Embedded Software Organization: Architecture and Design

Salvador Almanza-Garcia
Special Projects - Embedded Software
Vector CANtech, Inc., Novi MI, USA
IEEE SEM GOLD Vice-Chair
Objective

- To introduce basic concepts and examples of embedded software organization, from project planning, project structure, architecture and design

Note:

- The present material is intended for the audience attending the embedded systems workshop at Oakland University (mainly students). The content respect to methodology and/or source code is based on Author previous experience and current projects related to academics; it is not related and/or part of Vector CANTech Inc. products and/or tools.
Introduction
Definitions

Organization [From Dictionary]

- Something made up of elements with varied functions that contribute to the whole and to collective functions; an organism
- A structure through which individuals cooperate systematically to conduct business

Embedded Software Organization

- Planning methodology to create a coherent and effective structure in a software project, by categorizing different software components according to their specific characteristics, allowing to construct software systems that are reusable and portable through different hardware platforms and applications
Factors driving Embedded Systems Development:

- Demanding and intensive requirements
- Hardware complexity
- Reduced development times; products released to market as soon as possible
- Cost
- At no least important... Experience
Why Software Organization?

- The increasing complexity of system requirements as consequence of technology advancements in semiconductor industry

- Complex requirements critically impact the product life cycle. It is difficult to satisfy time-to-market demands (reduce development time and cost)

- Optimize and speed-up software development, without compromising safety, robustness and quality of the software components

- Improve software component reusability through multiple hardware platforms
Why Software Organization?

Embedded Systems Development Facts:

- Development time and engineering cost (i.e. NRE) become a problem, and the product falls behind schedule during the whole product life cycle.

- Industry recognizes the importance of making an effort to optimize and speed-up software development, without compromise robustness and quality of the software components created to fit into a specific software architecture.

- Software design and development are critical factors directly related to cost, time to market and the success of an embedded product.

- The increasing complexity of embedded systems implies an increase in specialization of function in the design team [Ganssle, 2008].
Embedded Software Architecture
More definitions…

Software Architecture

- The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. [Bass, Clements & Kazman, 2003]

- Collection of software components that follows an organized structure, and describes the overall system and it components behavior from a high-level design perspective

Embedded Software Architecture

- Structure and organization of multiple software components through different abstraction layers that intend to provide hardware independence, maximizes code reusability and propagates component behaviors, between multiple platforms of purpose-specific embedded computers
Abstraction

- Simplified view of a system containing only the details important for a particular purpose [Berzins & Luqi, 1991]

Embedded Software Abstraction

- Design methodology used to hide hardware architecture details from the application software domain by the isolation and encapsulation of relevant parameters that describe the behavior of an specific hardware entity, in order to facilitate software component reusability and portability

Software Component

- In software system, a software component is an entity with well defined behavior and interacts with other components and modules within the system
More definitions...

Software Interface

- A mechanism used by a software component or module to interact with the external world (i.e., analog/digital signals, RF) and other software components.

Coupling

- Degree of dependency between different software components within a system.

Cohesion

- Measures the degree of relationship between elements within a software component.
“All architecture is design, but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change” [Grady Booch]
Example of Layered Architecture

- Application
- EIFSW-IF
- EMBEDDED INFRA-STRUCTURE SOFTWARE (EISW)
- MCU Hardware
Python

Example of Layered Architecture

- EISW-IF
- EISW
- OS
- SYS MANAGEMENT
- MIDDLEWARE
- MIDDLEWARE LIBS
- HARDWARE ABSTRACTION
- MCU PERIPHERAL DRIVERS
- MCU HARDWARE
- APPLICATION
- EXTERNAL DRIVERS

© 2012. Vector CANtech, Inc.. All rights reserved. Any distribution or copying is subject to prior written approval by Vector.

