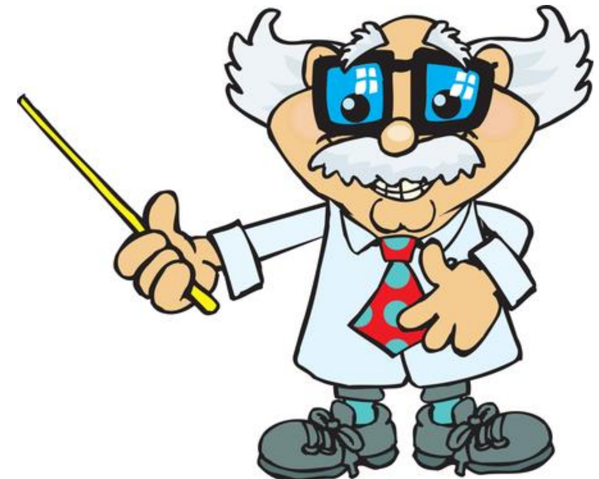
Bootloader Design Techniques for MCUs

Jacob Beningo, CSDP

BENINGO ENGINEERING

Session Overview

- Introduction
- The Boot-loader System
  - Local Models
  - Distributed Models
- Startup Branching
- Bootloader Behavior
- Resetting
- Memory Management
- Binary Formats



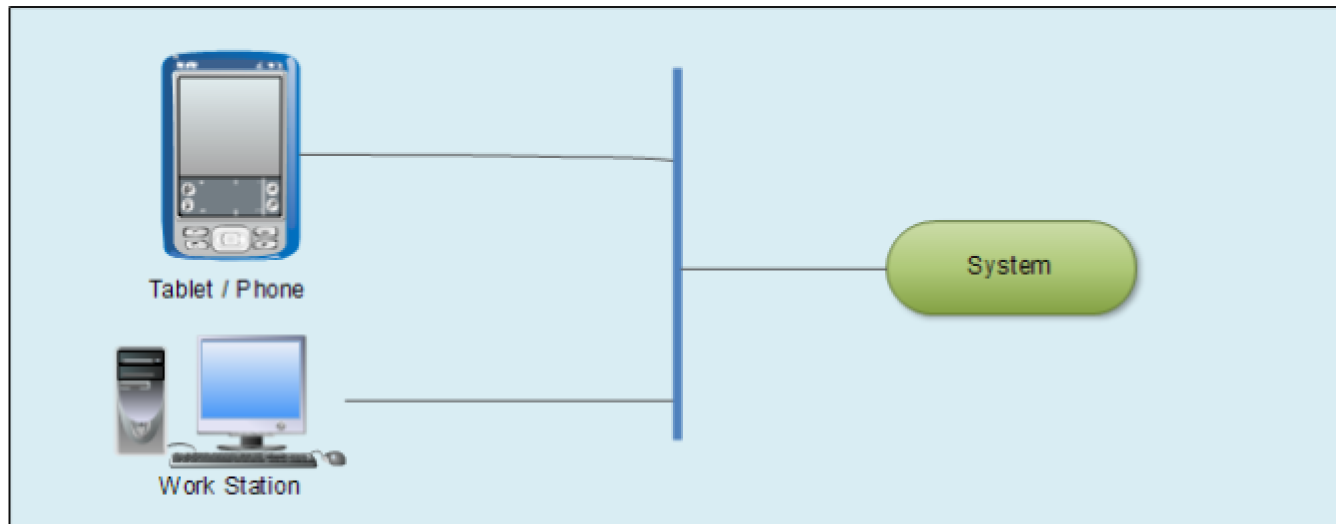
BENINGO ENGINEERING

Introduction

- Billions of microcontrollers sold per year
- Shortened development cycles
- Feature creep
- Intense competition
- Software bugs
- How to update software in the field when bugs are discovered or new features required?
- A boot-loader is an application whose primary purpose is to allow a systems software to be updated without the use of specialized hardware such as a JTAG programmer.

