Defence Science Journal, Vol. 63, No. 2, March 2013, pp. 214-222, DOI: 10.14429/dsj.63.4267

REVIEW PAPER

System Software Abstraction Layer - much more than Operating System Abstraction Layer

Sunita Awasthi Singh*, Bineesh P.K., and Satish Shetty K.

Defence Avionics Research Establishment, Bangalore– 560 093, India
**E-mail: sunitaawasthi.singh@gmail.com*

ABSTRACT

Current and future aircraft systems require real-time embedded software with greater flexibility compared to what was previously available due to the continuous advancements in the technology leading to large and complex systems. Portability of software as one of the aspects of this flexibility is a major concern in application development for avionics domain for fast development and integration of systems. Abstractions of the hardware platform which have been already introduced by the operating system community allow the software modules to be reused on different hardware and with different physical resources. Now operating system community has come up with an abstraction layer called operating system abstraction layer (OSAL) which along with the hardware abstraction unifies the OS architecture too. It provides a common set of primitives independent of the underlying operating system and its particular architecture. Factors such as reliability, scalability and determinism of any application largely depend on the design and architecture of the application. This is the most important and critical factor of real time systems such as mission computers of avionics systems, missile control system or control computers of space shuttle. It demands developer to perform feasibility of different software architecture to select the best alternative. Authors' analysis shows that to make any real time application more secure, scalable, deterministic, and highly portable, OSAL has to be extended to more than just operating system abstraction. This new view of OSAL will be called as system software abstraction layer (SSAL). In this paper, authors attempt to highlight the efficiency of SSAL as well as detailed description of its main features and design considerations. Authors have implemented the SSAL on top of two well known OS (WinCE and Vxworks) and performed extensive evaluations, which shows that it effectively reduces portability efforts while achieving simplicity, predictability, security and determinism. This paper presents in brief, the API functionalities, its components, implementation, interfaces, advantages and overheads along with a case study.

Keywords: Abstraction layer, API, Portability, secured, scalable, predictability, deterministic

NOMENCLATURE

API	Application programming interface
ARINC 653	Aeronautical Radio, INC. (ARINC) – Avionics application software standard interface
BSP	Board support package
HAL	Hardware abstraction layer
IMA	Integrated modular avionics
MC	Mission computer
OS	Operating system
OSAL	Operating system abstraction layer
PCD	Processing module control database
PM	Processing module
POSIX	Portable operating system interface
RTE	Real time executive
RTEMS	Real-time executive for multiprocessor Systems
RTOS	Real time operating system
SSAL	System software abstraction layer
SSP	System software package
XML	Extensible markup language

1. INTRODUCTION

The increasing avionics domain demands fast development and quick integration of software and hardware components. Portability of software components is a key issue, which becomes essential to achieve fast deployments of large and complex systems in current scenario.

The operating system abstraction layer (OSAL)[1,12] is a small layer of software used in the real-time software environment since many years. The main use of this layer is to isolate the embedded software from the real-time operating system (RTOS) so as to make it portable across all possible RTOS. It allows programs to run on different operating systems besides different hardware platforms. Hence it is independent of the underlying RTOS and hardware and is self-contained. Since many RTOS are not conformance to POSIX or other standards and also portability features of POSIX have its own limitations, the portability of the application is hard to achieve without such an abstraction layer. In addition, these standards will not make application developer free from RTOS jargon.

The fast and efficient development of software application demands the application developer to focus more on system

Received 14 September 2012, revised 08 February 2013, online published 23 March 2013

requirements of the domain application, functionality of the system and system integration rather than understanding RTOS features to implement these requirements. Today many commercial and custom developed RTOS are available in the market. Tuning embedded software as per the RTOS requirements from system to system has become a gigantic and time consuming task. Also, since many of the systems are developed in collaboration with many organizations across the world, it is difficult to maintain single RTOS for multiple systems. The best solution to these problems is to provide a common interface to embedded software regardless of RTOS or hardware to be used in the system. This interface is known as OSAL.

The OSAL is designed to be placed on top of the OS which translates system primitives from the original operating system into a unified API. A well designed OSAL provides implementations of an API for several real-time operating systems such as VxWorks, INTEGRITY, RTEMS, etc. To facilitate the use of these APIs, OSALs generally include a directory structure and set of makefiles that facilitates building a project for a particular OS and hardware platform. In particular, it addresses the discrepancies among different OS with respect to their functional API, hardware configuration mechanisms, resource management and handling of peripherals. At the same time, it also allows embedded software to be developed and tested on desktop workstations, providing a shorter development and debug time.

