Code Development Strategies

- Given: Problem Statement
- First step: Problem Specification
  - Text description: English, pseudocode
  - Graphical description: flowchart
- Next step: Code Development
  - Top-down
  - Bottom-up
  - Language selection

Coding Case Study – *CheckPressure*

Problem Statement:

Write a function named CheckPressure having two parameters: the current pressure of a system and the maximum pressure. Pressure is a 16-bit unsigned value. Current pressure is passed by value; maximum pressure, by reference. If the difference between the current pressure and maximum pressure is greater than PRESSURE_WARN, update system status and data registers by clearing bit 2 of an 8-bit port at 0x1038 and writing the difference to a 16-bit port at 0x1036.
Coding Strategy

Given the problem statement, the first step as an embedded programmer, or embedded software developer/designer (ESD), is to refine your level of understanding.

What is the context of the problem?
What operations are needed?
What is the sequencing of the operations?
What data will be operated on?
What are the types and sources of data?
What input/output is used?

Coding Strategy

You need to answer the questions and represent, or describe, the problem more completely.

A functional specification of the problem describes what the system does: the control flow, data flow, data attributes, modules and other behavioral characteristics of the system being designed (in this case, embedded software). For example:

- Text description: English, pseudocode
- Graphical description: flowchart, diagrams
Coding Strategy – Example

Specification example: Washing Machine

Suppose you are designing the embedded software for a washing machine. The next two pages show how you might begin to describe the system. You could make a list of what the system does. You could sketch a diagram that shows the interfaces between the program and the environment, i.e., the I/O. This is called the context for the program.

From Pont, Patterns for Time-Triggered Embedded Systems, Addison-Wesley and ACM Press, 2001

Coding Strategy – Example

 Specification example: Washing Machine

Text description of system operations and use:

1. The user fills the machine with dirty laundry and locks the door.
2. The user fills the detergent tank with appropriate detergent.
3. The user selects a wash program (e.g. ‘Wool’, ‘Cotton’) on the selector dial.
4. The user presses the ‘Start’ switch.
5. The door lock is engaged.
6. The water valve is opened to allow water into the wash drum.
7. If the wash program involves detergent, the detergent hatch is opened. When the detergent has been released, the detergent hatch is closed.
8. When the ‘full water level’ is sensed, the water valve is closed.
9. If the wash program involves warm water, the water heater is switched on. When the water reaches the correct temperature, the water heater is switched off.
10. The washer motor is turned on to rotate the drum. The motor then goes through a series of movements, both forward and reverse (at various speeds) to wash the clothes. (The precise set of movements carried out depends on the wash program that the user has selected.) At the end of the wash cycle, the motor is stopped.
11. The pump is switched on to drain the drum. When the drum is empty, the pump is switched off.
Coding Strategy – Example

Specification example: Washing Machine

Context diagram showing main process and I/O

The next page shows another level of detail about the system’s behavior. It uses a process-oriented graphical representation called a dataflow diagram. This is a common approach for embedded programs, in part because process-oriented languages such as C are commonly used.

The bubbles shows actions performed by the system, and the arcs show inputs and outputs to/from the actions, either data or control/status signals.

You are not expected to use this exact notation. That’s something you might learn more about later in a software engineering course. The purpose here is to illustrate possible approaches to specification.

There are other types of diagrams that represent aspects of the system, such as a state transition diagram. The point is that you need to develop and refine your understanding of the system behavior before you start with actual coding.
Coding Strategy – Example

**Specification example:**
Washing Machine

Dataflow diagram (DFD) showing actions and commands, data

Coding Case Study – **CheckPressure**

Problem Specification: Context Diagram
This is an elaboration on the problem statement.
Coding Case Study – CheckPressure

Problem Specification: Dataflow Diagram
This focuses on the action CheckPressure.