Local Single Device System

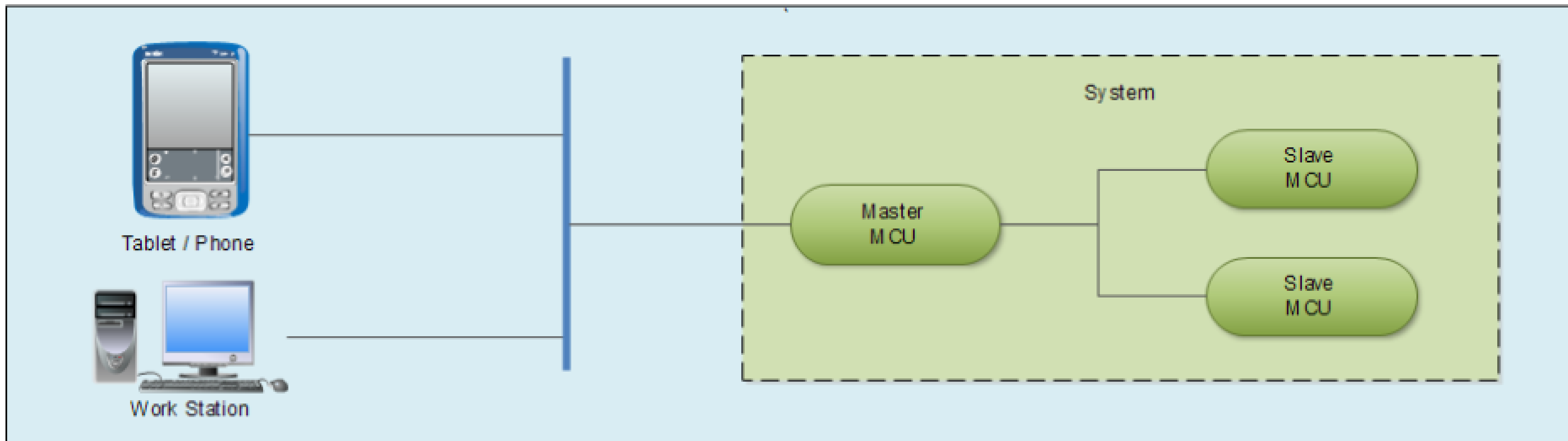
- Single MCU System (Traditional / Most Common)
- Flashing Method
  - Laptop / Workstation
  - Tablet or mobile device
  - USB Flash System



BENINGO ENGINEERING

Local Multiple Device System

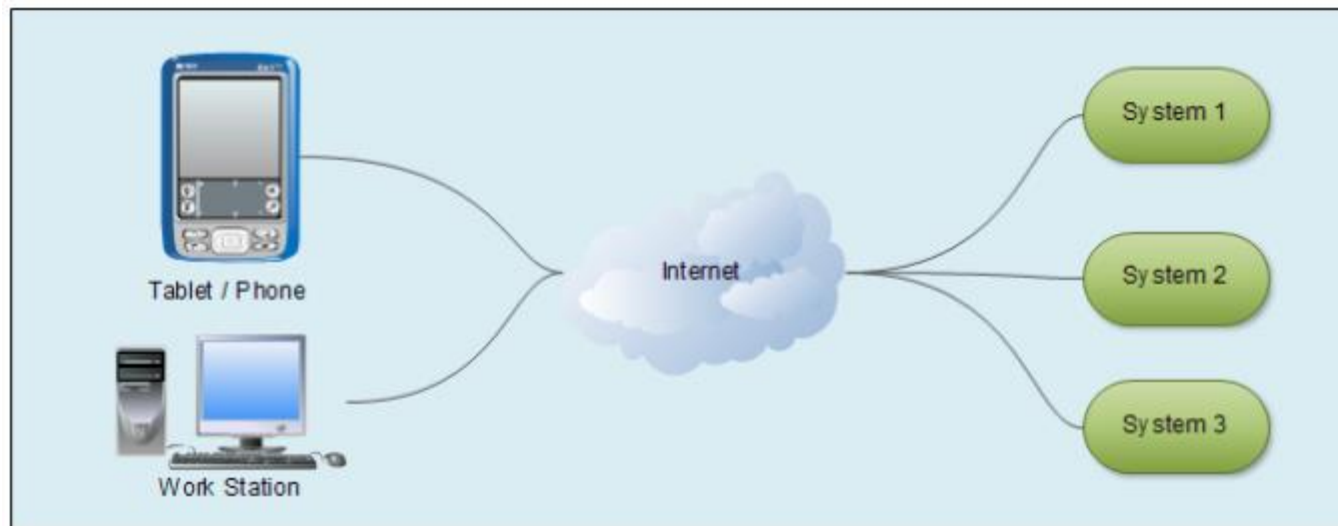
- Multi MCU System
- Flashing Method
  - Laptop / Workstation
  - Tablet or mobile device
  - USB Flash System
- Master MCU
  - Can be updated itself
  - Passes new application to slave devices and acts as the flash tool



