

A decorative graphic on the left side of the slide, consisting of a grid of hexagons. The hexagons are filled with various blue and green patterns, including circuit board traces, data lines, and abstract digital motifs. The hexagons are arranged in a way that they overlap and create a sense of depth.

Techniques to Make Your Code Run Faster and More Efficiently



Table of Contents

Table of Contents

1. Introduction
2. Inline Functions
3. Loop Unrolling
4. Bit Manipulation Instead of Arithmetic Division
5. Using Register Variables
6. Constant Propagation and Pre-calculation
7. Using the restrict Qualifier
8. Precomputed Lookup Tables
9. Optimizing Data Structures and Memory Alignment
10. Conclusion



1. Introduction

1. Introduction

In embedded systems, optimizing code is crucial for maximizing performance, minimizing power consumption, and ensuring real-time responsiveness. Unlike general-purpose programming, Embedded C operates in resource-constrained environments, requiring efficient coding techniques to squeeze out every bit of performance.

This article explores practical methods to speed up execution, including inline functions, loop unrolling, bit manipulation, DMA utilization, and data structure optimization.

Each technique is accompanied by real-world

1. Introduction

This article explores practical methods to speed up execution, including inline functions, loop unrolling, bit manipulation, DMA utilization, and data structure optimization. Each technique is accompanied by real-world code examples to illustrate its impact. Whether you're working with microcontrollers or embedded applications, these strategies will help you write faster, more efficient code while maintaining clarity and reliability in your embedded projects.



2. Inline Functions

2. Inline Functions

For small, frequently called functions, the overhead of a function call can be significant. The inline keyword suggests to the compiler that the function body should be inserted at each call site. This minimizes the call overhead and can lead to performance gains.

```
1  /* Example 1: Inline Functions */
2  /* Using the inline keyword to reduce function call
3  overhead for small operations */
4  static inline uint16_t max(uint16_t a, uint16_t b) {
5      return (a > b) ? a : b;
6  }
7
8  int main(void) {
9      uint16_t a = 10, b = 20;
10     // The compiler will likely inline this function call.
11     uint16_t m = max(a, b);
12     // Additional code...
13     return 0;
14 }
```




3. Loop Unrolling

3. Loop Unrolling

Loop overhead—such as incrementing counters and evaluating loop conditions—can sometimes be reduced by manually unrolling loops. This technique is particularly effective when the number of iterations is known and relatively small.

```
1 // Unrolling a loop to process multiple data
2 // elements per iteration
3 void process_data(uint8_t *data, int length) {
4     int i = 0;
5     // Assuming 'length' is a multiple of 4 for simplicity
6     for (; i < length; i += 4) {
7         // Process 4 elements per iteration to
8         // minimize loop overhead
9         data[i] = data[i] * 2;
10        data[i+1] = data[i+1] * 2;
11        data[i+2] = data[i+2] * 2;
12        data[i+3] = data[i+3] * 2;
13    }
14 }
```




4. Bit Manipulation Instead of Arithmetic Division

4. Bit Manipulation Instead of Arithmetic Division

Division operations can be relatively slow, especially on microcontrollers without hardware division support. When dealing with powers of two, bit-shifting offers a much faster alternative.

```
1 /* Using bit shifting to perform division by 2
2    (a power-of-two optimization) */
3 uint8_t divide_by_two(uint8_t value) {
4     // Right-shift by one is equivalent to division by 2.
5     return value >> 1;
6 }
```

Tip: Always verify that the data types and expected behaviors (e.g., handling of negative numbers) align with bit-level operations.



5. Using Register Variables

5. Using Register Variables

Historically, the `register` keyword was used to suggest that the compiler store frequently accessed variables in CPU registers rather than memory. Although modern compilers are very good at optimization, explicitly using `register` can still hint at performance-critical code areas.

```
1  /* Hinting the compiler to use registers for frequently used variables */
2  void add_arrays(const int *a, const int *b, int *result, int n) {
3      for (int i = 0; i < n; i++) {
4          register int tempA = a[i];
5          register int tempB = b[i];
6          result[i] = tempA + tempB;
7      }
8  }
```

Note: Many modern compilers ignore the `register` keyword, but in specific scenarios,

5. Using Register Variables

```
1  /* Hinting the compiler to use registers for frequently used variables */
2  void add_arrays(const int *a, const int *b, int *result, int n) {
3      for (int i = 0; i < n; i++) {
4          register int tempA = a[i];
5          register int tempB = b[i];
6          result[i] = tempA + tempB;
7      }
8  }
```

Note: Many modern compilers ignore the register keyword, but in specific scenarios, especially with older toolchains, it can still be beneficial.



6. Constant Propagation and Pre-calculation

6. Constant Propagation and Pre-calculation

If an expression in a loop doesn't change with each iteration, compute it outside of the loop. This not only improves execution speed but also makes your code clearer.

```
1  /* Moving constant calculations outside
2  the loop for efficiency */
3  #define MULTIPLIER 5
4
5  void scale_array(int *array, int length) {
6      // Pre-calculate constant factor.
7      int factor = MULTIPLIER;
8      for (int i = 0; i < length; i++) {
9          array[i] *= factor;
10     }
11 }
```

Insight: Modern compilers often perform

6. Constant Propagation and Pre-calculation

```
1  /* Moving constant calculations outside
2  the loop for efficiency */
3  #define MULTIPLIER 5
4
5  void scale_array(int *array, int length) {
6      // Pre-calculate constant factor.
7      int factor = MULTIPLIER;
8      for (int i = 0; i < length; i++) {
9          array[i] *= factor;
10     }
11 }
```

Insight: Modern compilers often perform constant propagation automatically. However, explicitly coding this way can prevent accidental inefficiencies and improve code readability.



