ISSN: 2277-9655
Impact Factor: 1.852

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

# AUTOSAR Multicore Operating System Implementation for MPC5668G

Devika K[*1], Syama R[2], Anurag R[3]
[*1,2] Department of Computer Science &Engineering, Sree Chitra Thirunal College of Engineering, Trivandrum, India
[3]Competency Manager, Tata Elxsi Ltd, Technopark, Trivandrum, India
k_devu@yahoo.co.in

## Abstract

The advanced features in hardware as well as in software field reflected in embedded system domain also. Microcontrollers used in automobiles are turned to multicore for supporting parallel execution. For efficient utilization of multicore microcontrollers it is necessary to design operating systems in order to assist features provided by hardware. Multicore operating systems shall be capable to handle each independent processing unit inside multicore microcontroller. Designing operating systems for multicore processors is very crucial because the improvement in performance depends strictly on the software implementation. AUTOSAR (Automotive Open System Architecture) is a standard that provide common platform for automotive applications. It also increases interoperability and interchangeability of software developed by different users. AUTOSAR provides guidelines or standards for development of software in automotive domain. AUTOSAR version 4.0 describes new ideas for implementing multicore operating system for automotive domain. New features added in AUTOSAR 4.0 are spinlock, Inter Os-Application Communication, multi-core startup/shutdown and other inter core services. This paper discusses the design and implementation details of multicore operating system for dual core processor MPC5668G.

**Keywords**: AUTOSAR, OSEK/VDX, Multicore, Microcontroller, Spinlock

## Introduction

Multicore processing is very important today because they facilitate parallel computing. It may be task level parallelism, data level parallelism, instruction level parallelism. This paper describes about task level parallelism. The application to be executed is divided into tasks and execution of tasks is distributed among different cores. In task parallelism each processor executes a process on same or different data. Each process may execute same or different code. In their execution different process may communicate with one another. This communication is achieved by passing data from one process to another. A crucial step in parallel programming is the allocation of tasks to the processors and the definition of their execution order [1].

Two open standards available in automotive domain for providing common platforms for software design are Open systems and corresponding interfaces for automotive electronics / Vehicle Distributed executive (OSEK/VDX)[2] and Automotive Open System Architecture (AUTOSAR)[3]. They are real time operating system standards. AUTOSAR OS is an extension of OSEK OS and it uses OSEK as basis.

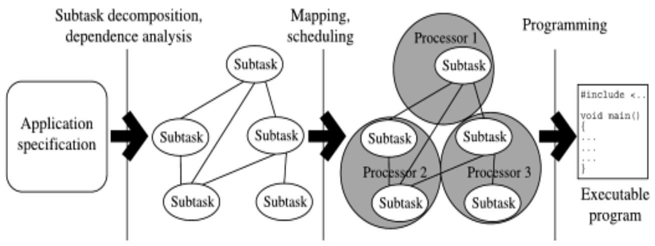

**Fig 1. Parallelization process in parallel programming**

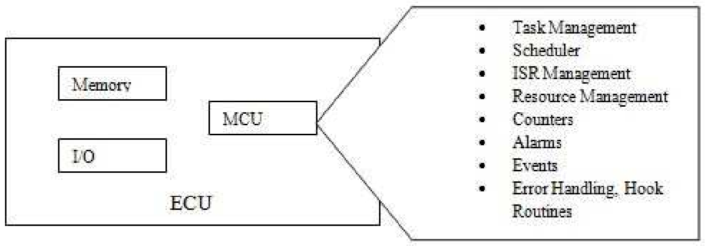

**Fig 2: OSEK OS Services**

Services provided by OSEK OS are shown in figure 2. They are

**A.** Task Management: OSEK OS provides two different types of tasks. They are basic tasks and extended tasks. Basic task releases the processor either in task termination or in execution of interrupt service routine as a result of interrupts. Extended tasks introduced one additional state called waiting state. The waiting state allows the processor to be released and to be reassigned to a lower-priority task without the need to terminate the running extended task [2].