Slide:
Software Layers Description

MCU Peripheral Drivers

- Internal device drivers
- Hardware access to MCU peripherals
- Provide MCU low-level abstraction
  - Hardware dependence is high, therefore, reuse is limited at this level
- Provide standard interfaces used by abstraction, OS and external driver layers
Software Layers Description

Hardware Abstraction Layer (HAL)

- Provides access to MCU hardware features through peripheral interfaces
- Hides hardware details not relevant to upper software layers
- Interfaces with MCU and external drivers in the low level side, and with HAL signal interface at the upper side
Middleware and System Management

- Facilitate the interaction between application components and other modules and/or components within the system:
  - Graphics Library
  - Networking
  - File Systems
  - Databases
  - Other Middleware components, i.e., off-the-shelf components

- Provides system management
  - Power Management
  - Memory management
  - Diagnostics

- Due to overhead, it is an optional layer
External Drivers

- Implements direct hardware access to external devices through MCU peripherals

- Meet all functional and timing requirements of the external devices

- Examples:
  - EEPROM (I2C™, SPI™, Microwire™, etc)
  - External ADCs (i.e. Delta-Sigma high-resolution converters)
  - Sensors and actuators
  - System Basis Chip (SBC)
  - I/O Driver ICs
Software Layers Description

Embedded Infrastructure SW Interface

- Provides to the application one more level of abstraction and hardware independence
- Translates logical signals into a meaningful format for the application
- Facilitates the communication between application software components and/or lower layer modules
- It is application specific
- Due to overhead, it is an optional layer
Software Layers Description

Application Layer

- Product specific functions
- Contains the software components that implements the desired functionality (unique) for a specific embedded computer system
- A high-level design methodology ignores the details of the hardware
- Reusability of application components strongly depends in the availability and efficiency of lower layers
Software Layers Description

OS

- Provides support for multi-tasking

- Task scheduling and synchronization

- If real-time OS (RTOS)
  - Context switching
  - Task preemption
  - Interrupt management
Software Layered Architecture

Advantages:

- Organized and modular design

- If properly applied, an architectural approach allows and facilitate distributed development; software components being developed by different teams or COTS from third parties (modular and scalable)

- Once a software architecture has evolved reaching an optimal level of maturity, the development process can be benefited by reducing development time and cost.

- Well defined architectures facilitate the usage of more advanced development techniques and tools, i.e., Model Based and Code Generation
Software Layered Architecture

Disadvantages:

- Modular layered software architectures and abstraction can consume significant resources in an embedded system in terms of memory and performance:
  - From few kilobytes of ROM/RAM to the order of several megabytes
  - From tenths of MHz to hundreds of MHz (even GHz)

- Transitioning from traditional embedded software development into a layered software architecture, can result in a large learning curve:
  - Adopting a new design and implementation methodology
  - Learning new tools

- Initially, the adoption of software layered architectures may result in a spike in cost and development time, making difficult its acceptance
Software Architecture Structure

Directory Structure (Simplified View)
Example - Dome Light Control: Software Architecture and Design

The Following example is for illustration purposes only, and to understand the presented concepts.
Example – Dome Light Control (Overview)

Software Layers Organization

*Door Monitor*  *Key Monitor*  *Battery Voltage Monitor*  *Dome Light Control*

EISW-IF

*OS*  *CAN COMM. Management*  *HwAbDiIn*  *HwAbPwm*  *HwAbAln*  *Dout IC Drvr*

CanDrv  CodDrv  McDrv  TimerDrv  AdDrv  SpiDrv

**MCU HARDWARE**

*CAN Trcvr*  *Drv Door Switch*  *Pass Door Switch*  *Batt. Voltage Sign. Cond.*  *Dome Light Hw Drv*

*Dome Light Bulb/LED*
Example – Dome Light Control (Overview)

Directory Structure
Example – Dome Light Control (Overview)

Example of Directory Structure (simplified)
Example – Dome Light Control (Overview)

Directory Structure

```
- AdcDrv
  - src
    - AdcDrv.c
  - inc
    - AdcDrv.h
  - doc
    - SDD_AdcDrv_Rev1p0.pdf

- CanDrv
  - src
    - CanDrv.c
  - inc
    - CanDrv.h
  - doc
    - SDD_CanDrv_Rev1p0.pdf

- GpioDrv
  - src
    - GpioDrv.c
  - inc
    - GpioDrv.h
  - doc
    - SDD_GpioDrv_Rev1p0.pdf
```
Example – Dome Light Control (Overview)

