UML and Function-Class Decomposition for Embedded Software Design

Abstract:
This class introduces a practical application of the UML diagrams and function-class decomposition (FCD) concept to requirements analysis, software architecture analysis and design, and software design and implementation for a complex embedded system. Based on the function-class decomposition concept, the UML diagrams for requirement analysis, and software architecture analysis and design are shown in detail. This is followed by decomposing the complex software architecture into UML class and state diagrams. Two detailed software implementation examples (including a application manager and a device driver) that include UML diagrams and C++ code are shown.

By Chai Kok-Soon, PhD
Koksoon58@yahoo.com

[ESC-305] UML and Function-Class Decomposition for Embedded Software Design, San Jose, 2006.

1) Introduction

The complexity and the application of the embedded systems are increasing significantly. Companies developing embedded systems are facing new challenges, ranging from security to software change requirements management. In particular, a significant number of the embedded products are real time systems, and many them find applications in safety critical systems that have be designed in very high quality. The intrinsic quality of embedded products can be attributed to the embedded software that powers and controls the functionalities of the products. Therefore, there must be a stringent software quality process utilized in order to design a high quality embedded product. Designing high quality software while meeting product life cycle requirements is certainly one of the most important challenges in embedded software design.

High quality embedded software cannot be designed by concentrating on satisfying the software requirements alone. The quality of embedded software shall include, but not limited to these two criteria, I) how well the embedded software satisfies the user requirements, II) how well the tools and process are deployed to design embedded software. Except for the simplest academic or commercial software products, the majority of the embedded software products are complicated in terms of the effort spent to develop them, and the software artifacts, including but not limited to requirements and design documents, source code or software models, and test plans that are used to design the software product. With the increased complexity of embedded software in terms of lines of code, and challenges faced by change requirements management and portability and reusability of the embedded software, it will continue to become more and more difficult to write embedded software that achieves the type of quality and speed of change demanded in the commercial environment.

Section 2 in this paper surveys some of the work done in the Unified Modeling Language (UML), object-oriented analysis and design, and function class decomposition. This section suggests in a practical perspective how to combine these three techniques specifically for large scale embedded software systems development. Section 3 suggests the application of functional decomposition, use case, collaboration and sequence diagram for system and software architecture specification. Section 4 describes how to map and decompose the software architecture into class, state and sequence diagrams. Section 5 describes three software implementation examples that include UML class diagrams and a C++ example. Section 6 concludes the application of these three techniques for large scale embedded software system design, and suggests further work for system and integration test.

2) Overviews of the OO and FCD, and UML

Object-oriented concepts were first introduced from object-oriented programming with the introduction of Simula and later the Smalltalk programming language [1]. It is termed for writing objects that combine attributes for representing and storing the states of the objects with operations to manipulate the attributes. Booch [2] started to advocate the use of objects during the entire software development cycle. The application of object-

oriented analysis to real time system has become very popular. For example, Veeraraghavan [3] shows an example of applying the OOA method to the analysis of signaling and control in broadband networks. The method starts by focusing on objects in the problem, rather than functions. In general, these methods identify objects in the embedded systems, such as printers, servers etc. and decomposing the large scale embedded software in embedded devices into software architecture.

In practice, it is difficult to design and manage the complexity of large-scale software systems using pure OO techniques. This is because pure OO techniques tend to design a software system from bottom-up [6], and has little guidance on the design of software architecture. The software architecture contains the functionalities and structure of software modules, and the relationship between these software modules [5]. For large software systems, the overall system structure becomes a central design problem [4]. Object-orientation only provides partial support for the analysis and design of layers, components, connectors, relationships between components, interfaces etc. that are important for the representation of software architecture. OO techniques also identify large number of objects at the initial stage of large scale software development project, which are not easily manageable. Chang proposes the application of the function class decomposition method that takes a functional view into account on the object-oriented basis [7]. This method concentrates on structured way to functionally group classes into software modules that are based on proven software architecture techniques, and engages the concepts of class decomposition in the object-oriented paradigms to derive decomposed classes from these software modules.

The following sections will explore the application of UML diagrams to analyze, design and represent embedded software, including device drivers and application modules. The requirements design product symbolizes a large scale and distributed embedded software. In practice, these types of systems are developed by multiple teams, including internal and external development teams and component-based development that involve complex coordination and change management. The UML and function-class decomposition technique examines this situation where is it getting more difficult to design highly competitive complex embedded software without the support of an advanced design process and technique.

3) Functional Decomposition for System and Software Architecture Design

This section applies the functional decomposition method to the system and software architecture to ultimately create modules in UML. The functions based on software modules are different from the pure OO method where objects are identified at this stage of software design.

3.1) Designing Complex Embedded System and Software

Most of the embedded systems interact with external I/O such as sensors, actuators and networks. Advanced distributed embedded system design starts with the system architecture and protocol design. The layering in the OSI model supports a systematic approach to decomposing complex embedded software to a number of independent but structured computations (such as processes, tasks or threads in embedded software) that interact with the external I/Os and networks.

Based partially on a structured approach to decomposing large embedded software, the design example demonstrates how to design a complex embedded software system for a product called Device Programmer. This section covers simple protocol design for embedded system that is mapped and decomposed to embedded software in the Device Programmer. This example does not intend to cover all the functionality of the Device Programmer but shows sufficient detail to decompose distributed embedded system intosoftware architecture, and then to allow detailed design bases in UML and functional-class decomposition. This example could be applied to other embedded applications such as distributed boot-loader, proprietary wireless devices, network gears design etc.