[Diagram of CheckPressure dataflow]

Problem Specification: Flowchart

[Flowchart of CheckPressure]
Assume that the caller is part of the Valve Control Process. Thus, CheckPressure is one of many functions in the program.
Coding Strategy

Once you have a better understanding of the problem, the next step is to get started with actual coding. However, you need to approach the coding with a plan in mind. Part of your code development plan should involve looking ahead to how you will test the code. With CodeWarrior, you have an excellent environment in which to test and debug your embedded programs. There are two common approaches to developing and testing a program:

- Top down
- Bottom up

Coding Strategy

Top down

- Start with the “big picture” of your program.
- What actions need to be performed by the main program?
- What is the flow of control for the actions, e.g., repetition, selection, sequencing, etc.?
  - Is there a main loop that repeatedly calls top-level actions?
- What triggers/controls the execution of the actions, or tasks?
  - For example: time, events, user input/output
- What global data are used by the tasks?
  - Is there a system mode or global state for all tasks, or does each task maintain its own state?
- For each task, should it be broken down into smaller subtasks?
  - For example: stages, states, or functions
Coding Strategy

Bottom up

- Start with individual components in the system, such as leaf functions or I/O devices.
- What does a function or device do, as defined by its specifications?
- What does a function or device need to do, as defined by the problem?
- How is the component used? What is its interface?
  - Who calls a function? What is the function’s prototype?
  - Who uses a device? What is the device’s programming model (registers, modes of operation, timing, etc.)?
- What algorithms and data structures are needed to use the function or device?
- How does the function/device map to system resources?

Coding Strategy

In most cases, you will use a combination of top-down and bottom-up approaches to develop and test your code. You will often start by thinking about the overall organization and flow of your code. Then you may turn your attention to the implementation of specific parts of the code.

For example, you might start by writing a main program that repeatedly calls a sequence of tasks, with each task using a certain time-slice. Taking the top-down approach, you might defer writing code for each task until later, and instead focus on getting the logic and timing of the main loop correct first. Thus, the tasks could be essentially empty to start with (i.e., dummy or stub routines).

Note that you could begin to write the task code as well, but it could be developed and tested separately.
Coding Strategy

Next, you might decide to focus on a particular device and develop the code to use that device. In this bottom-up approach, you will temporarily put aside the larger program, except for your understanding of how it uses the device, and write functions for the device, such as Init_Device, Read_Device, Write_Device, etc. You will make specific decisions about the implementation of the software, such as algorithms, variables, etc. You will debug this code to make sure that it works correctly.

Note that you could begin to write the top-level code as well as other functions, but these could be developed and tested separately.

The combined top-down/bottom-up strategy can be applied at different levels in the program. For example, any function with some amount of complexity might be developed and tested in a top-down/bottom-up fashion.

---

Coding Case Study – CheckPressure

Code Development and Test:

Top-down → Bottom-up

Main → Task1 → CheckPressure

... → TaskN → ...
Coding Case Study – ChecksPressure

Code Development and Test:

Top-down

In this example, we assume that a main program is written using a “super-loop” strategy:

- A main loop calls tasks or functions based on time, state, or events
- CheckPressure is called directly or indirectly (by calling a function that calls it)

The program is written completely in C. Then, as needed, parts of the program may be written in assembly.

Bottom-up

The design problem being focused on is the CheckPressure function. It should be coded as follows:

- Document the context
- Identify any coding rules or conventions for the target software
- Write the function in C
- Test the C function
- Convert the C to assembly, in segments or in full, depending on code complexity
- Test the assembly function
Coding Case Study –
CheckPressure

Code Development and Test:

Document the context
The context is determined by the problem statement and specs.

• The function receives Pressure Level and Max Pressure data via parameters and uses a global constant Threshold.
• The function conditionally writes Warning Status and Valve Setting data to memory-mapped status and data ports.

In C:

function prototype:
void CheckPressure (unsigned short sLevel, unsigned short * psMax)

global constants:
#define PRESSURE_WARN 1000
#define STATUS_PORT 0x00001038
#define DATA_PORT 0x00001036

Coding Case Study –
CheckPressure

Code Development and Test:
Let's continue with C coding, for now using no platform-specific coding rules.