To make embedded software more secured, scalable, predictable and deterministic in addition to portable, OSAL can be extended to more than operating system and hardware abstraction. This new view of OSAL will be called as system software abstraction layer (SSAL). SSAL can be viewed as a layer above OSAL where it extracts system specific requirements. It allows developers to develop and maintain same version of the embedded software across different systems of the same functionality but with different hardware and different OS. It also provides the desktop environment for the development of embedded software to the developer which in turn reduces the impact of potential hardware delays.

This concept has been implemented only for one system currently. We have attempted to address restrictions and scope of future expansion keeping in mind variety of systems. Also analogy of SSAL concept with ARINC653[2] concept is addressed in brief.

2. OBJECTIVE

The main objective of SSAL is to create only a single instance of abstraction layer which provides static or dynamic configuration of the system at start-up provides functionality services to the application such as memory, timer, message queue, and semaphore and spawns all instances of modules including other resources. It also provides an interface to configure the platform and its domain as per the system requirements.

The layers of the system which use SSAL are illustrated in Fig. 1. The SSAL is the top layer of the system software package (SSP) which completely isolates hardware and system software from real time application.

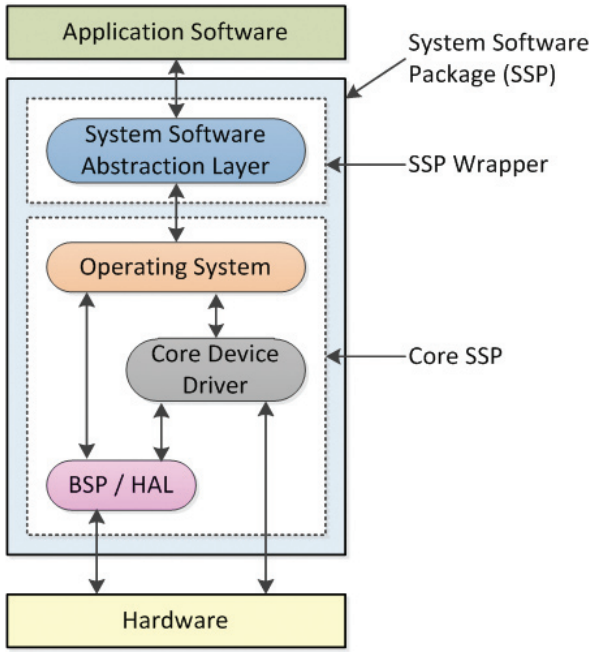

Figure 1. System software package layers.

The objective of this layer includes following main goals:

- Provide different infrastructures or frameworks for the application which can be configured based on system requirement.
- Provide static configuration mechanism to configure the resources to meet real time requirements of hard real time system.
- Provide simple or common API's for all device driver access isolating the complexity of the devices.
- Provide debug/control mechanism such as system monitor.
- Provide mechanism to develop and test embedded software in desktop environment with respect to the system functionality to achieve shorter development time and reduce hardware or RTOS IDE dependencies.

3. ADVANTAGES

- The developer is free from most of RTOS complexity and Software Architecture. As a result an avionics software developer can concentrate on Avionics functionality rather than Software terminology
- Allows reuse of software for different mission program with different hardware.
- Porting of application developed for one system to another with minimum or no changes.
- Maintenance of single version of the software for different RTOS or Hardware platform.
- Extending the software features without modifying the existing features

4. IMPLEMENTATION OUTLINE

The SSAL layer will contain a set of libraries or source code for interfacing to different RTOS. All these libraries will provide same API and functionality but

internally use API of different RTOS. The developer will select the required library through simple configurations based on RTOS used in the system. The layer will also contain different application frameworks or infrastructure files (library or source code) on which application is developed.

To select the required application framework, developer will update the configuration file. The layer will add most common device driver APIs like PCIe, Serial, ARINC429, and MIL–STD–1553B used in general avionics application and select required drivers in the configuration file. The specific device driver can be added in SSAL which will not demand any changes in application since the API will not change for any specific device.

SSAL will also have built-in debug system called system monitor which can be used by the application to monitor or debug the system. The system monitor will have PC based user interface software which will be communicating to the target system through serial or Ethernet. It will enable system engineers and software developers to have an insight view of the system. It will provide a utility to override normal behavior of the system for debug or lab purposes.

The configuration file can be a simple extensible markup language (XML) or C header files. The developer will configure the configuration file and using simple tools like make files or batch files can build SSAL layer in the form library or object file and will add this file to the application. The framework details are addressed in section 10 - Application framework.

5. CURRENT STATUS

At present, the SSAL is in early development stage. The SSAL has been partially implemented for Windows CE and VxWorks RTOS. Currently it provides only event and message oriented infrastructure for the application layer. System configuration is implemented only through header files. To evaluate the SSAL layer, a control computer application for an avionics program was developed over the SSAL. This development process proved that using SSAL, it is possible to achieve considerable reduction in application development time.

