (Task-ID & 0xF0) > 0 then task is enabled,
                                        // otherwise it is disabled

        }


        Struct SCHEDULE_CYCLE_TABLE
        {
            Integer N;                          // Length of Schedule_Cycle_Table = Number of
                                                // phases
            Integer   n;                        // Number of tasks to be scheduled
            integer Time_Slice_length;          // Duration of a time-slice in microseconds
            Integer Counter_Timer_ID;           // Which counter-timer engine is used for
                                                // running time-slice clock interrupts;
                                                // Convention is to use CT0;

            Integer   CT_Source_Option;         // Which clock source is selected to run the
                                                // Counter-Timer Engine

            Integer   CT_LIMIT_VALUE;           // Which determines the time-slice duration
            SCHEDULE Schedule[N];               // Schedule_Cycle_Table is an array of  N structures
            Boolean ENABLE_Schedule_Cycle ;             // This is used for turning on/off schedule
                                                        // cycle on a global level
            Integer  Current_Phase;                     // current phase of the table to be executed

            Integer * Kernel_Command_Queue_Address;             //pointer to command
                                                                //queue start
            Integer * Command_Queue_Read_Pointer;   // Current pointer to command
                                                    // to be executed;
            Integer * Command_Queue_Write_Pointer;  // pointer where command will be written
            Integer   Command_Queue_Size;           // Command Queue is a circular buffer
            Integer * Message_Address           // Pointer where the status/error messages
                                                //from the kernel can be put
```

```
                Integer   Message_Size                   // Message Area Size in Bytes
                Integer * Current_Message_Pointer // Current Pointer where messages will be put

                Char [ ]  Version Number;                // Book-keeping information useful for debugging
                Char [ ]   Algorithm Used;
                Char [ ]  Date of Creation;
                Char [ ]  List of  tasks and their data;
                Char [ ]  Authors;
                Char [ ]  Date of QA Approval;
                Char [ ]  Release Date
                Char [ ]  Release Version
        } Schedule_Cycle_Table, *Schedule_Cycle_Table_Pointer;
```

At the time of initialization, the following must be done:

```
        for(k = 0; k < N; k++)
        {
                Schedule_Cycle_Table.Schedule[k].phase = k;           // Set sequential index
                Schedule_Cycle_Table.Schedule[k].ENABLE_TASK = True;     // Enable all tasks
        }
        Schedule_Cycle_Table.Current_Phase = -1;
        Schedule_Cycle_Table.ENABLE_Schedule_Cycle = False;       // Multitasking will be started by
                                                                  // setting this value to be true
```

The pointer to the Schedule_Cycle_Table, *Schedule_Cycle_Table_Pointer is stored in a known location so that the table can retrieved easily. Since this table is a global resource, its access is protected by a semaphore lock so that race or blocking does not occur.

5.4 Time-Slice Clock and Its Interrupt Mechanism

The time-slice clock is generated with the Counter Timer unit 0, known CT0. CT0 has two 32 bit registers, CT0-Register0 Control and CT0-Register1 Counter register.

The CT0-Register0 Control register defines the clock source input to the counter timer unit with bits 5, 6, and 7. These three bits select one of the seven clock sources T1 - T7 generated by the Global Timing Generator. Bits 2, 3, 4 define which of these sources set the Run bit, namely bit 0 of CT0-Register0. These bits, collectively known as Start bits should be all set 0 to denote that none of the timing sources set the Run bit. The bit 1, known as the C it, should be enabled for continuous operation.

The CT0-Register1 is the counter register which holds a 16 bit LIMIT value and a 16 bit COUNTER value. The COUNTER value will be incremented at each clock and when it reaches the LIMIT value, the next clock will reset the COUNTER value to zero. At the same time, CT0 sets the interrupt bit - bit 0 of Cluster Hardware Register C2- also known as the Counter Timer Done register. An interrupt flag - bit30 of PSW - will be generated if bit0 of C2 and bit0 of C3 are enabled. An interrupt will be generated if bit30 of S4 is enabled in addition. Hence the initialization should do the following to ensure time-slice clock is generated properly:

```
Init_Time_Slice_Clock()
        {
                CT0-Register1_LIMIT = Schedule_Cycle_Table.CT_LIMIT_VALUE;
                                        // Set the appropriate limit value for the counter
                CT0-Register0_CLOCK = Schedule_Cycle_Table.CT_Source_Option &0x00000007;
                                        // The least significant three bits of CT_Source_Option
                                        // defines the timing source
                CT0-Register0_START = 0;
                                        // None of the timing source can start the operation
```

```
		CT0-Register0_C_bit = 1;
						// Though continuous mode is selected, the CT0 will start
						// running only when PE writes a 1 into CT0-Register0_bit 0
	}
```

When the kernel is ready to be run, PE writes 1 into CT0-Register0_bit 0 and the time-slice clock starts generating interrupts at the desired frequency. This can be done in one stroke as shown below:

```
Start_Time_Slice_Clock()
{
	CT0-Register0 = (Schedule_Cycle_Table.CT_Source_Option & 0x00000007)*32 + 3;
				// Do all at once and start the time-slice clock
}
```

Since CLA has only one interrupt signal, after receiving the interrupt, the interrupt service routine must decide which particular event has really caused the interrupt. This is tested in the following way:

```
Test_For_Time_Slice()
	{
		if (CT0-Register0_C_bit & S1_bit30 & S4_bit30 & C2_bit0 & C3_bit0)
			{
				if (Schedule_Cycle_Table.ENABLE_Schedule_Cycle ==True)
				{
					Do Time_Slice_Interrupt_Service();
				}
				else
				{
					Write_Message();
				}
			}
	}
```

The above routine is only a part of list of tests that the ISR has to do. Another routine that can wake up the kernel is the GWI:

```
Test_For_GWI()
	{
		if(S1_bit26 & S4_bit26)	//GWI bits
		{
			Init();	// Initialize kernel and load required functions
			if(New_Command = True)  // There must be somewhere a command flag to show
						// something new has happened
			{
				Get_Kernel_Command();   // Look in the kernel command queue
				Reset_Pointers(); // Clean up
				Execute_Command();	// Do it.
				Get_Application_Command();	// Scan all application
							// command queues
				Reset_Pointers();		// Clean up
				Execute_Command();
			}
		}
	}
```

At every time-slice interrupt the following events occur:

```
Time_Slice_Interrupt_Service( )
```

```
{
Disable_All_Interrupts();     // Shut off any interrupts that can occur! Now the kernel is running
                              // and no other tasks are running. According to our assumptions
                              // this is not REALLY necessary. But, hey, programmers are
                              // uncontrollable!! They may not follow our guidelines!

Save_Context();               // Save all registers in the Temp_Stack in CLA
                              // If CLA memory is precious and is needed for other important
                              //  storage, then Temp_Stack can be located in DRAM

Save_Temp_Stack();            // Save all interruptions that happened during
                              //  time-slice in the Task_Stack in DRAM if there is a Temp_Stack
Get_Phase();                  // Find the current phase and update the next phase

Get_Task_ID();                // This returns the Task-ID to be loaded for execution
                              // Task-ID = Schedule_Cycle_Table.Schedule[k].Task-ID
                              //ENABLE_TASK =
                              // Schedule_Cycle_Table.Schedule[k].ENABLE_TASK
                              // If ENABLE_TASK is false, then Task-ID = 0

If (Task-ID != Null_Task && Task-ID != House_Keeping_Task)
        {
        Clear_Temp_Stack();        // Prepare the Temp_Stack in CLA or where ever it is

        Load_Task_Stack(Task-ID);           // Load Task_Stack back into
                                                    // Temp_Stack; Set PC
        Enable_Time_Slice_Interrupt();
        Enable_Task_Interrupts(Task-ID);    // Enable only expected interrupts;
                                                    // Disable rest of interrupts
        Run_Task(Task-ID);                  // Set the Run Bit in PSW

        }
else if (Task = House_Keeping_Task)
        {
        Clear_Temp_Stack();
                                            // ENABLE_TASK must be true to accept kernel
                                            // commands; otherwise it is not possible
                                            // to control scheduling

        Load_House_Keeping_Task_Stack();
        Enable_House_Keeping_Interrupts();
        Enable_Time_Slice_Interrupt();
        Run_House_Keeping_Task();

        }
else if (Task = Null_Task)
        {
        Enable_Time_Slice_Interrupt();
        Wait_For_Time_Slice Interrupt();
        }
}
```

5.5 Level 0 Tasks

Here we describe a brief sketch of all the required low level functions. Some of these functions may require MTE operations which is not considered here.

