Excellent Integrated System Limited

Stocking Distributor

Click to view price, real time Inventory, Delivery & Lifecycle Information:

Digi International
20-101-0517

For any questions, you can email us directly:
sales@integrated-circuit.com

TECHNICAL NOTE

TN238

Rabbit Memory Usage Tips

This purpose of this technical note is to give the Rabbit Dynamic C user tips on how to use memory efficiently to better fit large programs into small memories. If your code and data are pushing the size limits of your target unit and still growing, this document should help. The code in here was primarily tested with Dynamic C version 8.30. Differences in older versions of software and hardware are briefly discussed at the end of this document. The Rabbit memory model is somewhat complex and worthy of longer discussions, which it gets in the chip user's manuals and other manuals. A complete understanding of the memory model can be helpful for advanced users who are trying to do advanced Rabbit tricks, such as make their own board with a non-standard memory configuration or their own programming tools, but the focus of this document is nuts & bolts tips to help squeeze your large program and data into a relatively small memory.

Summary of Tips for Maximizing Storage Space

Here, without explanations, are the tips for reducing memory usage. Click on the tips to go to the detailed explanations.

1. Use separate instruction & data space.
2. Use string literals sparingly.
3. Force functions to xmem or root only when necessary.
4. Make library functions nodebug.
5. Use auto variables for functions, but be careful about using auto arrays.
6. Use file compression to import large web pages.
7. Don't compile unneeded code in printf code.
8. Decrease or increase the DATAORG macro.
9. Use a second flash for code.
10. Enable inlining of internal I/O.
11. Choose "Optimize for Size" in the Compiler Options.
12. Reduce Debugger Functionality.
13. Use xalloc to allocate RAM in xmem for large buffers.
14. Use xmem constants for large constant tables.
15. Code efficiently.

Diagnosing Space Problems

Common Error Messages Related to Running Out of Space

The numbers in the **Possible fixes** line reference the tip list on page 1.

- **line 12: ERROR PROGRAM_NAME.C: Out of variable data space.**
  Cause: This is caused by too many static variables, structures and arrays:

  Possible fixes: 1, 3&8, 5, 12, 13

- **line 45: ERROR PROGRAM_NAME.C: Out of xmem code space, use a larger ROM/RAM.**
  Cause: This is caused by the compiler trying to put more functions in xmem than will fit.

  Possible fixes: 3, 4, 6, 7, 8, 9, 10, 11, 12, 15

- **line 29: ERROR PROGRAM_NAME.C: Out of root code space, try moving code to XMEM.**
  Cause: This is caused by the compiler trying to put more functions in root than will fit.

  Possible fixes: 1, 2, 3, 4, 8, 10, 11, 12, 14, 15

- **line 1: ERROR DKENTRY.LIB : Out of constant data space. (Separate I&D only)**
  Cause: This is caused by to the compiler trying to put more string literals and constants in root constant space than will fit.

  Possible fixes: 2, 8, 12, 14

Insufficient Stack Space Symptoms

Running out of stack space will cause a program crash that may or may not be accompanied by a run-time error. An “Unexpected Interrupt” run-time error can happen when stack corruption lands program execution on an 0xFF byte, which is both the op code for an unused RST vector and the contents of an erased flash byte. A “Target communication error” may occur instead - but stack corruption is not the only possible cause of that message, loss of target communication will generally occur when any program crash happens that kills the debug kernel and is not a run-time error that gets trapped.

Every time a function is called, the return address is pushed onto the stack along with the function arguments. It is better to pass pointers to large structures than the structures themselves to save stack space. The local variables, structures and arrays of the function called also take up stack space unless they are declared as “static” or the `#class static` directive has been used to change the default storage class from root to static.

Fixes

The sample program `STACKCHECK.C` shows how to do a run-time check (while debugging) for the stack getting low. Examine the section the MAP file generated by your program headed by

```
// Parameter and local auto symbol mapping and source reference.
```

for excessive use of auto arrays or large parameters being passed on the stack. The MAP file also shows a call graph of your program so you can examine the calling hierarchy.

Using the MAP and LST Files

A MAP file is created when any program is compiled. It has the same name as the program with the extension MAP, and is placed in the same directory as the program. The BIOS has a separate MAP file. The MAP file gives size and location of functions and data, and total data and code sizes. It is a good place to spot problems such as large auto arrays, root constant tables, or excessively large root functions. Unnamed string literals do not show up in the MAP file. Root constants will show up in the data section of the file with a segmented address like this:

```
10:0255
```

The 10 segment indicates that the 16th bit is inverted, no segment register is actually used by the MMU when accessing these root addresses.

