Optimizer

Reference Manual

Table of Contents

What is the Catalina Optimizer?

The Catalina Optimizer is a code optimizer that can substantially reduce the code size and increase the performance of programs compiled using the Catalina C compiler.

The Catalina C compiler is a free, open source, ANSI compliance C compiler for the Parallax Propeller. It can be downloaded from http://catalina-c.sourceforge.net/

Status

The Catalina Optimizer is released with each version of Catalina. You should only use the Optimizer that coincides with the same Catalina release. For a list of differences since the previous release, see the section later in this document titled What's new in this release?

Features

- Seamless integration with the Catalina C compiler;
- Five optimization levels;
- Linux and Windows versions;
- Optimizes C user programs as well as C library code, and also compatible LMM PASM functions;
- Works with all Catalina memory models (LMM, XMM, CMM, NMM);
- Typically reduces program code size by 10 to 15%;
- Typically improves program performance by 5% to 15%;
- Support for both Propeller 1 and Propeller 2 programs.

License

Early versions of the Catalina Optimizer were neither free nor open source. However, since Catalina version 4.9, the Optimizer is both. The source is included in each Catalina release.

Using the Optimizer

The Catalina Optimizer is not normally invoked independently – it is normally invoked internally by Catalina when the **–Ox** command line option is included in a Catalina command (where **x** is the optimization level – currently **1**, **2, 3, 4** or **5** are supported).

For example:

```
catalina -lc hello_world.c -O4
```

Three levels of code optimization are currently supported:

`-O1`	: basic optimization (single pass optimization)
`-O2`	: intensive optimization (performs a second optimization pass)
`-O3`	: same as -O2 but with automatic inlining
`-O4`	: same as -O3 but with automatic elimination of unused code
`-O5`	: same as -O3 but with load optimization

Note that unlike many other Catalina command line options, there can be no space between the **–O** and the level (**1**, **2**, **3**, **4** or **5**).

The single optimization pass (`-O1`) does most of the work, and typically reduces the size of code segments by 10%. The increase in overall program performance achieved typically ranges from 5% to 10% depending on the program itself (the larger savings are typical for programs that use the XMM memory model).

The second optimizer pass (`-O2`) can save an additional few percent in both space and execution time in larger programs.

The third option (`-O3`) searches for functions where inlining the body of the function would save the overhead of actually performing the function call – provided the inlining does not make the code size any larger (this optimization can make a very large difference in programs that are constructed from many small functions).

The fourth option (`-O4`) removes functions from the code segment – i.e. functions that are not referenced by the main program or any function referenced – directly or indirectly – by the main program.

The fifth option (`-O5`) optimizes load operations that can be performed in less space – e.g. loads that can be done in immediate mode rather than from Hub RAM.

Note that the optimizer works on ALL the C code in the program – including library functions (i.e. you don't need to maintain two versions of either the standard C libraries or your own libraries – one compiled with the optimizer and one without).

Using the Optimizer with BlackBox

Optimized code can still be used with the BlackBox debugger, except that inlining should probably not be used if you want to debug functions that may have been inlined – most debugger operations will work correctly on optimized code (e.g. break and single step), but those operations that depend on the debugger identifying the return address by reading the stack frame may (e.g. step out, step up) will not work correctly on inlined functions.

Optimizer Options

Although it is not intended to be invoked manually, if the **–v** (verbose) option is used in a Catalina compilation the `optimizer` can be seen being invoked with various options. So here is a list of the options supported by the program.