6. DIRECTORY STRUCTURE

The abstract directory hierarchy of SSAL is illustrated in Fig. 2. The options under each directory are not restricted to two or three levels or stages or adaptations. The figure shows a sample directory hierarchy. For example, OS components may include RTEMS, INTEGRITY, Linux, RTE in addition to VxWorks, Windows CE.

7. STANDARD API

The basic services provided by API are establish standard tasks, enable standard messaging mechanism, dynamic use of memory pools, enable timers and real time clock, exceptions handling, device drivers support, restart and watchdog mechanism. The sample API's for different functionality are listed in the Table 1.

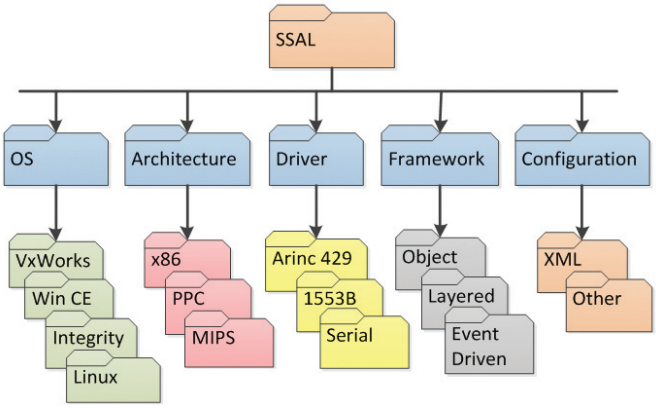

Figure 2. Directory structure.

Table 1. SSAL services

SSAL services	Application programming interfaces
Initialization	SysInit , SysMain
Task/Thread	SysActivateAllThreads, SysAttachWdHndls, SysCreateAllThreads , SysGetMyThrdId , SysGetMyThrdIdx , SysGetThrdIdByName, SysGetThrdIdxByName , SysGetThrdNameById
Queue	SysAllocMsg , SysFreeMsg , SysGetMsg, SysSndExMsg, SysSndMsg , SysSndMsgEx
Semaphore	SysSemGive , SysSemTake, SysCreateMutex
Event	SysRstTimeEvent,SysSetEvent, SysSet Time Event, SysWait4 Events
File system	SysFileInit, SysFileCreate, SysFileOpen, SysFileClose, SysFileRead, SysFileWrite, SysFileSeek, SysFileCopy, SysFileMove, SysFileRename, SysFileRemove
Misc	SysWait4Start, SysWatchDogRefresh, SysWatchDogStart , SysWaitForMultipleObjects , SysGetMyObjctDat

8. METRICS

Executable lines of code: 3500 approximately
Number of distinct BSP's: 2
Number of OS's supported: 2 – VxWorks, WinCE
Number of processors supported: x86, PPC, MIPS

9. OVERHEAD

9.1. Memory Overhead

The memory overhead of SSAL layer will be minimal. It may range from few kbytes to Mbytes. The current SSAL library which is for VxWorks RTOS with event based framework is only 54 kB size. The size of this version may not exceed 256 kB even after adding the new features.

9.2. Code Overhead

The SSAL use few internal data structure to maintain scalability and determinism. In addition to this, SSAL has code to implement RTE, framework, debug system

and other features. These modules may not be considered as overheads. It has few coding overheads to implement some API's which has different implementation across RTOS.

10. APPLICATION FRAMEWORK

The current framework design is based on events. In event driven framework, software modules are grouped into processing modules (PM). Each processing module has one master thread and various optional service threads. The communication between PMs is only through configured events and messages. The master thread of a processing module can send/receive message to/from other processing module. This will make each processing module independent from other. The master thread of all processing modules has identical structure. The SSAL also provides API's to maintain synchronization between processing modules. Basic services provided for processing modules are PM management; inter PM communication, intra PM communication and error handling. Figure 3 shows the architecture of this framework.

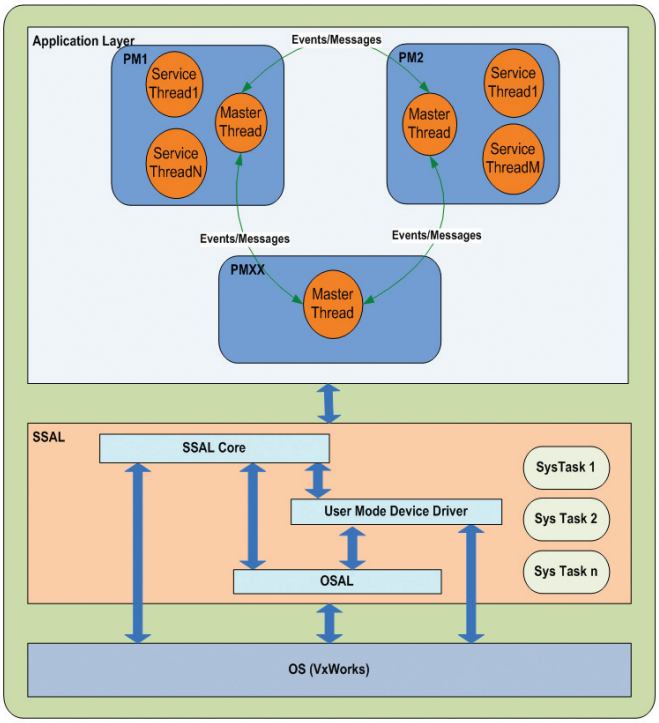

Figure 3. SSAL Events Driven Framework.

