International Journal of Innovative Research in Advanced Engineering (IJIRAE) ISSN: 2349-2163
Volume 1 Issue 10 (November 2014) www.ijirae.com

Implementation of Timing Specifications in Real-Time System

Ramya S
Department of Electronics & Telecommunication Engineering,
Bharath University

Abstract— A real-time embedded system generally has several periodic processes. The scheduling of these processes should be done using a real-time language that can make it easy to specify the timing requirements. Currently, this feature is not available in any of the established languages. This paper intends to propose an approach to implement timing specifications using the cyclic executive model in Ada programming language.

Keywords— Real-time system, timing specifications, scheduling, cyclic executive, Ada programming language

I. INTRODUCTION

A real-time system is one in which deadlines are very critical and hence scheduling plays an important part in the design of any real-time system. A real-time embedded system generally has several periodic processes. The scheduling of these processes should be done by a real-time language that can make it easy to specify the timing requirements. This aspect is currently not available in any of the established real-time languages [1]. There are several general-purpose scheduling tools such as AutoSys and TWS (Tivoli Workload Scheduler), which facilitate the specification of timing requirements for general-purpose systems. This paper aims to implement several timing specifications in a real-time language using the cyclic executive approach combined with table-driven scheduling concept.

The timing specifications that are proposed to be implemented are specifying the periodic scheduling of a task, the absolute time at which a given task is to begin execution, the maximum run time to be allocated to a particular task and the duration between two periodic tasks. If the system is unable to meet one or more of these requirements, it should raise an exception. Exception handlers can be written to take care of the expected errors and switch to a backup task. This project will be implemented using the Cyclic Executive model. A cyclic executive is a control structure or program for explicitly interleaving the execution of several periodic processes on a single CPU [2]. The interleaving is done in a deterministic manner so that execution timing is predictable. The process interleaving is defined by means of a cyclic schedule, which specifies an interleaving of actions that will enable processes to execute within their periods and deadlines. This project is proposed to be implemented using Ada (2012) programming language in GPS 2014.

II. REAL-TIME TASKS

Based on the way real-time tasks recur over a period of time, it is possible to classify them into three main categories: periodic, sporadic, and aperiodic tasks [3].

A periodic task is one that repeats after a certain fixed time interval. The precise time instants at which periodic tasks recur are usually distinguished by clock interrupts. Periodic tasks are therefore also referred to as clock-driven tasks. The fixed time interval after which a task repeats itself is called the period of the task. If T_i is a periodic task, then the time from 0 till the occurrence of the first instance of T_i (i.e. $T_i(1)$) is denoted by φ_i, and is called the phase of the task. The second instance (i.e. $T_i(2)$) occurs at $\varphi_i + p_i$. The third instance (i.e. $T_i(3)$) occurs at $\varphi_i + 2 * p_i$ and so on. Formally, a periodic task T_i can be represented by a 4 tuple $(\varphi_i, p_i, e_i, d_i)$ where p_i is the period of task, e_i is the worst case execution time of the task, and d_i is the relative deadline of the task. Most of the tasks carried out by a typical real-time system are periodic in nature.

A sporadic task is one that recurs at random instants. A sporadic task Ti can be is represented by a three tuple $T_i = (e_i, g_i, d_i)$ where e_i is the worst case execution time of an instance of the task, g_i denotes the minimum separation between two consecutive instances of the task, d_i is the relative deadline. The minimum separation (g_i) between two consecutive instances of the task implies that once an instance of a sporadic task occurs, the next instance cannot occur before g_i time units have elapsed. That is, g_i restricts the rate at which sporadic tasks can arise. The first instance of a sporadic task T_i is denoted by $T_i(1)$ and the successive instances by $T_i(2)$, $T_i(3)$, etc, as in the case of periodic tasks. Several sporadic tasks such as emergency message arrivals are highly critical in nature.

