Updating Code

Chapter 7

Program Memory Access

- Read only from program memory
- ATmega128 (avrgcc)
  - unsigned char curve[100] **PROGMEM** = {...};
  - Byte b = **pgm_read_byte**(&(curve[i]));
- PIC24E (xc11)
  - static **ROM** BYTE MY_DEFAULT_IP_ADDR[4];
  - **memcpypgm2ram**((void*)&AppConfig.MyIPAddr, (**ROM** void*)MY_DEFAULT_IP_ADDR, sizeof(AppConfig.MyIPAddr));
- Write to program memory: flash programming

Loading Code

- During development
  - Have a debugger (JTAG)
  - Have a code loading device
  - Flashing the chip
- After development (in deployment)
  - No debugger
  - Cannot use the code loading device in field
  - So, updating code is so different from flashing the chip.

Differences

- Have a code storage mechanism to store a new code image.
- Have a communication method to transfer the new code image.
- Program non-volatile but not-read-only code space memory.
- Have a scratch space memory to write the code to and make sure the upload is complete without corruption.
- Have a run space memory where the processor can execute code.

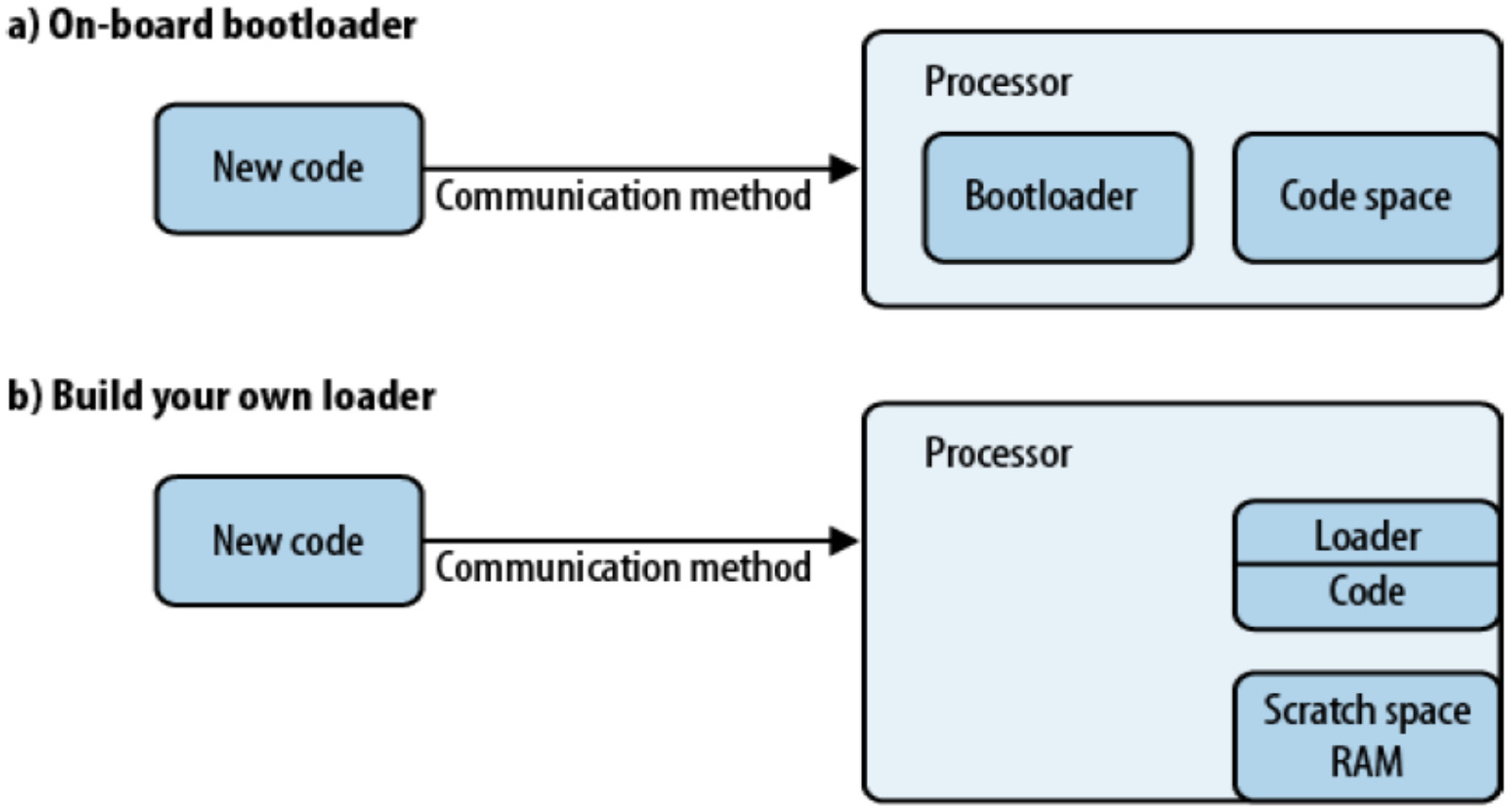
Key Components

- New code storage
  - EEPROM volumes
- Code transmission
  - USB, Network
- Code integrity check
  - CRC
- Scratch code space
  - Data RAM or another EEPROM volume
- Non-disruption controller
  - Bootloader in auxiliary flash

Main Issues

- What could go wrong?
  - Code block damaged
  - Flash rewrite errors
  - Power loss
- How to recover the system?
  - Brick

