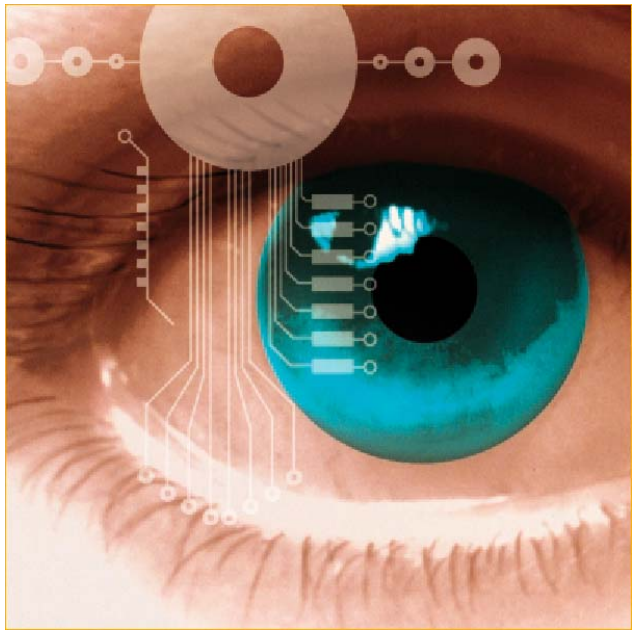
Benchmarking hints

How to tune
IAR Embedded Workbench
for best performance



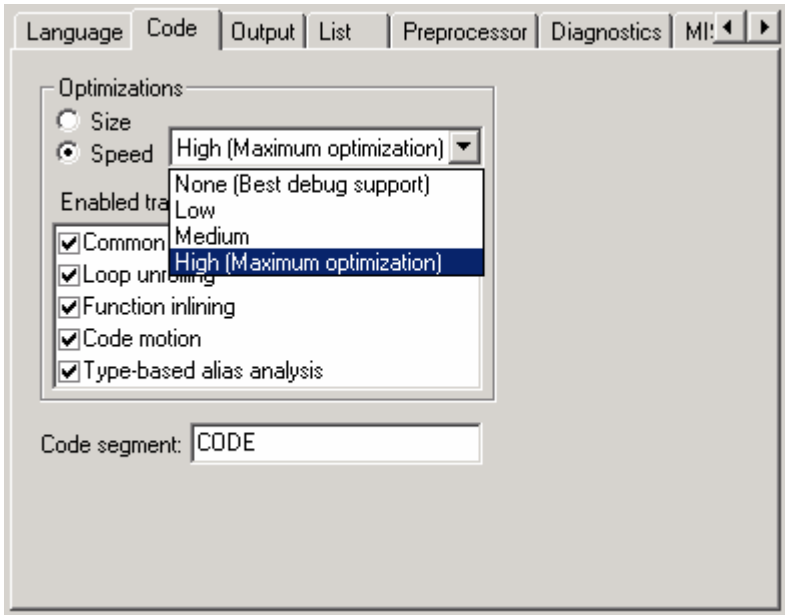
Optimization goal

Size or Speed?

Optimization level and type can be specified for the entire application and for individual files. In source code, the `#pragma optimize` directive allows you to do this even for individual functions.

The purpose of optimization is to reduce the code size and to improve the execution speed. When only one of these two goals can be satisfied, the compiler prioritizes according to the settings you specify.

Exploring the effects of the different transformations may lead to a better result. For example, the fact that ***Function inlining*** is more aggressive on speed optimization makes some programs smaller on the speed setting than on the size setting.