BENINGO ENGINEERING

Distributed Cloud Device System

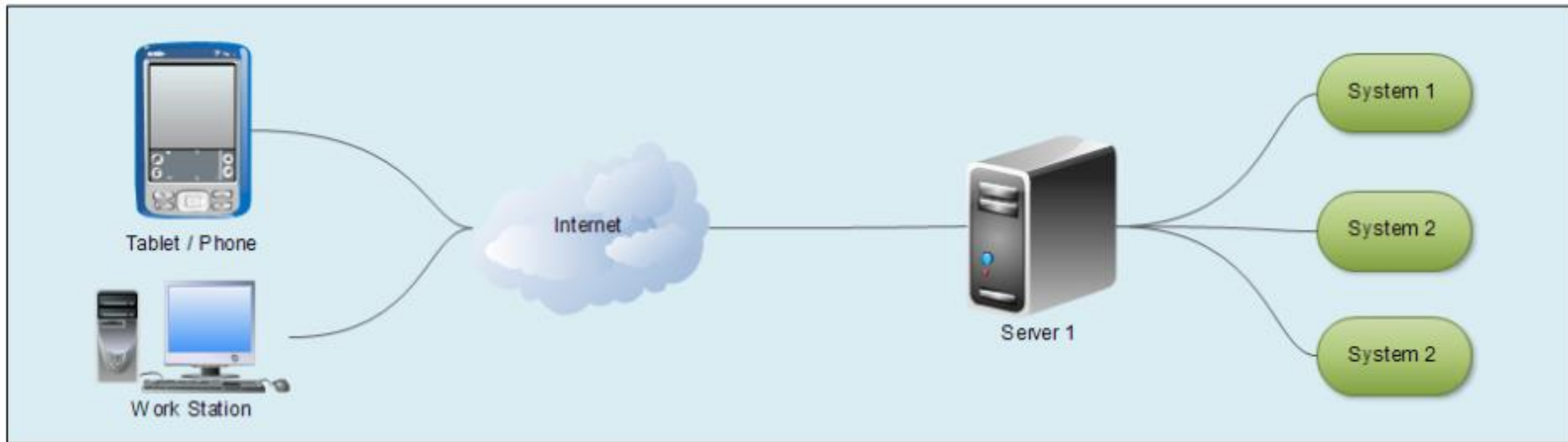
- MCU System
  - Single MCU Devices
  - Multi MCU Devices
  - Systems are internet enabled
  - Physical Separation from imaging tool
- Flashing Method
  - Internet Connected Devices
    - Tablets
    - Phones
    - Computers
    - Etc



BENINGO ENGINEERING

Distributed Cloud Device System 2

- MCU System
  - Single MCU Devices
  - Multi MCU Devices
  - Systems are not internet enabled
  - Physical Separation from imaging tool
- Flashing Method
  - Internet Connected Devices
    - Tablets
    - Phones
    - Computers
    - Etc



