CEC Continuing Education Center

DesignNews

Expert C Techniques to Master Baremetal Programming

DAY 3 : Baremetal Performance Analysis and Architecture Design

Sponsored by

DigiKey

BENINGO EMBEDDED GROUP

informa markets

Webinar Logistics

- Turn on your system sound to hear the streaming presentation.
- If you have technical problems, click “Help” or submit a question asking for assistance.
- Participate in ‘Group Chat’ by maximizing the chat widget in your dock.

THE SPEAKER

Jacob Beningo

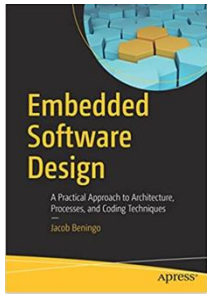
Jacob@beningo.com

jacobbeningo

Beningo Embedded Group – CEO / Founder

Focus: Software Architecture, Processes, and Dev Skills

At Beningo Embedded Group, we believe everyone deserves the skills to confidently advance their careers, meet deadlines, and deliver quality embedded systems. We provide modern strategies, insights, and hands-on training to equip developers and teams with the tools they need to succeed.



EMBEDDED SOFTWARE ACADEMY BY BENINGO

Visit **www.beningo.com** to learn more

Command Parsing

01

Command Parsers

The Tradition – if/else if/else

When it comes to parsing command packets or any messaging protocol in C, I often see developers use two parsing methods:

- if/else if/ else statements
- switch statements

```
if(Packet.OP_CODE == BOOT_EXIT)
{
   Bootloader_Exit(Packet);
}
else if(Packet.OP_CODE == ERASE_DEVICE)
{
   Bootloader_Erase(Packet);
}
else if(Packet.OP_CODE == PROGRAM_DEVICE)
{
   Bootloader_Program(Packet);
}
else if(Packet.OP_CODE == QUERY_DEVICE)
{
   Bootloader_DeviceQuery(Packet);
}
else
{
   Bootloader_UnknownCommand(Packet);
}
```

Command Parsers

The Tradition – switch

A more efficient way to implement the parser would be to use a switch statement!

The switch statement, when compiled, will generate the most efficient code to execute the statement, which usually results in a jump table.

The problem is that they can become complex and difficult to maintain as they grow to over a dozen or so commands.

```
switch(Packet.OP_CODE):
 case BOOT_EXIT:
    Bootloader_Exit(Packet);
 case ERASE_DEVICE:
    Bootloader_Erase(Packet);
 case PROGRAM_DEVICE:
    Bootloader_Program(Packet);
 case QUERY_DEVICE:
    Bootloader_DeviceQuery(Packet);
 default:
    Bootloader_UnknownCommand(Packet);
```

Command Parsers

Parsing using LUTs

A LUT is an array of a structure that contains all information necessary to find a command that needs to be executed and the function that should be executed for that command.

```
/**
* Defines the command structure used to
* parse a command packet
*/
typedef struct
{
   /* Stores the command */
   Command_t Command;

   /* Function pointer to command */.
   void (*function)( CommandPacket_t * Data);
}CommandRxList_t;
```

Audience POLL Question

What method do you use to parse commands?

a) if/else if / else
b) Switch statements
c) LUTs
d) Other

Building a Command Parser

02

Building a Command Parser

A Command List

Command_t is an enumeration that contains all the commands that we would expect to receive.

```
/**
* Defines the commands being received by the bootloader.
*/
typedef enum Command_t
{
   BOOT_ENABLE,    /**< Enter bootloader */
   BOOT_EXIT,      /**< Exit bootloader */
   ERASE_DEVICE,   /**< Erase application area of memory */
   PROGRAM_DEVICE, /**< Program device with an s-record */
   QUERY_DEVICE,   /**< Set the Slave Tx buffer with the current state */

   END_OF_COMMANDS /**< End of command list */
};
```

Building a Command Parser

A Command Table

A command table is an array of structures that contain:

- A human readable enumerated type
- A pointer to a command function

```
/**
* Defines an array of all of the supported commands
* and the function that should
* be executed when the command is received.
*/
const CommandRxList_t CommandList[] =
{
   { BOOT_EXIT,     Command_Exit },
   { ERASE_DEVICE,   Command_Erase },
   { PROGRAM_DEVICE, Command_Program },
   { QUERY_DEVICE,   Command_Query },
   { END_OF_COMMANDS, 0x00 },
};
```

Building a Command Parser

Pros and Cons

There are several advantages to using a table like this which include:

- Humans can easily read through the table quickly, which gives an at a glance look of the commands in the system.
- If a command needs to be added, a developer just needs to insert a new row into the table.
- If a command needs to be removed, a developer can just remove that row from the table.
- If a command operational code needs to change, it can be updated in the enumeration and the command table does not need to change.