3.2) System Requirements Analysis

The first design activity is to derive the requirements for the Device Programmer. A software requirements specification can combine text-based descriptions with use case diagrams to help analyze the product requirements of the Device Programmer. Figure 1 shows the use case diagram for the Device Programmer. The diagram shows the high level requirements that can be communicated to customers or the end users. The use case diagram shows that the Device Programmer has four major functions. The user calls the GetRemoteData use case in the Device Programmer to connect to a remote server. The ConnectServer use case establishes the connection between the Device Programmer and the Server. The DownloadData use case downloads configuration data from the Server to the Device Programmer. The ProgramDevices use case programs the devices connected to the Device Programmer with the configuration data. The use case diagram effectively shows that the Device Programmer is a networked device that programs devices by retrieving data from a remote server.

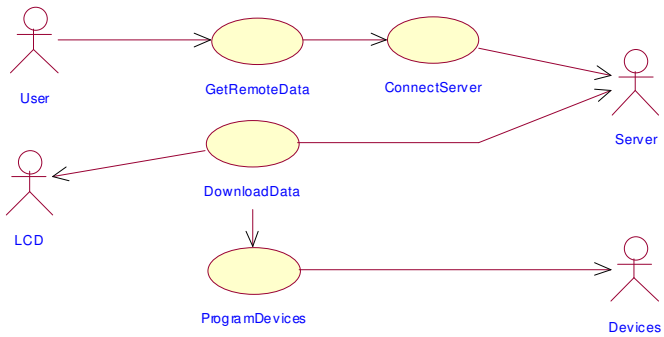

Figure 1: Use case diagram for the Device Program

The use case diagram shows what are the functionalities of the Device Programmer. Figure 2 shows that the Device Program uses the Ethernet and TCP/IP protocol to implement the GetRemoteData use case. The User presses power up button on the Device Programmer and the Device Programmer starts initialization, including the Ethernet connection. The Device Programmer goes into the CONNECTED state once it is successfully connected to the Server. The User initiates GetRemoteData function in the Device Programmer sending the GET_PROD_DATA command to the Server. The Server replies with a range of configuration data by replying the PRODUCT_DATA command to the Device Programmer. The Server issues a PRODUCT_DATA_COMPLETED command to the Device Programmer once the data transfer of the configuration data is completed.

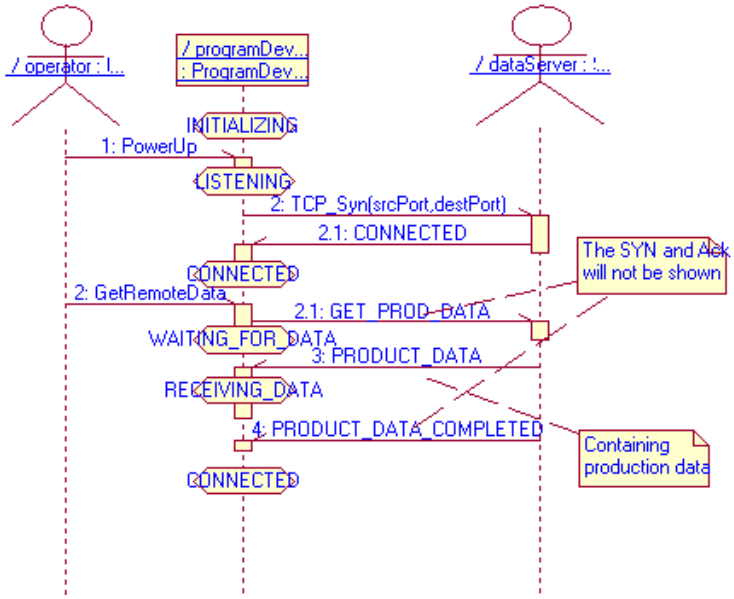

Figure 2: Sequence diagram showing the TCP/IP protocol

3.3) Software Architecture Design

The software architecture design following the concept of functional decomposition is commenced after the completion of the system requirements requirements analysis. This is an important phase to decompose the Device Programmer into a number of interconnecting high-level functions with structure. These functions should be connected with robust structure, typically a layering structure for the high-level functional decomposition of the Device Programmer. The arrangement of the functions in the layers may influence the software portability, reusability and maintainability of the embedded software. In order to design the embedded software to be portable to different hardware platforms, a hardware abstraction layer is introduced. The hardware abstraction layer should contain hardware driver functions that can be easily modified when porting to a new hardware platform. As the result, hardware related type defines (such as typedef unsigned long uint32_t) are normally located in a specific header file. More examples on this concept will be presented later on in this document.

The concept of portability also covers software platforms, particularly real-time operating systems (RTOS), which may differ on different projects. For companies that practice software reuse and aproduct line concept, the embedded software is designed to be shared across many different and generation of products. So an RTOS abstraction layer (ROSAL) containing the OS related functions from the embedded software is introduced. The communication layer, or middleware layer for some domains, is introduced for the grouping of communication related functions. This communication layer normally consists of the software implementation of protocols in the OSI layer from the data link layer to the transport layer. For example, this layer would contain the TCP, IP and PPP implementation of the TCP/IP protocol over the Ethernet network. The application layer, with the similar concept of the application layer in the OSI model, contains customized commands and functionalities for the implementation of protocols such as TCP/IP. However, for more complicated devices such as Device Programmer, the Application Layer also entails display management, user input management such as keypad management, multiple application management, data management, diagnostic management and the management of application layer for protocols such as TCP/IP, I2C etc.