```
Disable_All_Interrupts()
        {
```

```
                    // Clear higher order bits of PE Special Purpose Register S4;
                    for(k = 16; k < 32; k++) S4.bit(k) = 0;
                    // This will stop any interrupt being generated[23]
          }

Save_Context()
          {
                    // The current state of the machine is stored in a temporary area called Temp_Stack
                    // Temp_Stack is assumed to have the following elements:
                    // R_temp_Stack[0-31] to store GP registers
                    //S_temp[0-31] to store SP registers
                    // C1_temp,C3_temp,C5_temp, S4_temp to store interrupt mask registers
                    // task_temp to store task number

                    // Saving context involves the following:
                    for(k=0 ; k < 32; k++)
                    {
                              R_temp_Stack[k] = R[k];      // Save all General Purpose registers
                              S_temp_Stack[k] = S[k];    // Save all Special Purpose registers
                    }
                    C1_temp = C1;    // Save Interrupt mask registers
                    C3_temp = C3;
                    C5_temp = C5;
                    S4_temp = S4;
                    task_temp = Task_number;             // Save current task number
          }

Save_Temp_Stack()
          {
                    // Dump all temporary data into task stack

                    Task_Stack[task_temp].DSE_Interrupt_Enable =C1_temp;
                    Task_Stack[task_temp].CT_Interrupt_Enable = C3_temp;
                    Task_Stack[task_temp].Semaphore_Interrupt_Enable = C5_temp;
                    Task_Stack[task_temp].Interrupt_Enable = S4_temp;

                    for(k=0; k < 32; k++0
                    {
                              Task_Stack[task_temp].General_Purpose_Register[k] = R_temp[k];
                              Task_Stack[task_temp].Special_Purpose_Register[k] = S_temp[k];
                    }
          }


Get_Phase()
          {
                              integer test, phase, next_phase;
          Loop:               test = Read Semaphore register 0;  // How to do this is not defined yet in the
                                                                 // hardware spec.
                              if (test == 0)
                              {
                                        // we got the semaphore locked
                                        phase = Schedule_Cycle_Table.Current_Phase;
                                        next_phase = (phase + 1) mod N;
                                        Schedule_Cycle_Table.Current_Phase = next_phase;
                                                                    // Put next value for phase ready
```

[23] In CLAE Revision 6.0 document, page 9, second paragraph says that "...Note that the Software Interrupt and Trace Interrupt bits have no corresponding IE bits. Neither bit is maskable, and both are automatically reset upon execution of an RTI instruction". However, table 2-4 explicitly shows that these interrupts are maskable. Dave has agreed to make all interrupts maskable.

```
                        Return Phase;
                }
                else
                {
                        Wait for a little bit of time;   // We do not have a good
                                                         // way of doing this either; We do not want
                                                         // interrupts interfering!
                        go to Loop;
                }
        }

Get_Task_ID(Integer Phase)
        {
                integer test, Task-ID;
        Loop:   test = Read Semaphore register 0;  // How to do this is not defined yet in the
                                                   // hardware spec.
                if (test == 0)
                {
                        // we got the semaphore locked
                        Task-ID = Schedule_Cycle_Table.Schedule[Phase].Task-ID;
                        if(Schedule_Cycle_Table.Schedule[Phase].ENABLE_TASK = False)
                        {
                                Task-ID = 0;  // No enabling of task
                        }

                        Return Task_ID;
                }
                else
                {
                        Wait for a little bit of time;   // We do not have a good
                                                         // way of doing this either; We do not want
                                                         // interrupts interfering!
                        go to Loop;
                }
        }

Clear_Temp_Task()
        {
            for(k=0 ; k < 32; k++)
            {
                R_temp_Stack[k] = 0         // Clear all General Purpose registers
                S_temp_Stack[k] = 0         // Clear all Special Purpose registers
            }
            C1_temp = 0;      // Clear Interrupt mask registers
            C3_temp = 0;
            C5_temp = 0;
            S4_temp = 0;
        }

Load Task_Stack(Integer Task-ID)
        {
            // restore all registerfile back into CLA
            for(k=0; k < 32; k++)
            {
                S[k] = Task_Stack[Task-ID].General_Purpose_Register[k];
                R[k] = Task_Stack[Task-ID].Special_Purpose_Register[k];
            }
                C1 = Task_Stack[Task-ID].DSE_Interrupt_Enable;
                C3 = Task_Stack[Task-ID].CT_Interrupt_Enable;
                C5 = Task_Stack[Task-ID].Semaphore_Interrupt_Enable;
        }
```

```
Enable_Task_Interrupts( Integer Task-ID)
        {
                S4 =Task_Stack[Task-ID].Interrupt_Enable;           // This is the switch that turns on
                                                                    //all task interrupts
                Clear Interrupt bits();                                     // Use ANDN instruction to
remove
                                                                    // interrupt flags
        }

Run_Task()
        {
                S1_bit15 = 1;  // Set PE Run bit in the PSW and PC points to thr right place
        }

Enable_Time_Slice_Interrupt()
        {

                S4_bit 30 = 1;      // general turn-on
                C3_bit 0 = 1;       // CT enable bit
                Clear Interrupt bits();
        }

Wait_For_Time_Slice_Interrupt()
        {       int k,j,m;
                j =0;
                m = 1;
        loop:   if(m ==1)      // do something silly and waste time!
                {
                        for(k=0; k< 32000; k++)
                        {
                                j++;
                        }
                        m = j/32000;
                        goto loop;
        }

Load_House_Keeping_Task_Stack(Task-ID)
        {
                if(Task-ID == Schedule_Cycle_Table.N -1)
                {
                // restore all registerfile back into CLA
                for(k=0; k < 32; k++)
                {
                        S[k] = Task_Stack[Task-ID].General_Purpose_Register[k];
                        R[k] = Task_Stack[Task-ID].Special_Purpose_Register[k];
                }
                        C1 = Task_Stack[Task-ID].DSE_Interrupt_Enable;
                        C3 = Task_Stack[Task-ID].CT_Interrupt_Enable;
                        C5 = Task_Stack[Task-ID].Semaphore_Interrupt_Enable;
                }
        }


Enable_House_Keeping_Interrupts(Task-ID)
        {
                if(Task-ID == Schedule_Cycle_Table.N -1)
                {
                S4 =Task_Stack[Task-ID].Interrupt_Enable;           // This is the switch that turns on
                }                                                   //all task interrupts
```

```
        }

Run_House_Keeping_Task(Task-ID)
        {
                if(Task-ID == Schedule_Cycle_Table.N -1)
                {

                S1_bit15 = 1;  // Set PE Run bit in the PSW
                }
        }

Write_Message( Char * Text)
        {
                strcpy(Text, Schedule_Cycle_Table.Current_Message_Pointer); // Put the message
                Update_Pointer();
        }
```

5.6 Level 1 Tasks

Commands to the kernel are sent by the host or another task accepting read/write for commands to the kernel. Another PE could also attempt to control the kernel by placing commands in the command queue. Here is a list of items of work done during the house-keeping task:

```
House_Keeping_Task()
        {
                Read the Command Buffer Pointers;       // command buffer locations are defined in
                                                        // Schedule_Cycle_Table; By reading the
                                                        // read and write pointers, figure out if there
                                                        // new commands waiting; read all command
                                                        // buffers
                If there are new commands waiting
                        {
                        For each new command Do
                                {
                                        if (command = " Load APPSET-ID ")
                                        {
                                                Load_appset();    // Load the entire appset as it is
                                        }
                                        if command = " Load APPSET-ID #1 #2 #5 #7 ")
                                        {
                                                Load_appset(*argv);       // Load only specified
                                                                          // applications in the
                                                                          // taskset
                                        }
                                        if(command = "Create_Schedule")
                                        {
                                                Create_Schedule_Cycle_Table();    // Overwrite the
                                                                                  // new schedule
                                        }
                                        If (command = "start")      // commands are simple text strings
                                        {
                                                Start_Schedule();
                                        }

                                        If (command = "stop")
                                        {
                                                Stop_Schedule();
```

```
                    }

                    if(command = "enable(p1,p2)")
                    {
                              Enable_Task(p1,p2);
                    }

                    if(command = "disable(p1,p2)")
                    {
                              Disable_Task(p1,p2);
                    }
          }
     }
     else
     {
          Wait_For_Time_Slice_Interrupt();
     }
}
```

The individual calls for the command parsing are given below:

```
Start_Schedule ()
     {
          Schedule_Cycle_Table.ENABLE_Schedule_Cycle = True;
     }


Stop_Schedule ()
     {
          Schedule_Cycle_Table.ENABLE_Schedule_Cycle = False;
     }


Enable_Task(Integer Task-ID, Integer Phase)
     {
          if(Task-ID == Schedule_Cycle_Table.Schedule[Phase].Task-ID)
          {
               Schedule_Cycle_Table[Phase].ENABLE_TASK = True;
          }
     }

Disable_Task(Task-ID,Phase)
     {
          if(Task-ID == Schedule_Cycle_Table.Schedule[Phase].Task-ID)
          {
               Schedule_Cycle_Table[Phase].ENABLE_TASK = False;
          }
     }
```

5.7 Kernel Programs

The kernel program consists of two sub programs: kernel *Init* program which gets invoked at the starting time of the kernel and kernel *Run* program which becomes operational once the multitasking is started. Both these programs are built with function calls. Since CLA memory is precious, only these two programs and their function pointers are loaded in the CLA memory. The rest of actual programs reside only in DRAM. These two programs are made compact so that they take up very little space. Different functions are loaded into CLA as needed.

5.8 Kernel *Init* Program

The kernel *init* program is triggered by a GWI interrupt. The assumptions made at this point are:

1. The kernel function pointers are loaded into CLA Local Memory.
2. Commands are placed in the kernel command queue.
3. The App Registry is loaded in DRAM.