#define PRESSURE_WARN 1000
#define STATUS_PORT 0x00001038
#define DATA_PORT 0x00001036

/*************************************************************************/
// void CheckPressure(unsigned short sLevel, unsigned short * psMax)
// description:
// Checks the difference between the current pressure of system and
// a max value; if the result is greater than PRESSURE_WARN,
// bit 2 of port at 0x1038 is cleared and the difference is written to
// the 16-bit port at 0x1036
//
// params:
// unsigned short sLevel: current pressure of system
// unsigned short * psMax: maximum pressure
//
// returns: none
Coding Case Study –
CheckPressure

Code Development and Test:
The flowchart, drawn earlier, outlined what the function does.

```c
void CheckPressure(unsigned short sLevel, unsigned short * psMax)
{
    char *pStatus = (char *) STATUS_PORT;
    unsigned short *pData = (unsigned short *) DATA_PORT;
    short sDiff;

    //find the difference between the current and max pressure
    sDiff = sLevel - *psMax;

    //if the difference is greater than the threshold PRESSURE_WARN, clear
    //bit 2 of STATUS_PORT and write the difference out to DATA_PORT
    if(sDiff > PRESSURE_WARN){
        *pStatus = *pStatus & 0xFB;
        *pData = sDiff;
    }
}
```

Coding Case Study –
CheckPressure

Code Development and Test:
Now you should test the function. If you’ve already developed a larger program that
calls this function, you can proceed from there. However, you could also write a
separate “testbench program,” a main program that only calls this function. That would
help you isolate this function for individual component testing. Often, it is useful to do
individual component testing before integrating a component into the larger system.
Once in the larger system, you are testing more than just the component; you then have
interfaces to the component, interaction of the component with other components, etc.

To test CheckPressure, a testbench program should call the function with parameters
initialized in the code or by user input. You might also need to re-map the memory-
mapped I/O ports to actual memory spaces used by your test platform. You could
remove the predefined port addresses and let the compiler set up pointer variables. Or
you could initialize the pointers to LED or other output ports for testing. Then, run the
program on the test platform, using the IDE to single-step, watch variables, etc.
Coding Case Study – CheckPressure

Code Development and Test:

Testing done so far indicates whether the code does what it’s supposed to do with respect to the functional specification. That is, does it meet the functional requirements for the problem? There is another kind of testing that evaluates whether the code meets any nonfunctional requirements, such as performance (e.g., speed, power) or cost.

Often it is the nonfunctional requirements that lead an embedded programmer to write assembly language code. There are cases where an embedded programmer uses assembly language to get access to low-level features of the machine, such as a processor register. However, in most cases, assembly language is used because the programmer needs more control of the implementation – e.g., to make the code smaller, faster, or lower power.

At this point, the embedded programmer decides what C code should be converted to assembly code. This may be done incrementally, a segment or function at a time.

Coding Case Study – CheckPressure

Code Development and Test:

Suppose we have a constraint that CheckPressure must take less than a certain maximum amount of time to execute. When we measure the run time of the C function, we find that it exceeds the maximum allowed time. Perhaps, using the IDE, we even look at the assembly code generated by the compiler, and we notice some inefficiencies. We think we can do better if we code it by hand in assembly.

Note that this was a frequent scenario in earlier days when compiler technology was less mature. Nowadays, optimizing compilers know how to generate fast code. However, there still may be specific areas in which the compiler is not smart enough to optimize, and an embedded programmer might do better. For example, when a programmer knows a fast method for coding a calculation. Or when trying to meet specific power or current budgets in a battery-powered device. Or when trying to fit a particular memory footprint in a small-memory device.

Let’s convert function CheckPressure to assembly code.
Coding Case Study –
CheckPressure

Code Development and Test:

Let’s start by just writing the basic assembly code for the body of the function, without considering prologue or epilogue code. However, we should consider platform-specific coding rules, such as for register usage.

For the CodeWarrior IDE and PowerBox platform (MPC555 microcontroller, with PowerPC processor, etc.), there are specific rules to follow.

Without writing a complete function as a separate assembly module, we could start by writing inline assembly code inside the C function.

