Implementing Bootloaders on Renesas MCUs
Renesas Technology & Solution Portfolio
Microcontroller and Microprocessor Line-up

2010

1200 DMIPS, Superscalar
- Automotive & Industrial, 65nm
- 600µA/MHz, 1µA standby

32-bit High Performance
DSP, FPU with High Integration

1200 DMIPS, Performance
- Automotive, 40nm
- 800µA/MHz, 35µA deep standby

1200 DMIPS, True Low Power
- Industrial & Automotive, 130nm
- 144µA/MHz, 0.2µA standby

8/16-bit

10 DMIPS, Capacitive Touch
- Industrial & Automotive, 130nm
- 350µA/MHz, 1µA standby

50 DMIPS, Low Power
- Industrial & Automotive, 150nm
- 190µA/MHz, 0.3µA standby

10 DMIPS, Capacitive Touch
- Automotive, 40nm
- 190µA/MHz, 0.3µA deep standby

10 DMIPS, Capacitive Touch
- Automotive, 90nm
- 1mA/MHz, 100µA standby

165 DMIPS, FPU, DSC
- Automotive & Industrial, 90nm
- 600µA/MHz, 1.5µA standby

165 DMIPS, FPU, DSC
- Automotive, 65nm
- 600µA/MHz, 1.5µA standby

25 DMIPS, Low Power
- Industrial, 90nm
- 1mA/MHz, 100µA standby

200µA/MHz, 0.3µA deep standby

Floating Point Unit 32-bit
Digital Signal Unit 32-bit
MAC 48-bit
RMPA 80-bit
Barrel Shifter 32-bit
‘Enabling The Smart Society’

- In a smart society devices can add features and fix bugs after leaving store shelves. Having a bootloader in your system allows you to sleep better at night.

- **Firmware**
  - v1.2
  - v1.3
  - **✓ Fixed focusing bug**

- **Firmware**
  - v3.0
  - v4.0
  - **✓ Added Multitasking**
Agenda

- Quick bootloader overview
- The flash loader framework
- Design choices
- RX implementation
- RL78 implementation
- Lab
Quick Bootloader Overview
Quick Bootloader Overview

- What is a bootloader?
- What can a bootloader do?
  - Erase and rewrite user memory
  - Validate user memory
  - Communicate with the outside world
  - Download new application in case of failure
- What should a bootloader be?
Available Boot Options

- **Factory bootloader**
  - Cannot be modified
  - User code cannot run
  - Can program/erase everything

- **User boot area**
  - Programmed with USB from factory
  - Uses CPU rewrite code

- **User application**
  - Uses CPU rewrite code
The Flash Loader Project
What is the Flash Loader Project?

- Flexible system for implementing in-field-reprogramming in your own project
- Communications medium agnostic
- Modify to fit your system
- Retries & error checking built in
- Does not interfere with user application
Terms

Host

Load Image

Device

Storage
Parts of the Flash Loader Project?

- Flexible system that can be split into 2 parts:
  - Downloader
  - Bootloader
1: Using Flash Loader With Your Project
Add Flash Loader to your HEW Project

- Add flash loader project files

- Enables future updates

- Flash loader bootloader is separate project
Configure Flash Loader for Your System

- r_fl_app_header.c
- r_fl_comm_*type*.c
- r_fl_downloader.c
- r_fl_store_manager.c
- r_fl_memory_*type*.c
- r_fl_utilities.c

= Edit for your configuration
2: Making a Load Image
Making a Load Image

- S-Record ("MOT") files are inefficient
  - S-Record Converter

- Load image information:
  - Binary
  - Starts with header
  - Has N blocks
## Load Image Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid mask</td>
<td>1</td>
</tr>
<tr>
<td>Image ID</td>
<td>1</td>
</tr>
<tr>
<td>Version number</td>
<td>1</td>
</tr>
<tr>
<td>Size of load image</td>
<td>4</td>
</tr>
<tr>
<td>Max block data size</td>
<td>4</td>
</tr>
<tr>
<td>Load image CRC</td>
<td>2</td>
</tr>
<tr>
<td>Raw CRC</td>
<td>2</td>
</tr>
<tr>
<td>Address of 1st data block</td>
<td>4</td>
</tr>
<tr>
<td>Successfully stored?</td>
<td>4</td>
</tr>
</tbody>
</table>
3: Obtaining the Load Image
Transferring the Load Image

- Flash loader is medium agnostic
- Communications protocol is supplied
- Supported commands
  - Information request
  - Erase image from storage
  - Load image download
Storing the Load Image

- As Load blocks come in:
  - Check for errors
  - Store
- Storage area partitioned for load images
- If error occurs, retry is possible
What the Storage Area Looks Like

- Load images are in different ‘partitions’
4: Using the Load Image
Reboot & Flash

- Reboot when convenient
- Check for new load image
- If valid, reflash MCU
- Jump to user application and execute
Design Choices
Bootloader Decisions

- Why not store bootloader in same memory area as user application?
  - Bug in control code could erase everything
  - Application must be aware of bootloader

- Why not store load images in MCU memory?
  - Hard to make bootloader not interfere with user application
  - Bug in control code could erase currently running application
  - Limit application size or pay more
  - Move code to RAM
  - May have to use PIC and PID
  - Keep safe older revisions
RX Implementation
Flash Loader Bootloader

- Separate workspace
- Separate memory areas
- Can use same or different communications as FL downloader
- User Boot is special
Why is User Boot Special?

- Can execute from either reset vector on start-up
- Cannot accidentally erase
- Should have ‘one and done’ attitude
- No indirection tables!
Storing Load Images

- By default stored in external SPI flash
- RX cannot read from ROM while writing or erasing ROM
- Large stalls in user application or move to RAM
- No chance of losing image
- Cost of SPI flash is usually lower than doubling MCU ROM
- It’s so much easier!
RL78 Implementation
Dual Boot Blocks

- Has 2 independent boot blocks
- Can swap which one is used
- Removes problem of losing everything
- Still recommend external SPI flash
Using Boot Swap

- Download new image
- Swap to bootloader block
- Program in image
- Swap to application block
Summary

- Quick bootloader overview
- The flash loader framework
- Design choices
- RX implementation
- RL78 implementation
- Lab
Questions?
‘Enabling The Smart Society’

- In a smart society devices can add features and fix bugs after leaving store shelves. Have a bootloader in your system allows you to sleep better at night.

- **Firmware**
  - v1.2
  - v1.3
  - ✓ Fixed focusing bug

- **Firmware**
  - v3.0
  - v4.0
  - ✓ Added Multitasking