Pure assembly functions do not show the code size, but the size can be deduced by subtracting the function address from the address of the next function on the list. For example, here the size of `foo1` is 1d78-1d6e, or 10 decimal bytes:

```
1d6e        *   foo1                    \foofile.c
1d78        *   foo2                    \foofile.c
```

`.LST` files contain complete information about generated code. They show C source and the equivalent assembly code generated. Unnamed constants (for example, string literals) do show up in list files like so:

```
*** Constant Data ***
20d5  25 63 2C 20 64 72 6F 70 70 65 64 3D 25 64 2C 25 %c,
dropped=%d,%
20d6  64 0A 00 25 73 0A 00 48 65 6C 6C 6F 20 77 6F 72 d  %s  Hello
wor
20d6  6C 64 0A 00 00                                   ld
*** End Constant Data ***
```

List files are not generated by default because they tend to be large. The option can be turned on in the Compiler Options dialog box.

Using the Information Window

This window can be viewed by choosing “information” on the Window menu. It shows overall (BIOS and user program) memory usage for different segments.

For both the map files and the information window, a failed compile will not necessarily have accurate information. It is advisable to know how big your program is and how much room to spare there is before dropping in a large chunk of code.

Memory Usage Reduction Tip Details

You may want to read the Rabbit Memory Model and Glossary sections to better understand some of these tips.

1. Use separate instruction & data space.

The default compiler setting is to disable separate I&D space. To enable it, choose Project Options | Compiler from the Options menu and check the appropriate box before compiling your application. Enabling separate I&D space makes a full 0xD000 (53248) bytes available for root code, and the same amount of total space available for root data and root constants. Some of this available space is used by the BIOS. Without separate I&D enabled, the total space available for all root constants, root data and root code is 0xD000 bytes. Separate I&D space nearly doubles root memory.

Why would you *not* want to use separate I&D space then?

You may have code that has to run on the original R2000A chip, which does not support separate I&D. You may have already written code for later Rabbit chips that works fine but without separate I&D space, and now you want to expand your code and are afraid it will not work with separate I&D. Most code will work without any changes when you enable this feature. The exceptions are:

- Code that modifies ISR vectors

  ISR vectors work differently in the separate I&D space model because the vector table must be located in flash. To make ISR vector table entries changeable at run-time requires an extra relay vector, so it is preferable to set them at compile time as follows:

  ```
  #if __SEPARATE_INST_DATA__
     interrupt_vector timerb_intvec timerb_isr;
  #else
     SetVectIntern(0x0B, timerb_isr);          //init Timer B interrupt vector
  #endif
  ```

  See the assembly language section in the *Dynamic C User's Manual* for details about setting interrupt vectors in assembly code.

- Code that defines constants in assembly

  Constant tables in assembly language such as:

  ```
  #asm const
  myrootconstants::
  db 0x40, 0x41, 0x42
  #endasm
  ```

  require the `const` keyword as shown above in order to be referenced as data. The keyword will have no ill effect if separate I&D is not used.

2. Use string literals sparingly.

These do not show up in the MAP file, but can take a lot of space. Reuse the same strings when possible, because the compiler will only create one instance of literal strings if they are identical. If you are accustomed to debugging using a lot of `printf()` statements, you may find yourself using up too much root space with debug messages.

3. Force functions to xmem or root only when necessary.

Forcing code to root that does not need to be there takes away space for code that does. Use of

```
#memmap xmem
```

to force all functions not specified as root to xmem may cause xmem code space to run out and the compile to fail while there is plenty of root code space to spare. Putting C code in xmem is easy because the compiler does all the work. The compiler generates "bouncers" in root code for functions that are called via function pointer so that 16 bit function pointers will work.

Forcing Code to xmem

Forcing code to xmem may be necessary when more root code space is needed. Code is forced to xmem by putting the keyword `xmem` in front of a function definition and its prototype. The directive `#memmap xmem` can be used to put functions in xmem by default if `xmem` or `root` is not specified in the function definition and prototype. Pure assembly language functions will never be placed in xmem unless `xmem` is placed after the `#asm` directive:

```
#asm xmem
foo::
...
#endasm
```

Assembly sections within C functions will be compiled to xmem or root as the encapsulating function is.