**B.** Scheduler: The scheduler will organize the task execution sequence. OSEK supports multitasking so scheduling policy is important to determine which task will get the processor next. OSEK/VDX supports two scheduling policies they are full preemptive scheduling and non preemptive scheduling. Scheduling technique used is priority scheduling. If two tasks have same priority, then tasks are scheduled based on their arrival time.

**C.** Interrupt Management: OSEK/VDX supports two types of interrupts based on the Interrupt Service Routine (ISR). They are category 1 interrupts and category 2 interrupts. A category 1 ISR function does not contains any APIs supported by operating systems. Besides category 2 ISR functions make use of OS provided services. Rescheduling is not possible inside the ISR.

**D.** Resource Management: Resource management is necessary to ensure that two tasks cannot occupy the same resource at a time. Resource management prevents priority inversion and deadlock in some extend. Priority inversion can be avoided by Priority Ceiling Protocol (PCP) [3]. Each resource is assigned a static priority called ceiling priority. PCP temporarily raises the priority of the task to the ceiling priority of the resource if the priority of the task is less than the ceiling priority. The original priority of the task will be restored latter whenever the resource is released.

**E.** Counters and Alarms: Counter and alarms are used in OSEK operating system for processing recurring events. Counter is derived from hardware or software timer. Alarms are attached to the counter. More than one alarm can be attached to one counter. Alarm performs actions like task activation, set event and execution of callback functions at their expiry points.

**F.** Events: Events are used for synchronization among tasks. They are assigned only to extended tasks. Waiting for an event causes task enters into waiting state. When the event is set by another task, the state of earlier task will be changed from waiting to running or ready state.

**G.** Error Handling and Hook Routines: Hook routines are user defined code within the OS. They are called by the operating system on special situations depending on OS implementation. They have higher priority than all tasks and cannot be interrupted by category 2 interrupts. Different Hook routines used in OSEK are startup hook, shutdown hook, error hook, post-task hook and pre-task hook.

## AUTOSAR OS

The functionalities provided by AUTOSAR OS are backward compatible with OSEK OS. New concept introduced in AUTODAR is schedule table. Schedule table encapsulates a set of expiry points and thus address synchronization problem. Task activation and event setting are the actions associated with expiry elements. One or more actions can be assigned with one expiry element. Schedule table has a duration which is specified in ticks. OS will repeatedly process schedule table and will fire each expiry point in turn. Structure of schedule table is shown in Fig. 3. Schedule table may contain a set of tasks to activate and set of events to set. Task activations should be processed first then events. There are two types of schedule tables. Single shot schedule table will stop after executing all expiry points. Repeating schedule table will loops back to initial expiry point after processing final expiry point. The repeating period is equal to the duration of repeating schedule table. It is possible to start schedule table from an absolute or relative value of the counter driving the schedule table. Schedule table expiry elements are synchronized with the underlying counter. Two types of synchronization are possible. They are explicit synchronization and implicit synchronization [4].

AUTOSAR OS provides facility for stack monitoring. Stack monitoring will find out whether a task or ISR exceeded specified stack usage at the time of context switch. AUTOSAR supports all the operating system objects of OSEK/VDX OS. The collection of these objects (tasks, ISRs, alarms, schedule tables, counters, and resources) forms an OS application. The rights to access objects from applications might be granted during configuration. OS application may be trusted or non-trusted. Trusted applications are allowed to run in privileged mode and they have unrestricted access to memory and OS API. Non trusted applications have restricted access to memory and OS APIs. AUROSAR OS provides both memory and timing protection [4].