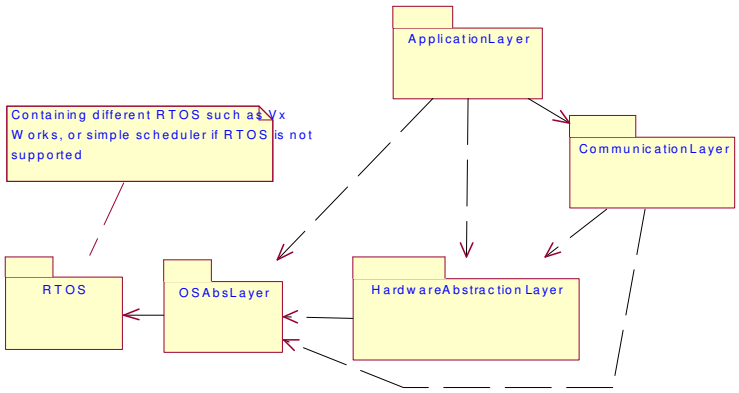

Figure 3: High-level software architecture of the Device Programmer

3.4) Functional Decomposition of the Application Layer

Figure 4 shows the functional decomposition of the ApplicationLayer clearly delineates different functions that will be further decomposed using the class decomposition method. Messages from other software layers are stored in the AppBuff and CommBuff software modules. The messages are retrieved and processed by the AppManager according to the data set in the dest_t field of the message. The messages are forwarded to the appropriate applications including ProgramApp, DataApp, DiagnosticApp etc. New applications can be easily added to the Application Layer with minor modifications in the AppBuff and CommBuff and AppManager.

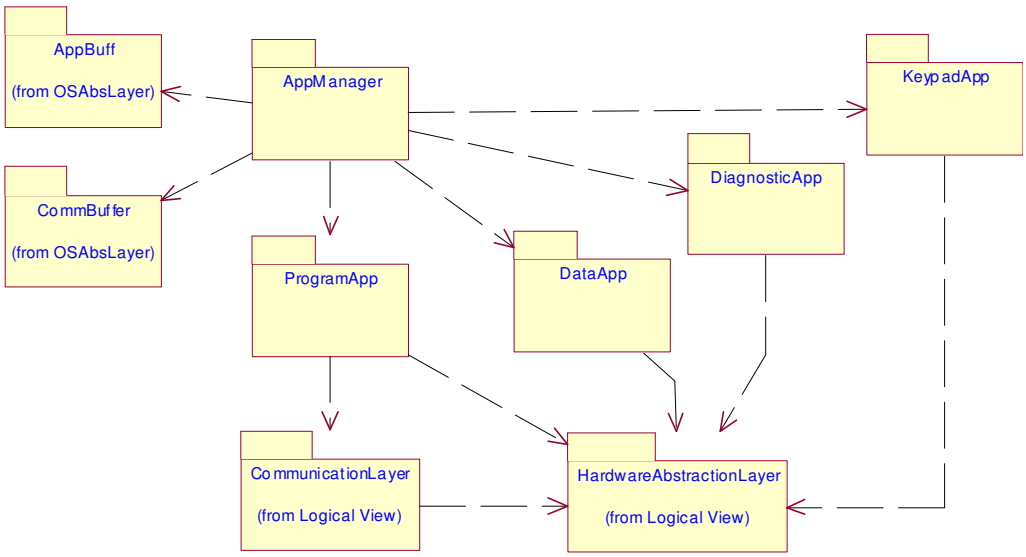

Figure 4: Functional decomposition of the Application Layer

3.5) Functional Decomposition of the HAL

Figure 5 shows the functional decomposition of the HAL using the software modules and class diagram in the UML. The HAL consists of a board support package (BSP) module containing functions to initialize the processor of the Device Programmer and all the hardware devices, including USB, RTC drivers etc. . The modules in the HAL communicate with the ApplicationLayer using the AppBuff, which is part of the ROSAL.

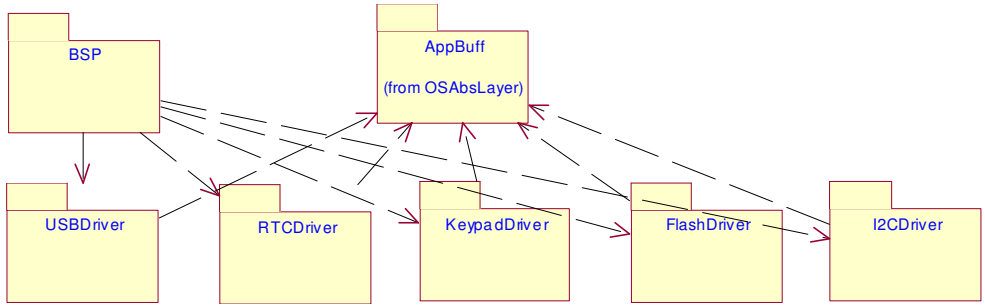

Figure 5: Functional decomposition of the HAL

3.6) Functional Decomposition of the Communication Layer

Figure 6 shows the functional decomposition of the Communication Layer. In this case, the development team has decided to outsource the TCPIP and USB software modules will not be developed internally. The modules are two of the commercial-off-the-self (COTS) components in the Device Programmer.