```
Init ()
{
        Allocate_CLA_Memory();   // Allocate space for scratch pad
        Read_Kernel_Command_Queue();
        Read_App_Command_Queue();
        Read_App_Registry();      // Read all entries
        Allocate_DRAM_Memory();          //Space for task stacks in DRAM
        Create_Task_Stacks();
        Load_Appset();    // Load the applications in the task stack
        Test_Schedule(); // Do Lehoczky-Sha-Ding test
        Create_Schedule_Cycle_Table();    // write out the schedule
        Init_Time_Slice_Clock();    // Set the clock
        Start_Time_Slice_Clock(); // Start it
        Start_Schedule();           // Start schedule
        Wait_For_Time_Slice_Interrupt();    // Wait for the clock
}
```

5.9 Kernel *Run* Program

Kernel *Run* program is the main activity of the kernel during multitasking. Kernel Run program is started with the arrival of the first time-slice interrupt:

```
Run()
{
{
        Disable_All_Interrupts();
        Save_Context();
        Save_Temp_Stack();
        Get_Phase();
        Get_Task_ID();

        If (Task-ID != Null_Task && Task-ID != House_Keeping_Task)
        {
                Clear_Temp_Stack();
                Load_Task_Stack(Task-ID);
                Enable_Time_Slice_Interrupt();
                Enable_Task_Interrupts(Task-ID);   // Enable only expected interrupts;
                                                   // Disable rest of interrupts
                Run_Task(Task-ID);                 // Set the Run Bit in PSW

        }
        else if (Task = House_Keeping_Task)
        {
        Clear_Temp_Stack();
        Load_House_Keeping_Task_Stack();
        Enable_House_Keeping_Interrupts();
        Enable_Time_Slice_Interrupt();
        Run_House_Keeping_Task();
```

```
            }
            else if (Task = Null_Task)
            {
            Enable_Time_Slice_Interrupt();
            Wait_For_Time_Slice Interrupt();
            }
    }
```

The run program is identical to the service routine for time-slice interrupt.

5.10 Kernel Organization in DRAM and CLA Local Memory

Kernel area is defined by two parameters for DRAM which indicates the high and low address of the kernel area. Similarly a high and low pointers define the CLA kernel area. Applications must make sure that they do not read or write into the kernel area.

5.11 List of Interrupt Services

The following is the list of Possible Interrupt Services required by the system. The priorities indicated refer to the order in which different interrupts are handled with in the ISR.

	Priority	Interrupt Service	Description
1	1	CT0-Time-Slice Interrupt	Drives the kernel to do multitasking
2	2	Global Wake Up Interrupt	Initializes the kernel
3	3	Hardware Break Point Interrupt	
4	4	Trace Interrupt	
5	5	Trap Interrupt	
6	6	CT1-Counter Timer Audio Frame Interrupt	
7	6	CT2- Counter Timer Audio Frame Interrupt	
8	6	CT3- Counter Timer Video Frame Interrupt	
9	6	CT4-CT15 Counter Timer Interrupts	Unused for the present
10	7	DSE 0 DONE Interrupt	
11	7	DSE 1 DONE Interrupt	
12	7	DSE 2 DONE Interrupt	
13	7	DSE 3 DONE Interrupt	
14	7	DSE 4 DONE Interrupt	
15	7	DSE 5 DONE Interrupt	
16	7	DSE 6 DONE Interrupt	
17	7	DSE 7 DONE Interrupt	
18	8	SMTC DONE Interrupt	
19	9	AMTC DONE Interrupt	
20	10	Software Interrupt (JSR)	
21	10	Software Interrupt - Divide By Zero	
22	10	Software Interrupt - Overflow	
23	10	Software Interrupt - Bounds Check	
24	10	Software Interrupt - Time Out	
25	10	Software Interrupt - Floating Point 1	

26	10	Software Interrupt - Floating Point 2	
27	10	Software Interrupt - Floating Point 3	
28	10	Software Interrupt - Memory fault	
29	10	Software Interrupt - Deadline Miss	
30	10	Software Interrupt - No data	

5.12 Model of Interrupt Service Routine (ISR)

Our model of ISR is a single Service Routine with a fixed interrupt vector location which will be pointed to whenever an interrupt occurs. The PE will always arrive at the same interrupt vector irrespective of the origin of interrupt. As shown in the table above there could be a variety of interrupt origins.

1. Each device or system requesting for interrupt raises its flag by setting a bit that can cause an interrupt. For example, DSE0 DONE bit is set to one when DSE0 wants to request an interrupt service. The PE has a choice of two different actions: It can accept the interrupt request or it can ignore the request.

2. The PE accepts the interrupt if the interrupt is enabled in the register S4 and other interrupt enable registers that are linked to the interrupt. Since there is a single hardware interrupt, a multilevel interrupt filtering is done by the ISR.

3. While an interrupt is being serviced, no new interrupts are allowed in.
4. Only when a Return From Interrupt (RTI) is executed, PE will accept new interrupt requests.
5. When RTI is executed, PE looks at all the bits that may lead to an interrupt generation. If any one of them is turned on, an interrupt is issued and PC is loaded with the interrupt vector.

6. It is the ISR that decides which of the possible causes of the interrupt to be serviced first. This decision stems from the placement of different filters for interrupts. this must correspond to the assumed priority structure of interrupts.

7. Before the interrupt is serviced, the bit that signaled the interrupt should be turned off to enable another interrupt from the same origin.

8. After the interrupt is serviced, a RTI instruction must be executed.
9. For every interrupt, the following sequence of actions are taken in the ISR:

```
If  (Interrupt is true) then
        {
                 if (interrupt caused by the priority 1 origin) then
                {
                        Turn- off the bit that signaled the interrupt;    // Notify that interrupt is
                                                                          //being served
                        Branch to the corresponding service;              // Go there
                        Return from Interrupt;                            // Enable Interrupts again
                }
                else if(interrupt caused by the priority 2 origin) then
                {
                        Turn- off the bit that signaled the interrupt;
                        Branch to the corresponding service;
                        Return from Interrupt;
                }
                else if(interrupt caused by the priority 3 origin) then
                {
                        Turn- off the bit that signaled the interrupt;
```

```
                Branch to the corresponding service;
                Return from Interrupt;
        }
        ...
        ...
        ...
        else if(interrupt caused by the priority 10 origin) then
        {
                Turn- off the bit that signaled the interrupt;
                Branch to the corresponding service;
                Return from Interrupt;
        }

}
```

The order in which the above "if then else" statements are executed is important They must correspond to the priorities assigned to the interrupts. The interrupting devices simply turn on their bits and leave them there. When PE returns from a RTI instruction, it can look at the next pending interrupt signal and activate it. Different ISRs written by application developers are merged together as shown above. The important thing is the placement of different ISRs written by different programmers. The placement is guided by the priority of interrupt service.

5.13 Interactive Multitasking

By interactive multitasking, we mean that kernel function can be altered interactively such as starting a new application or halting another etc. The kernel is amenable to such modifications by increasing the frequency of house-keeping task and monitoring the inputs in the application command queues. Since testing for schedulability can be done on the fly, including or deleting an application from the schedule becomes easy. The variations on this theme can easily be implemented with the current structure.

5.14 Estimation of DRAM Storage Requirements

From the descriptions in section 5.3 and fig. 5, we arrive at the following storage requirements in DRAM:

<u>App Registry:</u> Let there be k appsets in the app registry each with n applications. App registry is a static storage space where the appsets are kept. Then these k appsets need a total storage space of bytes. For example, if the app registry contains 3 appsets each with 7 applications, then the total storage requirement is bytes.

<u>Task Stacks:</u> There needs to n task stacks for each application in the appset. Only one appset is loaded at any any time. Each task stack needs 352 bytes. For n=7, this translates into 2464 bytes.

<u>Schedule Cycle Table:</u> This table contains important schedules needed for multitasking. The storage requirement for a Schedule Cycle Length of L is given by bytes if we do not use a compressed format. With compression, the storage becomes bytes. For n = 7 applications, a lower bound of L = 20000 is reasonable. This amounts to 60232 bytes.

<u>Message and Command Buffers:</u> These buffers are used to get the messages from the kernel and write kernel commands. An adequate size of 256 bytes for each of these buffers is recommended. This amounts to 512 bytes.

<u>Interrupt Service Routine:</u> The model of the interrupt service routine is that it contains all the service functions in a monolithic structure pointed out by a single fixed interrupt vector. There are 8 DSE DONE interrupts, 16 CT Done Interrupts, 2 MTC interrupts, 1 Global wake up interrupt, 1 hardware break point interrupt, 1 Trace and 1 trap interrupts, 1 software interrupt, and about 10 software interrupts generated by

numerical exceptions - totalling 41 interrupts. Assuming an average size of 256 bytes for each service routine, we have 256.41 = 10 496 bytes. This is definitely close to an upper bound rather than a typical value.

Kernel functions: We assume a total of 50 functions, each of size 256 bytes to include all kernel functions. This amounts to 12800 bytes. Once again this is an upper bound.

Prime Number Table: This table contains the first 170 prime numbers. The storage needed is 170.4 = 680 bytes.

Hence the total storage size comes to 270,568 bytes for the entire kernel. This does not take into account of the storage required for the applications and their command queues. The majority of the storage space, about 89%, goes to storing the schedule cycle table.

	Storage Requirement in Bytes for k=3, L = 20000 and n = 7 in DRAM
App Registry	3384
Task Stacks	2464
Schedule Cycle Table	60232
Message and Command Buffers	512
Interrupt Service Routine	10496 (Upper bound)
Kernel Functions	12800 (Upper bound)
Prime Number Table	680
Total	90568

5.15 List of Kernel Functions

The following are the list of kernel functions required; they are pointed by function pointers in CLA Memory.

	Function	Description
1	Read_Kernel_Command_Queue()	
2	Read_App_Command_Queue()	
3	Allocate_DRAM_Memory()	Allocation of DRAM
4	Allocate_CLA_Memory()	
5	Read_App_Registry()	Read all application parameters
6	Create_Task_Stacks()	
7	Test_Schedule()	Do Lehoczky-Sha-Ding test
8	Create_Schedule_Cycle_Table()	
9	Load_Appset(Appset-ID)	
10	Load_Appset(*argv)	
11	Start_Schedule()	
12	Stop_Schedule()	
13	Enable_Task(Task-ID,Phase)	
14	Disable_Task(Task-ID,Phase)	
15	Wait_For_Time_Slice_Interrupt()	
16	Disable_All_Interrupts()	
17	Save_Context()	
18	Save_Temp_Stack()	
19	Create_Temp_Stack()	

20	Get_Phase()	
21	Get_Task_ID(Phase)	
22	Clear_Temp_Stack()	
23	Clear_Task_Stack()	
24	Load_Task_Stack(Task-ID)	
25	Enable_Task_Interrupts()	
26	Run_Task()	
27	Enable_Time_Slice_Interrupt()	
28	Load_House_Keeping_Task_Stack()	
29	Enable_House_Keeping_Interrupts()	
30	Run_House_Keeping_Task()	
31	Write_Kernel_Message()	
32	Write_App_Message()	
33	Init_Time_Slice_Clock()	Setting of CT0 Counter Timer
34	Start_Time_Slice_Clock()	
35	Test_For_Time_Slice()	Same as the part in ISR
36	Test_For_GWI()	Same as the part in ISR
37	Time_Slice_Interrupt_Service()	Kernel *Run* Function
38	GWI_Service()	Kernel *Init* Function
39	Dump_Status()	Message dump for kernel
40	Resume_Schedule()	

5.16 Kernel Area Definition

6.0 Schedule Design Tool

A schedule design tool written in Visual Basic is available for designing application-specific real-time kernels. The tool gives valuable information regarding task schedulability and delivers a working task schedule cycle for implementation. As many as seven tasks can be loaded and their temporal relationships can be understood from the tool. There are facilities to perform period transformation and change the parameters at will. Both Lehoczky-Sha-Ding test and slot scheduling are done. Figs 6 and 7 show two views of this tool.

6.1 Schedule Design Tool Code