Architectures of Loading Code



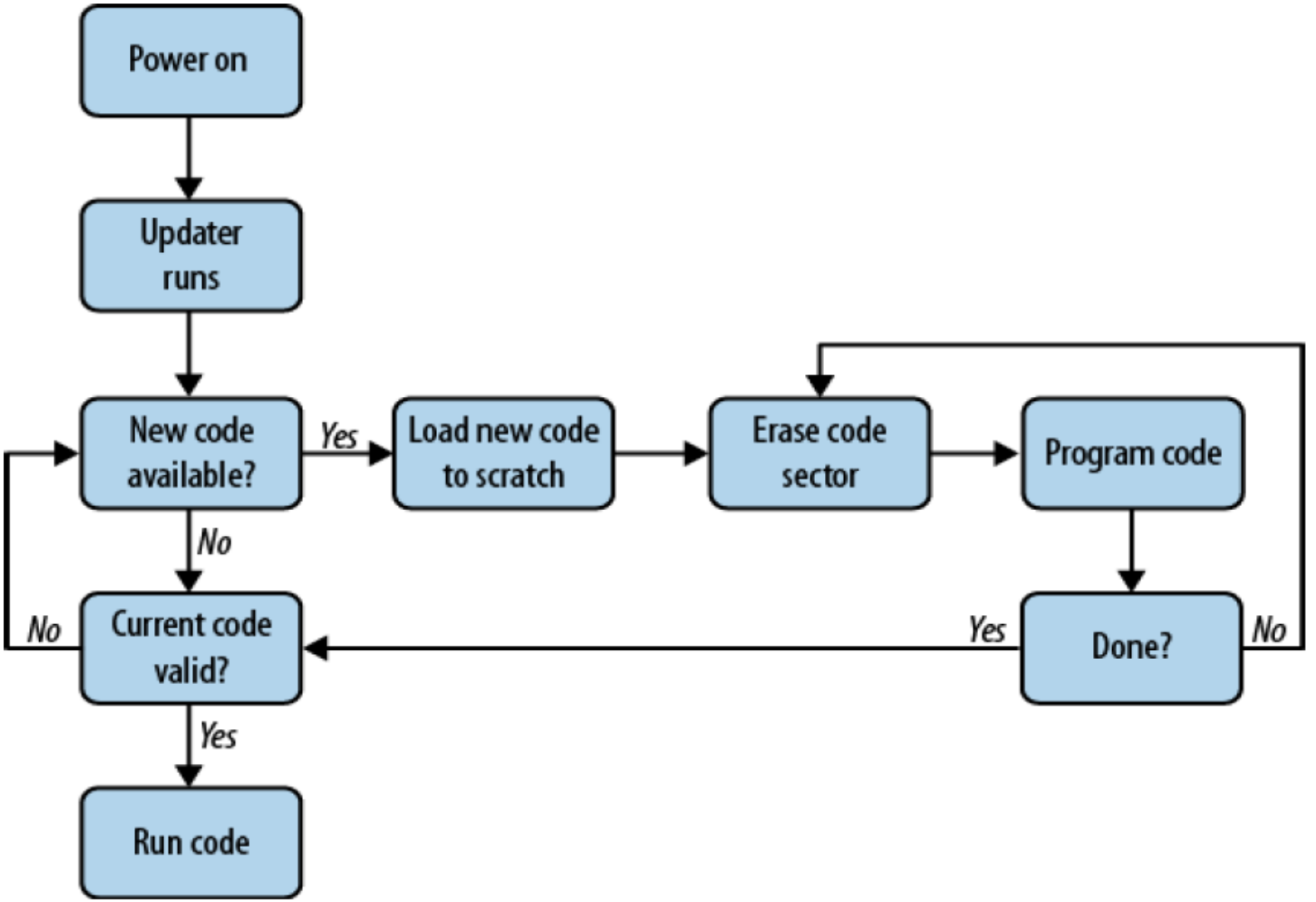
On-board Loader

- Set the I/O pin to connect the new code from an external storage to the system
- Load code from an external storage to the code space.

Build Your Own Updater

- A resident program in the code space that could reprogram the rest of the memory
- Process
  - **Erase a sector (always, mcu specific)**,
  - Read the new code into some scratch memory,
  - Use chip-specific functions to write the code to that sector,
  - Repeat until all runtime sectors are complete.
- Issues
  - New code is corrupted from the source or dropped in communication.
  - New code storage or writing fails (e.g. power loss)

Build Your Own Updater



Example: Case24

- Update: embedded design
  - The update program is an embedded app.
  - It updates a running app.
- Functionality (operations)
  - Detect if a running app exists
  - Update on booting
    - The running app does not support update
  - Detect the running app's version
  - Verify CRC of apps in code memory
  - Diagnose code memory
  - Reboot the running app
  - Code transmission: usb or network

Example: Case24

- Requirements (quality)
  - Robust: not breaking if fail
    - The running app can be overwritten only when everything else is correct and ready
- Resource constraints
  - Code memory size must be twice of the app size
    - Code storage of the new app must be separated from the running app
  - The new app image must be divided and sent block by block
    - Both usb and network have size limits on packets.

Example: Case24

- Main components
  - Updater in device (aux flash)
    - App: main.c
    - Controller: update.c
    - Communication: USB/*.c
    - Volume manager: volume.c