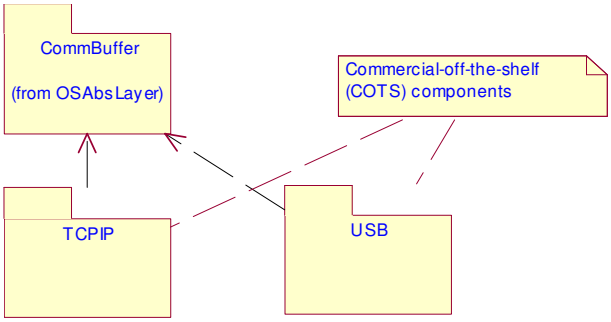

Figure 6: Functional decomposition of the Communication Layer

3.7) Scenario Analysis

Figure 7 shows the scenario analysis of the GetRemoteData use case to design the software architecture of the Device Programmer. The figure shows the messages between different functions that implicitly represent the relationship between the software modules.

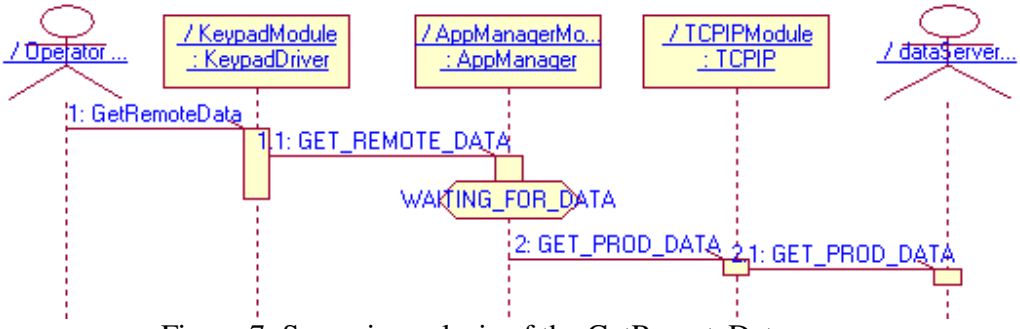

Figure 7: Scenario analysis of the GetRemoteData use case

4) Class Decomposition for Software Detail Design

With the completion functional decomposition using of some well-structured s software modules in UML, the object-oriented design and implementation using the class decomposition method is commenced.

4.1) Class Decomposition of the AppManager module

Figure 8 shows the class decomposition of the AppManager module into classes using the object-oriented design and programming method. The AppManager is decomposed into AppManagerInt, AppManager and AppMgrStateMachine classes with interface with the AppBuff and ProgramAppBuff classes. The AppManagerInt class is the interface class for the AppManager module. Other software modules execute the initialize() function in the AppManagerInt interface class to execute the Run() function in the AppManager class. The AppManagerInt, AppManager and AppMgrStateMachine are implemented following the Singleton design pattern. This should ensure that there is single instance for the classes AppManagerInt, AppManager and AppMgrStateMachine class.

The AppManager executes the Run() function that calls the GetMsg() function in the AppBuff class pending the availability of messages from other software layers. The AppManager calls the SendMsg() function of the AppMgrStateMachine class passing the message with an event. The AppMgrStateMachine consists of the logic to implement the state machine shown in Figure 9. The state machine consists of variables specifying an array of current states, next states, events and executing functions. For example, the initial state of the state machine is READY, and if the event AppBuff::PROGRAM_APP_ID occurs then the ProcProgApp() function will be executed. The ProcProgApp() function may pass the message to the ProgramApp module by executing the SendMsg() function. The existing state of the state machine is now in PROC_PROGRAM_APP_MSG. The event AppBuff::PROC_END will cause the state machine to execute the Null() function with its current state changes from PROC_PROGRAM_APP_MSG state to READY state.