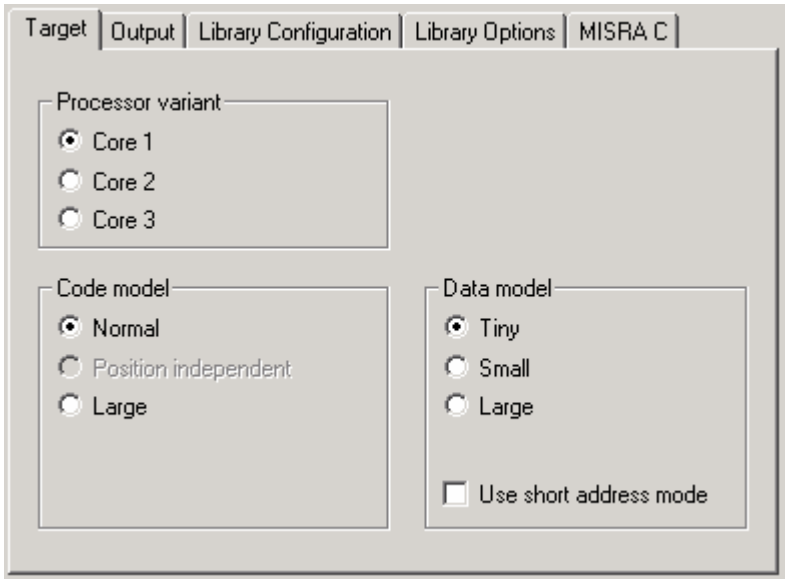
Memory model

Memory model

Choose the smallest possible memory model

Benefits:
- Smaller addresses
- Smaller instructions
- Smaller pointers

⇒More efficient
⇒Less code



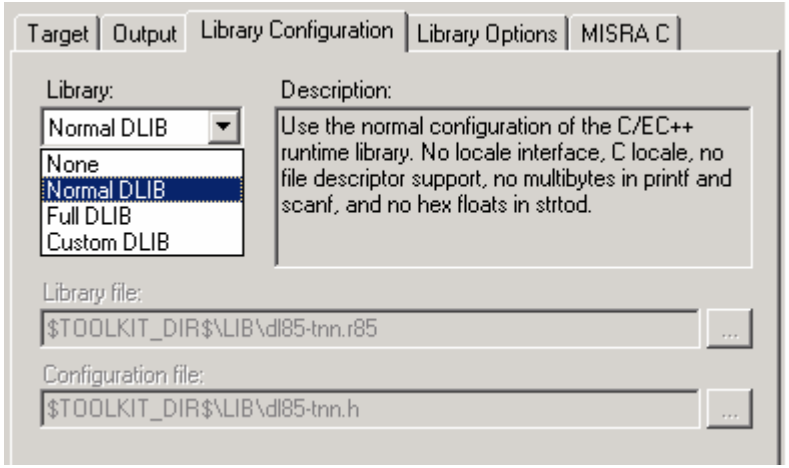
Runtime environment

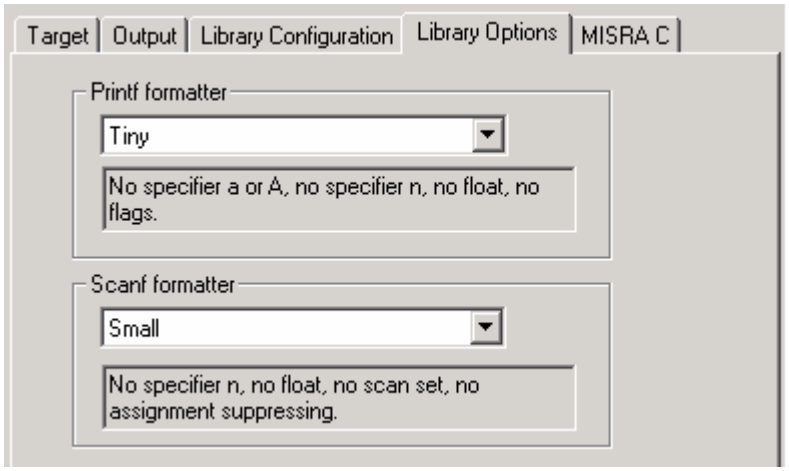
Adapting the runtime environment

•By default, the runtime libraries are compiled at highest size optimization level. You should rebuild them if you are optimizing for speed!