An aperiodic task is similar to a sporadic task in many ways. An aperiodic task can arise at random instants. However, in case of an aperiodic task, the minimum separation g_i between two consecutive instances can be 0. This means that two or more instances of an aperiodic task might occur at the same time instant. Also, the deadline for an aperiodic task is expressed as either an average value or is expressed statistically. Aperiodic tasks are generally soft real-time tasks.

Aperiodic tasks can recur in quick succession. It therefore becomes very difficult to meet the deadlines of all instances of an aperiodic task. When several aperiodic tasks recur in a quick succession, there is a bunching of the task instances and it might lead to a few deadline misses.

III. Scheduling Techniques

There exist several schemes of classification of real-time task scheduling algorithms. A popular scheme classifies the real-time task scheduling algorithms based on how the scheduling points are defined. The three main types of schedulers according to this classification scheme are clock-driven, event-driven, and hybrid, as shown in figure 1. The clock-driven schedulers are those in which the scheduling points are determined by the interrupts received from a clock. In the event-driven ones, the scheduling points are defined by certain events which precludes clock interrupts. The hybrid ones use both clock interrupts as well as event occurrences to define their scheduling points. In this paper, clock-driven scheduling algorithm will be considered for implementation of timing specifications in real-time system.

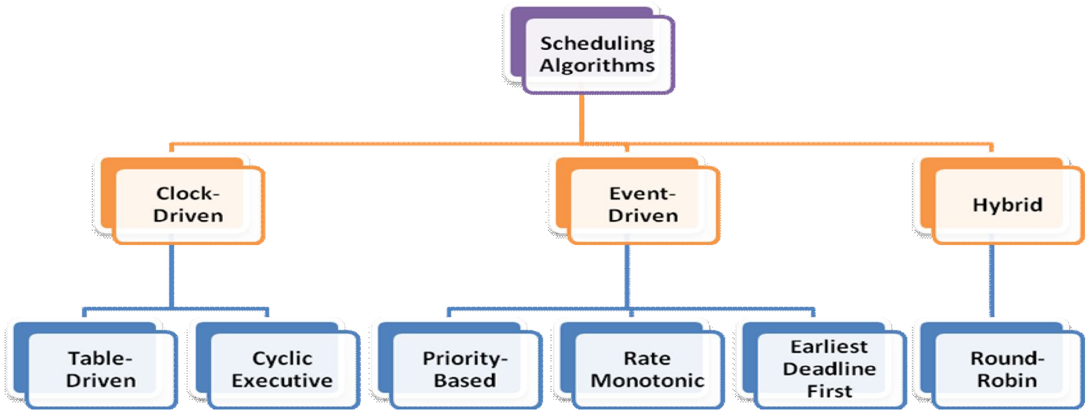

Fig. 1 Classification of scheduling algorithms

Clock-driven schedulers make their scheduling decisions regarding which task to run next only at the clock interrupt points. Clock-driven schedulers are those for which the scheduling points are determined by timer interrupts. Clock-driven schedulers are also known as offline schedulers because the schedule is fixed before the system starts to run. In other words, the scheduler pre-determines which task will run when. These schedulers therefore incur very little run time overhead. Nonetheless, a major limitation of this class of schedulers is that they cannot adequately handle aperiodic and sporadic tasks given that the exact time of occurrence of these tasks cannot be predicted. This type of schedulers is consequently also called static schedulers. Clock-driven schedulers can be classified as table-driven and cyclic schedulers.

Table-driven schedulers usually pre-compute which task would run when, and store this schedule in a table at the time the system is designed or configured. Instead of the scheduler computing the scheduling information automatically, the application programmer is given the liberty to select a user-defined schedule for the set of tasks in the application and store the schedule in a table (called schedule table) to be used by the scheduler at run time.