BENINGO ENGINEERING

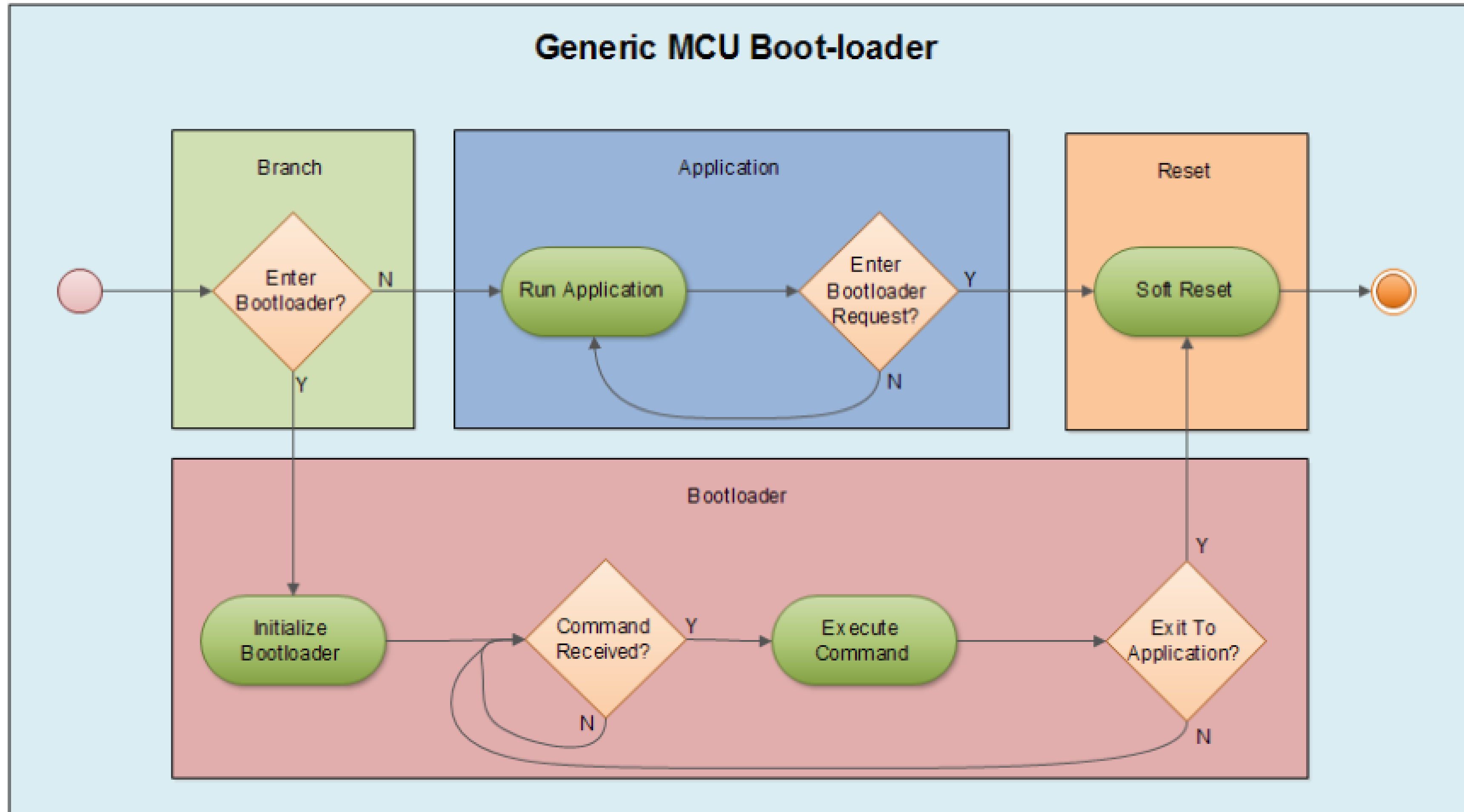
Requirements

Each boot-loader will have its own unique set of requirements based on the type of application; however, there are still a few general requirements that are common to all boot-loaders

1) Ability to switch or select the operating mode (Application or boot-loader)
2) Communication interface requirements (USB, CAN, I2C, USART, etc)
3) Record parsing requirement (S-Record, hex, intel, toeff, etc)
4) Flash system requirements (erase, write, read, location)
5) EEPROM requirements (partition, erase, read, write)
6) Application checksum (verifying the app is not corrupt)
7) Code Security (Protecting the boot-loader and the application)

BENINGO ENGINEERING

Bootloader System



BENINGO ENGINEERING

Startup Branching

The simplest method often used in example code to enter the boot-loader is to use GPIO.