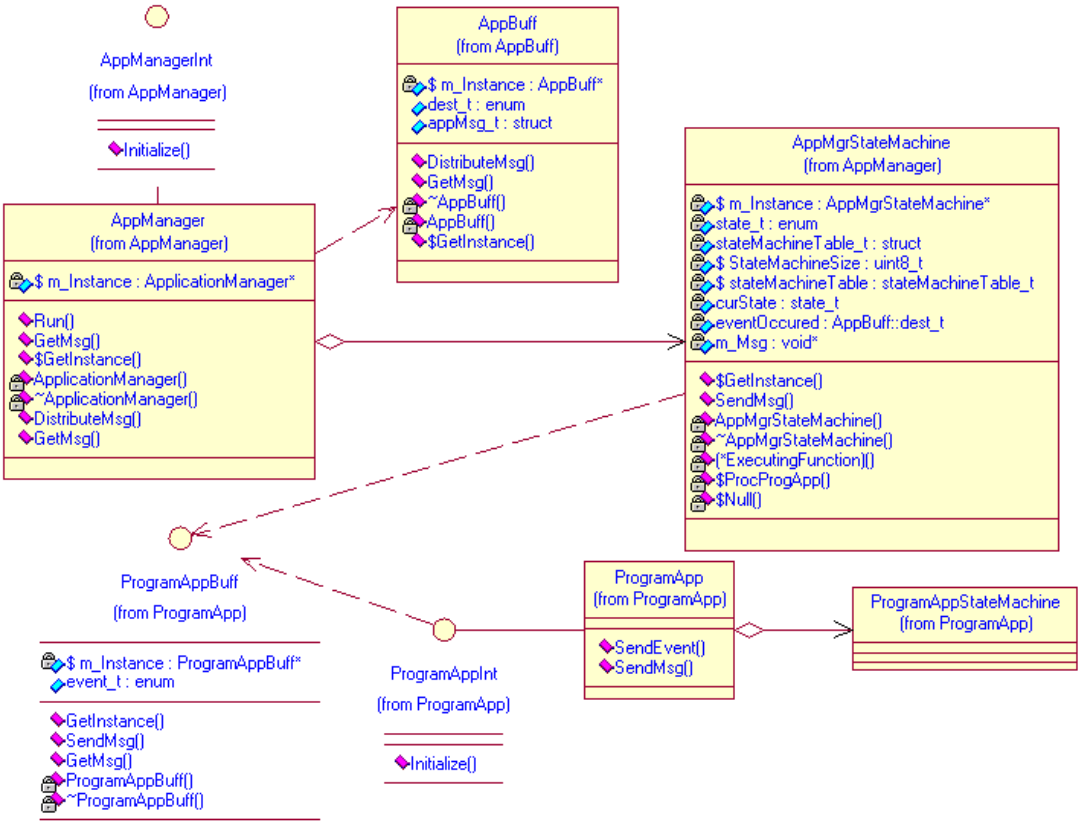

Figure 8: Class decomposition of the AppManager module

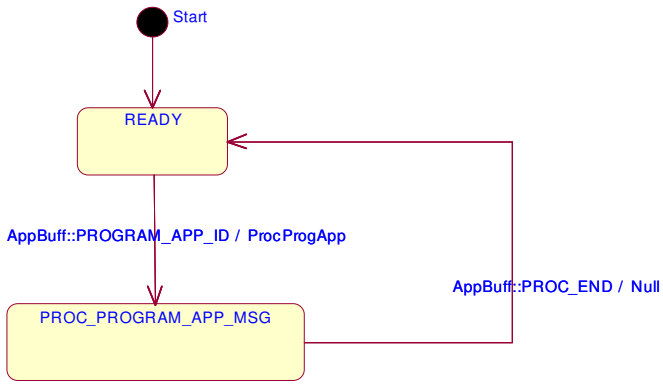

Figure 9: State diagram of the AppMgrStateMachine class

4.2) RTOS Abstraction Layer (ROSAL)

Figure 10 shows the class decomposition in the RTOS abstraction layer (ROSAL). The ROSAL uses AppBuff and CommBuff receive messages transmitted by the HAL and Communication Layer to the Application Layer. The OSAL class diagram also consists of the FlashOSServices class that provides an abstraction between OS specific functions with the FlashDriver.

