TN219

Root Memory Usage Reduction Tips

Customers with programs that are near the limits of root code and/or root data space usage will be interested in these tips for saving root space. The usage of root code and data by the BIOS in Dynamic C 7.20 increased from previous versions. A follow-on release will reduce BIOS root space usage, but probably not to the level of usage in previous versions.

Increasing Available Root Code Space

Increasing the available amount of root code space may be done in the following ways:

- **Use #memmap xmem**

  This will cause C functions that are not explicitly declared as "root" to be placed in xmem. Note that the only reason to locate a C function in root is because it modifies the XPC register (in embedded assembly code), or it is an ISR. The only performance difference in running code in xmem is in getting there and returning. It takes a total of 12 additional machine cycles because of the differences between `call`/`lcall`, and `ret`/`lret`.

- **Increase DATAORG**

  Root code space can be increased by increasing `DATAORG` in `RabbitBios.c` in increments of 0x1000. Unfortunately, this comes at the expense of root data space, but there are ways of reducing that too.

- **Reduce usage of root constants and string literals**

  Shortening literal strings and reusing them will save root space. The compiler, starting with version 7.20, automatically reuses identical string literals.

  These two statements :

```
printf ("This is a literal string");
sprintf (buf, "This is a literal string");
```

  will share the same literal string space whereas:

```
sprintf (buf, "this is a literal string");
```

  will use its own space since the string is different.

- **Use xdata to declare large tables of initialized data**

  If you have large tables of initialized data, consider using the keyword `xdata` to declare them. The disadvantage is that data cannot be accessed directly with pointers. The function `xmem2root()` allows xdata to be copied to a root buffer when needed.

```
// This uses root code space
const int root_table[8] =
{300,301,302,103,304,305,306,307};
// This does not
xdata xdata_table {300,301,302,103,304,305,306,307};
main(){
   // this only uses temporary stack space
   auto int table[8];
   xmem2root(table, xdata_table, 16);
       // now the xmem data can be accessed
       // via a 16 bit pointer into the table
}
```

  Both methods, `const` and `xdata`, create initialized data in flash at compile time, so the data cannot be rewritten directly.

- **Use xstring to declare a table of strings**

  The keyword `xstring` declares a table of strings in extended flash memory. The disadvantage is that the strings cannot be accessed directly with pointers, since the table entries are 20-bit physical addresses. As illustrated above, the function `xmem2root()` may be used to store the table in temporary stack space.

```
// This uses root code space
const char * name[] =
   {"string_1", . . . "string_n"};
// This does not
xstring name {"string_1", . . . "string_n"};
```

  Both methods, `const` and `xstring`, create initialized data in flash at compile time, so the data cannot be rewritten directly.

- **Turn off selected debugging features**

  Starting with Dynamic C 7.20, watch expressions, breakpoints, and single-stepping can be selectively disabled to save some root code space. From Dynamic C's main menu, select "Options" and then "Project Options." Look on the Debugger tab in the resulting Options dialog.

- **Place assembly language code into xmem**

  Pure assembly language code functions can go into xmem starting with Dynamic C 7.20:

```
#asm
foo_root::
   [some instructions]
   ret
#endasm
```

  The same function in xmem:

```
#asm xmem
foo_xmem::
   [some instructions]
   lret      ; use lret instead of ret
#endasm
```

  The correct calls are `call foo_root` and `lcall foo_xmem`. If the assembly function modifies the XPC register with

```
LD XPC, A
```

  it should not be placed in xmem. If it accesses data on the stack directly, the data will be one byte away from where it would be with a root function because `lcall` pushes the value of XPC onto the stack.

Increasing Available Root Data Space

Increasing the available amount of root data space may be done in the following ways:

- **Decrease DATAORG**

  Root data space can be increased by decreasing `DATAORG` in `RabbitBios.c` in increments of 0x1000. This comes at the expense of root code space.

- **Use #class auto**

  The default storage class of Dynamic C is static. This can be changed to auto using the directive `#class auto`. This will make local variables with no explicit storage class specified in functions default to auto. If you need the value in a local function to be retained between calls, it should be static. The default program stack size is 2048 (0x800) bytes if not using µC/OS-II. This could be increased to 0x1000 at most. It already is increased if the TCP/IP stack is used. The code to change it is in `program.lib`:

```
#ifndef MCOS
   #define DEFAULTSTACKSIZE 0x1000  ; increased from 0x800
#else
   #define DEFAULTSTACKSIZE 0x200
#endif
```

  Deeply nested calls with a lot of local auto arrays could exceed this limit, but 0x1000 should ordinarily be plenty of space. Using more temporary stack space for variables frees up static root data space for global and local static variables.

- **Use xmem for large RAM buffers**

  `xalloc()` can be used to allocate chunks of RAM in extended memory. The memory cannot be accessed by a 16 bit pointer, so using it can be more difficult. The functions `xmem2root()` and `root2xmem()` are available for moving from root to xmem and xmem to root. Large buffers used by Dynamic C libraries are already allocated from RAM in extended memory.