•Select the required level of support for certain standard library functionality like, `locale`, `file descriptors`, and `multibytes` by choosing the appropriate library configuration.

•Select library options for `scanf` input and `printf` output formatters according to your needs. The smallest formatters are not selected by default.



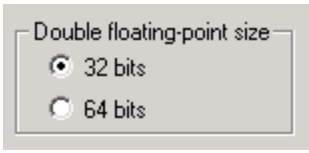


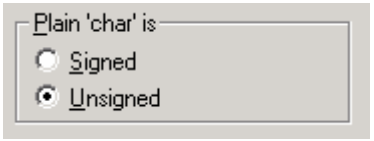
Data types

Data types have big influence on code size/speed

- Choose the smallest data types
- Use unsigned char if possible

  ⇒ Allows bit operations to be performed instead of arithmetic



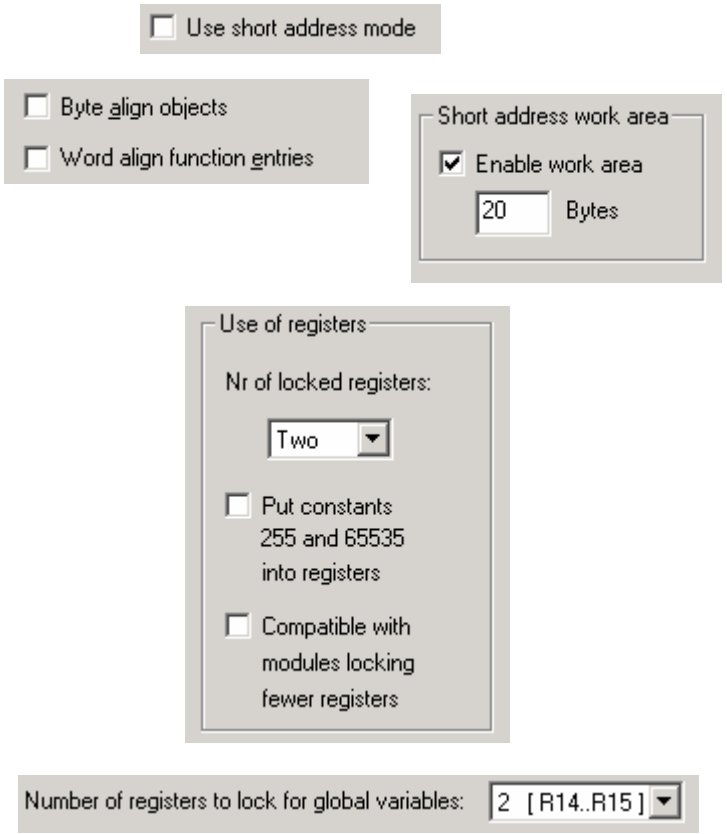


Target-specific options

Check for target-specific options that gain performance

Example:

- Efficient addressing modes
  ⇒efficient memory accesses
- Locking registers for constants/variables
  ⇒more efficient code for operations on registers than on memory
- Even align functions entries
  ⇒even aligned instructions gain speed
- Byte align objects
  ⇒requires less memory for storage but might give bigger code



Benchmark code

Use relevant benchmark code

•Embedded systems benchmarks shall address the characteristics of embedded programs.

•Real applications are usually good for benchmarks but, make sure that the code can execute. IAR XLINK will remove un-referenced code and variables but not all linkers have this ability.

•Make sure that the test code is not affected by the test harness (test support functions). The example to the right is actually benchmarking `printf()`.

•Compare linked code. One compiler may inline code where another makes a library call.

•Use an application that you are familiar with!

```
unsigned long fib(unsigned long x)
{
 if (x > 2)
  printf("%ld\n",x); // Test harness
  return(fib(x-1)+fib(x-2));
 else
  return(1);
}
```