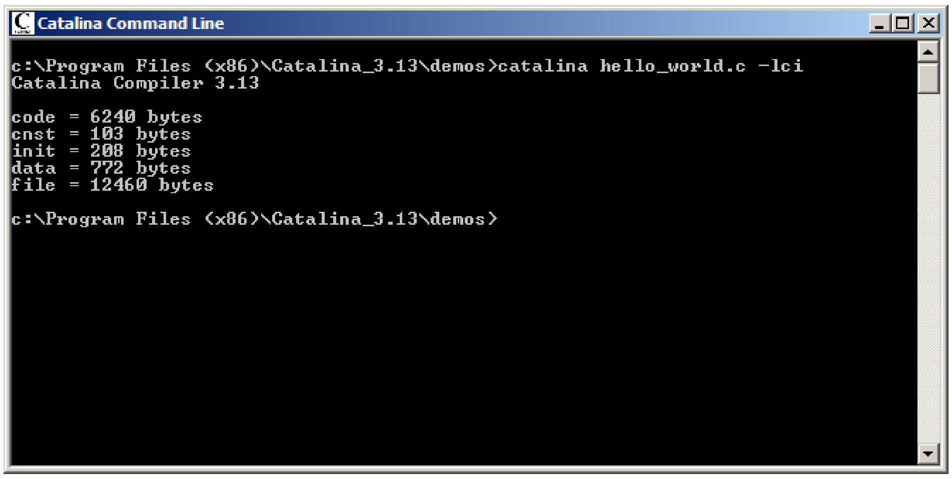
```
-? or -h  print a helpful message (and exit)
-b        binary output (default)
-d        generate listing
-e        eeprom output;
-ilevel   set information level (note no space!)
-L path   path to libraries
-M size   memory size to use (default is 64k)
-o name   output optimized results to file 'name'
-Olevel  optimization level (-O1, -O2, -O3, -O4, -O5 - note no space!)
-u        do not remove intermediate output files
-p vers   Propeller version (1 or 2)
-v        verbose (output information messages)
-w        enable warnings
```

Optimizer Example

A normal Catalina compilation with no optimization, such as:

```
catalina -lc hello_world.c
```

might produce the following output:



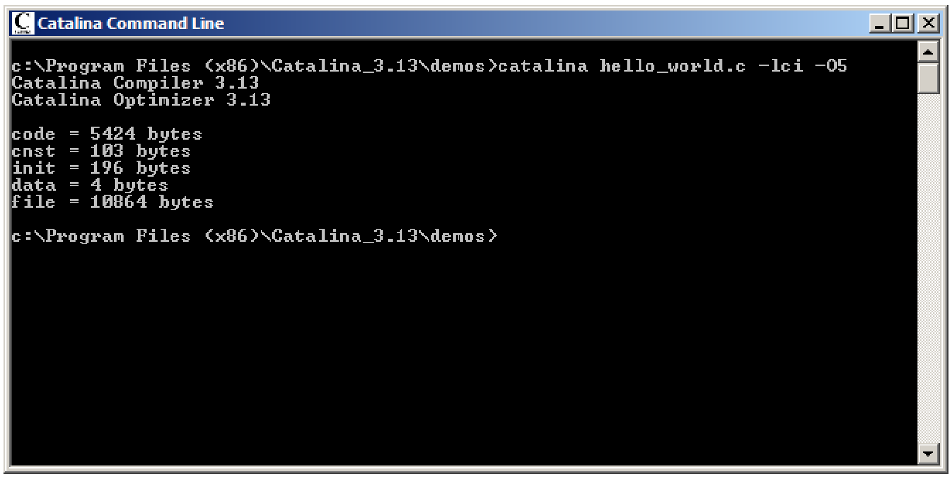
```
c:\Program Files (x86)\Catalina_3.13\demos>catalina hello_world.c -lci
Catalina Compiler 3.13

code = 6240 bytes
cnst = 103 bytes
init = 208 bytes
data = 772 bytes
file = 12460 bytes

c:\Program Files (x86)\Catalina_3.13\demos>
```

Note the code segment size for this program is 6240 bytes.