Forcing Code to root

Forcing code to root is necessary if the code is going to change the XPC segment register, or if more xmem space is needed. Code is forced to root by putting the keyword `root` in front of a function definition and its prototype. The directive `#memmap root` can be used to put functions in root by default if `xmem` or `root` is not specified in the function definition and prototype, but this directive should generally not be used.

4. Make library functions nodebug.

The default for functions is `debug`. After functions are debugged, put the keyword `nodebug` in front of the function definition (it is not needed in the prototype). This saves two bytes for each C statement in the function, because the compiler inserts an RST 28h and a NOP instruction between each C statement in debug functions as a hook into the debugger.

5. Use auto variables

The default storage class is auto. Using autos instead of statics leaves more space available for static data in the data segment. Without separate I&D enabled, this can also free up more root code by allowing `DATAORG` to be increased by one or more 0x1000 byte increments. With separate I&D enabled, this gives more space for root constants.

Functions using only autos are *generally* reentrant (but shared I/O resources also effect reentrancy).

Large auto arrays or large structures passed by value can overflow the stack. It is more efficient in terms of both stack space and speed to pass pointers to large structures as function arguments than to pass the structures themselves. The sample program `STACKCHECK.C` shows how to use an assert macro to check the stack space while debugging.

6. Use file compression to import large web pages.

If your application serves some large web pages that are #ximported into the program, try using `#zimport` instead. This imports the files compressed. The HTTP server has decompression integrated into it already. #zimport'ing small files is not efficient, and may actually increase file size. The run-time decompression code will increase code size. For more information on using file compression with Dynamic C, see Technical Note TN234, "File Compression (Using #zimport)."

7. Don't compile unneeded code in printf code.

If an application doesn't use the float conversion specifiers in `printf()` calls, add

```
#define STDIO_DISABLE_FLOATS
```

to the application. This can save as much as 1.5K bytes of root code space and 3.7K bytes of xmem code space.

Later versions of Dynamic C will compile-switch-out the unneeded code in the function `doprnt()` if this macro is defined. For earlier versions, make the following code change in `doprnt()` in the library `\LIB\STDIO.LIB`:

```
goto __DC_loop;
#ifndef STDIO_DISABLE_FLOATS          // ADD THIS LINE
case 'f' :
case 'g' :
     .
     .
     .
     goto __DC_loop;
#endif                                // AND THIS LINE
case 'X' :
case 'x' :
```

The new `STDIO.LIB` also has a new sprintf/printf format specifier, `%ls`, that allows use of physical addresses for xstrings without copying to root first. It is used in one of the included sample programs. This new feature will be a part of future Dynamic C releases.

8. Decrease or increase the DATAORG macro.

`DATAORG` is a macro that is `#define`'d near the top of the BIOS. It can be adjusted up or down in 0x1000 byte increments. With separate I&D, adjusting it up gives more space for root constants and less for root variables. Without separate I&D, adjusting it up gives more space for root code and constants and less space for root variables.

9. Use a second flash for code.

The macro `USE_2NDFLASH_CODE` is #define'd near the top of the BIOS, but commented out by default. If you have a board that has two flash chips and your code does not fit in the first flash, uncomment out the definition. Otherwise the compiler will not know about the extra flash. The second flash may not be used for a file system if it is used for code.

10. Enable inlining of internal I/O.

This is the default setting for this compiler option. This means that the code to execute the I/O operation is compiled without a function call. Use constant parameters for all except the `data` parameter for write functions to ensure in-lining. These three statements use 16, 16 and 9 bytes respectively without in-line I/O enabled, and 14, 7 and 6 bytes with in-lining.

```
WrPortI(PADR,&PADRShadow,data);    //&PADRShadow is constant (to the compiler)
WrPortI(PADR,NULL,data);           //PADR is W/R so doesn't need a shadow register
RdPortI(PADR);
```

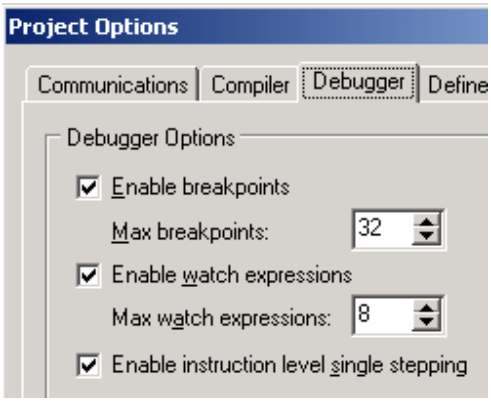
In addition, if the library functions are never compiled, their code sizes will be saved, and in-line code is *much* faster.