```
VERSION 5.00
Begin VB.Form Form1
   AutoRedraw      =   -1  'True
   Caption         =   "Rate-Monotonic Scheduler For CLA Micro-Kernel"
   ClientHeight    =   8355
   ClientLeft      =   60
   ClientTop       =   345
   ClientWidth     =   10320
   LinkTopic       =   "Form1"
   ScaleHeight     =   8355
   ScaleWidth      =   10320
   StartUpPosition =   3  'Windows Default
   Begin VB.CommandButton Command5
      Caption         =   "Transform Task"
      Height          =   615
```

```
      Left            =   3720
      TabIndex        =   16
      Top             =   1320
      Width           =   1215
   End
   Begin VB.CommandButton Command4
      Caption         =   "Delete Task"
      Height          =   615
      Left            =   3720
      TabIndex        =   14
      Top             =   480
      Width           =   1215
   End
   Begin VB.CommandButton Command3
      Caption         =   "Clear"
      Height          =   615
      Left            =   2640
      TabIndex        =   12
      Top             =   1320
      Width           =   975
   End
   Begin VB.HScrollBar HScroll1
      Height          =   615
      Left            =   2880
      Max             =   100
      Min             =   1
      TabIndex        =   10
      Top             =   3000
      Value           =   1
      Width           =   2175
   End
   Begin VB.ListBox List2
      Height          =   1230
      Left            =   5640
      TabIndex        =   9
      Top             =   1560
      Width           =   4335
   End
   Begin VB.TextBox Text1
      Height          =   375
      Left            =   1680
      TabIndex        =   7
      Top             =   240
      Width           =   615
   End
   Begin VB.ListBox List1
      Height          =   1230
      Left            =   5640
      TabIndex        =   6
      Top             =   120
      Width           =   4335
   End
   Begin VB.CommandButton Command2
      Caption         =   "Do RM Schedule"
      Height          =   615
      Left            =   2760
      TabIndex        =   5
      Top             =   2160
      Width           =   2055
   End
   Begin VB.CommandButton Command1
      Caption         =   "Add Task"
```

```
      Height         =   615
      Left           =   2640
      TabIndex       =   4
      Top            =   480
      Width          =   975
   End
   Begin VB.TextBox Text3
      Height         =   375
      Left           =   1680
      TabIndex       =   1
      Top            =   1920
      Width          =   615
   End
   Begin VB.TextBox Text2
      Height         =   375
      Left           =   1680
      TabIndex       =   0
      Top            =   1080
      Width          =   615
   End
   Begin VB.Line Line4
      X1             =   2400
      X2             =   2400
      Y1             =   0
      Y2             =   2880
   End
   Begin VB.Line Line3
      X1             =   5280
      X2             =   5280
      Y1             =   0
      Y2             =   2760
   End
   Begin VB.Label Label6
      Caption        =   "   Victor Ramamoorthy                    February 1998                               CLA Micro-RT
Kernel Design  "
      Height         =   375
      Left           =   1800
      TabIndex       =   15
      Top            =   8040
      Width          =   6855
   End
   Begin VB.Label Label5
      Height         =   855
      Left           =   5760
      TabIndex       =   13
      Top            =   3000
      Width          =   4095
   End
   Begin VB.Label Label4
      Height         =   495
      Left           =   360
      TabIndex       =   11
      Top            =   3120
      Width          =   2295
   End
   Begin VB.Line Line2
      X1             =   0
      X2             =   10200
      Y1             =   2880
      Y2             =   2880
   End
   Begin VB.Line Line1
```

```
      X1              =   0
      X2              =   2400
      Y1              =   960
      Y2              =   960
   End
   Begin VB.Label Label3
      Caption         =   "Time-Slice (ms)"
      Height          =   255
      Left            =   240
      TabIndex        =   8
      Top             =   360
      Width           =   1335
   End
   Begin VB.Label Label2
      Caption         =   " Computation  (ms)"
      Height          =   495
      Left            =   240
      TabIndex        =   3
      Top             =   1920
      Width           =   1335
   End
   Begin VB.Label Label1
      Caption         =   "Period (ms)"
      Height          =   255
      Left            =   360
      TabIndex        =   2
      Top             =   1200
      Width           =   855
   End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Private Sub Command1_Click()
'If period <> 0 And computation <> 0 And N < 7 Then
    'N = N + 1
    'T(N) = period
    'C(N) = computation

    'List1.AddItem "Task No. = " & N & "  Period = " & T(N) & "  Computation = " & C(N)

    'T(N) = period
    'C(N) = computation
    List1.AddItem period & "/" & computation & "/  ->  " & "   Period = " & period & "  Computation = " &
computation
    N = List1.ListIndex + 1

End Sub

Private Sub Command2_Click()
Dim i, j, index, address, addressmax, addressmin, k As Integer
Dim MIN As Double
Dim noofcycles, extra As Integer
Dim tot, used, percent, percentt As Double
Dim str1, str2, str3 As String
Dim time, test As Double
Dim ratio As Double
Dim ivar, factor As Integer
' Get all info from list1 control
' Parse the list data string: Look for /
```

```
lsdresult = False
For i = 0 To List1.ListCount - 1
    str1 = List1.List(i)
    j = InStr(1, str1, "/", 0)
    str2 = Mid(str1, 1, (j - 1))
    t(i + 1) = CDbl(str2)
    k = InStr(j + 1, str1, "/", 0)
    j = InStr(1, str1, "/", 0)
    str3 = Mid(str1, (j + 1), (k - j - 1))
    c(i + 1) = CDbl(str3)
Next i
N = List1.ListCount

j = 0
For i = 1 To N
    If t(i) <> 0 And c(i) <> 0 Then
        j = j + 1
        Tcopy(j) = t(i)
        Ccopy(j) = c(i)
    End If
Next i
N = j  ' Redefine value of N as the original may contain holes
For j = 1 To N
    MIN = Tcopy(1)
    index = 1
    For i = 1 To N
        If Tcopy(i) <= MIN Then
            MIN = Tcopy(i)
            index = i
        End If
    Next i
    ST(j) = MIN
    Tcopy(index) = 3200000
    SC(j) = Ccopy(index)
Next j

If timeslice <> 0 Then
    For i = 1 To N

        ratio = ST(i) / timeslice   'upper floor function
        ivar = Fix(ST(i) / timeslice)
        If (ratio - ivar) > 0 Then
            factor = ivar + 1
        Else
            factor = ivar
        End If
        QT(i) = factor
        ratio = SC(i) / timeslice
        ivar = Fix(SC(i) / timeslice)
        If (ratio - ivar) > 0 Then
            factor = ivar + 1
        Else
            factor = ivar
        End If

        QC(i) = factor
    Next i
End If
List2.Clear
For i = 1 To N
    List2.AddItem "Priority = " & i & "  Period = " & QT(i) & "(ts)  Computation = " & QC(i) & "(ts)"
 Next i
```

```
For i = 1 To schedulelength
   schedule(i) = 0
Next i

For i = 1 To N

   If i = 1 Then
      noofcycles = CInt(schedulelength / QT(i))
      For j = 1 To noofcycles - 1
         For k = 1 To QC(i)
            address = (j - 1) * QT(i) + k
            schedule(address) = i
         Next k
      Next j
   Else
      noofcycles = CInt(schedulelength / QT(i))

      For j = 1 To noofcycles - 1
            extra = 0
            addressmin = (j - 1) * QT(i) + 1
            addressmax = j * QT(i)
            For address = addressmin To addressmax
               If schedule(address) = 0 And extra < QC(i) Then
                  schedule(address) = i
                  extra = extra + 1
               End If
             Next address

      Next j

   End If
Next i



' Lehoczky-Sha-Ding test


If N <> 0 Then
   time = 0
   For i = 1 To N
      time = time + QC(i)
   Next i

' time contains the seed

looplsd:    test = 0

         For i = 1 To N
            ratio = time / QT(i)
            ivar = Fix(time / QT(i))
            If (ratio - ivar) > 0 Then
               factor = ivar + 1
            Else
               factor = ivar
            End If
            'test = test + (Fix(time / QT(i)) + 1) * QC(i)
            test = test + factor * QC(i)
         Next i

         If test <> time And test <= QT(N) Then
```

```
            time = test
            GoTo looplsd
        Else
            If test <= QT(N) And test = time Then ' Convergence and within deadline
                lsdresult = True
            Else
                lsdresult = False
            End If
        End If