However, when optimization is included, such as:

```
    catalina -lc hello_world.c -O5
```

the following output might be produced:



```
c:\Program Files (x86)\Catalina_3.13\demos>catalina hello_world.c -lci -O5
Catalina Compiler 3.13
Catalina Optimizer 3.13

code = 5424 bytes
cnst = 103 bytes
init = 196 bytes
data = 4 bytes
file = 10864 bytes

c:\Program Files (x86)\Catalina_3.13\demos>
```

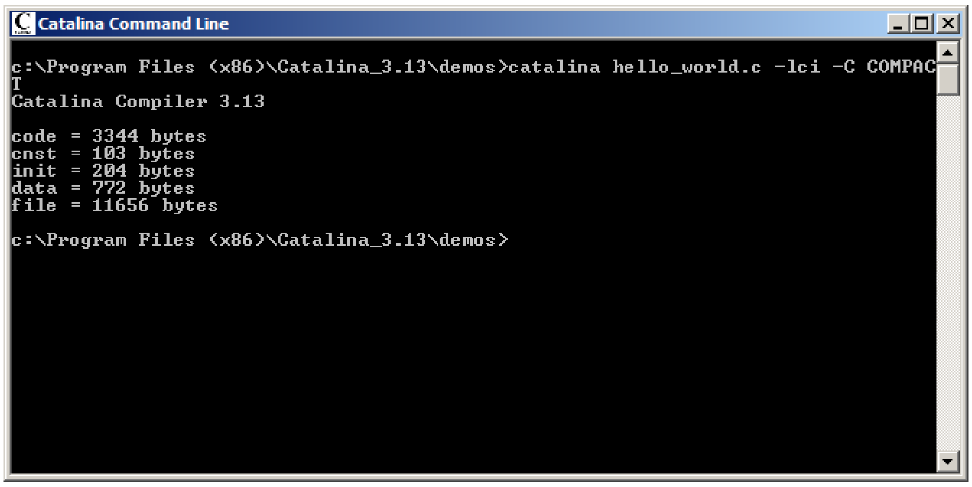
The code segment size is now 5424 bytes – a reduction in code size of 13%.

Note that when the Catalina Optimizer is used, the Catalina compilation process can take longer than normal.

The optimizer also works for CMM (Compact Memory Model) programs. For example, compiling the same program in compact mode, but with no optimization:

```
    catalina -lc hello_world.c -C COMPACT
```

the following output might be produced:



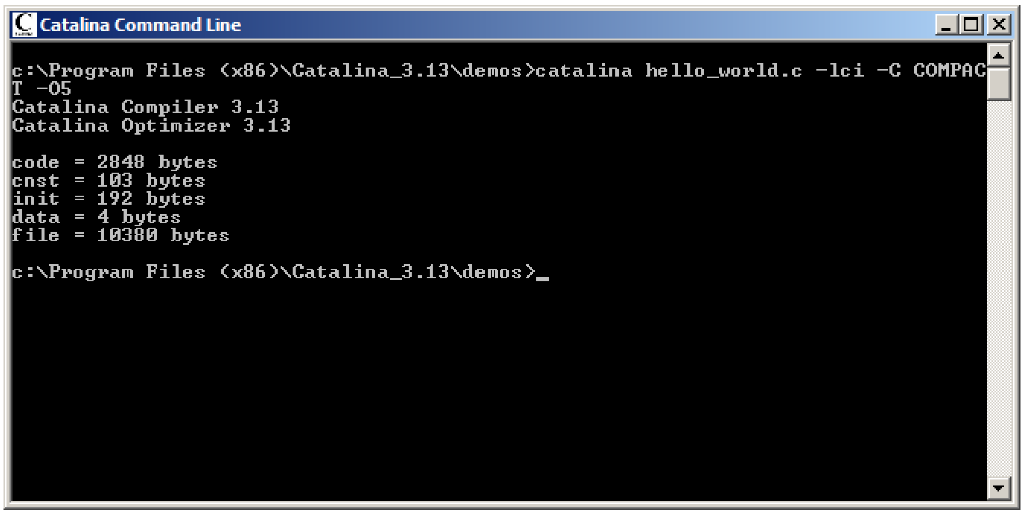
```
c:\Program Files (x86)\Catalina_3.13\demos>catalina hello_world.c -lci -C COMPAC
T
Catalina Compiler 3.13

code = 3344 bytes
cnst = 103 bytes
init = 204 bytes
data = 772 bytes
file = 11656 bytes

c:\Program Files (x86)\Catalina_3.13\demos>
```

The code segment size is 3344 bytes.