## AUTOSAR Multicore OS Specifications

AUTOSAR version 4.0 specifies extension to AUTOSAR OS and provides mechanism to support multicore micro-processors. The Multi-Core OS in AUTOSAR is not a virtual ECU concept, instead it shall be understood as an OS that shares the same configuration and most of the code, but operates on different data structures for each core [5]. Object (Task, Counter, Alarm, Resource ….) IDs should be unique across all cores. Every object ID should be referring to one entity independent of the core it is residing. This applies to all objects without considering whether they are shared or not. The OS can be entered on each core in parallel. TASKs and ISRs cannot dynamically change cores by means of the scheduling algorithm.

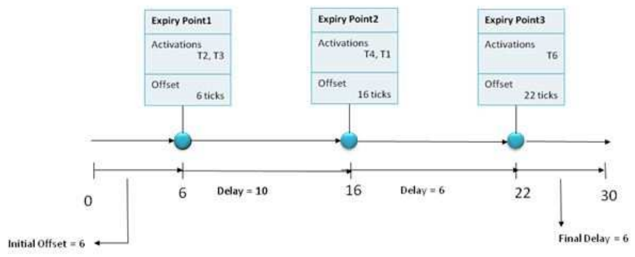

**Fig. 3 Structure of Schedule Table**

**A.** Locatable Entity (LE): LE is the term used to refer the object which is contained entirely on one core. The assignment of objects to cores is defined at configuration time.

**B.** Start-up: Core start-up is depending on the hardware. Hardware starts only one core called master core. All other cores remain in halt state until they are started by the software. In this master slave start-up behavior the master core does not need software activation. After startup StartOS() API will be called on each core. StartOs provides two synchronization points. One is before the execution of start-up hook and another is after the execution of application specific StartupHook and before the scheduler is started. Fig. 4 shows the multicore master-slave start-up behavior. StartOS() API is called by passing application mode as argument. In multicore OS all the cores shall be run in same application mode.

**C.** Shutdown: AUTOSAR 4.0 introduced a new API ShutdownAllCore which facilitates simultaneous shutdown of all cores. The API executes application specific shutdown hook followed by a synchronization point. All cores should be synchronized before calling shutdown hook. ShutdownOS() API is called on each core to shutdown that particular core.

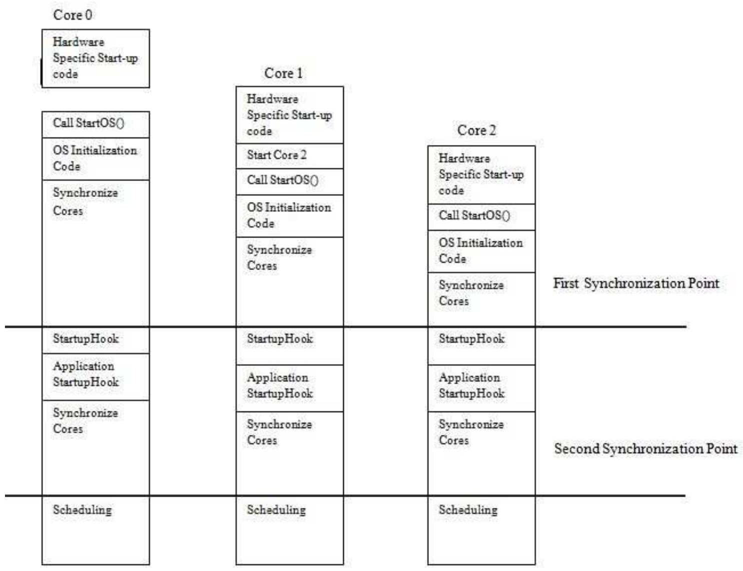

**Fig. 4 AUTOSAR Multicore Start-up behavior**

The hardware assigns a physical core id to each core. This physical core id cannot be used as an index in case of core specific variables. So mapping from physical core id to logical core id is necessary. Mapping of applications and other software components is based on the logical core id. Counters can be incremented only by the core on which it is assigned. Counters on one core can drive alarms and schedule tables residing on same core.

