Appendix A:

Improving Code Size With the MPLAB C18 Compiler

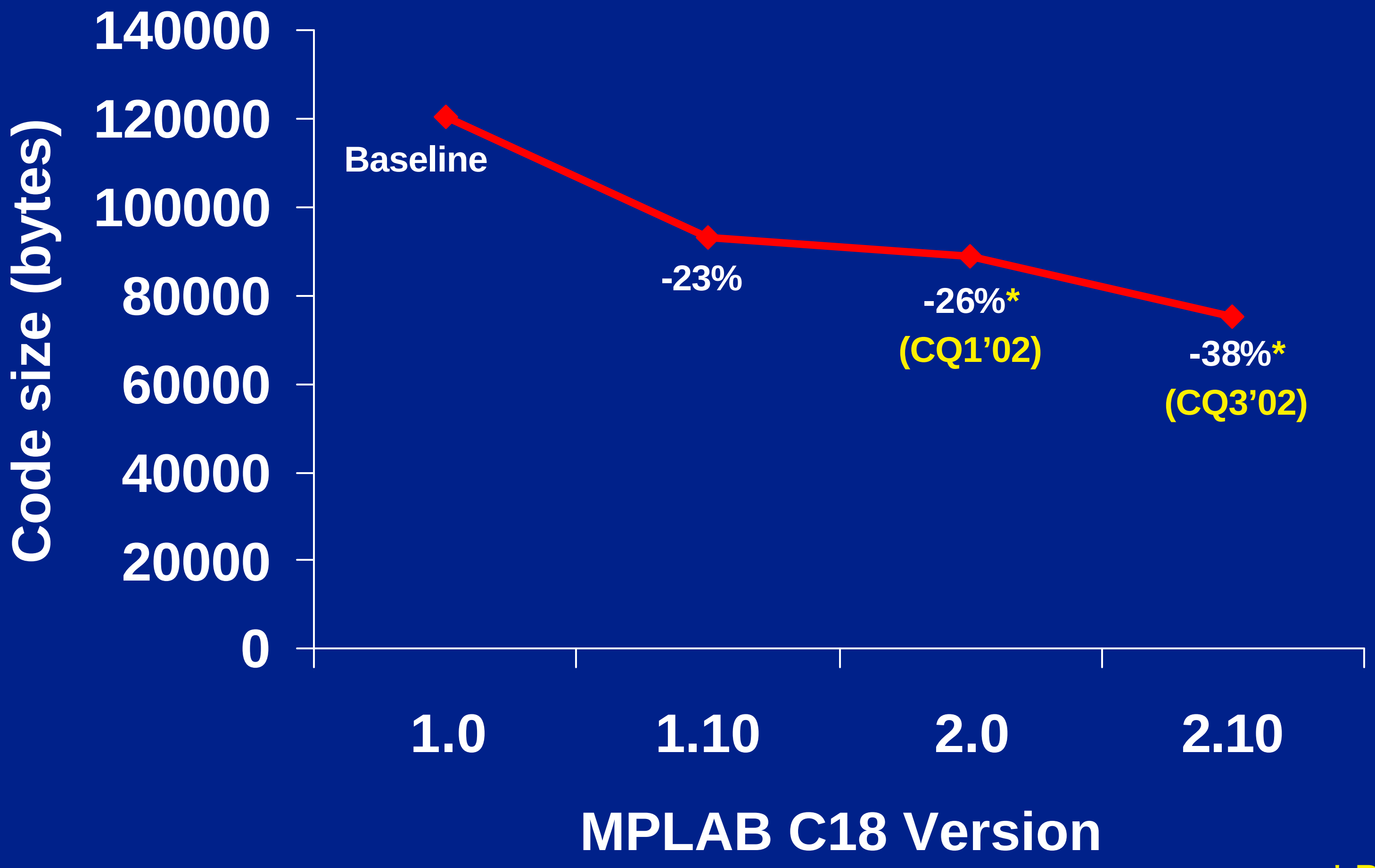
Our Goal: *To understand how to reduce C application code size on PIC18 MCUs through intelligent use of MPLAB C18 and careful structuring of C code.*