11. Choose "Optimize for Size" in the Compiler Options.

This especially saves code space in programs that do a lot of long integer arithmetic by calling rather than in-lining arithmetic utility functions.

12. Reduce Debugger Functionality.

The project options on the Options menu allow reduction of some debugging functionality to save code and data space.



This can save as much as 5.7 Kilobytes in root code, 1.8K in xmem code, 0.59K root data, and 0.45K in root constants.

13. Use xalloc to allocate RAM in xmem for large buffers

Dyanmic C libraries generally already do this. The library `RS232.LIB` does not, but a version of it that does accompanies this technical note. See the accompanying sample programs.

14. Use xmem constants for large constant tables.

Use `xdata` and `xstring` instead of root constants. This is especially applicable to programs that have text menuing systems. See the sample programs accompanying this note.

15. Code efficiently.

There are a number of ways to increase code efficiency.

Compute complex pointer expressions once.

For example, this:

```
Ptr1->Ptr2->item1 = x;
Ptr1->Ptr2->item2 = y;
Ptr1->Ptr2->item3 = z;
Ptr1->Ptr2->item4 = u;
```

is more efficiently coded as:

```
Ptr3 = Ptr1->Ptr2;
Ptr3->item1 = x;
Ptr3->item2 = y;
Ptr3->item3 = z;
Ptr3->item4 = u;
```

Apply the same idea to any complex expression. You can examine generated code in the assembly window to see whether it is worth adding a new variable and line code to reduce overall code size.

Use ints as opposed to longs.

When possible, ints are much more efficient. Do not use floats unless you have to.

Reuse code via functions when possible.

It only takes a few cycles and code bytes to call a function! The new `RS232.LIB` included with this note is an example of how much space can be saved by abstracting similar functions to combine them.

Learn assembly language.

While it is usually not practical to code everything in assembly, if you have some code that needs to be fast, there's nothing like assembly language for speed, and you get the bonus of smaller code.

Learn correct use of modules.

If you create your own libraries, learn how to use headers correctly. Incorrect usage of headers could result in code being compiled that is not actually called by the program, and data space being allocated that is not used. The Modules section of the *Dynamic User's Manual* explains the use of modules.

Sample Programs and Libraries

All of the code provided with this technical note is provided as samples only, and not as part of normal Dynamic C releases.

The first three groups of files described here are in the accompanying ZIP file. All should extracted to a folder named `TNXMEM` under the root directory of your Dynamic C installation.

If you are using Dynamic C 8.30, you can set the User defined LIB directory file in the advanced compiler options to the `LIB.DIR` in that folder. For other versions, just add these lines to the end of the `LIB.DIR` file you are using:

```
TNXMEM\CBUF.LIB
TNXMEM\XMALLOC.LIB
TNXMEM\MALLOC.LIB
```

Modified Library Files

These files are modified versions of like-named files from Dynamic C version 8.30. If these libraries are not compatible with your version Dynamic C. simply make the changes specified.

Table 1. Modified Dynamic C Library Files

Library File	Description
`STDIO.LIB`	Has a small modification for tip 7. It also demonstrates a new sprintf/printf format specifier, %ls, that allows use of physical addresses for strings. See detailed discussion of tip 7.
`XMEM.LIB`	Has new functions `xgetfloat()` and `xsetfloat()` (`xgetfloat` is needed by the sample program `SIN_TABLE.C`)
`LIB.DIR`	References for new libraries and the above libraries.
`RS232.LIB`	This is a version of the standard Dynamic C library that saves lots of code space by abstracting the similar API functions and ISRs of the different ports so there is one version of each instead of six. It also permits the use of xmem buffers for the port read and write buffers and so can save considerable amount of root data space. The API is backwards compatible, but new functions, `serXopen2()`, must be used to initialize the ports to use xmem buffers. This library requires `CBUF.LIB` to work.
`CBUF.LIB`	Circular buffer routines for buffers in xmem. The API is backwards compatible with the older version, but the functions are forced to root so they can manipulate the XPC. Calls to the functions from C code will work for old code, but calls from assembly code should be changed to call from `lcall`. The only currently released Dynamic C library that calls functions in the library from assembly is `RS232.LIB`, so these two new versions are a pair.

New Library Files

New libraries need to be added to `LIB.DIR` file or the user specified LIB directory file before they can be #used in a program.