    - Flash access: NVMem.c
  - Controller in host
    - App: bootload.c
    - Communication: hidapi.c
    - Volume manager: volmanager.c

Example: Case24

- Why need a volume manager?
- Flash (code memory) layout
  - each address has 2 bytes or 1 word
  - One instruction has 4 bytes. The MSB is 0x00.
    - 0x00FFFFFF at two consecutive addresses
  - 0x000000 - 0x0001FF : interrupt vector table, every interrupt takes 2 addresses (except reset), so 255 interrupts, reset takes the first four addresses
  - 0x000200 - 0x02ABFF : primary program
  - 0x7FC000 - 0x7FFFF9 : auxiliary program
  - 0x7FFFFA - 0x7FFFFF : auxiliary interrupt vector, and the alterative reset

Example: Case24

- Flash structure
  - Main flash
    - Divided into two volumes
      - Each volume has 0x010000 words (0x020000 bytes)
    - App volume
      - Store the run time app code
    - Up volume
      - Store the new app code to update
  - Aux flash
    - Store the updater code
- Reset target vector selector to aux flash
  - #pragma config RSTPRI = AF

Example: Case24

- Volume structure
  - A volume has 32 pages.
  - A page has eight rows, 1024 instructions, or 3072 bytes. Each page is aligned at 3072 bytes.
  - A row has 128 instructions or 256 words or 384 bytes. Each row is aligned at 384 bytes.
- Operation:
  - Erase a single page
  - Program either two instruction words or up to 128 instruction words

Example: Case24

- App volume structure
  - Code size: 0x00FF00
    - Start address: 0x000000
    - End address: 0x00FEFF
  - Meta info of app
    - Size: 0x000100
    - Start address: 0x00FF00
    - End address: 0x010000
    - APP_PROG_FLAG at 0x00FFC0
- Up volume structure
  - Same as the app volume structure
  - Start address: 0x010000