Suggestion #1

Use the latest version of MPLAB C18

Code Size Comparison

Default Options



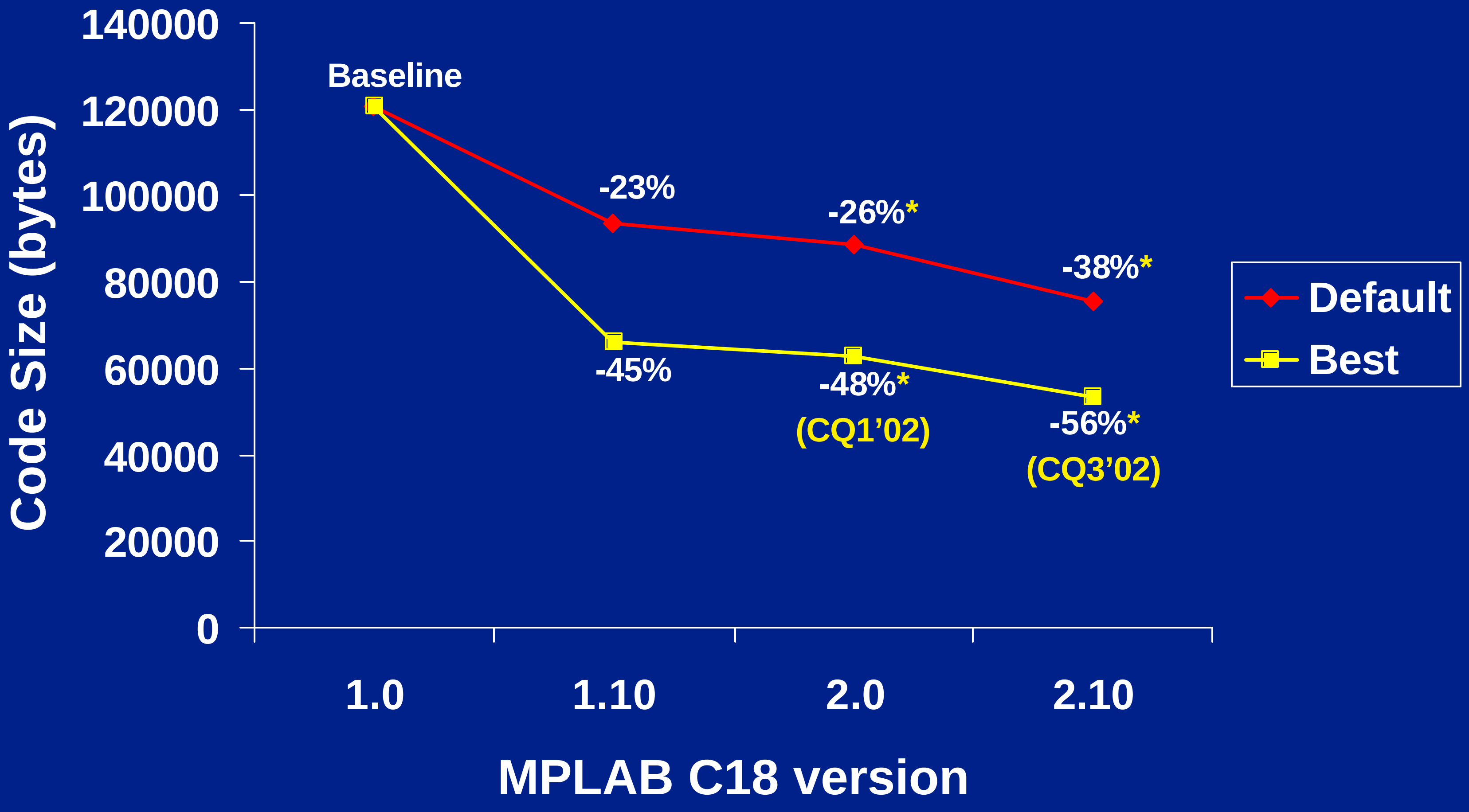
* Projected

Suggestion #2

Carefully select command-line options

Code Size Comparison

Choosing Command-Line Options



Command-Line Options

LFSR Use

- MPLAB-C18’s **`-lfsr`** switch enables use of the LFSR instruction
- Currently, MPLAB-C18 assumes that LFSR shouldn’t be used without the **`-lfsr`** switch given
- The switch should always be used when it is known that the LFSR errata doesn’t exist on the targeted part

Command-Line Options

Optimizations

- All of MPLAB-C18's optimizations currently target code size
- Optimizations should be enabled for smallest code size
- NOTE: Optimizations may interfere with MPLAB debugging

Command-Line Options

Memory Model

- MPLAB-C18 has two memory models:
  - `-ms`: small memory model (pointers to program memory are 16-bits wide)
  - `-ml`: large memory model (pointers to program memory are 24-bits wide)
- Use `-ms` whenever possible

Suggestion #3

Select appropriate storage class for data

Command-Line Options

Data Storage Class

- Default storage class for parameters and local variables is **`auto`**
  - Parameters are passed on the software stack
  - Locals are located on the software stack

Using auto Variables

Example - calculate the expression (a + b):

```
movlw      offset(a)
movff      PLUSW2, tmp
movlw      offset(b)
movf       PLUSW2
addwf      tmp
```

6 program words

(not counting prolog/epilog)

Command-Line Options

Data Storage Class

- C also provides for **`static`** local variables
- MPLAB-C18 extends C with **`static`** parameters (available in v1.10 and later)
- For example:

```
char add( static char a, static char b )
{
   static char result;
   result = a + b;
   return result;
}
```

Using static Variables

Example - calculate the expression (a + b):

```
movlb        b*
movf         b
addwf        a
```

*likely target for optimization

3 program words

(no prolog/epilog required)

`static` Gotchas

- Gotcha #1 - Reentrant code

  Variables may overwrite themselves

  - Recursion (function calls itself)
  - Function called (directly or indirectly) from main() and an ISR.