Cyclic schedulers are extremely popular and are used extensively in the industry. A large majority of all small embedded applications being manufactured presently are based on cyclic schedulers. Cyclic schedulers are simple, efficient, and are easy to program. An example application where a cyclic scheduler is normally used is a temperature controller. A cyclic scheduler repeats a pre-computed schedule. The pre-computed schedule needs to be stored only for one major cycle. Each task in the task set to be scheduled repeats identically in every major cycle. The major cycle is divided into one or more minor cycles. Each minor cycle is also referred to as a frame. The scheduling points of a cyclic scheduler occur at frame boundaries. This means that a task can start executing only at the beginning of a frame.

Both table-driven and cyclic schedulers are important clock-driven schedulers. A scheduler needs to set a periodic timer only once at the time of application initialization. This timer continues to give an interrupt exactly at every frame boundary. But in table-driven scheduling, a timer has to be set every time a task starts to run. The execution time of a typical real-time task is usually of the order of a few milliseconds. Therefore, a call to a timer is made every few mill Seconds. This represents a significant overhead and results in degraded system performance. Therefore, a cyclic scheduler is more efficient than a table-driven scheduler. This is a reason probably why cyclic schedulers are so overwhelmingly popular especially in embedded applications. However, if the overhead of setting a timer can be ignored, a table-driven scheduler is more proficient than a cyclic scheduler because the size of the frame that needs to be chosen should be at least as long as the size of the largest execution time of a task in the task set. This is a source of inefficiency, since this results in processor time being wasted in case of those tasks whose execution times are smaller than the chosen frame size.

IV. GENERAL-PURPOSE SCHEDULING TOOLS

The most common scheduling tools used for general-purpose jobs are AutoSys and TWS (Tivoli Workload Scheduler). A brief study of these tools will give an idea of the features that can be implemented in real-time systems using Ada programming language. AutoSys operates on Unix/Linux, Windows and AIX platforms while TWS operates on Unix/ Linux, Windows, IBM iSeries, IBM zOS [4].

An AutoSys system consists of an event server, event processor and remote agent. The event server is a database that contains all the information pertaining to the AutoSys job schedule. The event processor is a process that polls the database for information, evaluates job dependencies to determine when and if jobs should run, and communicates with the Remote Agent on the AutoSys client to start running the job. A combination of an Event Server and one or more Event Processors is called an Instance. A set of related jobs is grouped into a box. The scheduling information is specified both at the box level as well as the job level. There are several timing specifications that can be mentioned using AutoSys scheduler. The basic attributes defined in an AutoSys job are job type, frequency, start time, maximum allowed execution time, command to be executed and the predecessor job.

TWS is scheduler in which a set of associated jobs is called a job stream or schedule. The scheduling features are similar to AutoSys, except that TWS operates on a day by day basis. The schedules to be executed are loaded for every production day. Any job which does not execute as per the plan gets carried forward to the next day's plan. In contrast, AutoSys will have only one version of the job. If the job does not get completed for the day, a new version does not get loaded. TWS allows the user to specify several timing requirements while defining a schedule. The absolute start time of the job or schedule can be mentioned in the time zone of the TWS instance. The maximum run time allowed for a job can be set so that an alert is sent to the user notifying about the long run of the job. Also, it is possible to set a deadline for a job in order to alert the user if the job has not completed its run within a stipulated time.

V. ADA PROGRAMMING LANGUAGE

Ada is a modern programming language designed for large, long-lived applications – and embedded systems in particular – where safety and security are critical. Ada is seeing significant usage worldwide in high-integrity / safety-critical / high-security domains including commercial and military aircraft avionics, rail transport systems, air traffic monitoring and control, and medical devices [5]. The most prominent features of Ada include strong typing, modularity mechanisms (packages), run-time checking, parallel processing (tasks, synchronous message passing, non-deterministic select statements, and protected objects), exception handling, and generics [6]. Ada's tasking features allows the user to express common real-time idioms such as clock-driven or periodic tasks and event-driven tasks, and the Real-Time Annex provides a number of facilities that allows avoiding of unbounded priority inversions. Apart from these, a protected object locking policy is defined that uses priority ceilings. This feature, in particular, has an efficient implementation in Ada, where mutexes are not required, in view of the fact that protected operations are not allowed to block any task. Ada 95 defined a task dispatching policy that essentially requires tasks to run until blocked or preempted, while Ada 2005 introduced several others such as the Earliest Deadline First scheduling policy. Ada 2012 has introduced furthermore powerful assertion mechanisms and predicates in the language.