Based on the rules, the parameters will be in registers r3 and r4. After that, it’s up to the programmer to allocate stack space or registers as needed for local variables or temporaries. We might choose volatile registers for temporary data and nonvolatile registers for local variables.

```c
#define PRESSURE_WARN 1000
#define STATUS_PORT 0x00001038
#define DATA_PORT 0x00001036

void CheckPressure(unsigned short sLevel, unsigned short * psMax) {
    char *pStatus = (char *) STATUS_PORT;
    unsigned short *pData = (unsigned short *) DATA_PORT;
    short sDiff;
    ...
}
```

Register usage:
- r3: sLevel
- r4: psMax
- r14: *psMax
- r31: pStatus
- r30: *pStatus
- r29: pData
- r28: *pData
- r27: sDiff
- r12: temporary

These constants are used as labels in the assembly code.
Coding Case Study – CheckPressure

Code Development and Test:

Register Usage Table:

<table>
<thead>
<tr>
<th>Source Variable, Type</th>
<th>Value</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>sLevel, unsigned short</td>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>psMax, pointer to us</td>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>*psMax, us</td>
<td>R14</td>
<td></td>
</tr>
<tr>
<td>pStatus, pointer to char</td>
<td>R31</td>
<td></td>
</tr>
<tr>
<td>*pStatus, char</td>
<td>R30</td>
<td></td>
</tr>
<tr>
<td>pData, pointer to us</td>
<td>R29</td>
<td></td>
</tr>
<tr>
<td>*pData, us</td>
<td>R28</td>
<td></td>
</tr>
<tr>
<td>sDiff, short</td>
<td>R27</td>
<td></td>
</tr>
<tr>
<td>temporary</td>
<td>R12</td>
<td></td>
</tr>
</tbody>
</table>

Local variable initialization:

```c
char *pStatus = (char *) STATUS_PORT;
unsigned short *pData = (unsigned short *) DATA_PORT;
```

CheckPressure:

```assembly
lis r31, STATUS_PORT@h
ori r31, r31, STATUS_PORT@l
lis r29, DATA_PORT@h
ori r29, r29, DATA_PORT@l
```

Notice that a 16-bit port address could be specified as an operand in an instruction, e.g.,

```assembly
stb r30, STATUS_PORT@l(0)
```

However, a 32-bit address must be loaded into a register.
Coding Case Study – CheckPressure

Code Development and Test:

Next, convert the pressure difference calculation:

```c
// find the difference between the current and max pressure
sDiff = sLevel - *psMax;

// R27 = R3 - *R4, or R27 = R3 - R14
// R3, R4 are passed in as parameters; dereference R4 into R14 using load half-word
// calculate difference, subtract works on registers only
lhz r14, 0(r4)
sub r27, r3, r14
```

Notice that the subtract instruction does not affect the condition register. Thus, another instruction is needed to evaluate the condition in the if statement. The compare instruction does a subtraction to set the condition codes.

```c
If (sDiff > PRESSURE_WARN){
    *pStatus = *pStatus & 0xFB;
    *pData = sDiff;
}
```

Next, convert the threshold comparison and if statement:

```c
If (R27 > PRESSURE_WARN) {
    // R31 = R31 & 0xFB; or *R31 = R30 & 0xFB;
    *R29 = R27;
    cmpwi r27, PRESSURE_WARN ;compare sDiff-PRESSURE_WARN
    bhi done ;if >=, branch to done to skip code
    lbz r30, 0(r31) ;read STATUS_PORT
    andi r30, r30, 0xFB ;clear bit 2
    stb r30, 0(r31) ;update STATUS_PORT
    sth r27, 0(r29) ;write sDiff to DATA_PORT
    done:
}
```
Coding Case Study – CheckPressure

Code Development and Test:

Completed code segment for CheckPressure:

```assembly
CheckPressure:

ils r31, STATUS_PORT@h ;initialize port pointers
ori r31, r31, STATUS_PORT@l
lis r29, DATA_PORT@h
ori r29, r29, DATA_PORT@l
lhz r14, 0(r4) ;calculate sDiff
sub r27, r3, r14
cmpwi r27, PRESSURE_WARN ;compare sDiff-PRESSURE_WARN
ble done ;if <=, branch to done to skip code
lhz r30, 0(r31) ;read STATUS_PORT
andi r30, r30, 0xFB ;clear bit 2
sth r30, 0(r31) ;update STATUS_PORT
sth r27, 0(r29) ;write sDiff to DATA_PORT
done: blr
```

Now that you have a complete code segment for CheckPressure, you could test this code, using either inline assembly code or by inserting the assembly code into the StartAsm code module.