10.1 Messaging

Messaging is the method for communication between PMs. Each PM can send a message to any other PM utilizing SSAL services. A message contains the originating PM, destination PM, message length, and message code and payload data. Each PM has an incoming queue of messages; the size of the queue is statically configured during PM creation. The message queue holds the pointer to messages allocated from message pools that are pending to the PM. Sending a message to a PM is signalled to the PM by raising the message event to the receiving PM.

10.2 Events

Events are the main trigger of processing module to become ready from wait state. A processing module may wait for multiple events at once; the first event comes, wakes the PM and starts a process.

10.3 Software watchdog

The SSAL software watchdog subsystem is used to monitor the processing modules and verify that processing modules are capable of handling events. Each processing module defines the maximum time, in milliseconds, in which the PM must signal that it is ready to handle events, this is done by calling the watchdog service routine. The SSAL verifies that all processing modules serve the software watchdog and if a processing module watchdog expires, it is a software watchdog event that causes a software restart.

The below code segment is the outline of the master thread of any processing module.

```
void  PM_xyz_master(void)
    {
        /* Prepare this PM for working */
        PM_xyz _Init();
    /* Wait for other PMs to be ready – if Sync. Flag is true
    for this PM, then this call will be blocked till all the PMs
    in the system become ready else return immediately*/
    SysWait4Start();
    /* Start the software watchdog monitor for PM*/
    SysWatchDogStart();
    /* Start the event handling */
    while( TRUE )
        {
          /* Wait for events from SSAL */
           event = SysWait4Events (…);
           /* Refresh the software watchdog */
           SysWatchDogRefresh();
           /* Handle the event */
            switch( event )
              {
               /* High Priority Msgs */
                case HIGH_PRI_MSG_EV:
               /* Get the msg Pointer */
               msg_ptr= SysGetMsg();
               /* Handle the Message */
               PM_xyz_HandleHighPrioMsgs(msg_ptr);
               /* Free the message buffer */
               SysFreeMsg(&msg_ptr);
               break;
               /* Low Priority Msgs */
               case LOW_PRI_MSG_EV:
                   /* Get the msg Pointer */
                   msg_ptr= SysGetMsg();
                   /* Handle the Message */
                   PM_xyz_HandleLowPrioMsgs(msg_ptr);
                   /* Free the message buffer */
                   SysFreeMsg(&msg_ptr);
                   break;
        /* Timer Event *
        case TMR_EV:
                   /* Handle Timer Event */
                   PM_xyz_HandleTimerEvent();
```

```
                break;
        default:
        break;
        } /* Event Handling */
    } /* While */
} /* PM_xyz_master */
```

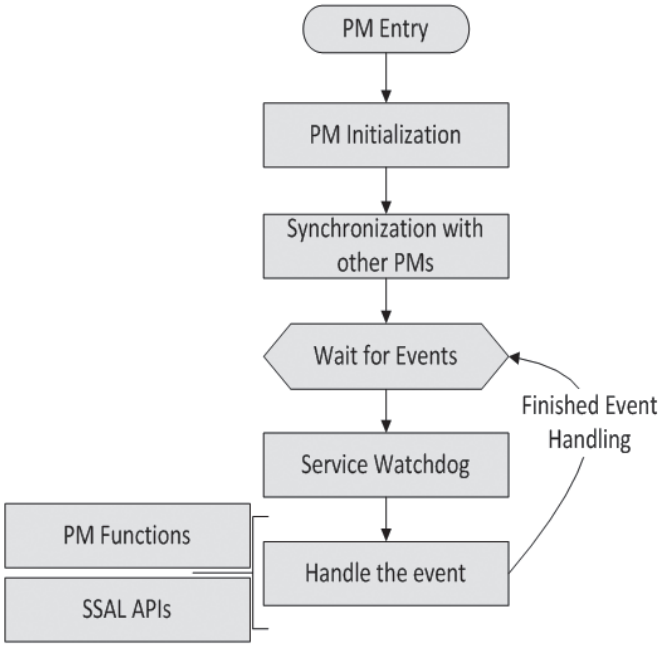

Figure 4. Processing module master thread architecture.