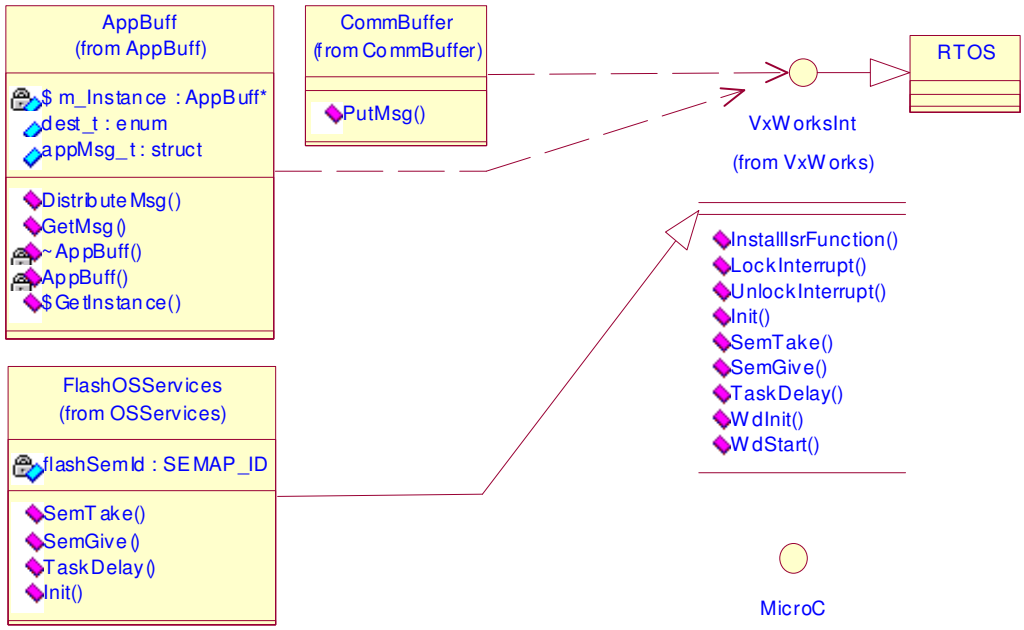

Figure 10: Class decomposition of the OS Abstraction Layer

4.3) Class Decomposition of the FlashDriver module

Figure 11 shows how to design the class diagram of the FlashDriver module achieving the aim of design for reuse and portability. The FlashDriver module provides three major operations to read, write and erase memory . The FlashInt class interfaces to the FlashDevice and is dependent on some OS services to synchronize the device driver states. The FlashDriver contains a Flash abstract class consisting of the ATACompactFlash and CfiFlash for Compact flash and Common Flash Interface (CFI) based flash respectively. The ATACompactFlash consists of ATA-based operations such as ATADrive() with variables initializing and specifying the specifications of a ATA-based Compact flash connected to the Device Programmer. The ATA-based commands are implemented by the ATACommand() function. However, the ATACompactFlash class does not contain any hardware related register information as these registers are declared and modified in the STR71ATAPort that directly manipulates the registers of the STR71-based controller. Therefore, FlashOSServices and STR71ATAPort are grouped under the OSServices and STR71Hardware module respectively easing the migration of the FlashDevice from STR71 hardware platform and a RTOS software platform to a new hardware and software platform.

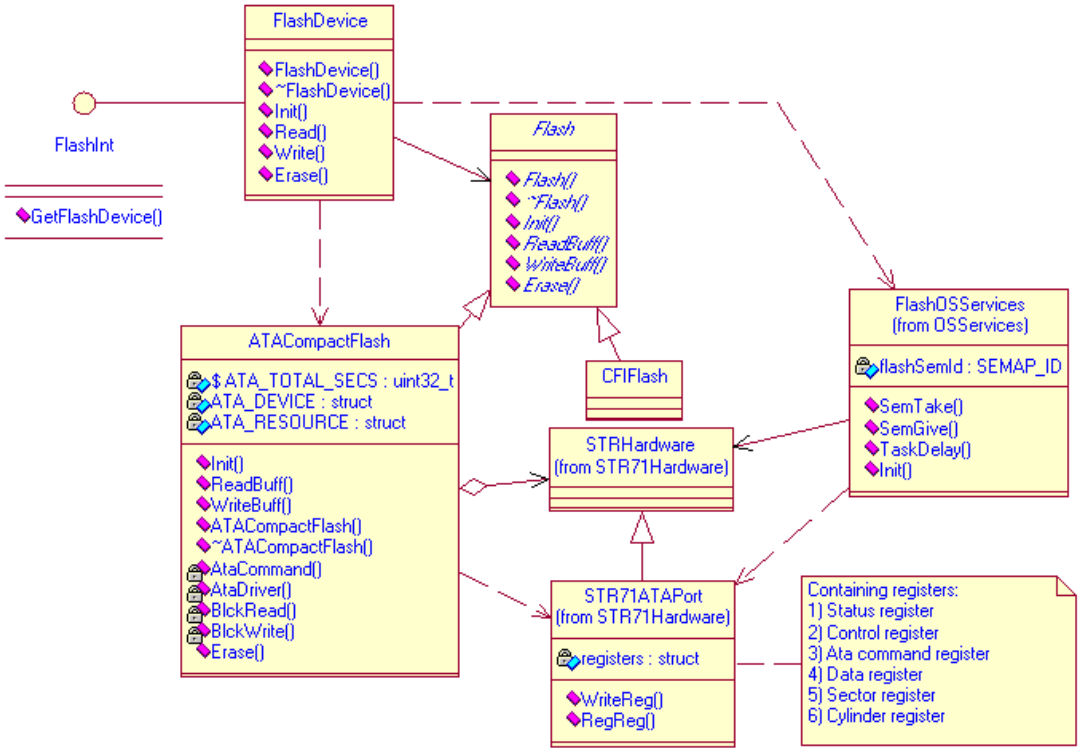

Figure 11: Class decomposition of the FlashDriver

5) Implementation for ApplicationManager and Device Drivers

This section shows how the class diagrams are implemented in the C++.

5.1) ApplicationLayer

5.1.1) C++ Implementation for the ApplicationManagerClass

Figure 12 is the header file for the ApplicationManager class based on the class diagram shown in Figure 8. It is implemented based on the Singleton design pattern ensuring that there is only one instance of the ApplicationManager class. Other software modules instantiate the ApplicationManager class using the APP_MANAGER that calls the ApplicationManager::GetInstance() returning the only instance of the ApplicationManager class. The source file for the ApplicationManager class is shown in Figure 13. The ApplicationManager class contains a state machine implemented as APP_MGR_SM. The ApplicationManager class executes the SendMsg() function to pass events to the APP_MGR_SM. The APP_MGR_SM will react to the event according to the basic theory of a state machine shown in Figure 9.

```
#define APP_MANAGER (ApplicationManager::GetInstance())

class ApplicationManager
{
public:
        static ApplicationManager* GetInstance();
        AppBuff::appMsg_t* GetMsg();
        void Run();

private:
        static ApplicationManager* m_Instance;
        ApplicationManager(void);
        ~ApplicationManager(void);
};
```

Figure 12: ApplicationManager.h

```
ApplicationManager* ApplicationManager::m_Instance = NULL;

ApplicationManager* ApplicationManager::GetInstance()
{
        if (m_Instance==NULL)
        {
                m_Instance = new ApplicationManager;
        }
        return m_Instance;
}

AppBuff::appMsg_t* ApplicationManager::GetMsg()
{
        return (APP_BUFF->GetMsg());
}

void ApplicationManager::Run()
{
        AppBuff::appMsg_t* appMsg;

        while (1)
        {
                appMsg = (AppBuff::appMsg_t*)GetMsg();
                APP_MGR_SM->SendMsg(appMsg);
        }
}
```