As we will soon see in the implementation, there are several additional advantages to using a command table such as:

- They can be navigating quickly when looking for the correct command to execute.
- They minimize code complexity
- They are easy to maintain and adapt
- They are scalable and reusable.

Building a Command Parser

Executing a Command

```
void Command_Process(CommandPacket_t const * const Packet)
{
   const CommandRxList_t * CmdListPtr;
   uint8_t CmdIndex = 0;

   // Loop through the command list and see if there is a match.
   // It will loop until a null pointer is found in the table.
   CmdListPtr = CommandList;

   while(CmdListPtr->function != NULL || CmdListPtr->Command != Packet.OpCode)
   {
      CmdListPtr++;
      CmdIndex++;
   }

   // Verify that we found a match and that the function is not NULL
   if(CmdListPtr->function != NULL)
   {
      // Execute the command and parse out the command byte
      (*CmdListPtr->function)(Packet);
   }
}
```

Audience POLL Question

Can you see the power of function pointers for improving the scalability and performance of your application?

a) Yes
b) No
c) Working on it . . .
d) other

03 Measuring Performance

Measuring Performance

Toggling I/O

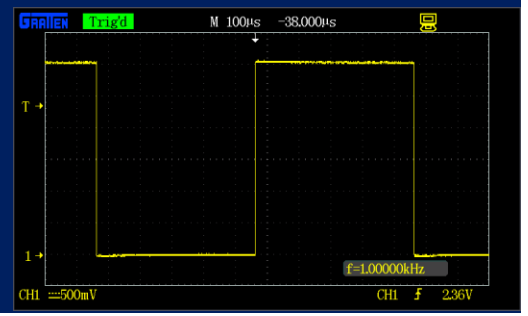
Why Use an I/O Line?

- Provides a real-time, hardware-level indication of system activity.
- Useful for measuring:
  - Task execution times.
  - Interrupt latencies.
  - System idle time and load.
- Easy to integrate with minimal software overhead.

```
// Measure function execution time
GPIO_SetPin(GPIO_PIN);

// Task to measure
PerformTask();

GPIO_ClearPin(GPIO_PIN);
```



Measuring Performance

Internal Timer with Printf

Why Use an Internal Timer?

- Precision:
  - Provides highly accurate time measurements based on system clock.
- Flexibility:
  - Can measure any section of code or system event.
- Software-Only Approach:
  - No external tools or I/O lines are required.
  - Ideal for systems where GPIO-based measurements are not feasible.

```
uint32_t start_time, end_time, elapsed_time;

// Start the timer
start_time = DWT->CYCCNT;

// Execute the task to measure
PerformTask();

// Stop the timer
end_time = DWT->CYCCNT;

// Calculate elapsed time in cycles
elapsed_time = end_time - start_time;

// Print the result
printf("Elapsed time: %lu cycles\n", elapsed_time);
```

Measuring Performance

Tracealyzer

Why Use Tracealyzer?

- Comprehensive Insights:
  - Visualize task execution, context switches, and CPU utilization in real time.
- Non-Intrusive Analysis:
  - Minimal impact on system performance compared to manual logging or printf.
- Advanced Debugging:
  - Identify bottlenecks, race conditions, and unexpected delays quickly.
- Pre-built Integration:
  - Works seamlessly with many RTOSes, but also supports baremetal systems through manual event logging.

Audience POLL Question

What technique do you like the most?

a) I/O toggling
b) Internal Timer w/ printf
c) Tracealyzer
d) other

Next Steps

04

Going Further

https://beningo.mykajabi.com/

Download Function Pointer Example Code:

- Additional Video on Baremetal Scheduling
- Scheduling Lab Instructions
- Scheduler Source Code

Additional Resources

Please consider the resources below:

- Jacob's Blogs
- Jacob's CEC courses
- Embedded Software Academy

- Embedded Bytes Newsletter
  - http://bit.ly/1BAHYXm

Consulting | Coaching | Training

www.beningo.com

Next Steps

- ✓ Mastering Function Pointers for Baremetal Systems
- ✓ Building Cooperative Schedulers and Command Parsers
- ✓ Baremetal Performance Analysis and Architecture Design
- Assertions and Validation Techniques for Baremetal
- Managing Interrupts in Baremetal Systems

CEC Continuing Education Center

DesignNews

Thank You

Sponsored by

DigiKey

BENINGO EMBEDDED GROUP

informa markets