Table 2. New Dynamic C Library Files

Library File	Description
`FSER.LIB`	A very small footprint, very fast serial driver. The API is not backwards compatible with `RS232.LIB`. It doesn't have support for modes other than N81, or flow control, but it does support SPI.
`XMALLOC.LIB`	A system that allows dynamic allocation and freeing of xmem RAM using physical addresses.
`MALLOC.LIB`	A system for dynamic allocation and freeing of root or xmem RAM using 16 bit pointers.

New Sample Programs

The sample programs in this table are included in the ZIP file accompanying this technical note.

Table 3. Sample Programs that Use Memory Reduction Techniques

Sample Program	Description
`XSTRING2.C`	A method of displaying a table of string literals stored as xstrings.
`SIN_TABLE.C`	Demonstrates using a large look-up table in xmem for a math calculation.
`SINDATA.BIN`	Data table file imported by `SIN_TABLE.C` program.
`CXBUF.C`	Using a circular xmem RAM buffer.
`MALLOC.C`	A basic demonstration of `malloc()` and `free()`.
`MALLOCTEST.C`	Allocating and freeing xmem and root RAM buffers via 16 bit pointer. Includes a visual representation of the heap that can show problems that can occur with dynamic memory allocation.
`STACKCHECK.C`	A method of checking the stack
`XMALLOC.C`	A basic demonstration of `xmalloc()` and `xfree()`.
`XMALLOCTEST.C`	Allocating and freeing xmem RAM buffers via physical addresses.
`SERXMEM.C`	Sample program demonstrating the use of the included `RS232.LIB` with xmem serial buffers.

Other Sample Programs

These sample programs are part of normal Dynamic C releases and illustrate some of the concepts discussed in this technical note. They are not included in the accompanying ZIP file.

`\SAMPLES\XMEM\XDATA.C` - How to put various types of numeric and character data into xdata constants.

`\SAMPLES\XMEM\XSTRING.C` - Simple way to access uniform sized xstrings.

`\SAMPLES\XMEM\XMEMDEMO.C` - Demonstration of `xmem2root()` and `root2xmem()`.

`\SAMPLES\ZIMPORT\ZIMPORT.C` - Demonstration of the zimport compiler directive.

`\SAMPLES\TCPIP\HTTP\ZIMPORT\ZIMPORT.C` -This program uses the `ZIMPORT.LIB` library to compress web pages that are served by the HTTP server.

Glossary/Definitions

BIOS - The program that contains start-up code, PC-target communications protocols and debug functionality. The standard BIOS source code is in `/BIOS/RABBITBIOS.C`. This program is compiled and loaded before the user program.

Data segment - The logical address space in RAM used for global and static variables. The top of the data segment is always 0xCFFF in Dynamic C programs. The bottom is defined by the BIOS macro `DATAORG`. The physical address corresponding to the logical address is computed from the value in the DATASEG I/O register.

Extended memory (abbreviated xmem) - logical addresses in the range 0xE000 - 0xFFFF. The corresponding physical address is computed from the value in the XPC register. Dynamic C makes use of extended flash memory for code, but does not allocate RAM for static variables or auto variables in xmem. Any memory in flash or RAM in the 1 MB physical address space may accessed with xmem. (Writing to flash requires special function calls of course.)

Logical address - The 16 bit address used by the large majority of Rabbit instructions to access code and data in a 64K space.

Physical address - The 20 bit address in the 0000-0xFFFFF range that the 16 bit logical address maps into.

Root constants - String literals and initialized variables. In Dynamic C, these normally go into flash.

Root memory - This general term refers to the root code/constant, stack, and data segments.

Root segment - This more specific term refers to the logical addresses below the data segment where the logical and physical addresses are the same.

Separate Instruction & Data space - A memory mapping that nearly doubles the root memory available because code and data can share the same logical addresses and but map to different actual physical addresses. See Appendix A: Rabbit Memory Model.

Stack - Memory used to store passed parameters, local auto data, function call return addresses and data “pushed” onto it with the assembly language instruction push. Normally the logical addresses between 0xD000 and 0xDFFF in Dynamic C programs.

Stack segment - The Rabbit CPU permits a bigger segment, but Dynamic C programs only map 0xD000-0xDFFF as the stack segment.

Segment registers - Three I/O registers, SEGSIZE, DATASEG and STACKSEG, and one special processor register, XPC, are used to map logical addresses to physical addresses. Any of the segment registers can be loaded with a new valid value at any time, but this has to be done with great care. For example, if code is executing in the xmem segment and the XPC is changed, then execution will not continue at the next instruction, but instead will continue at the location in physical memory where the logical address of the next instruction maps to. This is because the PC (program counter) register holds a logical address.