Ada is a programming language that is suitable for all kinds of development requirements. It has built-in features that directly provide support to structured, generic, object-oriented, distributed and concurrent programming. Ada lays distinctive emphasis on good software engineering practices that scale really well to very large software systems (millions of lines of code, and very large development teams). In addition to offering support for good software development practices, which are applicable to general-purpose programming, Ada has a powerful set of specialized features supporting low-level programming for real-time and safety-critical embedded systems. These features include machine code insertions, low-level access to memory, address arithmetic, bit manipulations, control over bitwise representation of data, and a well-defined, statically provable concurrent computing model called the Ravenscar Profile.

Ada is designed to support the construction of long-lived, highly reliable software systems [7]. This programming language includes several provisions to define packages of related types, objects, and operations. The packages may be parameterized and the types may be extended to support the construction of libraries of reusable, adaptable software components. The operations may be implemented as subprograms using the conventional sequential control structures, or as entries that include synchronization of concurrent threads of control as part of their invocation. Ada supports object-oriented programming by providing classes and interfaces, inheritance, polymorphism of variables and methods, and generic units. The language treats modularity in the physical sense as well, with a facility to support separate compilation. The language provides extensive support for real-time, concurrent programming, and includes facilities for multi-core and multi-processor programming. Ada allows signaling of errors as exceptions, which can be handled explicitly. The language also covers systems programming, which requires precise control over the representation of data and access to system-dependent properties. Also, a predefined environment of standard packages is provided, including facilities for string manipulation, input-output, numeric elementary functions, random number generation, and definition and use of containers.

Ada defines a set of pragmas that can be used to supply additional information to the compiler. These language defined pragmas are implemented in GNAT and work as described in the Ada Reference Manual. GNAT is a free, high-quality, complete compiler for Ada, integrated into the GCC compiler system. Ada also allows implementations to define additional pragmas whose meaning is defined by the implementation. GNAT provides a number of these implementation-defined pragmas, which can be used to extend and enhance the functionality of the compiler. Three steps are needed to create an executable file from an Ada source file through GNAT [8]. First, the source file(s) must be compiled. Then the file(s) must be bound using the GNAT binder. Finally, all appropriate object files must be linked to produce an executable. All three steps are most commonly handled by using the gnatmake utility program that, given the name of the main program, automatically performs the necessary compilation, binding and linking steps. Although the command line interface such as gnatmake alone is sufficient, a graphical Interactive Development Environment (IDE) can make it easier for the user to compose, navigate, and debug programs. GPS (GNAT Programming Studio), the GNAT graphical IDE, is a powerful and simple-to-use IDE that streamlines the software development process from the initial coding stage through testing, debugging, system integration, and maintenance. GPS invokes the GNAT compilation tools using information contained in a project (also known as a project file) - a collection of properties such as source directories, identities of main subprograms, tool switches, etc., and their associated values.