Figure 13: ApplicationManager.cpp

5.1.1) C++ Implementation of the state diagram for the ApplicationManager Class

Figure 14 shows the source code for the state machine of the ApplicationManager class. AppMgrStateMachine implementing the state machine shown in Figure 9 consists of:

- state_t containing all the states of the state machine
- stateMachineTable_t containing the variables of a state machine including the current states, events, next states and the functions to be executed.
- void (*ExecutingFunction)(void) is a function pointer executing a specific function based on the existing state and an incoming event. For example, stateMachineTable[] in Figure 15 shows that the ExecutingFunction points to the ProcProgApp() if the current state is READY and the event is AppBuff::PROGRAM_APP_ID.
- static uint8_t StateMachineSize is the size of the state machine on compiled time.
- static stateMachineTable_t stateMachineTable[] contains an array of current states, events, next states and functions forming the state machine.
- curState is the variable referring to the current state of the state machine.
- static void ProcProgApp(void) and static void Null(void) are two of the functions that will be executed by the state machine.

```
#define APP_MGR_SM AppMgrStateMachine::GetInstance()

class AppMgrStateMachine
{
public:
        static AppMgrStateMachine* GetInstance();
        void SendMsg(AppBuff::appMsg_t* appMsg);

private:
        AppMgrStateMachine(void);
        ~AppMgrStateMachine(void);
        static AppMgrStateMachine* m_Instance;

        typedef enum
        {
                READY,
                PROC_PROGRAM_APP_MSG,
        } state_t;

        typedef struct
        {
                state_t curState;
                AppBuff::dest_t event;
                state_t nextState;
                void* functionCall;
        } stateMachineTable_t;

        void (*ExecutingFunction)(void);
        static uint8_t stateMachineSize;
        static stateMachineTable_t stateMachineTable[];
        state_t curState;
        AppBuff::dest_t eventOccured;
        void* m_Msg;

        /* Function call for the state machine */
        static void ProcProgApp(void);
        static void Null(void);
};
```

Figure 14: AppMgrStateMachine.h

Figure 15 shows part of the source code of the AppMgrStateMachine class with GetInstance() function and destructor omitted from the code for simplification. The AppMgrStateMachine class contains stateMachineTable[] with an array of stateMachineTable_t. The stateMachineTable_t type contains data for current state, event, next state and function call. The stateMachineSize is a static function that changes according to the size of the array for stateMachineTable[]. The SendMsg() function runs the loop with two conditions, I) the run time value of the current state is stored in curState, II) telling the compiler that the loop begins at 0, and stops at stateMachineSize unless stateMachineTable[stateArray].curState in the state machine is equal to the current state in curState and the stateMachineTable[stateArray].event is equal to the event that just occurs.

In other words, the SendMsg() function attempts to match the event that it receives against the current state with its event of the state machine. If the event matches the state machine will change to a new state according to the Next State field in the state machine, and a function call is executed as the result.

```
AppMgrStateMachine::stateMachineTable_t AppMgrStateMachine::stateMachineTable[] =
{
/* Current State */             /* Event */                  * Next State */              /* Function Call */
{READY,                         AppBuff::PROGRAM_APP_ID,  PROC_PROGRAM_APP_MSG,(void*) ProcProgApp },
{PROC_PROG_APP_MSG,             AppBuff::PROC_END,          READY,                       (void*) Null        }
};

uint8_t AppMgrStateMachine::stateMachineSize
(sizeof(AppMgrStateMachine::stateMachineTable)/sizeof(AppMgrStateMachine::stateMachineTable_t));

AppMgrStateMachine::AppMgrStateMachine(void)
{
        curState = READY;
}

void AppMgrStateMachine::ProcProgApp(void)
{
        // Processing the ProgramApp
        PROG_APP_BUFF->SendMsg(APP_MGR_SM->m_Msg);
}

void AppMgrStateMachine::SendMsg(AppBuff::appMsg_t* appMsg)
{
        m_Msg = (void*)appMsg;
        eventOccured = appMsg->destID;

        for (uint8_t stateArray = 0; stateArray < stateMachineSize; stateArray++)
        {
                if (stateMachineTable[stateArray].curState == curState)
                {
                        if (stateMachineTable[stateArray].event == eventOccured)
                        {
                                curState = stateMachineTable[stateArray].nextState;

                                // Pointing to a function
                                ExecutingFunction = (void(*)(void)) stateMachineTable[stateArray].functionCall;

                                if (ExecutingFunction != NULL)
                                {
                                        // Executing a function call
                                        (*ExecutingFunction)();
                                }
```

Figure 15: Part of the AppMgrStateMachine.cpp

5.2) RTOS Abstraction Layer

Figure 16 shows the header file for the AppBuff acting as an OS abstraction layer between the Application Layer with the HAL. The AppBuff class consists of the following variables:

- DistributeMsg() function. Software layers interfacing to the Application Layer using this AppBuff class by executing the DistributeMsg() function. The DistributeMsg() function calls appropriate RTOS message passing functions passing messages to the buffer, which is then received by the ApplicationManager class.
- dest_t. The software layers indicate the destination of the message by specifying a specific destination, including ProgramApp module specified by dest_t.
- appMsg_t. The destID variable is the destination of the message, and the appMsgHeader contains specific command coding that specifies how a specific application (such as ProgApp etc.) should process the data of the message.

```
#define APP_BUFF AppBuff::GetInstance()

class AppBuff
{
public:

        typedef enum
        {
                PROGRAM_APP_ID,
                FILE_APP_ID,
                PROC_END
        } dest_t;

        typedef struct
        {
                dest_t destID;
                uint32_t appMsgHeader;
                uint32_t dataLength;
                uint8_t* data;
        } appMsg_t;

        static AppBuff* GetInstance();
        void DistributeMsg(dest_t destId, uint32_t dataLength, void* data);
        appMsg_t* GetMsg();

private:
        AppBuff(void);
        ~AppBuff(void);
        static AppBuff* m_Instance;
};
```

Figure 16: AppBuff.cpp

5.3) FlashDriver

Figure 17 shows the header file of the FlashDriver shown in Figure 11. The header consists of a flashDevice_t for the ATA compact flash and CFI-based flash. The GetFlashDevice() function in the FlashDevice class accepts parameter passing from external class, and initialize and returns a FlashDevice instance referring to the ATA_PPC405_FLASH and CFI_ALTERA_FLASH respectively. The external class will be able to run code referring to the type of flash it initializes.

```
#define FLASH_INT FlashInt::GetInstance()

class FlashInt
{
public:

        typedef enum {
                ATA_PPC405_FLASH,
                CFI_ALTERA_FLASH
        } flashDevice_t;

        FlashDevice* GetFlashDevice(flashDevice_t flashDevice);
        FlashInt* GetInstance();

private:
        FlashInt(void);
        ~FlashInt(void);
};
```

Figure 17: FlashInterface.cpp

```
FlashDevice* FlashInt::GetFlashDevice(flashDevice_t flashDevice)
{
        FlashDevice* m_flash = NULL;
        switch (flashDevice)
        {
                case FlashInt::ATA_PPC405_FLASH:        // Initialize ATA-based Compact flash
                {
                m_flash = new FlashDevice(flashDevice);
                break;
                }
                case FlashInt::CFI_ALTERA_FLASH:        // Initialize Cfi-based flash
                {
                m_flash = new FlashDevice(flashDevice);
                break;
                }
                default:
                m_flash = NULL;
                break;
        }
        return m_flash;
}
```

Figure 18: GetFlashDevice Function

6) Conclusion and Future Work

The function-class decomposition method based on UML presents a top down and bottom up approach to embedded software design. This design method has fulfilled many design requirements as follows:

i) Design for test. The FCD approach allows the application of a systematic approach to component and system integration test. For example, an application can read message passing in the ROSAL and state machine to check that the system is behaving according to the state diagram in Figure 9 and sequence diagram in Figure 7.

ii) Design for distributed development. The functional decomposition results in a well-defined software architecture, and well-delineated software modules that would allow an easier distributed development. For example, different

development teams can develop different software modules according to sequence diagrams showing the inter-module relationship, and using the state diagram to develop correct software modules.

iii) Design for maintainability. UML diagrams could be developed in a software tool that supports model-driven software development, thus ensuring that the software design models are in synchronized with the software implementation.

iv) Design for reuse. The FCD approach groups design models that are software platform dependent into the ROSAL, and design models that are hardware platform dependent into the HAL. Thus ensuring that most of the software modules are hardware and platform independent allowing an effective way for software reuse. UML and model-driven software development are also provides an infrastructure for better software reuse.

v) Design for change management. The flexible software architecture and OO implementation of state diagram provide a flexible software structure for change management including adding new states to state machines, adding new applications to the ApplicationLayer etc.

The systematic large-scale software decomposition has fulfilled many design requirements as stated above. Future work should include examples how to use tools to integrate these design for X factors into the software design life cycle. For example, it is possible to use sequence diagrams and state machines to write software module and component test cases, and either automatically or manually compare the test results executed in an actual target with these diagrams. Furthermore, it is also possible to analyze inter-module messages and state machine events in the target verifying that the product satisfies the requirements according to the specification in the sequence and state diagrams.

7) References

[1] A. Goldberg and A. Kay. Smalltalk 72 Instruction Manual. Xerox PARC, 1976.

[2] G. Booch. Object oriented development. IEEE Transactions on Software Engineering, SE-12(2):211-221, February 1986.

[3] M. Veeraraghavan and T. F. La Porta. Object-oriented analysis of signaling and control in broadband networks. International Journal of Communication Systems, 7(2):131 147, April 1994.

[4] R. J. Allen, A Formal Approach to Software Architecture (1997), IFIP Congress, Vol. 1.

[5] Len Bass, Rick Kazman, Paul Clements, Software Architecture in Practice, Addison-Wesley Professional, 2003.

[6] Alexander Kossiakoff, William N. Sweet, Systems Engineering Principles and Practice, Wiley-IEEE, 2002.

[7] Carl K. Chang, Jane Cleland-Haung etc., Function-Class Decomposition: A Hybrid Software Engineering Method, Computer, Volume 34, Issue 12 (December 2001), Pages: 87 – 93, IEEE Press.