Example: Case24

- Meta info structure
  - VOL_META_PL in volume.h
  - Align every three bytes (0x00FFFFFF)
  - Info
    - Major version number
    - Minor version number
    - Revision number
    - Crc
    - Page number
    - Row number
    - Code size
    - App name

Example: Case24

- Communication commands (com com)
  - Two-way communication
    - VOL_CMD_TYPE in volume.h
    - From host to device
    - From device to host
  - Com com structure
    - VOL_COMMAND in volume.h
  - Payload structure in volume.h
    - VOL_CODE_PL
    - VOL_META_PL
    - VOL_ERROR_PL
    - VOL_READ_PL

Example: Case24

- Com handler
  - VOL_STATUS in volume.h
  - HandleVolCmd in volume.c
- Update flow
  - Reboot
  - Check if run-time app is valid
    - Count down and start the app
  - Wait to get new app
    - ST_IDLE -> ST_START -> ST_ERASE -> ST_META -> ST_CODE -> ST_PROGRAM -> ST_RUN

Complete Code In Flash

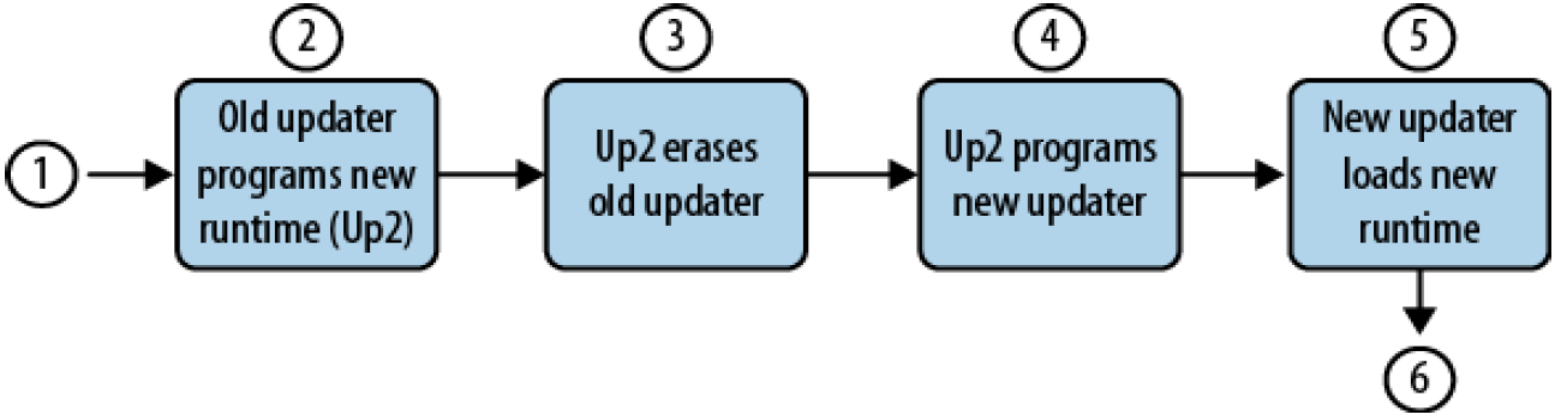
- Main flash (App)
  - IVT (in assembly)
    - Reset at 0x0
  - Init (in assembly)
    - Data & BSS
    - Constants
  - App (main, in C)
    - App init
      - Device drivers
    - Scheduler (while)
      - EventHandler (while)
      - Event queues
  - Libs (in assembly)
  - ISRs (in C)
- Aux flash (Updater)
  - IVT (in assebmly)
    - Reset
  - Init (in assembly)
    - Data & BSS
    - Constants
  - Updater (main, in C)
    - Up init
    - App test
    - Updating (polling)
  - Libs (in assembly)

Update Updater

- Two-pass update
- Pass1
  - The older updater updates a new runtime.
    - This new runtime is not application.
  - The new runtime updates the new updater.
- Pass2
  - The new updater updates a new runtime again.
    - This new runtime is the application.