When optimization is included:

```
catalina -lc hello_world.c -C COMPACT -O5
```

the following output might be produced:



```
c:\Program Files (x86)\Catalina_3.13\demos>catalina hello_world.c -lci -C COMPAC
T -O5
Catalina Compiler 3.13
Catalina Optimizer 3.13

code = 2848 bytes
cnst = 103 bytes
init = 192 bytes
data = 4 bytes
file = 10380 bytes

c:\Program Files (x86)\Catalina_3.13\demos>_
```

The code segment size is now 2848 bytes – a reduction in code size of 15%.

Optimizer Phases

The Catalina Optimizer consists of the main program (**catoptimize** for LMM, XMM, NMM and **cmmoptimize** for CMM), which invokes up to fourteen separate optimization phases, represented by separate executable (**catopt_0** .. **catopt_13** for LMM, XMM, NMM and **cmmopt_0 .. cmmopt_13** for CMM). These phases cannot be used independently – each phase (except the first) typically requires the information generated by one or more of the preceding phases. The phases executed, and the order in which they are executed, depends on the optimization level specified. The numbering of the phases is arbitrary, and has nothing to do with the optimization level selected.

The first phase is for optimizing loads, but (since it was added most recently) is invoked only when optimization level 5 is specified:

cxxopt_0 optimize the load instructions in the program to use smaller instructions where possible.

The next three phases are for inlining functions, and are invoked only when optimization level 3 (or greater) is specified:

`cxxopt_1` identify all the potentially 'inlinable' function calls in the program.

`cxxopt_2` identify all 'inlinable' functions, and generate an 'inline' version. To be inlinable, a function must:

not use the stack for arguments, *and*
not require any local variables, or use the frame pointer, *and*
not have its address taken elsewhere in the program, *and*
either be leaf function (i.e. one that makes no other calls),
or be called once only,
or be less than 6 longs in size.

`cxxopt_3` replace all 'inlinable' function calls with an equivalent 'inlined' versions of the function.

The next three phases are for optimizing jumps. They are invoked when optimization level 1 (or greater) is specified:

`cxxopt_4` identify all the absolute jumps in the program that can potentially be replaced by relative jumps. To do this, an intermediate version of the program must be compiled (to find out exactly how long each of the jumps are!)

`cxxopt_5` identify all the relative jumps that are within the range that can be optimized.

`cxxopt_6` replace all the absolute jumps with relative jumps if they are within range.

The next three phases are for removing redundant instructions. They are invoked when when optimization level 1 (or greater) is specified:

`cxxopt_7` identify all function calls in the program.

`cxxopt_8`	identify functions that don’t spill arguments to stack (and therefore don’t require the BC argument to be passed).
`cxxopt_9`	remove redundant assignments to BC for functions that don’t use it.
`cxxopt_10`	remove redundant assignments from functions where the value assigned is not used again before the function returns.

The next two phases are for removing unreferenced code and data. They are invoked only when optimization level 4 (or greater) is specified**:**

`cxxopt_11`	identify all function calls and address loads in the program.
`cxxopt_12`	identify the functions used by each function, or whose address is taken.
`cxxopt_13`	remove functions not explicitly required.
`cxxopt_14`	(Propeller 2 CMM only) insert “alignl” PASM keywords where required.

Finally, if optimization level 2 (or greater) is specified, phases 4, 5 and 6 are then repeated – this is because the removal of redundant instructions sometimes allows additional absolute jumps to be turned into relative jumps (in LMM, XMM or NMM because the destination of the jump now falls into the range -511 .. +511, or in CMM in the range -255 .. 255).

Then the final version of the program is assembled.