Directory Structure

- **McuDrv**
  - src
    - McuDrv.c
  - inc
    - McuDrv.h
  - doc
    - SDD_McuDrv_Rev1p0.pdf

- **TimerDrv**
  - src
    - TimerDrv.c
  - inc
    - TimerDrv.h
  - doc
    - SDD_TimerDrv_Rev1p0.pdf

- **HAL**
  - src
    - HwAbDin.c
    - HwAbDout.c
    - HwAbAin.c
    - HwPwm.c
  - inc
    - HwAbDin.h
    - HwAbDout.h
    - HwAbAin.h
    - HwPwm.h
  - doc
    - SDD_HALDIn_Rev1p0.pdf
    - SDD_HALDout_Rev1p0.pdf
    - SDD_HALDAin_Rev1p0.pdf
    - SDD_HALPwm_Rev1p0.pdf

© 2012. Vector CANtech, Inc.. All rights reserved. Any distribution or copying is subject to prior written approval by Vector.
Example – Dome Light Control (Overview)

Directory Structure

- **CanComm Mngmnt**
  - **src**
    - CanCommMngr.c
    - CanDiagMngr.c
  - **inc**
    - CanCommMngr.c
    - CanDiagMngr.c
  - **doc**
    - SDD_CanCommMngr_Rev1p0.pdf
    - SDD_CanDiagMngr_Rev1p0.pdf

- **OS**
  - **src**
    - Sched.c
  - **inc**
    - Sched.h
  - **doc**
    - SDD_Sched_Rev1p0.pdf

- **DoutIcDrv**
  - **src**
    - DoutICDrv.c
  - **inc**
    - DoutICDrv.h
  - **doc**
    - SDD_DoutICDrv_Rev1p0.pdf
Example – Dome Light Control (Overview)

Directory Structure

EISWIF
- src
  - inc
    - EISWIF_DomeLightCtrl.c
    - EISWIF_DoorMonitor.c
    - EISWIF_KeyMonitor.c
    - EISWIF_BattVoltMonitor.c
  - doc
    - SDD_EISWIF_DomeLightCtrl_Rev1p0.pdf
    - SDD_EISWIF_DoorMonitor_Rev1p0.pdf
    - SDD_EISWIF_KeyMonitor_Rev1p0.pdf
    - SDD_EISWIF_BattVoltMonitor_Rev1p0.pdf

DLC
- src
  - inc
    - App_DomeLightCtrl.c
    - App_DoorMonitor.c
    - App_KeyMonitor.c
    - App_BattVoltMonitor.c
  - doc
    - SDD_DomeLightCtrl_Rev1p0.pdf
    - SDD_DoorMonitor_Rev1p0.pdf
    - SDD_KeyMonitor_Rev1p0.pdf
    - SDD_BattVoltMonitor_Rev1p0.pdf
Directory Structure
Directory Structure
Example – Dome Light Control (Overview)

Software Component Structure

<table>
<thead>
<tr>
<th>Software Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>swc.c</td>
</tr>
<tr>
<td>swc.h</td>
</tr>
<tr>
<td>swcCfg.c</td>
</tr>
<tr>
<td>swcCfg.h</td>
</tr>
</tbody>
</table>
Example – Dome Light Control (Overview)

Software Component Structure

```
SCW1

swc1.c

swc1.h

swc1_cfg.c

swc1_cfg.h

SCW2

swc2.c

swc2.h

swc2_cfg.c

swc2_cfg.h
```
Example – Dome Light Control (Overview)

Component Interaction

DomeLightControl

Interface

- CanCommMngr
  - CanRxMSg
  - CanTxMSg
  - Std Interface
  - HwAbCommIf
    - CanDrv

- DoorMonitor
  - DrvDoorSwitchState
  - PassDoorSwitchState

- KeyMonitor
  - KeySwitchState

- BattVoltageMonitor
  - BattVoltageState

- DomeLightBrightness

Interface

- DoutICDrv

Interface

- BattVoltage

Interface

- BattVoltageRaw

Interface

- SpiDomeLightCtrlICmd

MCU Peripherals

- CAN Trcvr
- Din Hw Signal Conditioning
- DoutICPwrDrv
- Drv Door Switch
- Pass Door Switch
- Dome Light Bulb/LED
Example – Dome Light Control (Overview)