The parameters need to be passed correctly in registers R3 and R4, and the ports need refer to actual memory addresses or output ports for the PowerBox.

Once these are set up appropriately, you should be able to single-step through this code similar to your C code and verify that it works correctly.

It should be possible to use the same testbench program.
Now let's finish the code for the function by writing the prologue and epilogue assuming the function will be used in a larger program including both C and assembly modules and/or call other functions. We need to follow the EABI rules used by CodeWarrior.

The prologue should:

- Preserve the Link Register on the runtime stack
- Preserve nonvolatile registers used by the function
- Reserve space for local variables (if allocated on stack)
- Follow rules for the stack pointer and stack frame (stack space used by the function)
  - SP must be word-aligned; you can read/write byte or half-word data on the stack, but SP itself must always be word-aligned (multiple of 4).
  - Stack frame must be double-word aligned; that is, its size in bytes must be a multiple of 8 (or in words, a multiple of 2).
  - Stack frame must have 2 frame header words at the base or lowest address.

The epilogue should:

- Pop everything that was pushed in the prologue, in reverse order
- Remove the space by incrementing SP

The prologue and epilogue can update the SP for each push and pop, or alternatively, can update the SP for all stack frame space at once. See coding options.

In addition to the prologue and epilogue code, appropriate directives or qualifiers need to be inserted in the C and assembly modules. For example:

- In C, use `extern`
- In assembly, use `.import`, `.export`
Cultural Case Study – CheckPressure

Code Development and Test:

Here is what the stack frame for CheckPressure should look like after the prologue:

- Each cell in the picture is one word wide.
- Saved registers could be accessed relative to SP; see notation \( N(SP) \).

Prologue for function CheckPressure:

```
#define SP, r1
.export CheckPressure

CheckPressure:
// Prologue
addi SP, SP, -32 ; allocate stack frame space
mflr r0 ; get LR ("move from LR" into r0)
stw r0, 36(SP) ; save LR
stw r14, 28(SP) ; save r14
stmw r27, 8(SP) ; save bank of NVs starting with r27
```

Coding ISU, CPRE 211, F02 38
Coding Case Study –
CheckPressure

Code Development and Test:
Epilogue for function CheckPressure:

done:
    // Epilogue
    lmw r27, 8(SP) ; restore bank of NVs starting with r27
    lwz r14, 28(SP) ; restore r14
    lwz r0, 36(SP) ; get the saved LR
    mtlr r0 ; restore LR ("move to LR" from r0)
    addi SP, SP, 32 ; deallocate stack frame space
    blr ; exit and return to caller using LR

Coding Strategy

The CheckPressure case study began with a problem statement, introduced a functional specification of the system and function, and walked through a code development and test process, starting with C and ending with assembly code.

You should be able to follow this same basic process for any software module that you are developing.
Coding Strategy – Example

Bigger Example: Digital Camera

The process just covered is a simplified version of designing an embedded system. Recall the parts of an embedded system; there may be custom ICs in addition to a processor. Thus, the design of an embedded system typically involves more than code development for a processor. Many decisions are made along the way, e.g.:

Which microcontroller will be selected for the code to run on? E.g., is a 32-bit microcontroller needed, or could an 8-bit microcontroller be used?

Is custom hardware or additional processors needed to meet requirements or constraints?

The digital camera example walks you through the design process. Notice that embedded programming is an essential part of the process.

See class.ee.iastate.edu/cpre211/handouts/esd_ch7.ppt

Don’t worry about the algorithm details. But take a look to appreciate the role of specification, the advantages of starting with C, and the iterative nature of design.

See also class.ee.iastate.edu/cpre211/handouts/esd_ch8_FSM.ppt for another look at the role of specification, in this case, using a finite-state machine specification.