- If GPIO == HIGH then enter application
- If GPIO == LOW then enter boot-loader

The advantages are

- the code can be implemented in assembly
- the branch can be executed very quickly
- it is very simple (too simple for most applications)

The disadvantages are

- Susceptibility to start-up noise
- Requires the use of GPIO for a dedicated function
- Accidental entry into boot-loader by unsuspecting customer

Startup Branching

Simple assembly branching code used in an S12X

```
; ----------------------------------------------------------------------
brclr $0259, $01, GoBoot        ; if PP0 == 0 then start the boot-loader
                                ; if PP0 == 1 then start the application
ldd   AppResetVect              ; Load the Application Reset Vector
ldx   AppResetVect
jmp    0,x                      ; jump to the application
GoBoot:
lds  #StackTop
jmp  main ;
; ----------------------------------------------------------------------
```


BENINGO ENGINEERING

Startup Branching

The code in the previous slide has a potential flaw. What happens if the boot-loader has been added to the system but the application has not yet been flashed???

When a microcontroller flash section has been erased it will be erased to all 1's. This means that if the reset vector is all 1's, we know that this reset vector is not valid and that the application has not yet been programed. By performing an extra check on the application reset vector the programmer can prevent the code from branching to a non-existent application

BENINGO ENGINEERING

Startup Branching

```
; ------------------------------------------------------------------
brclr $0259, $01, GoBoot       ; if PP0 == 0 then start the boot-loader
                               ; if PP0 == 1 then start the application
ldd   AppResetVect             ; Load the Application Reset Vector
cpd   #$ffff                   ; Compare it to 0xFFFF
beq   _GoBoot                  ; if the application reset vector is not
                               ; available then start the bootloader
ldx   AppResetVect
jmp   0,x                      ; jump to the application

_GoBoot:
lds  #StackTop
jmp  main                      ; Continue Boot-loader startup
; ------------------------------------------------------------------
```


BENINGO ENGINEERING

Startup Branching

In most cases you will not find a GPIO triggered boot-loader implemented on a production product. Instead, a common method used to detect a request to enter the boot-loader is to change an EEPROM value.

Nearly every embedded system stores configuration values in some type of EEPROM whether its on-chip, off-chip or part of some emulated EEPROM in flash. Storing a byte or a word configuration value for boot status is then a natural place to store which mode the system should boot into.

Startup Branching

```
; -----------------------------------------------------------------
ldd   AppResetVect          ; Load the Application Reset Vector
cpd   #$ffff                ; Compare it to 0xFFFF
beq   _GoBoot               ; if the application reset vector is not
                            ; available then start the bootloader
ldd   EepromProgStatus      ; Read the programmed status byte from eeprom
cpd   #'B'                  ; Compare it to 'B' for boot-load
beq   _GoBoot               ; if Status == 'B' for Boot-loader then jump to
                            ; boot-loader, otherwise continue to the application
ldx   AppResetVect
jmp   0,x                   ; jump to the application

_GoBoot:
lds  #StackTop
jmp  main                   ; Continue Boot-loader startup
; -----------------------------------------------------------------
```

Startup Branching

Integrating the branching code into the boot-loader allows a number of more sophisticated checks to be performed from within a higher level language than assembly.