Inter OS-Application Communication (IOC) is used in AUTOSAR 4.0 for communication between applications. If applications resides in different cores then IOC become inter-core communication. IOC may not necessarily require cross core communication. All communication must be routed through RTE (Real Time Environment) on server and on receiver side. IOC supports 1:1 communication and N:1 communication, i.e, one sender one receiver communication and multiple sender and receiver communication. IOC provides both unqueued and queued communication services.

## MPC5668G Architecture

MPC 5668G is Freescale's dual core microcontroller. It is a 32-bit microcontroller with power PC architecture. It supports all automotive communication protocols. It provides 592 KB static RAM and 2 MB flash memory. 592 KB RAM is split into two blocks. MPC 5668G cores are e200z6 and e200z0. e200 processors are designed deeply for embedded control applications that require low cost solutions rather than maximum performance. It is an automotive focused product designed to satisfy need for integration of electronic features within the vehicle. It can be used as automotive gateway. Both cores have same instruction set, same set of registers and same data representation. MPC5668G offers symmetric multiprocessing because the RAM is shared between both cores [6].

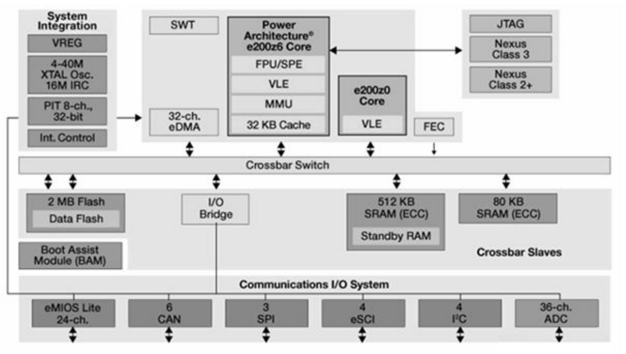

**Fig. 5 : MPC5668G Block diagram**

## Implementation Details

**A.** Core Identification: e200z6 and e200z0 processor register array contains a processor control register called Processor Version Register (PVR). For Z6 core PVR value is 0x8112_0000 and for Z0 core PVR value is 0x8171_0000. The GetCoreID() API will read the value of PVR register. Depending upon the known values of PVR register each core is differentiated by GetCoreID(). GetCoreID will return the logical core id of each processor. For example, Z6 core logical id is 0 and Z0 core logical id is 1. Logical id s assigned by the OS developer.

**B.** Start-up: e200z6 is the master core. So after booting, Z6 core will be active. Master core will activate the slave core by calling StartCore(Z0). When calling StartCore(Z0) the Z0 core will be activated by resetting the value of reset vector register(Z0VEC) of Z0 core. After startup each core will call StartOS() API with application mode as argument.

**C.** Synchronization: As discussed above StartOS() will synchronize the cores twice. Synchronization point is defined between cores to make sure that system is initialized properly [7]. Synchronization is performed by using a counter and a semaphore. Counter is a global variable that indicate how many cores had to be synchronized. At the synchronization point, each core will increment the counter variable and test whether the counter value equal to the total number of cores. If counter value is less than total number of cores, the core will wait.

**D.** Scheduling: Each core have different scheduler and priority ring buffer for scheduling tasks assigned to it. Core on which the task to be executed is specified by the configuration file. If task is assigned to Z6 core, the task is added to the priority ring buffer of Z6 and scheduler of Z6 will handle task execution.

**E.** Inter-core task Activation: Task activation is a type of alarm expiry action. Task assigned to one core can be activated by an alarm residing in different core. When a task is activated, the task id will be added to the priority ring buffer corresponding to the core on which it should be executed. The scheduler of that core will consider the task at the next scheduling point. AUTOSAR multicore OS supports inter-core event handling also.

**F.** Spinlock: Spinlock implementation uses atomic set-and-test functionality provided by the hardware. Spinlock is used to protect critical section and shared memory. Spinlock may lead to deadlock. A task can be protected from deadlock either by wrapping the spinlock with SuspendAllInterrupts() API or by preventing nested spinlock calls. Nested spinlock calls can be avoided by assigning numbers to resources in increasing order. The resource numbers will be added to a linked list. When a task trying to acquire multiple resources, it can take resources only in the order in which they are stored in the linked list.

