Technical Note

TN203

Porting a Program to Dynamic C

Introduction

Dynamic C has a number of improvements and differences compared to many other C compiler systems. This application note gives instructions and suggestions for porting a C program from other compilers to Dynamic C.

Generally, other C compilers require a large number of include files, potentially different for each source file, to provide function prototypes. Dynamic C function prototypes are automatically handled by the Dynamic C source libraries. The major porting task is reorganizing the source files and function prototypes into Dynamic C libraries.

There are some differences in the scoping rules between Dynamic C and various other compilers intended for embedded systems. These mostly result from the fact that Dynamic C compiles the entire program from source code and source libraries rather than compiling numerous modules independently and then linking them as a separate step.

Dynamic C follows the ISO/ANSI C standard when feasible and desirable. Because the standard does not take into account the special needs of embedded systems, it is necessary to depart from the standard in some areas and desirable in others. The standard does not take into account important embedded systems issues such as read only memory and embedded assembly language. For this reason, practical compilers intended for embedded systems do not completely comply with the standard, but use it as a guide.

As a rough estimate, it will require a day’s work to port 1,000 lines of source code. This does not include resolving differences in I/O hardware, I/O drivers and startup code.

This document does not discuss Dynamic C extensions to the C language, such as costatements, since they will not appear in code you port from another compiler.

Modular Programming with Dynamic C Libraries

Many programs consist of a number of files (`*.c` files in some compilers) that may be compiled separately. The programs may also have libraries of code blocks that are included optionally, depending on whether they are needed for a particular configuration. In addition, there are usually include (`*.h`) files that provide function prototypes, structure definitions and, often, various macros. The definitions in the include files effectively have global scope because they are included in every file where they are used. Dynamic C does not support file-scoped variables. In other compilers, file scoped global variables are declared with the keyword **`static.`** These global variables only become a problem in porting if there are symbol name conflicts between such variables declared in different file scopes. Simply rename the variables to resolve conflicts.

A C program has the following elements:

- Function prototypes, macro definitions, structure and typedef definitions
- Source code (C functions) and global data definitions
- Supplied libraries and user-defined libraries
- Assembly language (non-portable)

Dynamic C supports these same elements and, if you like, your program can be split between separate files. In other compilers, the separate modules have `.c` extensions (or whatever file type is used) and the include filenames have `.h` extensions. In Dynamic C, only the main program file ends with a `.c` extension (the file with the function "main"). If desired, this program file could contain your entire program except for the standard libraries and the BIOS. The alternative is to break your program up into separate modules called libraries or `.lib` files. These library files encourage modular design since Dynamic C compiles these library files together with your main program file.

Dynamic C improves upon the concept of header files by using special "header" sections. for two primary reasons. First, function prototypes and global data declarations are placed in the header section next to the implementation of the associated function for convenient reference. Second, this organization also allows the compiler to generate smaller code, since it conditionally compiles smaller sections of code depending on whether or not the symbols in that section are needed (e.g., referenced in the main program file).

Dynamic C library files serve the same purpose as object module libraries because sections of code will be compiled as a part of the user's program if they are needed to satisfy undefined global symbols. When a program is compiled under Dynamic C the compilation is as follows:

1. The BIOS, if it has not already been compiled, is compiled. Usually the BIOS is compiled and downloaded to the target at the start of a session; then you can recompile your program without having to recompile the BIOS. The BIOS is recompiled only if it crashes or compiler options change that require a recompile (e.g., changing memory devices). You can add modules from your program to the BIOS to avoid having to recompile them frequently.
2. All the global definitions contained in library "headers" are compiled. This normally includes function prototypes, macros and structure definitions. Global variables may be included in the library headers, but normally they are defined in the user's code between function definitions.
3. The libraries are scanned to locate modules that need to be compiled to satisfy undefined symbols. This continues until all library modules needed by your program have been compiled. Note that all compilation is continuous from start to finish and is done from C source code, then transformed to machine code resident in the target. There are no intermediate object modules. This method of doing the compile has profound advantages related to code size, debugging, and simplicity.

Here is an example of how you would port a simple module to Dynamic C.

The file **`libfoo.c`** contains function **`foo`**. Header file **`libfoo.h`** contains **`foo`**’s prototype, a macro, and a global variable.

`libfoo.c`

```
void foo(){ /* some code ... */ }
```

`libfoo.h`

```
#define FOOMAC 1
int globalFoo;
void foo();
```

Here is the Dynamic C version. The header and **`.c`** files are combined into one LIB file. A LIB file can contain multiple functions and headers. A header section can contain multiple function prototypes.