`static` Gotchas

- Gotcha #2 - Function pointers

  Address of parameters not known at compile time

- Function pointers may not be used with functions containing static parameters

`static` Gotchas

- Gotcha #3 - Matching declarations

  All declarations must use explicit storage class if not all files are compiled with the same default

- Example:

```
char add( char a, char b );
```

Will only work if the default storage class is identical in both the declaring and defining files.

`static` Gotchas

- What if one of the “static Gotchas” applies to your code?
  - Best case: use `-ol` on all files and explicit `auto` storage class as needed.
  - Intermediate case: Use `-ol` on as many files as possible and explicit storage classes as needed.
  - Worst case: Don’t use `-ol`, but use explicit `static` storage class as much as possible.

Command-Line Options

Data Storage Class

- MPLAB-C18 v2.0 and later extends C with the **`overlay`** storage class for local variables
  - Behaves identically to the **`static`** storage class, except:
  - RAM locations are overlaid by the linker when possible based on a call tree analysis
  - Default storage class can be set to **`overlay`** using the **`-sco`** option

MICROCHIP
MASTER'S

Suggestion #4

Choose smallest data type possible

MPLAB-C18 Data Types

Type	Min Value	Max Value
`unsigned char`	0	255
`signed char`	-128	127
`unsigned int`	0	65,535
`signed int`	-32,768	32,767
`unsigned short long`	0	16,777,215
`signed short long`	-8,388,608	8,388,607
`unsigned long`	0	4,294,967,295
`signed long`	-2,147,483,648	2,147,483,647

Using Appropriate Data Types

$$c = a + b$$

char:

```
MOVLB      b
MOVF       b,0,1
ADDWF      a,0,1
MOVWF      c,1
```

(4 words)

int:

```
MOVLB    a
MOVF     b,0,1
ADDWF    a,0,1
MOVWF    c,1
MOVF     high(b),0,1
ADDWFC   high(a),0,1
MOVWF    high(c),1
```

(7 words)

Suggestion #5

Use access RAM for your variables

Variable Allocation

Using Access RAM

- MPLAB-C18 allows for efficient use of unbanked RAM with the **`near`** type specifier
- RAM variables will default to **`near`** by using the **`-oa`** option
- Compiler won't emit **`movlb`** instructions for accessing these variables

Variable Allocation

Using Access RAM

- Use the **`near`** specifier for the most frequently accessed variables
- Gotcha: as with **`static`** and **`overlay`**, prototypes must match definitions

Suggestion #6

Keep definitions in same file with references

Variable Allocation

Defining Variables

- MPLAB-C18 can be more aggressive optimizing variables in the files where they are defined.

Source code:

```
char a, b, c;
void foo( void )
{
   c = a + b;
}
```

Machine code:

```
MOVLB     b
MOVF      b,0,1
ADDWF     a,0,1
MOVWF     c,1
```

(4 words)

Variable Allocation

Defining Variables

- MPLAB-C18 must be more conservative with externally-defined variables

Source code:

```
extern char a, b, c;
void foo( void )
{
   c = a + b;
}
```

Machine code:

```
MOVLB     b
MOVF      b,0,1
MOVLB     a
ADDWF     a,0,1
MOVLB     c
MOVWF     c,1
```

(6 words)

Suggestion #7

Use #pragma varlocate

Using `#pragma varlocate`

- Use #pragma varlocate to tell the compiler what bank a variable is located in

Source code:

```
extern char a, b, c;
void foo( void )
{
   c = a + b;
}
```

Machine code:

```
MOVLB     b
MOVF      b,0,1
MOVLB     a
ADDWF     a,0,1
MOVLB     c
MOVWF     c,1
```

(6 words)

Using `#pragma varlocate`

- Improves MPLAB-C18 banking optimizer

Source code:

```
#pragma varlocate 3 a, b, c
extern char a, b, c;
void foo( void )
{
   c = a + b;
}
```

Machine code:

```
MOVLB      b
MOVF       b,0,1
ADDWF      a,0,1
MOVWF      c,1
```

(4 words)

Using `#pragma varlocate`

Gotcha: *has no impact on how variables are actually allocated*

Suggestion #8

Replace Common Expressions With Variables

Common Sub-Expression Elimination

- Applies to all types of expressions

Source code:

```
MY_STRUCT s[10];
for(i=0; i<10; i++)
{
    s[i].a = i;
    s[i].b = 34;
}
```

Code size:

```
10 words to calculate s[i]
2 words to assign i
10 words to calculate s[i]
3 words to assign 34

    = 25 words total
```

Common Sub-Expression Elimination (Contd.)

Source code:

```
MY_STRUCT s[10];
MY_STRUCT *p = &(s[0]);

for(i=0; i<10; i++)
{
  p->a = i;
  p->b = 34;
  p++;
}
```

Code size:

```
0 words to calculate s[i]
6 words to assign i
7 words to assign 34
4 words to increment p

    = 17 words total
```

Suggestion #9

Don’t Use a Variable When a Constant Will Do

Constant Evaluations

- Pre-calculate all values that can be determined at compile-time.

Original source:

```
a = 2;
b = 17 + 52 * a;
c = b;
```

Transformed source:

```
c = 121;
```