**G.** IOC : The IOC allows one or more data item transfer per atomic communication operation. A data item can either be a value for atomic basic data types or a reference for user defined data structures. IOC send function writes data to the buffer in memory area which is shared by both sender and receiver. Concurrent access to the data in buffer is protected by using spinlock. Each atomic communication has to be specified in the IOC Configuration Description in a standardized XML format. Sender-Receiver communication can be performed in two semantics; first one is data semantics (last-is-best semantics) and second is event semantics (First-is-best semantics). 'IsQueued' attribute of data element is used to distinguish between data and event semantics. In event semantics this attribute is set to true and in data semantics this attribute will be set to false [8]. There is one description block per communication operation specifying
- Unique identifier
- Data type(s)
- Sender information
- Receiver information
- Name of callback function on receiver side in case of notification.
-Communication is queued or unqueued (last is best)
- In queued communication: Length of the queue

Software components send items in queued semantics or unqueued semantics. RTE_Send_<id>(data) call will be mapped to IocSend_<Id> (<data>). Fig. 5 shows IOC with shared memory. Concurrent access to shared memory results in data inconsistency. It is very important to ensure data consistency. One way to ensure data consistency is task blocking strategy [8]. Data consistency can be achieved by protecting the code which is handling shared memory using spinlock.

**a)** Last-is-Best Semantics

This semantics is used for data transmission. Buffer used for this communication use a single element queue or a single memory location. Each call to send function will overwrite data in the buffer. i.e, the old value will be replaced by the new value. Read and write operations should be atomic. For data consistency, read and write operations are protected by interrupt disabling and spinlock. Read access always return last received data. When new data received, previous value will be discarded without considering whether it was read or not. On sender side, AUTOSAR supports two functions which are using Last-is-Best semantics. They are

IOCWrite_Id (data) : - used to transfer single data element

IOCWriteGroup_Id(data1, data2, …) :- used to transfer multiple data elements

Receiver side functions are

IOCRead_Id (data) : - used to read single data element

IOCReadGroup_Id(data1, data2, …) :- used to read multiple data elements

**b)** Event Semantics

Events are important and they have to be handled appropriately. Loss of events cannot be tolerated. So isQueued attribute should be set to true. The received events have to be buffered into a queue. The queue should have a fixed length which is specified in configuration file. Send operation will put the event to the end of the queue. Receive operation will read event from front of queue. If queue is full received event shall be discarded.