Build Your Own Updater

- Modifying the Resident Updater

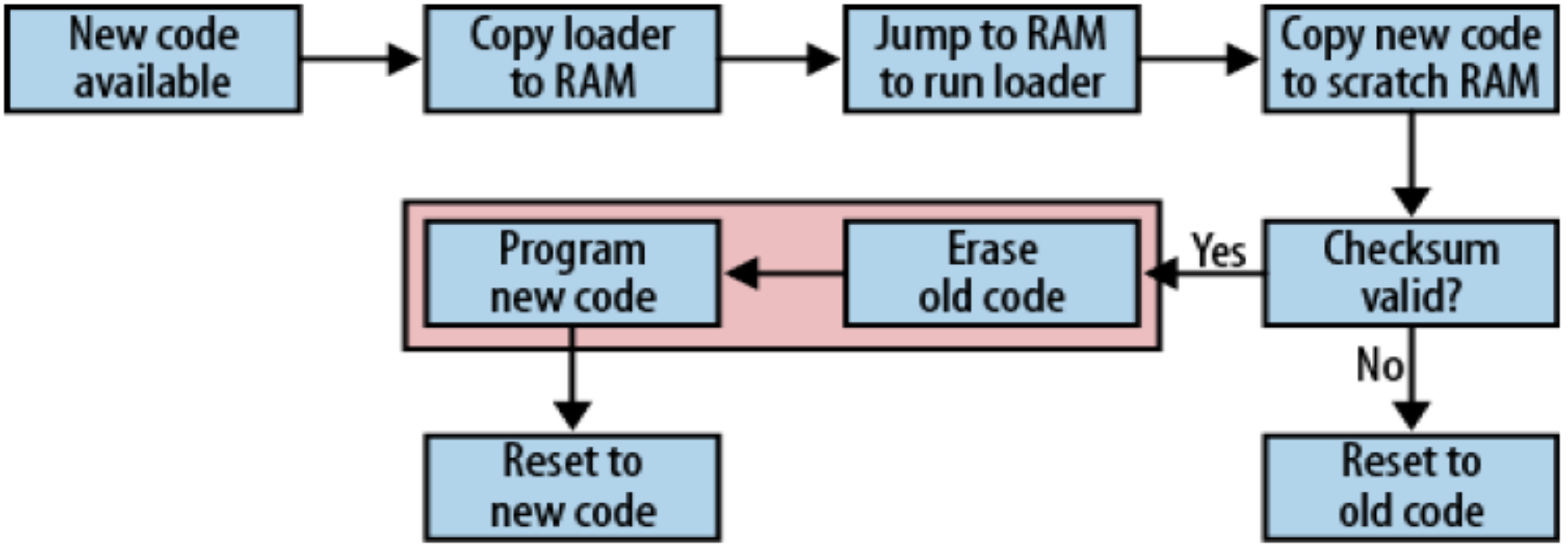


Stage	Storage mechanism	Updater area of memory	Runtime code area of memory
1	Up2	Old updater*	Old runtime
2	(Nothing)	Old updater	Up2*
3	New updater	(Erased)	Up2*
4	(Nothing)	New updater	Up2*
5	New runtime	New updater*	(Erased)
6	(Nothing)	New updater	New runtime*

Brick Loader

- Issues of your own updater
  - Don’t have enough code space to devote to the updater
  - The updater functionality changes regularly
  - The code space doesn’t allow sector erases.
- Solution : brick loader
  - Store the loader image in an external storage.
  - Brick : A system is unable to ever load valid code.
    - The power is lost during the period between erasing the flash and having the new code fully loaded, the system is not recoverable in the field.

Brick Loader



Brick Loader

- Loading process
  - With the runtime code, copy the loader from the storage to RAM.
    - Allocate some RAM specifically and only for the loader code.
    - Build the loader program to run at a certain address (its base address) and calling functions at certain other addresses via customizing linker script.
    - Make the loader program smaller than the size of the available RAM.
    - Stop all subsystems of the running code.

Brick Loader

- Loading process
  - Run the loader in RAM
    - Make sure that the loader is correct and complete with a checksum.
  - Copy the new runtime code from storage to a scratch area.
  - Erase the old runtime code and program the new.
    - Brick could occur, but scratch ram alleviate the problem.
  - Reset the processor to run the new code.

Security

- What to protect
  - A secret algorithm in the code
  - The ability to create or verify code and data
  - The integrity of the hardware
- Security measures
  - Turn code-read protection on
    - The processor can still execute the code.
    - The debugger (or the loader) cannot read the code space.
  - Encrypt and sign the new application code
  - Encrypt and sign the new loader