11. PROCESSING MODULE CONTROL DATABASE

All processing modules (PMs) of application are defined in the form of PM control database (PCD). This static configuration ensures reliability of the software. The PCD will be generated using a data structure. The PCD data structure as an example is shown below in Table 2.

Table 2. PCD Data Structure

Member	Description
pm_type	Application/socket/system
stack_size	Stack requirements for PM in bytes
pm_param	PM name, entry function, etc
priority	Priority of master thread of PM
thread_h	Handle of master thread
wd_flg	If Software watchdog required
wd_timeout	watchdog timeout
sync_flg	If this PM needs to wait for other PMs to get ready
sync_done	Sync done
ev_fil[EV_NOE]	Event information of this PM
msg_q	Message queue of PM
io_info	I/O device attached to this PM
sock_info	Network info if PM of socket type
eol	To mark the end of the PM in PCD list

11.1 Initialization

The global PM file holds all PM information and first entry of PCD is always 'InitThread' which is responsible for complete system initialization and activating all other PMs of the PCD. The API - sysMain will activate only InitThread. The OS user entry function (WinMain – Windows, usrAppInit - VxWorks) will call SSAL entry function 'SysMain' and this will create first PM of PCD (highest priority task) which is the InitThread.

11.2 Start-up Sequence

The start-up sequence is described in the Fig. 5 given below. The OS_UserEntry depends on OS used in the system. For example in VxWorks it is usrAppInit().

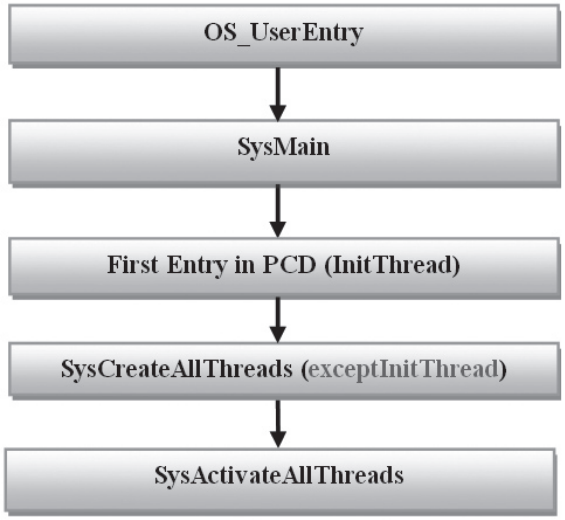

Figure 5. start-up sequence.

11.3 Monitoring

The InitThread will start monitoring master thread of all PM after it activates all threads. The watchdog handler of InitThread will handle time out of any PM based on criticality of the fault occurred in PM.

12. APPLICATION FRAMEWORK - A CASE STUDY

Fig. 6 shows the major processing modules of the application software and the data flow between them. This framework implemented in one of control computer (CC) and figure describes interfaces of CC to external sub systems.The developer should avoid OS API calls and OS dependent header files in application software with the current framework. However, developer is free to use standard header files like string.h, stdio.h, etc.

12.1 Handling Outgoing Message

As an example we would explain the concept of handling an outgoing Message from the system. The outgoing messages are handled using a set of SSAL threads called PMR (PM Routers) and two set of tables –one global SSAL Route table and one local outgoing messages table for each PM Router (communication interface).

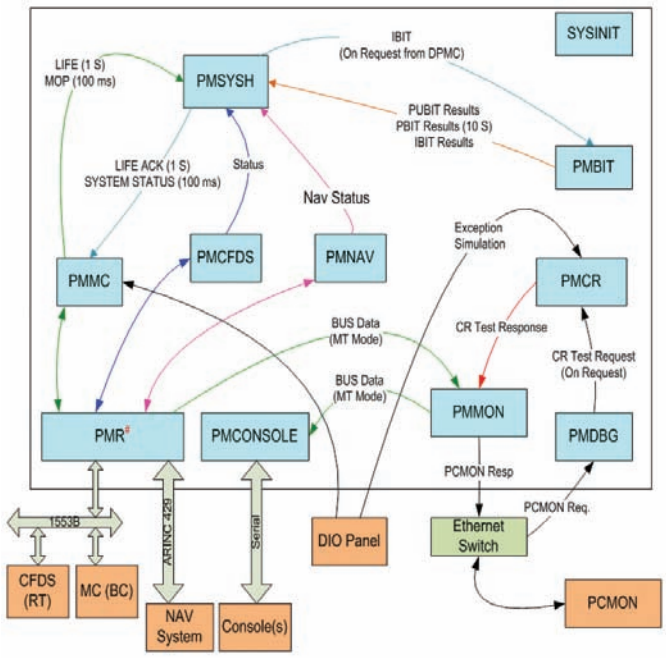

Figure 6. Application framework architecture.