Else
    lsdresult = False
End If


' Compute Utilization of the processor after assignment

tot = 0
used = 0
percent = 0
For i = 1 To schedulelength - 1000
    tot = tot + 1
    If schedule(i) <> o Then
        used = used + 1
    End If
Next i

percent = (used / tot) * 100

percentt = 0

For i = 1 To N
    percentt = percentt + QC(i) / QT(i)
Next i

percentt = percentt * 100

' Computing sequence length by prime number factorization
For i = 1 To 170
    work(i) = 0 ' reset all arrays
    expo(i) = 0
Next i

Dim inputvar, integervar, remainder As Integer
For i = 1 To N

    inputvar = QT(i)  ' Save it
    ' Check if QT(i) is prime
    For j = 1 To 170
        If inputvar = Prime(j) Then
            expo(j) = expo(j) + 1
            GoTo yesprime:
        End If
    Next j

Notprime:   ' A composite number
    For j = 1 To 170    ' Reset working array
        work(j) = 0
    Next j

breakup:
    For j = 1 To 170
```

```
        integervar = inputvar \ Prime(j)
        remainder = inputvar Mod Prime(j)

        If remainder = 0 And integervar > 1 Then
            work(j) = work(j) + 1
            inputvar = integervar
            GoTo breakup:
        End If

        If remainder = 0 And integervar = 1 Then
            work(j) = work(j) + 1
            GoTo finished:
        End If
    Next j

finished:
    For j = 1 To 170
        If work(j) > expo(j) Then expo(j) = work(j)
    Next j


yesprime:
Next i

' Find the LCM
seqlength = 1
For i = 1 To 170
    If expo(i) > 0 Then
        For j = 1 To expo(i)
            seqlength = seqlength * Prime(i)
        Next j
    End If
Next i


Label5.Caption = "Processor Utilization = " & Format(percent, "00.00") & " %  (" & Format(percentt, "00.00")
& " %)" & Chr(13) & " Schedule Cycle Length = " & seqlength & Chr(13) & "Lehoczky-Sha-Ding Test Result :
"
If lsdresult = True Then
    Label5.Caption = Label5.Caption & Chr(13) & " Task Set is Schedulable"
Else
    Label5.Caption = Label5.Caption & Chr(13) & " Task Set is NOT Schedulable"
End If

Open "C:\kernel-schedule.txt" For Output As #1
Print #1, " Kernel Schedule   " & Now
Print #1, "    Time Slice = " & timeslice & " (ms)"
For i = 1 To N
    Print #1, "Priority: " & i; "   Period = " & QT(i) & "(ts)"; "    Computation = " & QC(i) & "(ts)"
Next i
Print #1, "----------------------------------------------"
Print #1, " Processor Utilization = " & Format(percent, "00.00") & " %  (" & Format(percentt, "00.00") & " %)"
Print #1, " Schedule Cycle Length = " & seqlength
If lsdresult = True Then
    Print #1, " LSD test: Task Set is Schedulable"
Else
    Print #1, " LSD test: Task Set is Not Schedulable"
End If
Print #1, "  "
Print #1, "-----------------------------------------------"
Print #1, " "
For i = 1 To schedulelength
```

```
    Print #1, i; schedule(i)
Next i
Close #1
End Sub

Private Sub Command3_Click()
' Clear command
Dim i As Integer
For i = 1 To N
    t(i) = 0
    c(i) = 0
Next i
List1.Clear
List2.Clear

N = 0

End Sub

Private Sub Command4_Click()
' Delete a task pointed by list1
If Deleteaction = True And Deleteindex >= 0 And Deleteindex <= List1.ListCount Then
    List1.RemoveItem Deleteindex
    List2.Clear
    'T(Deleteindex + 1) = 0
    'C(Deleteindex + 1) = 0
    Deleteaction = False
End If


End Sub

Private Sub Command5_Click()
' Transform task
Transformaction = True
Transformindex = Deleteindex
Form1.Hide
Form2.Show
End Sub

Private Sub Form_Load()
Dim i As Integer
Dim str As String
N = 0
numoftasks = 0
schedulelength = 10000
List1.Clear
List2.Clear
ymax = 6000
ymin = 4500
xmin = 500
ygridmin = 4400
ygridmax = 8000
Deleteaction = False
Transformaction = False
Open "C:\prime.txt" For Input As #2
    For i = 1 To 170
        Line Input #2, Prime(i)
       ' If IsNumeric(str) = True Then Prime(i) = CInt(str)
    Next i
Close #2
```

```
End Sub

Private Sub HScroll1_Change()
Dim i, j, k, position, MAX, MIN, noofcycles, address, addressmax, addressmin As Integer
Dim seqlength As Double
Dim Nlimit As Integer
i = HScroll1.Value
position = HScroll1.Value
MAX = position * 100
MIN = (position - 1) * 100
Label4.Caption = "Min = " & MIN + 1 & "   Max = " & MAX
Form1.Cls
' Plot the grids
For i = 1 To 100
   Line (xmin, ygridmin)-(xmin + i * 90, ygridmax), RGB(100, 100, 100), B
Next i

For i = 10 To 100 Step 10
   Line (xmin, ygridmin)-(xmin + i * 90, ygridmax), RGB(0, 180, 0), B
Next i
For i = 0 To 100 Step 10
   j = MIN + i

      CurrentX = xmin + i * 90 - 200
   CurrentY = ygridmin - 300
   Print Format(j, "#####")
Next i

' Limit display plots
Nlimit = N
If Nlimit > 7 Then Nlimit = 7


' Plotting the task cycles for all tasks

For index = 1 To Nlimit
   For j = 1 To schedulelength
      spare(j) = 0
   Next j


   noofcycles = CInt(schedulelength / QT(index))
   For j = 1 To noofcycles - 1
      For k = 1 To QC(index)
         address = (j - 1) * QT(index) + k
         spare(address) = index
      Next k
   Next j


   For i = 1 To 100
      If spare((position - 1) * 100 + i) = 0 Then
         Line ((i - 1) * 90 + xmin, ymin + index * 200)-(i * 90 + xmin, (ymin + index * 200 + 100)), RGB(0, 0, 0),
BF
      ElseIf spare((position - 1) * 100 + i) = 1 Then
         Line ((i - 1) * 90 + xmin, ymin + index * 200)-(i * 90 + xmin, (ymin + index * 200 + 100)), RGB(255, 0,
0), BF
      ElseIf spare((position - 1) * 100 + i) = 2 Then
         Line ((i - 1) * 90 + xmin, ymin + index * 200)-(i * 90 + xmin, (ymin + index * 200 + 100)), RGB(0, 255,
0), BF
      ElseIf spare((position - 1) * 100 + i) = 3 Then
```

```
        Line ((i - 1) * 90 + xmin, ymin + index * 200)-(i * 90 + xmin, (ymin + index * 200 + 100)), RGB(0, 0,
255), BF
      ElseIf spare((position - 1) * 100 + i) = 4 Then
        Line ((i - 1) * 90 + xmin, ymin + index * 200)-(i * 90 + xmin, (ymin + index * 200 + 100)), RGB(255,
255, 0), BF
      ElseIf spare((position - 1) * 100 + i) = 5 Then
        Line ((i - 1) * 90 + xmin, ymin + index * 200)-(i * 90 + xmin, (ymin + index * 200 + 100)), RGB(255, 0,
255), BF
      ElseIf spare((position - 1) * 100 + i) = 6 Then
        Line ((i - 1) * 90 + xmin, ymin + index * 200)-(i * 90 + xmin, (ymin + index * 200 + 100)), RGB(0, 255,
255), BF
      ElseIf spare((position - 1) * 100 + i) = 7 Then
        Line ((i - 1) * 90 + xmin, ymin + index * 200)-(i * 90 + xmin, (ymin + index * 200 + 100)), RGB(255,
255, 255), BF
      End If
  Next i
Next index

' Plotting combined schedules of all tasks

  For i = 1 To 100
    If schedule((position - 1) * 100 + i) = 0 Then
      Line ((i - 1) * 90 + xmin, ymin + 1600)-(i * 90 + xmin, (ymin + 1600 + 100)), RGB(0, 0, 0), BF
    ElseIf schedule((position - 1) * 100 + i) = 1 Then
      Line ((i - 1) * 90 + xmin, ymin + 1600)-(i * 90 + xmin, (ymin + 1600 + 100)), RGB(255, 0, 0), BF
    ElseIf schedule((position - 1) * 100 + i) = 2 Then
      Line ((i - 1) * 90 + xmin, ymin + 1600)-(i * 90 + xmin, (ymin + 1600 + 100)), RGB(0, 255, 0), BF
    ElseIf schedule((position - 1) * 100 + i) = 3 Then
      Line ((i - 1) * 90 + xmin, ymin + 1600)-(i * 90 + xmin, (ymin + 1600 + 100)), RGB(0, 0, 255), BF
    ElseIf schedule((position - 1) * 100 + i) = 4 Then
      Line ((i - 1) * 90 + xmin, ymin + 1600)-(i * 90 + xmin, (ymin + 1600 + 100)), RGB(255, 255, 0), BF
    ElseIf schedule((position - 1) * 100 + i) = 5 Then
      Line ((i - 1) * 90 + xmin, ymin + 1600)-(i * 90 + xmin, (ymin + 1600 + 100)), RGB(255, 0, 255), BF
    ElseIf schedule((position - 1) * 100 + i) = 6 Then
      Line ((i - 1) * 90 + xmin, ymin + 1600)-(i * 90 + xmin, (ymin + 1600 + 100)), RGB(0, 255, 255), BF
    ElseIf schedule((position - 1) * 100 + i) = 7 Then
      Line ((i - 1) * 90 + xmin, ymin + 1600)-(i * 90 + xmin, (ymin + 1600 + 100)), RGB(255, 255, 255), BF
    End If
  Next i


' Checking deadlines for individual tasks

For index = 1 To Nlimit        ' index is the variable for a task
  For j = 1 To schedulelength
    spare(j) = 0       ' Clean spare storage
  Next j


  noofcycles = CInt(schedulelength / QT(index))