Door Monitor Component Interaction
Example – Dome Light Control (Overview)

Key Monitor Component Interaction

```
KeyMonitor
  EISWIF_readDrvDoorSwitch()
  ↓
EISWIF_KeyMonitor
  HAL_readDin(KEYSTATE)
  ↓
HwAbDin
  GPIO_readDin(port, pin)
  ↓
GpioDrv
  ↓
GPIO DINs
```

© 2012. Vector CANtech, Inc.. All rights reserved. Any distribution or copying is subject to prior written approval by Vector.
Example – Dome Light Control (Overview)

Battery Voltage Monitor Component Interaction

```
BatteryVoltageMonitor
  EISWIF_readBatteryVoltageState()

EISWIF_BatteryVoltageMonitor
  HAL_readAin(BATT_VOLT)

HwAbAin
  Adc_readAin(channel)

AdcDrv

A1Ns
```
Dome Light Control Component Behavior
Example – Dome Light Control (Overview)

Dome Light Control Component Behavior

stm DomeLightControl

Initial
T0 [Power Mode == OFF]

DOME_LIGHT_NORMAL_OPERATION

DOME_LIGHT_OFF

TF [DomeLightBrightness <= CaDomeLightBrightnessMin]

TB [DomeLightOnRequest == LIGHT_OFF_REQUEST]

TA [DomeLightOnRequest == (LIGHT_ON_DOOR_REQUEST OR LIGHT_ON_KEYOUT_REQUEST)]

DOME_LIGHT_BRIGHTNESS_CONTROL

DOME_LIGHT_ON

TC [CaDlcBrightnessControlEnabled == TRUE]

DOME_LIGHT_PROTECTION

DOME_LIGHT_UNDERRIGHTURE

TM [DomeLightProtectionRequests == NO_PROTECT]

TG [DomeLightProtectionRequests == UNDER_VOLTAGE_PROTECT]

TK [DomeLightProtectionRequests == OVER_VOLTAGE_PROTECT]

DOME_LIGHT_OVERVOLTAGE

TL [DomeLightProtectionRequests == UNDER_VOLTAGE_PROTECT]

TJ [DomeLightProtectionRequests == OVER_VOLTAGE_PROTECT]

T1 [Power Mode != OFF]

Final
Example – Dome Light Control (Overview)

Dome Light Control Component Behavior
Dome Light Control Component Behavior

stm DoorMonitor

DOOR_LIGHT_OFF_REQUEST

DOOR_LIGHT_ON_REQUEST

T0 [Power Mode == OFF]

TB [DoorClosedTimer > 0 AND All Doors are CLOSED]

TC [DoorClosedTimer Expired OR VehicleSpeed > CaVehicleSpeedMax]

TA [Any Door Transition from CLOSED to OPEN AND VehicleSpeed <= CaVehicleSpeedMin]

TC [Any Door is OPEN]

T1 [Power Mode != OFF]

Final
Dome Light Control Component Behavior

DomeLightVoltageMonitor

Initial

T0 [Power Mode == OFF]

DOME_LIGHT_NO_PROTECT

TA [BatteryVoltageStatus == UNDER_VOLTAGE]

DOME_LIGHT_UNDER_VOLTAGE_PROTECT

TB [BatteryVoltageStatus == NORMAL_VOLTAGE]

DOME_LIGHT_NORMAL_VOLTAGE_PROTECT

TC [BatteryVoltageStatus == OVER_VOLTAGE]

TE [BatteryVoltageStatus == OVER_VOLTAGE]

TF [BatteryVoltageStatus == UNDER_VOLTAGE]

DOME_LIGHT_OVER_VOLTAGE_PROTECT

T0 [Power Mode == OFF]

[Power Mode != OFF]

Final
Example – Dome Light Control (Overview)

Door Monitor Component Behavior

stm CpSaFrontDoorMonitor
Drv DoorMonitor

Initial

[Init]