12.2 SSAL Route Table

This table is used by SSAL to decide where to send the external messages. This table should be updated with the entries so that the messages for the external computers interface can be send to required PMR (R1553, RARINC or RSERIAL). For example if system has to send a message to a mission computer (MC) and if this message has to go through 1553B which is handled by R1553, then an entry has to be added in the SSAL route table to send the messages from PMMC to R1553. If this is done, then PMMC can send the message to MC (using SysSendExMsg API) and this will internally come to R1553, who handles it depending on the outgoing message table entry.

12.3 Outgoing Messages Table

This table consists of information regarding the outgoing messages in the system. It has multiple instances based on the number of priority levels supported by PMR. The contents of the table are inserted in the system initialization through function 'PMR_Register_Ext_Msg'.

Table 3. PMR outgoing message table

Member	Description
PM_Name	Name of the application PM which is registering
Message header	when the message comes to PMR, how it can identify it
Priority level	Priority for the message
Hardware interface	message will go to which hardware interface (since single thread of PMR can handle multiple hardware interface)
Hardware interface specific parameters	It will vary based on the hardware interface, for ARINC it can be the transmit or receive channel number, For 1553B, it can be which controller number and which sub address etc.

Example Scenario

Consider that PMMC has to send a message to MC (External computer) namely MSG_PMMC_2_MC_LIFE_ACK through the 1553B interface (Controller-1, SA–2). The following steps are to be done:

1. In SSAL route table, an entry has to be made so that messages to MC are routed to PMR. This is done when SSAL is built (the route table is built into the SSAL)
2. Register the message – Add entry in outgoing message table

a. lr_msg_hdr.sq_msg_code = MSG_PMMC_2_MC_LIFE_ACK;
b. lr_life_ack.sq_chnl = MUXBUS_CHNL_0;
c. lr_life_ack.ub_SA = 0x02;
d. lsq_dpr_err = PMR_Register_Ext_Msg("PMMC", &lr_msg_hdr, MSG_TYPE_PMR_MED, DRV_IF_MUXBUS, & lr_life_ack, &lsq_drv_err);

Once this step is done successfully, the PMMC can send message to MC and this will go through PMR as shown in the fig. 7.

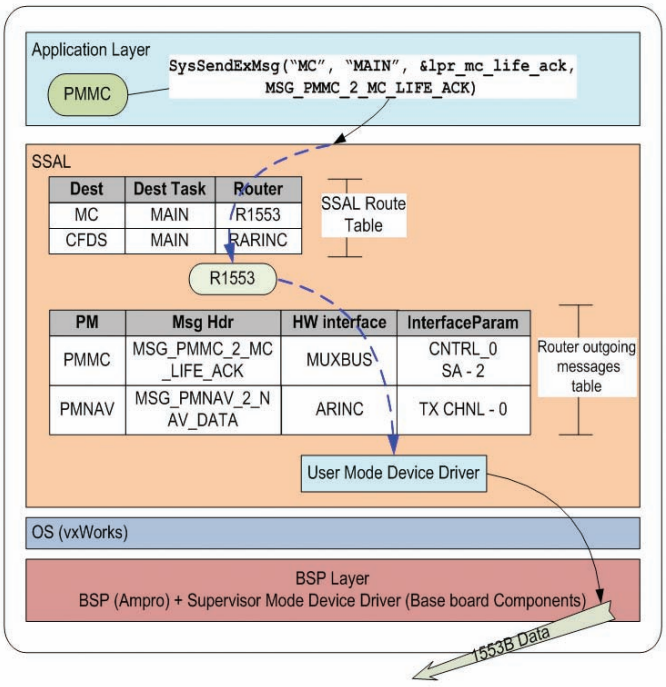

Figure 7. PMMC to MC (driver interface) message transfer through PMR.

12.2 Handling Incoming Messages

The incoming messages are handled using the table,

1. PMR incoming message table – Separate tables are maintained per interface basis

a. ARINC incoming table (separate table for each Receiver channel)
b. 1553B incoming table (separate table for each controller)

Table 4. PMR incoming message table

Member	Description
PM_Name	Name of the application PM which is registering
Message header	Message header to be attached while sending the message to the PM
Priority level	Priority for the message
Message ID	ID for the message
Hardware interface specific parameters	It will vary based on the hardware interface, for ARINC Table this field is not required, for 1553B which Sub Address

Incoming Messages Table

This table consists of information regarding the incoming messages in the system. It has multiple instances based on the number of hardware interfaces supported by PMR (R1553, RARINC or RSERIAL). The contents of the table are inserted in the system initialization through 'PMR_Register_PM_forData'.