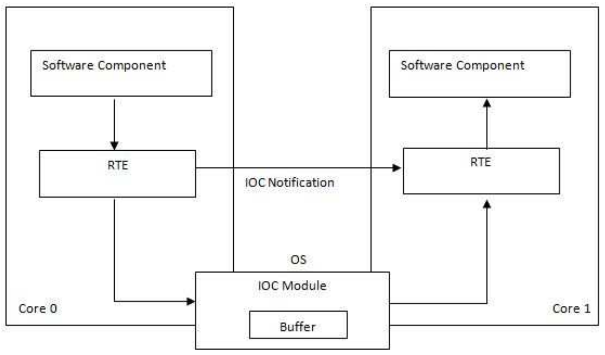

**Fig. 5 Inter OS-Application Communication through RTE**

Sender side event semantic functions are

IOCSend_Id (data) : - used to transfer single event

IOCSendGroup_Id(data1, data2, …) :- used to transfer multiple events

Receiver side functions are

IOCReceive_Id (data) : - used to read single data element

IOCReceiveGroup_Id(data1, data2, …) :- used to read multiple data elements

H. Building and Debugging

Compiler used is Wind River Diab compiler[9]. It provides the control and flexibility required to meet the demands of embedded software development. It is compliance with latest ANSI and ISO standards for C and C++. It supports wide variety of target architectures including Power PC. Lauterbach is used for debugging the program. Lauterbach PowerTrace and Trace32 software debugger provide instruction trace, memory monitoring capabilities, and kernel mode debugging. These capabilities enable the software developer to diagnose real-time software failures and memory corruption issues such as stack overflow and wild pointers. For providing on-chip debug logic, processors using JTAG (Joint Test Action Group) interface. JTAG is IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture. It allows developers to communicate with the chip and to perform operations like single stepping and breakpointing. Multicore processors required debugging of multiple cores via a shared debugging interface. Multicore processors does not provide separate JTAG interface for each core. Lauterbach provides provision to address several cores via joint JTAG interface and synchronous debugging. TRACE32 is an emulator system for MCUs that provides emulation memory and a bus state analyzer.

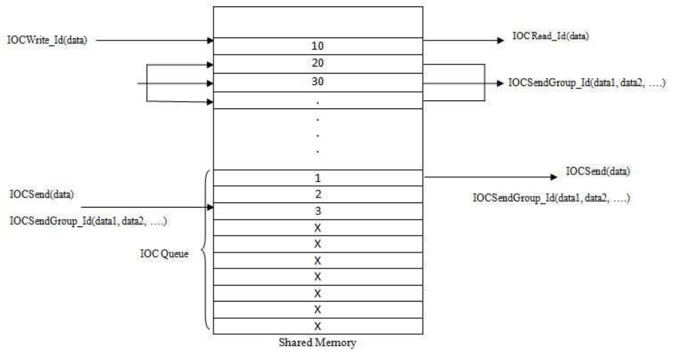
IOCWrite_Id(data)
IOCRead_Id(data)
10
20
30
IOCSendGroup_Id(data1, data2, ....)
IOCSend(data)
IOCSend(data)
IOCSendGroup_Id(data1, data2, ....)
IOCSendGroup_Id(data1, data2, ....)
1
2
3
X
X
X
X
X
X
X
IOC Queue
Shared Memory

**Fig. 6 IOC Functions using shared memory**

## Conclusion

Due to the increasing demand for electronic security and comfort in vehicles it is necessary to use more multi core ECUs. It is the only way to increase the performance. AUTOSAR 4.0 is a first step made in this direction. In the area of code migration from single core code to multi-core code there is an enormous development pressure of automated solutions. How far and fast multi core processors spreading depend on a number of criteria. Multicore operating system is used in cars if there is a large need for more processing power and the performance advantage of multi processor ECUs. Multi core ECUs can only cover the market, if the costs of the software development are not growing dramatic. This can only succeed if existing single core code can be migrated cost-effectively on multi core systems. AUTOSAR 4.0 is the preliminary version of multicore operating system. Additional features that can be added in future are dynamic allocation of tasks to cores, direct access from BSW modules to IOC services, activation of task on receiver side as a result of IOC notification.

## References

[1] Quinn Michael J, Parallel Programming in C with MPI and OpenMP McGraw-Hill Inc. 2004. ISBN 0-07-058201-7
[2] OSEK/VDX Operating System specification 2.2.3
[3] R. Rajkumar and J. Lehocsky. Priority Inheritance Protocols: an Approach to Real-Time Synchronisation. IEEE Transaction on Computer, 39(9):1175–1185, 1990.
[4] AUTOSAR Specification of Operating System Version 3.0.2
[5] AUTOSAR Specification of Multicore OS Architecture version 4.0
[6] MPC5668x Microcontroller Reference Manual for MPC5668E/MPC5668G, Document Number: MPC5668XRM, Rev. 2, 09/2008
[7] Configuring a Mixed Asymmetric Multicore Application for StarCore DSPs, Freescale Semiconductor Application Note, Document Number: AN4155, Rev. 0, 10/2010
[8] AUTOSAR Specification of RTE Software, version 1.0.1
[9] Wind River Compiler for PowerPC, User's Guide 5.7