7. Using the restrict Qualifier

7. Using the restrict Qualifier

When working with pointers, using the restrict qualifier tells the compiler that the pointed-to data will not be aliased elsewhere. This allows the compiler to optimize memory accesses more aggressively—often resulting in faster code.

```
1 void vector_add(const float * restrict a, const float * restrict b,  
2                float * restrict result, size_t n) {  
3     for (size_t i = 0; i < n; i++) {  
4         result[i] = a[i] + b[i];  
5     }  
6 }
```

Insight: By assuring the compiler that the pointers do not overlap, you enable better scheduling of memory operations and potentially more efficient use of registers.



8. Precomputed Lookup Tables

8. Precomputed Lookup Tables

Complex computations, such as trigonometric functions or logarithms, can be expensive if computed repeatedly. Precomputing values and storing them in a lookup table can save processing time at runtime. This approach trades off memory usage for speed—an often acceptable compromise in embedded systems.

```
1 // Lookup Table for Fast Trigonometric Computation
2 #include <math.h>
3 #define TABLE_SIZE 256
4 #define PI 3.14159265f
5
6 float sin_table[TABLE_SIZE];
7
8 void init_sin_table(void) {
9     for (int i = 0; i < TABLE_SIZE; i++) {
10         sin_table[i] = sinf(i * (PI / (TABLE_SIZE - 1)));
11     }
12 }
```


8. Precomputed Lookup Tables

```
1 // Lookup Table for Fast Trigonometric Computation
2 #include <math.h>
3 #define TABLE_SIZE 256
4 #define PI 3.14159265f
5
6 float sin_table[TABLE_SIZE];
7
8 void init_sin_table(void) {
9     for (int i = 0; i < TABLE_SIZE; i++) {
10         sin_table[i] = sinf(i * (PI / (TABLE_SIZE - 1)));
11     }
12 }
13
14 float fast_sin(int index) {
15     // Ensure index is within bounds
16     index %= TABLE_SIZE;
17     return sin_table[index];
18 }
```

Tip: Use lookup tables where the slight inaccuracy from discretization is acceptable compared to the performance gain.



9. Optimizing Data Structures and Memory Alignment

9. Optimizing Data Structures and Memory Alignment

Choosing the right data types and aligning data structures can reduce memory access time. Using the smallest data type needed not only minimizes memory footprint but can also improve cache utilization on systems with caches. Aligning structures properly may also reduce the number of memory cycles required to access data.

```
1 // Efficient Data Structure Alignment
2 typedef struct __attribute__((packed, aligned(4))) {
3     uint16_t id;
4     uint8_t value;
5     uint8_t flag;
6 } SensorData;

void process_sensor_data(SensorData *data, size_t count) {
```

9. Optimizing Data Structures and Memory Alignment

```
1 // Efficient Data Structure Alignment
2 typedef struct __attribute__((packed, aligned(4))) {
3     uint16_t id;
4     uint8_t value;
5     uint8_t flag;
6 } SensorData;
7
8 void process_sensor_data(SensorData *data, size_t count) {
9     for (size_t i = 0; i < count; i++) {
10         if (data[i].flag) {
11             // Process data[i].value as required
12         }
13     }
14 }
```

Tip: Carefully consider the data types you use and their alignment. While compilers often handle alignment automatically, explicitly specifying it can sometimes yield performance benefits on certain hardware.



10. Conclusion

10. Conclusion

Optimizing Embedded C code is often a balancing act between raw speed, memory usage, and code maintainability.

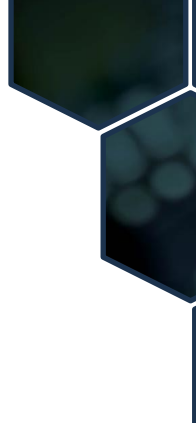
As with any optimization effort, it's crucial to:

- **Profile your code:** Measure performance before and after applying changes.
- **Understand your hardware:** Tailor optimizations to the specific capabilities and limitations of your target platform.
- **Balance trade-offs:** Ensure that improvements in speed do not excessively complicate the codebase or jeopardize maintainability.

10. Conclusion

- **Understand your hardware:** Tailor optimizations to the specific capabilities and limitations of your target platform.
- **Balance trade-offs:** Ensure that improvements in speed do not excessively complicate the codebase or jeopardize maintainability.

By combining these strategies with the ones previously discussed, you'll be well-equipped to squeeze every bit of performance out of your embedded systems while keeping your code clear and maintainable.



"Thanks for watching! If you enjoyed this video, make sure to hit the like button and subscribe to stay updated with my latest content."

Don't forget to check out my other videos for more tips and tutorials on Embedded C, Python, hardware designs, etc. Keep exploring, keep learning, and I'll see you in the next video!"