Example Scenario

Consider that PMMC has to receive a message from MC (External computer) namely MSG_MC_2_PMMC_LIFE through the 1553B interface (Controller - 1, SA – 2). The following steps are to be done:

1. Register the message - Add entry in Incoming Message Table
   a. lr_msg_hdr.sq_msg_code = MSG_MC_2_PMMC_LIFE;
   b. lr_life.sq_chnl = MUXBUS_CHNL_0;
   c. lr_life.ub_SA = 0x02;
   d. lsq_dpr_err = PMR_Register_PM_forData ("PMMC", &lr_msg_hdr, MSG_TYPE_PMMC_MED, MSG_ID_MC_LIFE, DRV_IF_MUXBUS, & lr_life, &lsq_drv_err);

   Once this is done successfully, the PMMC will receive message from PMR whenever the controller 1 (at SA 2) detect any data activity happening. In such event, MUXBUS driver will generate the data event (generated by the handler attached to MUXBUS driver by PMR) to PMR, which in turn will collect the data and send to PMMC using SSAL message interfaces.

13. ANALOGY WITH ARINC 653

Continuous growth in the aerospace industry has encouraged the avionics systems to utilize the increased processing power, communication bandwidth and hosting multiple federated applications into a single integrated platform. This technology has been realized as integrated modular avionics (IMA) which has emerged as a platform for integrating multiple avionics applications of varying severity levels on a common shared integrated computing environment[5,6]. IMA platform is realizable with well integrated ARINC 653[3] based real time operating system with time and memory partitioning with application executive (APEX) libraries[4].

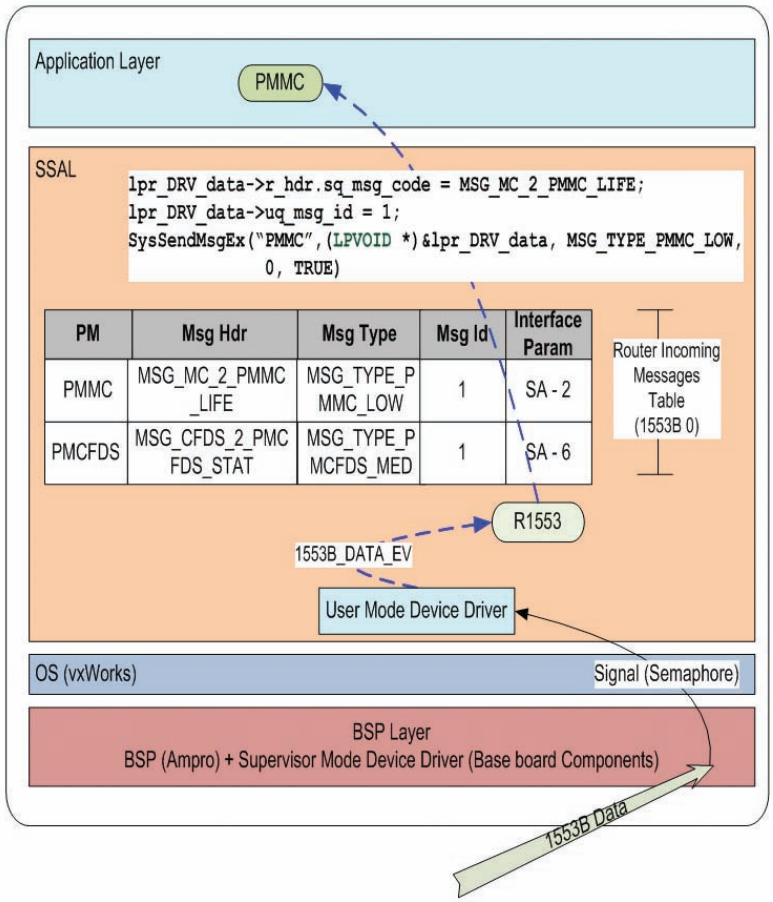

Figure 8. MC to PMMC (driver interface) message transfer through PMR.

ARINC 653 defines an API called application executive (APEX) to decouple the RTOS platform from the application software. It provides an abstraction layer managing the timer and space partitioning constraints of the platform and an interface to configure the platform and its domain.

Each application software is individually contained in a partition and has its own memory space. It also has a dedicated time slot allocated by the APEX API. Within each partition multitasking is allowed. The APEX API provides services to manage partitions, processes and timing, as well as partition/process communication and error handling.

The main goals for designing IMA based systems[8] are technology transparency, scheduled maintenance and incremental updates. Basic features of an IMA system[8] are layered architecture using standard programming interface layers to hide hardware and applications from one another, static or dynamic reconfiguration of applications, protection mechanisms among applications, to allow applications to be inserted or altered without impact on the rest of the system, flexible scheduling to meet the deadlines of all the applications, for each viable configuration and when system is upgraded, code re-use and portability, an operating system to manage the applications, physical integration of networks, modules and IO devices and design for growth and change.