- Image Checksum
- Back-door access through tool presence

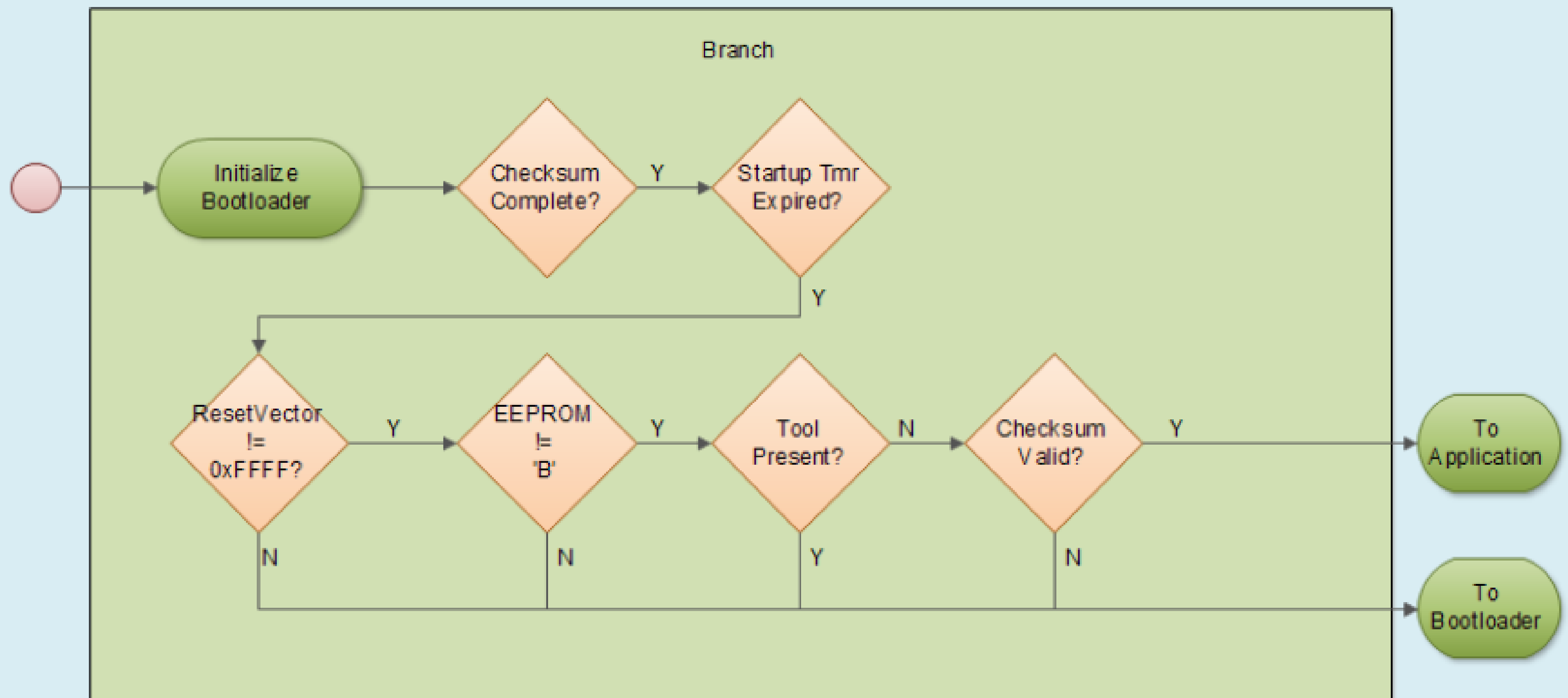
In addition to the standard branch checks

- Reset Vector Present
- EEPROM byte set

The branch logic task runs at a periodic rate until the checksum has been completed and a tool detection timer has expired

BENINGO ENGINEERING

Startup Branching



BENINGO ENGINEERING

Startup Branching

```
; ------------------------------------------------------------------------
//  When the checksum has completed and the timer has expired for waiting for a
//  programming tool to respond, perform the branch checks.
if((Checksum_Complete == TRUE) && (StartUpTmr == EXPIRED))
{
    if((*ResetVector != 0xFFFF)      &&    // Does app reset vector exist?
      (Status != 'B')                &&    // EEPROM status set?
      (Boot_ToolPresent != TRUE)     &&    // Tool present?
      (Checksum_Valid != FALSE))           // Checksum valid?
   {
      App_LoadImage();
   }
   else
   {
      Boot_LoadImage();
   }
}
; ------------------------------------------------------------------------
```