Typical Optimizer Results

The amount of code that is saved by each optimization level can vary tremendously depending on the type of program. However, the following is quite typical:

Optimization Level 1: 10%

Optimization Level 2: 1-2%

Optimization Level 3: 2-3%

Optimization Level 4: 1-5%

Optimization Level 5: 1-5%

Optimization level 1 is fairly consistent at saving around 10% of code size purely due to the replacement of absolute jumps by relative jumps, and the elimination of redundant instructions.

Optimization level 2 may make no additional difference over and above optimization level 1 in small programs – in such programs it is typical that all possible jump optimizations have already been made in level 1. However, in larger programs it can often save an additional few percent of total code size.

Optimization level 3 may make no difference at all in programs that essentially consist of a single ‘main’ function, but in programs that consist of a relatively large number of small function calls, this optimization level can save up to 30% of the total code size, because of the overheads involved in making each function call.

Optimization level 4 will only make a significant difference if there are unused functions or variables in the program. However, this is often the case for C code that has been automatically generated, or in code that is still under development.

Optimization level 5 will make a significant difference for most programs.

Note that for optimization levels 1 to 3, it is only the *code segment size* that can be reduced – the size of any data segments (e.g. strings, initialized variables, jump tables etc) *cannot* be reduced unless there are unused functions or unused variables, and optimization level 4 is used..

Also, in a typical Catalina binary, quite a few kilobytes can be taken up by the kernel and various drivers and plugins. However, this space is typically re-used as heap or stack at run time.

All this means is that you will not always see an apparent 15% reduction in the size of the final binary file even when the optimizer has removed 15% of the code. However, you will see significant performance improvements – usually on the same magnitude as the code savings.

For example, on the standard C **Dhrystone** integer benchmark program, the Catalina optimizer reduces the code size by around 15%, and increases the dhrystone performance by 8%. [1]

[1] Note that previous versions of the optimizer achieved better *apparent* improvements – this is because the Catalina compiler and kernels have themselves improved over the same time, leaving less scope for optimization to improve things further.

Optimizing hand-coded LMM PASM

The Catalina optimizer optimizes all code in the program, including library functions and hand-coded LMM PASM functions. However, when optimizing hand-coded LMM PASM, that PASM must be formatted so as to be recognizable by the Optimizer.

The following formatting rules must be obeyed:

1. No tabs embedded in the instructions;
2. All labels must start in column 1;
3. All comment markers `'` `{` or `}` must appear in column 1;
4. All opcodes must be in lower case. For example:

   mov *not* **MOV**
5. All special registers **BC**, **RI**, **FP** and **SP** must be in upper case (this does not apply to the general purpose registers r0 .. r23). For example:

   FP *or* **BC** *not* **fp** *or* **bc**
6. All LMM primitive names must be in upper case. For example:

   LODA *not* **loda**
7. Single spaces before the conditionals, between the conditionals and the opcode, between the opcode and the operands, and between the operands (with no space before the comma separating operands). For example:

   ```
    if_z mov FP, #1
   ^    ^   ^   ^   single spaces should appear here
   ```
8. No comments or blank lines between the parts of an LMM primitive. For example:

   ```
    jmp #LODA
                     no blank line or comment should appear here
    long @my_variable
   ```

If in doubt, follow the code format used by the generated code or the hand-coded library functions as closely as possible.

For example, here is an actual (but quite trivial) Catalina library function that will be recognized as being able to be 'inlined' by the Catalina Optimizer:

```
C__ina
 mov r0, INA
 jmp #RETN
```

If the `-O3` option is selected when optimizing, all calls to the library function **_ina()** will be replaced by the single inline instruction:

```
 mov r0, INA
```

This not only saves one instruction in the body of the function itself, it saves up to 3 instructions for each invocation of the function.

Reporting Bugs

Please report all Optimizer bugs to ross@thevastydeep.com.

Where possible, please include a *brief* example program that demonstrates the problem. Ideally, this will be in the form of a program that works correctly without optimization, but fails to compile or run correctly when optimization is used.