   For j = 1 To noofcycles - 1
          extra = 0
          addressmin = (j - 1) * QT(index) + 1
          addressmax = j * QT(index)
          For address = addressmin To addressmax
            If schedule(address) = index Then
              extra = extra + 1
            End If
           Next address

          If extra = QC(index) Then
```

```
                For address = addressmin To addressmax
                    spare(address) = 2    'Green color for meeting deadline
                Next address
            Else
                For address = addressmin To addressmax
                    spare(address) = 1      ' Red color for NOT meeting the deadline
                Next address
            End If

    Next j

    For i = 1 To 100

        If spare((position - 1) * 100 + i) = 1 Then
            Line ((i - 1) * 90 + xmin, ymin + 1800 + (index * 200))-(i * 90 + xmin, (ymin + 1800 + 100 + index *
200)), RGB(255, 0, 0), BF
        ElseIf spare((position - 1) * 100 + i) = 2 Then
            Line ((i - 1) * 90 + xmin, ymin + 1800 + (index * 200))-(i * 90 + xmin, (ymin + 1800 + 100 + index *
200)), RGB(0, 255, 0), BF

        End If
    Next i

Next index


End Sub







Private Sub List1_DblClick()
Deleteindex = List1.ListIndex
Deleteaction = True
Transformaction = True

End Sub

Private Sub Text1_Change()
If IsNumeric(Text1.Text) = True Then
    timeslice = CDbl(Text1.Text)
Else
    Text1.Text = ""
    timeslice = 0
End If
End Sub

Private Sub Text2_Change()
If IsNumeric(Text2.Text) = True Then
    period = CDbl(Text2.Text)
Else
    Text2.Text = ""
    period = 0
End If

End Sub
Private Sub Text3_Change()
If IsNumeric(Text3.Text) = True Then
    computation = CDbl(Text3.Text)
```

```
Else
    Text3.Text = ""
    computation = 0
End If
End Sub


VERSION 5.00
Begin VB.Form Form2
   Caption         =   "Transform Task"
   ClientHeight    =   1770
   ClientLeft      =   60
   ClientTop       =   345
   ClientWidth     =   2460
   LinkTopic       =   "Form2"
   ScaleHeight     =   1770
   ScaleWidth      =   2460
   StartUpPosition =   3  'Windows Default
   Begin VB.CommandButton Command1
      Caption         =   "OK"
      Height          =   375
      Left            =   840
      TabIndex        =   1
      Top             =   1320
      Width           =   855
   End
   Begin VB.TextBox Text1
      Height          =   375
      Left            =   600
      TabIndex        =   0
      Top             =   720
      Width           =   1215
   End
   Begin VB.Label Label1
      Caption         =   "Input the Scale Factor Below:"
      Height          =   375
      Left            =   360
      TabIndex        =   2
      Top             =   240
      Width           =   1815
   End
End
Attribute VB_Name = "Form2"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Private Sub Command1_Click()
Call transform_task
Form2.Hide
Form1.Show
End Sub

Private Sub Text1_Change()
If IsNumeric(Text1.Text) = True Then
    scalefactor = CDbl(Text1.Text)
Else
    scalefactor = 0
    Text1.Text = ""
End If

End Sub
```

```
Attribute VB_Name = "Module1"
Global period, computation, timeslice As Double
Global numoftasks As Integer
Global t(1 To 100), c(1 To 100), Tcopy(1 To 100), Ccopy(1 To 100) As Double
Global ST(1 To 100), SC(1 To 100) As Double
Global QT(1 To 100), QC(1 To 100) As Integer
Global N As Integer
Global schedulelength As Integer
Global schedule(1 To 10000) As Byte
Global spare(1 To 10000) As Byte
Global ymax, ymin, xmin, ygridmin, ygridmax As Single
Global Deleteindex, Transformindex As Integer
Global Deleteaction, Transformaction As Boolean
Global lsdresult As Boolean
Global scalefactor As Double
Global Prime(1 To 170), expo(1 To 170), work(1 To 170) As Integer
Public Sub transform_task()
Dim str1, str2, str3 As String
Dim j, k As Integer
Dim p, c, pnew, cnew As Double
If Transformaction = True And Transformindex >= 0 And Transformindex <= Form1.List1.ListCount - 1 Then
    str1 = Form1.List1.List(Transformindex)
    j = InStr(1, str1, "/", 0)
    str2 = Mid(str1, 1, (j - 1))
   p = CDbl(str2)
    k = InStr(j + 1, str1, "/", 0)
    j = InStr(1, str1, "/", 0)
    str3 = Mid(str1, (j + 1), (k - j - 1))
    c = CDbl(str3)
    pnew = p * scalefactor
    cnew = c * scalefactor
    Form1.List1.RemoveItem Transformindex
    Form1.List1.AddItem pnew & "/" & cnew & "/  ->  " & "   Period = " & pnew & "  Computation = " & cnew
End If
End Sub
```

6.2 Prime Number Table

The file "Prime.txt contains a list of first 170 prime numbers used in the calculation of schedule cycle length:

2	71	167	271
3	73	173	277
5	79	179	281
7	83	181	283
11	89	191	293
13	97	193	307
17	101	197	311
19	103	199	313
23	107	211	317
29	109	223	331
31	113	227	337
37	127	229	347
41	131	233	349
43	137	239	353
47	139	241	359
53	149	251	367
59	151	257	373
61	157	263	379
67	163	269	383

389	547	691	859
397	557	701	863
401	563	709	877
409	569	719	881
419	571	727	883
421	577	733	887
431	587	739	907
433	593	743	911
439	599	751	919
443	601	757	929
449	607	761	937
457	613	769	941
461	617	773	947
463	619	787	953
467	631	797	967
479	641	809	971
487	643	811	977
491	647	821	983
499	653	823	991
503	659	827	997
509	661	829	1009
521	673	839	1013
523	677	853	
541	683	857	

7.0 Guide for Application Programmers

In this section, we will describe the actual use of the kernel and applications. There are several issues that need to be kept in mind while writing applications. They are detailed in the following sections.

7.1 Extracting Task Parameters: Method 1

The first question that comes to mind when integrating an appset with the kernel is regarding the extraction of task parameters $\{C_i, T_i, P_i\}$ describing an application in an appset. Given an application program, there should be a way of arriving at the corresponding task parameter set $\{C_i, T_i, P_i\}$. Since we use only a simplified parameter set with only three parameters - out of which only two parameters are independent. The priority parameter is chosen automatically by the Rate-Monotonic assignment within the appset. Hence for each application we need to find only two parameters, computation time, and the period, T_i.

Out of these two, the period T_i is mainly decided by the nature of the application. If the application happens to be, for example, a MPEG decoder, then the value of the period is very likely to be 33 milliseconds corresponding to 30 frames/sec video output expected from the decoder. Similarly for an audio application, a different audio frame period is useful to denote the action of the application. The same is true for a telecommunication application which is expected to maintain a constant rate of input and output data. Hence finding out about the period is an easy job. It is almost given from the definition of an application.

Deriving the second parameter C_i from an application program is far from trivial. In general, an application can be broken down as interconnection of *Modules*[24], where each module is a small program unit with a measurable execution time. The execution time of a module could be its worst case execution time or its actual execution time, if known. The application designer is expected to have good understanding of the execution time of each module in the application. Then the application program can be described as a *Task Flow Graph*[25] of its component modules. The task flow graph is composed of four types of subgraphs: *Chain* subgraph, *AND*-subgraph, *OR*-subgraph and *Loop*-subgraph. Because of this macro structure of a multimedia application, it is possible to derive the computation time C_i from the module execution time and the task flow graph. However, there are data dependent branches and loops along with intermodule communication delays and double buffering schemes that foil simple schemes for calculating the net C_i from the component module values. Either probabilistic branch and loop models are to be incorporated or good engineering approximations must be carried out to simplify the problem. Currently this very topic has become of interest to researchers and a good deal of public domain literature exits[26]. Fortunately, there always exists a loop-subgraph that corresponds to the fundamental period of the application. Within this *fundamental loop*, one can concentrate on the modules that get activated for every fundamental period and derive an upper bound of C_i. If the application is written from scratch, then division into modules and tracking them inside loops and branches become feasible. Since the application designer is forced to provide the necessary performance goals for the entire application, the designer is also forced to make measurements at the module level as well as the application level. Hence this method of *micro structure modeling* of the application program becomes relevant both in the design phase as well as the final encapsulation phase into an appset.

7.2 Extracting Task Parameters: Method 2

The other approach in estimating the computation time is simple and straightforward. It does not attempt to invoke micro structure modeling, but a *macro structure measurement* model. If is the fundamental period of an application program, then the computation time C_i must be very much smaller than T_i, i.e., $C_i << T_i$. If this were not true, then multitasking would be impossible as all of the system resources are required just to manage a single application.

Let us assume that $C_i << T_i$. If we can remove the timing constraint of the application to deliver the output at every fundamental period of T_i milliseconds by disabling all timing related mechanisms such as time stamps, interrupts etc., and allow the application to run in the *Single-Application-free-running* mode - possibly by providing special arrangements for carefully chosen test data set that would stress the system - then the application would have a maximum frame rate of instead of the design goal of . Since there is only one application running at the target platform, there is no interference from other applications. Some special precautions also must be taken to ensure that the

---

[24] D.T.Peng and K.G.Shin, "Modelling Concurrent Task Execution in a distributed system for Real-Time Control", IEEE Transactions on Computer, vol 36, no. 4, pp 500-516, April 1987.

[25] C.J.Hou and K.G.Shin, "Module Allocation with Timing and Precedence Constraints in Distributed Real-Time Systems", IEEE Proceedings of 13th Real-Time Systems Symposium, pp. 146-155, December, 1992.

[26] Chao-Ju Hou and Kang G. Shin, " Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems", IEEE Transactions on Computers, vol. 46, No. 12, December 1997.

application behaves exactly as before only at a greater frame rate of $f_{\max}$. In most cases, this involves very few modifications such as directing the input and output signals to different files instead of ports. In fact, such a testing procedure is routinely done for MPEG decoders and other related communication software. The power of this approach lies in the fact that (1) it is easy to conduct the test, and (2) it does not require complex modeling of the application which becomes tricky with buffering mechanisms, branch/loops, communication delays and data dependencies within modules. All that is required is a small modification in the application and a carefully chosen test data file which is long enough to give statistically significant accuracy in estimating the maximum frame rate $f_{\max}$. Also note that this method makes no assumptions about the internal structure of the application and hence is likely to be robust. Once $f_{\max}$ is estimated then on the assumption of global linearity, the estimate of C_i is

. There may be only one problem with this estimate. Since we are measuring the quantity $f_{\max}$ as an average over a long interval, local disturbances due to time-varying nature of compression products are averaged out. Hence $\hat{C}_i$ tends to be an average value rather than a maximum.

This can be simply corrected by an ad-hoc factor of safety such as

where .

Though the method of direct measurement is preferred over the first method of micro structure modeling due to our ignorance of the actual system functions, it is possible to combine both these approaches: One can measure $\hat{C}_i$ by actual test set up and improve the measurement by modeling the fundamental loop carefully to account for peak excursions.

7.3 Generating Appsets

Let us say, we have *n-1* applications with parameter sets obtained from models or direct measurements described in sections 7.1 and 7.2. We will add a house-keeping task with as the *n*th application to be included in the appset. Given the task set of size *n*, the generation of the appset follows the algorithm given below:

1. Choose a time-slice duration, $T_{Time-Slice}$ in milliseconds.
2. Compute for $i = 1, 2, 3 \ldots n$.
3. Compute for $i = 1, 2, 3, \ldots n$.

4. Compute $t(0) = \sum_{i=1}^{n} Q[C_i]$
5. Update $t(i) = \sum_{i=1}^{n} Q[C_i] \cdot \left\lceil \frac{t(i-1)}{Q[T_i]} \right\rceil$ for $i = 1,2,3\cdots$
6. if $t(i+1) = t(i)$ and $t(i+1) \le Q[T_n]$,

   then the task set is schedulable. Go to Step 8.

   Otherwise go to step 7.
7. Find new solutions by

   (a) Use Period Transformation to scale the periods, i.e., find a scalar value *a*, such that

   for *i* = 1, 2, 3 ...*n,* and

   for *i* = 1, 2, 3 ...*n.* Then go back to step 4. Or

   (b) Use a new value for $T_{Time-Slice}$ and go to step 2.
8. Express $Q[T_i] = p_1^{e_1(i)} \cdot p_2^{e_2(i)} \cdot p_3^{e_3(i)} \cdots p_M^{e_M(i)}$ for $i = 1,2,3,\cdots n$, where $p_1, p_2, p_3, \cdots p_M$ are the first *M* prime numbers. Find *L*: $L = p_1^{\max_{i=1,2,3,\cdots n}\{e_1(i)\}} \cdot p_2^{\max_{i=1,2,3,\cdots n}\{e_2(i)\}} \cdot p_3^{\max_{i=1,2,3,\cdots n}\{e_3(i)\}} \cdots p_M^{\max_{i=1,2,3,\cdots n}\{e_M(i)\}}$.
9. If $L <$ 32000, then go to step 10. Otherwise go to step 7.
10. Update the time-slice by taking into account of the overheads: First save . Then Update

    .

    If or the number of times the iteration was performed was large, then stop: We have found the solution. Otherwise go to step 1.

The algorithm attempts to find a schedule cycle length less than 32000 so that we use a 16-bit number to represent the schedule cycle. It also takes into account of the overhead costs of task switching. Once a solution is arrived at, the appset can be generated by including the correct number of applications that are guaranteed to meet the real-time deadlines. The design tool described in section 6.0 is very useful in this process.

The actual appset is defined by filling the array entries below and storing it in DRAM:

```
Struct APP_SET
{
        Char [ 32] Appset Name;
        Char [ 32] Description;
        Char [ 8] Version;
        Char [ 8] Date of Release;
        Char [ 32] Author;
        Integer APPSET_ID;
```

```
		Integer Number_of_apps;
		Integer Schedule_Cycle_Length;
		Integer Number_Of_House_Keeping_Tasks;
		Integer House_Keeping_ID;			// Is it -1 or something else?
		Integer Utilization;
		Integer   *Schedule_Cycle_Pointer;		// Where Schedule_Cycle_Table is kept
		Integer Time-Slice;		// Time-Slice in milliseconds
		APP   *apps[n];		// Pointers to apps structure
	}
```

APP structure actually knows where applications are residing:

```
	Struct APP
	{
		Char [ 32] Application Name;
		Char [ 32] Author;
		Char [ 8] Release Date;
		Char [ 8] Version;
		Integer APP-ID;		// Same as Task-ID
		Integer Task_Period;
		Integer Computation;
		Integer Priority;

		Integer   * App_Text_Address;	// Address Pointer to Code Text
		Integer   App_Text_Length;	// Size of Text
		Integer   * App_Data_Address;	// Address Pointer to
						//Data required by
						//the Application

		Integer   App_Data_Length;	// Size of Data
		Integer   * App_Stack_Address;	// Pointer to Application
						//Stack maintained by the
						//   Application

		Integer   App_Stack_Size; // Size of Application
					//Stack required by
					//the Application

		Integer   * App_Status_Address;	// Pointer to Status section

		Integer   App_Status_Size;// Size of Status
		Integer   *App_Message_Area;
		Integer   App_Message_Area_Size;
	} apps[n];
```

7.4 Merging ISRs

Sections 5.11 and 5.12 describe the interrupt service routines expected by the system. For each application, application programmer is expected to supply the relevant set of ISRs intended for that application. When appset is formed, The corresponding ISRs for the applications are merged into a single ISR while preserving the priority structure. The table in section 5.11 explains the list of interrupts and their priorities. For example, an application with a ISR for DSE 1 DONE interrupt will be placed along with other ISRs having the priority 7.

7.5 MTE Transfers: Polling

MTE transfers can be done in two different ways: using a polling strategy or by using interrupts. From the real-time performance point of view, it is preferable not to have interrupts. In that circumstance, it is necessary to use a polling procedure.

The idea in this method is to write to a well known address after the transfer is finished. When PE is ready to query the result of data transfer, it reads the location and finds out if indeed the data transfer has taken place. To facilitate the polling, we need write additional information about which task issued the command and what is the command queue number.

Suppose that we need to write a block of data to DRAM. Then the following parameter block can be used:

```
Struct MTC_WRITE
        {
                int     MTC_Command1;  //  MTC Parameter Block Command Field
                                       //  bits 24-31; bit 31 = 1 to continue after this block
                                       //bits 24-25 set to 00 signifying no interrupt
                int     Line_Count1;   // bits 12-23
                int     Byte_Count1;   // bits 0-11
                int     Destination_Address;     // where the data must be moved
                int     Source_Address1;                  // from where
                int     Destination_Delta1;      // increment value
                int     Source_Delta1;           //increment value

                int     MTC_Command2;            // MTC command for result block
                                                 // bits 24-31; bit 31 = 0
                                                 // bits 24-25 set to 00.
                int     Line_Count2;             // equal to zero
                int     Byte_Count2;             // equal to 8
                int     Result_Address;          // Where the result of action must be written
                int     Source_Address2;         // where the result block is stored
                                                 // before transfer
                Int     Task-ID;       // Which task issued the request
                Int     Write_Sequence_Number; // When did it issue the request
                int     Destination_Delta2;      // equal to 1
                int     Source_Delta2;           // equal to 1
        }
```

The above data structure would be converted as two consecutive MTC parameter blocks: the first one would contain the actual details of the block of data to be written. The second one will be the one containing Task-ID and Write Sequence Number for the write command. The MTC parameter blocks can be then:

```
Struct MTC_Block
        {
                int     word1;
                int     word2;
                int     word3;
                int     word4;
                int     word5;
                int     word6;
                int     word7;
                int     word8;
        }
```

where

```
MTC_Block.word1 = (MTC_WRITE.Byte_Count1 & 0x00000FFF) | (MTC_WRITE.Line_Count1&0x00000FFF)<<12 | (MTC_WRITE.Command1 & 0x000000FF)<<24 ;

MTC_Block.word2 = MTC_WRITE.Destination_Address;
MTC_Block.word3 = MTC_WRITE.Source_Address1;
```

```
MTC_Block.word4 = (MTC_WRITE.Source_Delta1 & 0x0000FF) | (MTC_WRITE.Destination_Delta1 &0x0000FFFF)<<16;

MTC_Block.word5 = (MTC_WRITE.Byte_Count2 & 0x00000FFF) | (MTC_WRITE.Line_Count2 &0x00000FFF)<<12 | (MTC_WRITE.Command2 & 0x000000FF)<<24 ;