BENINGO ENGINEERING

Bootloader

- Requirements
  - Command driven vs image driven
  - Commands
    - Lock/Unlock Flash
    - Read/Write Configuration
    - Image/Record Data
    - Switch to Application
  - Image Driven
    - Continuously loops through image
    - Completely Autonomous

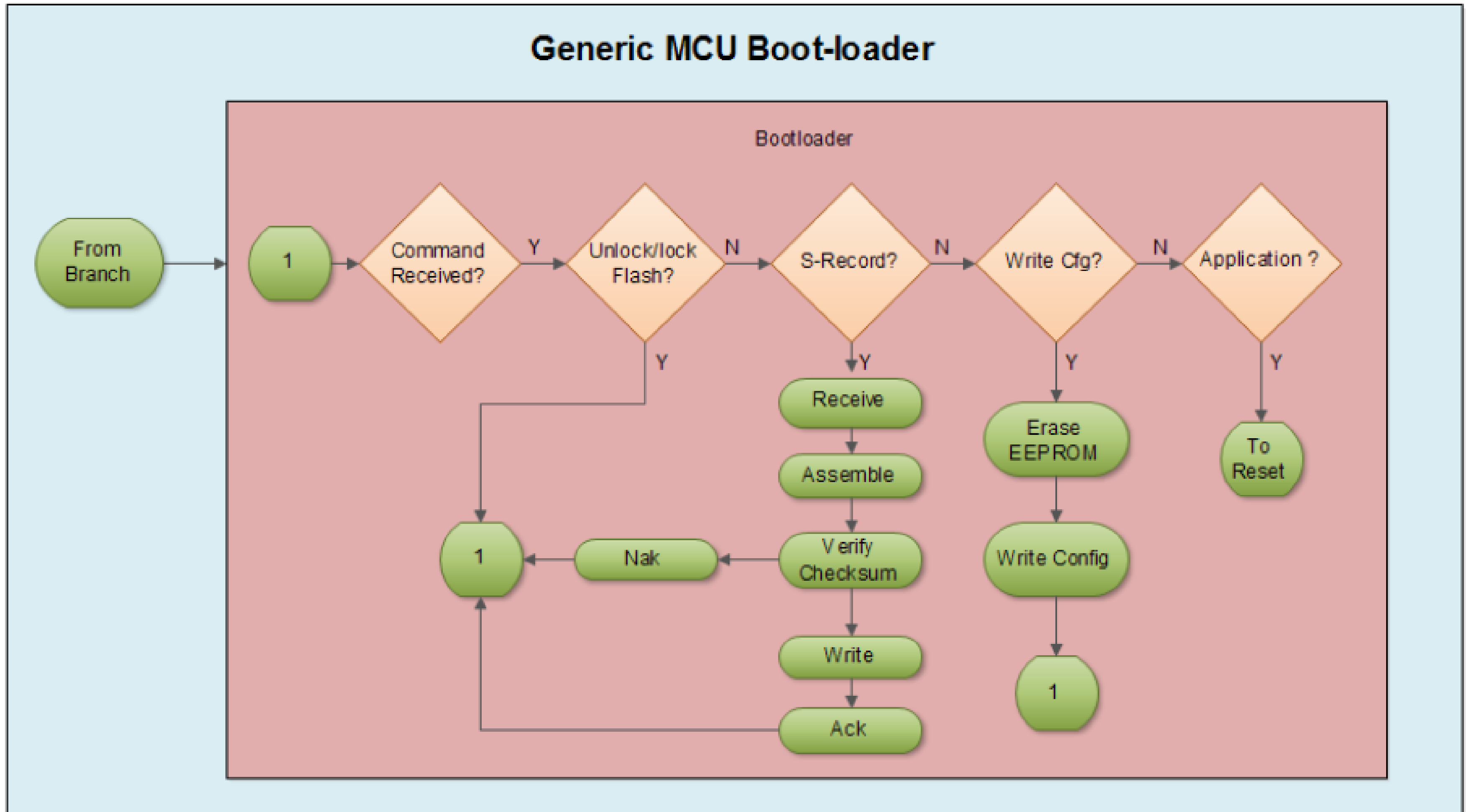
BENINGO ENGINEERING

Bootloader

- Assembling the Image
  - A block of image data is usually larger than can be directly communicated
  - Memory region broken up into separate packets
  - Packets need to be reassembled and validity checked
  - Steps
    - Receive image packets
    - Reassemble into image block
    - Verify Checksum
    - Write
    - Acknowledge
  - Repeat until completed