VI. Design Approach

This paper intends to propose a design approach for the implementation of timing specifications using a real-time language in a generic manner that can be employed for any real-time embedded application. The real-time language chosen for this implementation is Ada (2012) because of its extensive features as discussed in the previous section. This paper considers three tasks for the program flow. There is an Ada program written corresponding to each task, namely, *task_1.adb*, *task_2.adb* and *task_3.adb*. The scheduling information for these tasks is coded in *scheduler.adb*. All these individual programs are linked together in a single project named *Real_Time_Scheduler.gpr*. The programs for each of the tasks can have an implementation of any functionality such as printing a message or doing a particular computation. The tasks are scheduled using the cyclic executive model combined with table-driven approach. The scheduling details are stored in a table, known as the schedule table, which is created during the development phase. This is a one-time activity so that scheduling information remains constant throughout the lifecycle of the application. The timing specifications such as maximum run time, absolute execution start time, relative execution start time and duration between the execution of two tasks is stored in the schedule table. A project named *Real_Time_Scheduler.gpr* is created which contains the src, obj and include folders. The source programs *scheduler.adb*, *task_1.adb*, *task_2.adb* and *task_3.adb* are created in the src folder. The *task_n* programs have the source code for the respective functionality to be performed. The scheduler.adb program contains the logic to schedule the three tasks in a cyclic manner by querying the schedule table, which contains the timing specifications of each of the three tasks. The semantic is checked to create the ALI files, after which the source programs are compiled and the object files are created in the obj folder. The project is then built to create the executable application *schedule*. This application can be launched directly from the obj folder to view the output of the scheduler project.

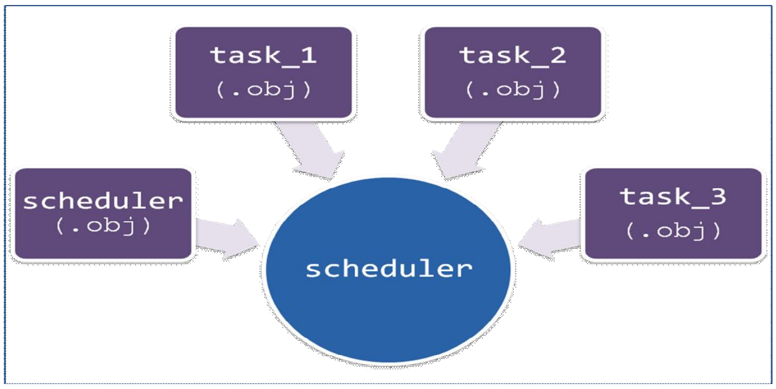

Fig. 2 Real time scheduler project in Ada

VII. Conclusion

The most vital requirement of any real-time system is the ability to meet deadlines. In order to achieve this goal, it is imperative that there exist a mechanism to implement timing specifications in a real-time language through an efficient and optimized approach. This paper discussed the prospect of making use of the cyclic executive model using Ada programming language. The features implemented can be found in general-purpose scheduling tools and have been simulated in a similar manner using Ada programs for a real-time system. Future implementations of this project may consider making the scheduler generic so that any number of tasks can be scheduled by the Ada project.

ACKNOWLEDGMENT

The author wishes to thank Ms. Rachelin Sujae, Department of Electronics and Telecommunication Engineering, Bharath University for her guidance and support towards this work.

REFERENCES

[1] C. M. Krishna, Kang G. Shin, *Real-Time Systems*, The McGraw-Hill companies, Inc., New York, 2010.
[2] T. P. Baker, Alay Shaw, "The cyclic executive model and Ada", IEEE Real-Time Systems Symposium, 1988, Proceedings.
[3] "Real-Time Task Scheduling – Part 1". [Online]. Available: http://nptel.ac.in/courses/108105057/Pdf/Lesson-29.pdf
[4] "List of job scheduler software". [Online]. Available: http://en.wikipedia.org/wiki/List_of_job_scheduler_software#Job_scheduling_products_without_ERP_support
[5] "The Ada Programming Language". [Online]. Available: http://www.adacore.com/adaanswers/about/ada/
[6] Clifton A. Ericson, II, *Concise Encyclopedia of System Safety: Definition of Terms and Concepts*, John Wiley & Sons, Inc.. Hoboken, New Jersey, 2011.
[7] "Ada Reference Manual". [Online]. Available: http://www.ada-auth.org/standards/12rm/html/RM-TTL.html
[8] "GNAT User's Guide". [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-3.4.4/gnat_ugn_unw.pdf