MTC_Block.word6 = MTC_WRITE.Result_Address;
MTC_Block.word7 = MTC_WRITE.Source_Address2;
MTC_Block.word8 = (MTC_WRITE.Source_Delta2 & 0x0000FF) | (MTC_WRITE.Destination_Delta2 &0x0000FFFF)<<16;
```

When the transfer is over, the Task-ID and Sequence Number will be written at the Result_Address. The PE can read this address and infer if the transfer indeed has taken place or not. PE can keep polling the address or check it during the next task period.

7.6 MTE Transfers: Interrupts

MTE transfer with interrupt activation is similar to description in the section 7.5 but with minor modifications. The command word bits 24-25 are set to 01 so that an interrupt is generated. Instead of writing the result to a memory location, the result can be made explicit by reading a semaphore register. For each task, a unique semaphore register can be assigned. A semaphore register provides a test function for resource allocation.

If a semaphore register reads zero, then the resource that it corresponds to, namely MTC, is available for reading or writing. The semaphore register is set to one immediately after it is read. The MTC can be programmed to do the following sequence: (1) perform read or write and (2) after transfer, write into the semaphore register. When a semaphore is written, it is set to zero and generates an interrupt. PE can check the semaphore register for zero and infer that transfer has taken place.

7.7 Error Handling

It is the responsibility of the application to handle all run time errors and exceptions. The default action of the kernel when nothing is specified is to halt the processing.

7.8 Memory Protection

Currently no memory protection schemes are suggested. The application programmer is expected to watch out for the applications not entering the kernel space. In future, another task can be spun off to do this function.

7.9 Style Guide for Application Programmers

Because of the special requirements of CLA architecture, certain care must be taken to integrate application programs with the kernel software. The following constitute a set of guidelines that application programmers must adhere to:

1. Take care of all interrupt services and the kernel does not provide any service with respect to application handling. The main task of the kernel is to do effective multitasking within the constraints of CLA system. Kernel is oblivious to any interrupt request other than (1) time-slice clock interrupt and (2) GWI .
2. The ISRs written by the application programmers must be merged to form a single monolithic ISR as described in the section 5.11 and 5.12.

3. All MTE transfers must be handled by the application. It can be done using either interrupts or by message passing. See the discussion in section 5.2. We will only amplify the method of using interrupts here:
    (a) Allocate statically a different semaphore register to different tasks
    (b) Tasks are switched on AMTC and SMTC depending on the phase of the schedule cycle. Any commands placed on MTCs must write to the corresponding semaphore register (of the task at hand) before generating interrupts. Then the semaphore registers can be used as a means to perform interrupt filtering.
    (c) Track the MTE transfers at the beginning of each time-slice to see if the transfer is done.
4. Take care of all numerical exceptions and error handling. The kernel provides no services in this context.
5. Use the schedule design tool and manual measurements of critical loops to arrive at a sensible schedule cycle.
6. Use worst case values as task parameters in case of ignorance. This can be improved in the next iteration or by actual measurements
7. Take care not write into kernel area in DRAM as well as in CLA Local Memory.

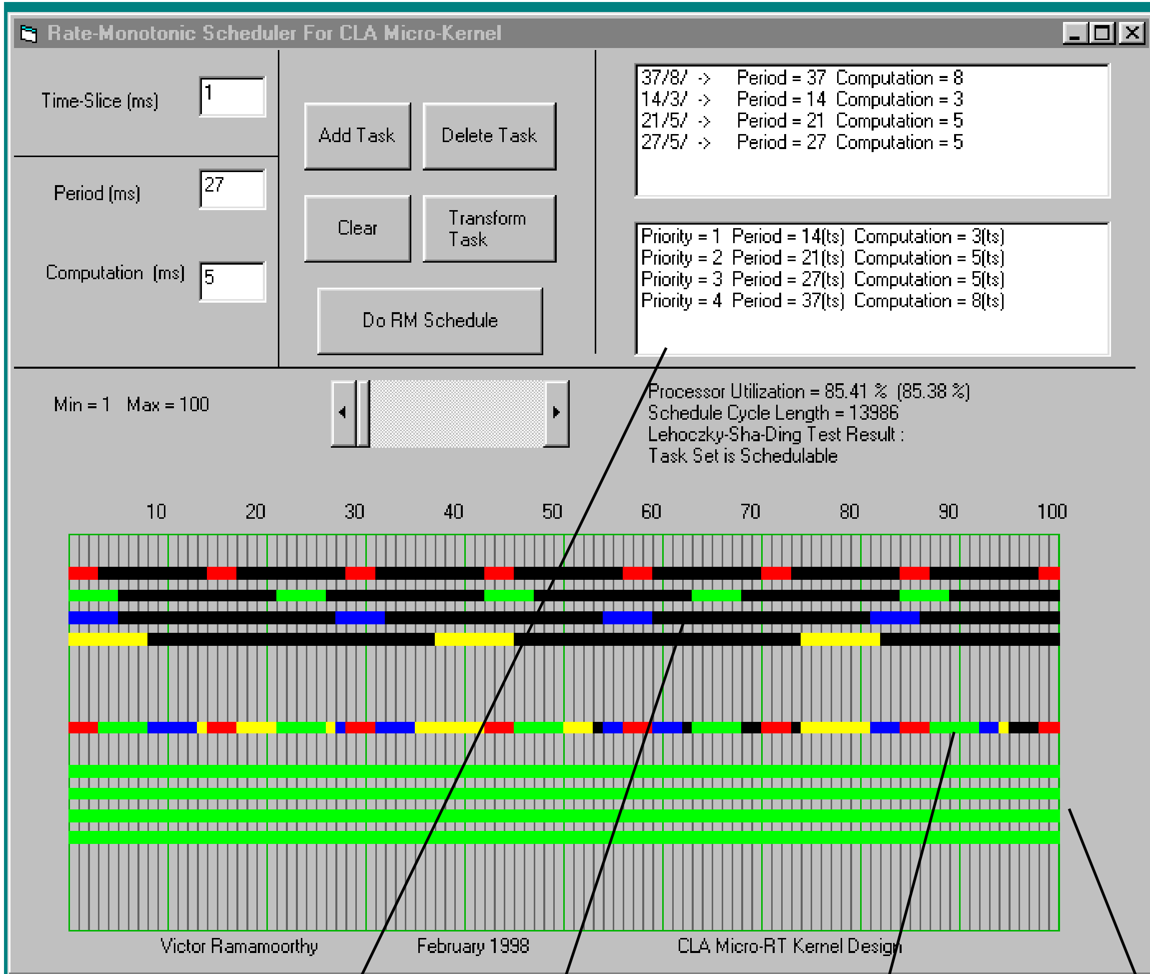

Fig 6. A view of schedule design tool

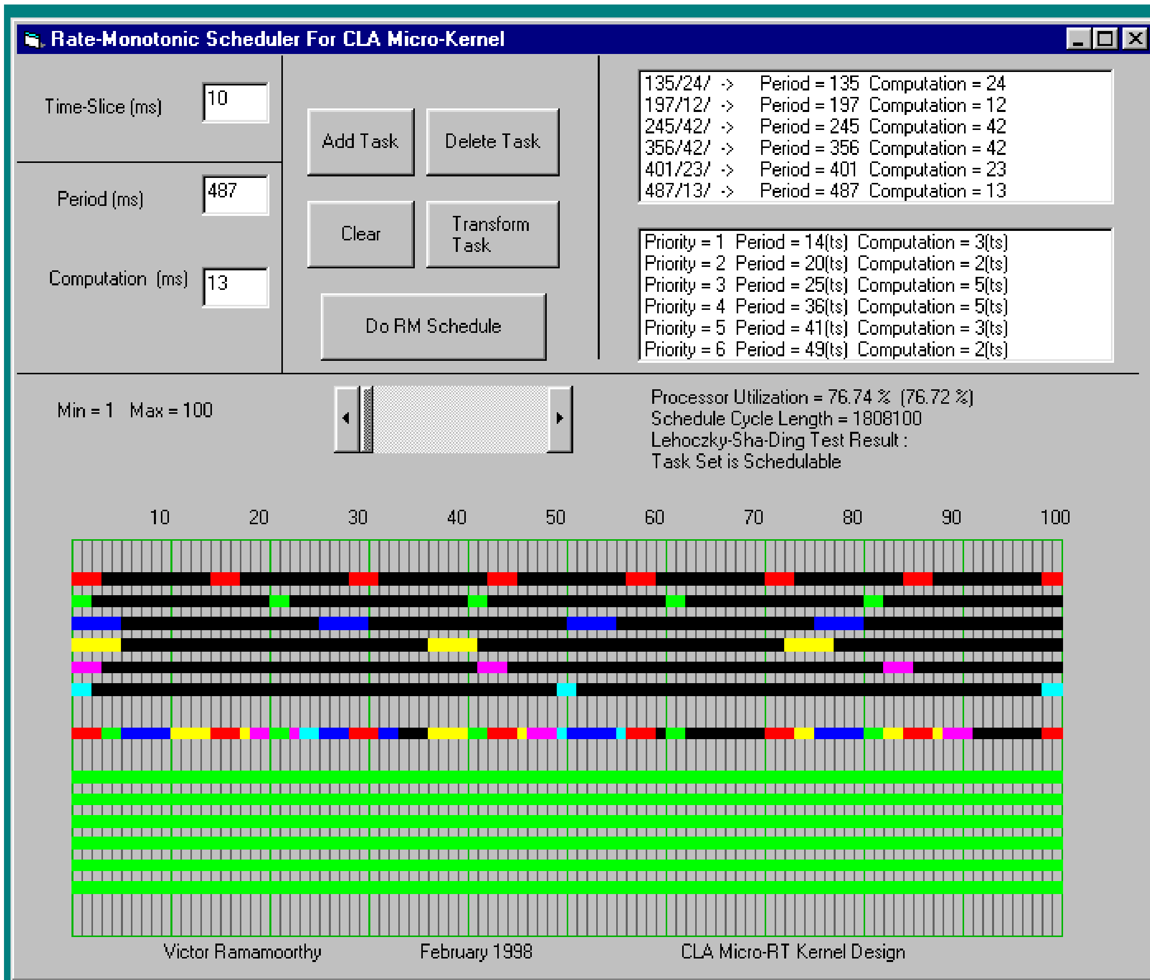

Fig. 7 Another View of the tool.