DRV_DOOR_CLOSED

TA [FrontDrvDoorSwitchState == SWITCH_OPEN]

TC [FrontDrvDoorSwitchState == SWITCH_CLOSED]

TA [FrontDrvDoorSwitchState == SWITCH_OPEN]

DRV_DOOR_OPEN

Initial

[Init]

Initial

[Init]

PASS_DOOR_CLOSED

TA [FrontPassDoorSwitchState == SWITCH_OPEN]

TC [FrontPassDoorSwitchState == SWITCH_CLOSED]

TA [FrontPassDoorSwitchState == SWITCH_OPEN]

PASS_DOOR_OPEN

Final

Final

© 2012. Vector CANtech, Inc.. All rights reserved. Any distribution or copying is subject to prior written approval by Vector.

Slide:
Example – Dome Light Control (Overview)

Door Monitor Component Behavior

**Diagram Description:**
- **Initial State:** T0 [From Any State]
- **Primary States:**
  - **DOOR SWITCH SCAN**
  - **DOOR SWITCH DEBOUNCE**
  - **DOOR SWITCH STATE UPDATE**
- **Transitions:**
  - TB [Door Switch Transition Detected]
  - TC [Door Switch Transition Invalid]
  - TG [Update Completed AND Door Switch State Not Changed]
  - TF [Door Switch Transition Detected]
  - TE [Door Switch Transition Valid]
  - TD [Decrement DoorSwitchDebounceTimer]
- **Final State:**
Key Monitor Component Behavior

Diagram:

- **stm CpSaKeyMonitor**
- **Key Monitor**
- **Initial**
- **KEY_OUT_POSITION**
- **KEY_IN_POSITION**
- **TA** [Key State transition to IN Position]
- **TC** [Key State transition to OUT Position]
- **Final**
Example – Dome Light Control (Overview)

Key Monitor Component Behavior

stm KeyInputDebounce

Initial

T0 [From Any State]

KEY_INPUT_SCAN

TA [Scanning For Key Input Transitions]

TC [Key Input Transition Invalid]

KEY_INPUT_DEBOUNCE

TB [Key Input Transition Detected]

KEY_INPUT_STATE_UPDATE

TG [Update Completed AND Key Input State Not Changed]

TF [Key Input Transition Detected]

TE [Key Input Transition Valid]

TD [Decrement KeyInputDebounceTimer]

Final
Battery Voltage Monitor Component Behavior

stm CpApBatteryVoltageMonitor

Initial

BATTERY_NORMAL_VOLTAGE

BATTERY_UNDER_VOLTAGE

BATTERY_OVER_VOLTAGE

Final

TA [BatteryVoltage < CaBatteryVoltageMin]

TB [(BatteryVoltage >= CaBatteryVoltageMin) AND (BatteryVoltage <= CaBatteryVoltageMax)]

TC [BatteryVoltage > CaBatteryVoltageMax]

TE [BatteryVoltage > CaBatteryVoltageMax]

TF [BatteryVoltage < CaBatteryVoltageMin]
So... all this to turn-on a light???
Conclusions

- The embedded world requires more sophisticated design methodologies that can satisfy current market demands; it is important to develop robust architectures (HW/SW) that allow more efficient and low cost implementations.

- The even more exigent and complex requirements trigger the advances of semiconductor industry, providing more powerful processors; therefore, software development becomes a more complex task.

- The evolvement of software requires the application of more advanced software engineering concepts.

- Embedded Software development is not a trivial task; a new culture of development is emerging and requires immediate attention.
Conclusions

- The embedded world requires more sophisticated design methodologies that can satisfy current market demands.

- The even more exigent and complex requirements trigger the advances of semiconductor industry, providing more powerful processors; therefore, software becomes more complex as well.

- The evolvement of software requires the application of more advanced software engineering concepts.

- Embedded Software development is not the task of electrical engineers anymore; neither pure software engineers; new culture of development is emerging.
Thank you... Any question?
Thank you for your attention.

For detailed information about Vector and our products please have a look at:

www.vector.com

Author:

Salvador Almanza, Special Projects(PES)
salvador.almanza@vector.com

Vector CANtech, Inc.