`libfoo.lib`

```
/*** Beginheader foo */

int globalFoo;
#define FOOMAC 1
void foo();

/*** endheader */

void foo(){  /* some code ... */ }
```

To use **`libfoo.lib`** in Dynamic C, an entry must be made in the file **`LIB.DIR`** in Dynamic C’s root directory:

```
LIB\LIBFOO.LIB
```

The source file that uses **`libfoo.lib`** needs the line:

```
#use libfoo.lib
```

added to the top of the file. Alternatively, this line could be added to the file **`default.h`**. The file **`default.h`** is used during both the BIOS and program compile to bring in the standard Dynamic C libraries.

Default Storage Class

The default storage class for local variables in most compilers is auto. In Dynamic C, the default storage class is static. Whether variables are auto or static usually determines if the function is reentrant or not. If you are converting a function to Dynamic C, just adding “auto” to the unspecified variable declaration will take care of everything except initialized variables.

Starting with Dynamic C 7.02, the **`#class`** directive will allow you to change the default storage class.

Initialized Variables

In Dynamic C, when a variable is initialized upon declaration, e.g. `int x = 0`, it is treated as a constant and is stored in flash. In order to facilitate porting, the Dynamic C compiler generates a warning that the variable is stored in flash. Using the `const` keyword for initialized variables eliminates the warning. In Dynamic C, the `const` keyword use is restricted to data with global or static scope. In other words, the `const` keyword is not applicable to auto variables, function parameters, or function return types. Refer to Chapter 13 of the *Dynamic C User's Manual* for more details on using the `const` keyword.

The correct way to initialize a static variable that may be changed by the program is to use a `#GLOBAL_INIT` section. A `#GLOBAL_INIT` section may appear in any function, and may be used to initialize local static variables as well as global static variables. All `#GLOBAL_INIT` sections are executed once, before `main()` is entered. The following code shows the right way and the wrong way to initialize a static variable:

```
const int x = 0;        // x will be put in code space in flash

int y;                  // y is a normal variable stored in RAM
int *xp;

main(){

   // This is the correct way to initialize a static variable. The #GLOBAL_INIT
   // section  should be placed after variable declarations and before the first
   // executable statement in any function it is used in.
   #GLOBAL_INIT {y=1;}

   // the compiler is smart enough to flag the following two statements as errors
   x = 1;
   *(&x) = 1;

   xp = &x;

   // this statement produces a run-time error if run-time pointer checking is enabled.
   *xp = 1;
}
```

Note that other C compilers will automatically initialize *all* static variables to zero that are not *explicitly* initialized before entering the main function. Dynamic C programs do not do this because in an embedded system you may wish to preserve the data in battery-backed RAM on reset.

Function Pointer Arguments

Dynamic C does not support argument lists in function pointer types in prototypes. This doesn't mean you can't use arguments in indirectly called functions, it just means the compiler will not check that the argument list is correct.

```
foo (int (*foobar) () );
```

is the correct syntax for a function pointer in a prototype whether or not foobar takes arguments.

Example:

```
// This program outputs "3".
int func(int x, int y);
int foo (int (*foobar) () );
main(){
   printf("%d\n",foo(func));
}


int foo (int (*foobar) (){
   return (*foobar) (1,2);
}


int func(int x, int y){
   return x+y;
}
```

Block Scope Variables

There are no block scope variables in Dynamic C. Variable declarations inside functions must appear before executable statements, e.g.

```
foo(){
   int x;
   while(1){
      int y;              // Incorrect!!!
   /* some code ... */
   }
}
```

#if defined(), #if !defined ()

These are not supported. Instead, use **`#ifdef`** and **`#ifndef`**, respectively.

sizeof operator

Prior to Dynamic C 7.20, the **`sizeof`** operator in Dynamic C did not match the ANSI C specification. Under ANSI, the expression used in the **`sizeof`** operator is evaluated by the compiler for its type, but no code is generated for run-time evaluation of the expression. For example, in the following program fragment (assuming **`i`** is 2 bytes), ANSI C would output "2 0" whereas "2 1" would be output by Dynamic C in versions prior to 7.20.

```
i = 0;
x = sizeof(i++);
printf ("%d %d", x, i);
```

Enum Types

Starting with Dynamic C 7.20 the **`enum`** keyword is supported.

Bit-field Types

Dynamic C does not currently support bit-field types. Code using bit-fields must be modified to use standard integral types and bitwise operations.

Switch Statements

Case constants in switch statements need to be two byte integral values.

String Constant Concatenation

Some C compilers concatenate quoted strings on separate lines:

```
char string[] = G
   "12345678901234567890"
   "12345678901234567890"
   ;
```

Dynamic C requires the use of a backslash as a continuation character to concatenate the next line:

```
char string[] =
   "12345678901234567890" \
   "12345678901234567890"
   ;
```

Z-World, Inc.

2900 Spafford Street
Davis, California 95616-6800
USA

Telephone: (530) 757-3737
Fax: (530) 757-3792

www.zworld.com

Rabbit Semiconductor

2932 Spafford Street
Davis, California 95616-6800
USA

Telephone: (530) 757-8400
Fax: (530) 757-8402

www.rabbitsemiconductor.com