As of now, SSAL cannot be considered to be ARINC 653 compliant. It is currently designed for non ARINC653 systems. It does not provide partition management kernel in OS hence cannot fulfil basic requirements of spatial and temporal partitioning. It fulfils only standard API requirement similar to ARINC 653 APEX from configurability, portability, flexibility, reliability, security and determinism point of view. Also it gives a future scope of growth and changes in design. It provides configuration mechanism including error management while allocating resources during the start up of the system. It can be used in systems where higher determinism is required with a custom real time executive to avoid RTOS overheads in run time.

14. FUTURE PLAN

To make real time application development avail powerful features of SSAL, many functionalities have to be implemented in future. Some of the functionalities are:

- Provision in SSAL for many RTOS environment selection such as Integrity, Windows.
- The configuration through XML file or simple graphical user interface (GUI).
- Different types of frameworks selection for different levels of real time application should be provided in future versions.
- SSAL can be extended to provide real time executive in simple real time systems where no OS is used.
- Conversion from thread model to process model
- Shared memory API
- Extensive monitor and debug system

15. CONCLUSIONS

The SSAL project is started with a vision of bringing all real time application development across DRDO labs under a common platform. This will avoid work duplicity and facilitate porting of applications developed for one system to other with minimum or no changes. SSAL interface with ARINC 653 has to be thought carefully. The SSAL development is tightly coupled with RTOS. Since it is capable of providing real time executive to systems without RTOS, it resembles a custom minimal RTOS in many features. This feature of SSAL has led to the development plan of a common in-house RTOS for all the real time application development in research organizations across the nation. This will avoid not only long term dependency on RTOS vendors but also overheads of RTOS. However to achieve the vision of SSAL, the research community across the nation has to provide many contributions to this project. We are looking forward to suggestions and ideas from scientist community to make SSAL project a vision to reality.

ACKNOWLEDGMENTS

Authors thank Shri P. M Soundar Rajan, Director DARE, for his continuous moral support and motivation and also Wg. Cdr Sendhil Kumar for his technical support and valuable comments on this paper. Authors acknowledge their gratitude towards the members of System Software Division (SSD), Technology Group of Defence Avionics Research Establishment (DARE) for their interactive support from time to time.

REFERENCES

1. OS abstraction layer. http://osal.sourceforge.net/ (Accessed on 20/12/2011)
2. 653P3 - Avionics Application Software Standard Interface, Part 3, Conformity Test Specification. https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=704(Accessed on 14/06/2012)
3. ARINC Specification 653-1, Avionics Application Standard Interface, Published by Aeronautical Radio Inc Software, October 2003.
4. ARINC. ARINC Specification 653-2: Avionics Application Software Standard Interface Part 1 - Required Services. Aeronautical Radio INC, Maryland, USA. 2005.
5. ARINC Specification 651: Design Guidance for Integrated Modular Avionics. Aeronautical Radio, Inc, Annapolis, MD, November 1991. Prepared by the Airlines Electronic Engineering Committee.
6. DO 297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations, RTCA Inc, RTCA Inc, November 2005.
7. John Rushby, Partitioning in Avionics Architectures: Requirements, mechanisms and Assurance, SRI International, Menlo Park, California, NASA/CR-1999-209347, June 1999.
8. What is integrated modular avionics (IMA)? http://www-users.cs.york.ac.uk/~philippa/IMA.html (Accessed

on 15/01/2012)
9. José Rufino and Sérgio Filipe, AIR Project Final Report, DI-FCUL, TR–07–35
10. Wind River Systems Inc, Wind River System Viewer user's guide, ver 4.7, 2005.
11. An operating system abstraction layer for portable applications in wireless sensor networks. http://rts.eit.uni-kl.de/fileadmin/publication_files/SERNA_SAC10.pdf(Accessed on 05/07/2011)
12. OS abstraction layer (OSAL). http://opensource.gsfc.nasa.gov/projects/osal/index.php(Accessed on 06/10/2011)

Contributors

Mrs Sunita Awasthi Singh obtained her BE (Elect. Instrumentation Engg.) from BIET, Jhansi, India in 1996. She is working as a Scientist at Defence Avionics Research Establishment (DARE), Bangalore. Her area of expertise is development of real time embedded system software for various Fighter Aircrafts. Her core areas of interest are: Integrated modular avionics, board support package development, RTOS, device drivers.

Mr Bineesh PK obtained his BTech (Computer Science & Engg.) from NSSCE, Palakkad, in 2004. He is working as a Scientist at DARE, Bangalore, on real time embedded system software development. His areas of interest includes: RTOS, device drivers, and board support package.

Mr Satish Shetty K has obtained AMIETE (Computer Science & Engg.) from IETE New Delhi, India in 2010. He is currently working as Senior Technical Assistant at DARE, Bangalore. His areas of interests are: RTOS, device drivers, and avionics communication standards.