BENINGO ENGINEERING

Bootloader



BENINGO ENGINEERING

Resetting the System

- How to reset the system
  - Watchdog timer
    - Infinite loop
    - Illegal write to register
    - Soft reset command
  - Manual software reset
  - Notify user to power cycle

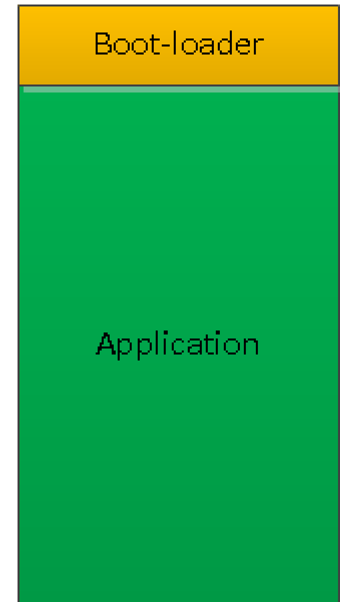
```
void Wdt_Reset(void)
{
        SWT.SR.R = 0x0000FFFF; // invalid key

        return;
}
```


BENINGO ENGINEERING

Memory Partitioning

Microprocessor flash space will need to be partitioned or sectioned off in the linker file in such as way that two memory maps exist.

- Boot-loader space
- Application space

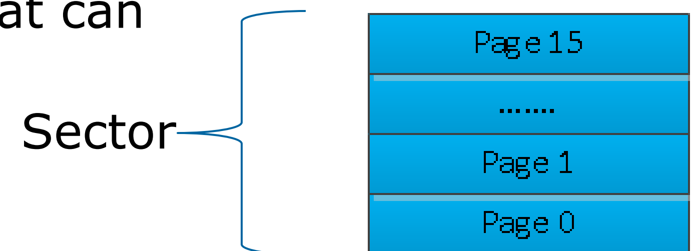


Flash is typically organized into Pages and Sectors.

A page is typically 256 bytes. There are usually 16 pages in a Sector.

Sectors are usually the smallest "quanta" that can be erased.

There is usually 4 kB in a Sector.



BENINGO ENGINEERING

Reset Vectors

- The reset vector is the location in memory where the first instruction for the application is located. When a processor is first started up it begins program execution at the address stored in the reset vector.
- For a system with a boot-loader, this address will be the branching code or entry into the boot-loader itself. So if the processor reset vector is already used by the boot-loader, how on earth does code branch to the application code? Where is the application reset vector stored?
- The application developer can leave the reset vector in its default location so that the application can be debugged without the boot-loader.
- The boot-loader programming tool or the boot-loader itself should relocate the application vector to a predetermined location within flash. This location can be selected relatively arbitrarily. It should be carved out in the linker file.

```
APP_RESET   : origin = 0x3F5FFB, length = 0x000002
```


BENINGO ENGINEERING

Application Binary File Formats

Binary Formats

- Motorola S-Records
- Intel Hex file format
- COFF(Common Object File Format)

Record type Address Checksum

```
S00600004844521B
S3220000000000000000000000000000000000000000000000000000000000000000DD
S31A0001FFEB000000000000000000000000000000000000000000FA
S70500000000FA
```

Header record
Data records
Termination record
Checksum
Byte count
Address for S3 records

BENINGO ENGINEERING

Contact Information

P.O. Box 400
Linden, Michigan 48451

E : **jacob@beningo.com**
T : **810-844-1522**
: **Jacob_Beningo**
: **Beningo Engineering**
: **JacobBeningo**
EDN : **Embedded Basics**

ARM Connected Community

Jacob Beningo
Principal Consultant