Static data/variables - Global variables or variables declared inside functions that are declared "static." These variables reside in the data segment and exist for the life of the program. If the default storage class is changed to static with the directive `#class static`, local variables (those declared inside functions) not specifically declared as auto or static will be static. Global variables (those declared outside functions) can only be static, but the static keyword is not needed for globals.

String literal - An unnamed quoted string, for example: `printf("string literal");`

Auto variables - Variables declared inside functions that are not declared "static," (unless the directive `#class static` has declared the default storage class to be static) or are explicitly declared "auto." These variables reside in the stack function only while the function is being executed.

xmem - short for extended memory, also a keyword to force code to be compiled to xmem.

Appendix A: Rabbit Memory Model

For a description of the Rabbit memory model, see technical note, TN202, “Rabbit Memory Management in a Nutshell.”

Computing Physical Addresses

Abbreviations - LA - Logical Address, **PA -** Physical Address

Let the SEGSIZE register = XYh where X is the high nibble and Y is the low nibble. The following algorithm determines the physical address that a logical address maps to.

```
If LA >= E000h
   PA = LA + (XPC * 1000h)

Else If LA >= X000h
   PA = LA + (STACKSEG * 1000h)

Else If LA >= Y000h
   PA = LA + (DATASEG * 1000h)

Else PA = LA
```

Example: (typical non separate I&D space mapping)

if,

SEGSIZE register = 0xD6
STACKSEG register = 0x88
DATASEG register = 0x91
XPC register = 0x04

then,

Logical address 0x1000 = physical address 0x01000
Logical address 0xE800 = physical address 0x12800
Logical address 0x6080 = physical address 0x97080
Logical address 0xD400 = physical address 0x95400

How physical addresses map to memory devices is determined with MIU registers (MB0CR, MB1CR, MB2CR and MB3CR) and the MMIDR register. See the chip user’s manual for details.

Extended vs. Root Memory

Code *runs* exactly as efficiently in extended memory (assuming that the xmem code is mapped to memory with same speed as root code) as it does in root. *Calling* code in xmem and returning from it requires only an extra 12 CPU cycles. The main reasons for running code in root are that the code is an ISR, which must be in root, or the code manipulates data in xmem and therefore modifies the XPC segment register. Changing the XPC is not permitted when the code that changes it is running in xmem. The main reason for running code in xmem is that most programs will not fit entirely into root. The Dynamic C compiler handles the details for putting C code in xmem and calling it. A little bit of extra care must be taken to correctly call xmem code and return in assembly code, as well as to access stack data, since an xmem call pushes an extra byte on the stack.

Assembly Language Caveats

The assembler will promote a `call` to `lcall` but it will not demote `lcall` to `call`:

```
root int foo1();
xmem int foo2();

main(){
   asm call foo2                    //  the actual instruction generated is lcall
   asm lcall foo1                   // the instruction generated is also lcall
}

root int foo1(){}
xmem int foo2(){}
```

This code will compile to root and work:

```
#asm
foo::
   ld hl,(sp + 2)                   // load left argument of foo call to hl
   ld ix,(sp + 4)                   // load right argument of foo call to ix
ret
#endasm

main(){   foo(1,2);   }
```

This code will compile to xmem and work:

```
#asm xmem
foo::
   ld hl,(sp + 3)                   // load left argument of foo call to hl
   ld ix,(sp + 5)                   // load right argument of foo call to ix
   lret                             // long return
#endasm

main(){   foo(1,2);   }
```

This code will work whether `foo` is compiled to root or xmem, the C compiler handles the details of the return, and the assembler adjusts the constant @SP according to whether `foo()` is in root or xmem:

```
foo(int x, int y)                    // can be preceded with "xmem" or "root"
{
#asm
   ld hl,(sp + @SP + x)              //  load left argument
   ld ix,(sp + @SP + y)              //  load right argument
#endasm
}
main(){ foo(1,2); }
```

Xmem Data

Dynamic C uses 16-bit addresses only for pointers; there are no far pointers at the time of this writing. Data handling is, therefore, more complex. The source code provided with this technical note has examples that make use of xmem data including a sample malloc/free implementation and a sample circular buffer API. Malloc'ed xmem data can only be manipulated via 16 bit pointers from root code, or with LDP instructions and the functions that use them (See API functions for Xmem below).

Storage Class - auto vs. static

More use of auto variables and arrays means more root data space for static variables and arrays, but overuse of auto data, especially large auto arrays, can lead to stack overflow. No stack checking is performed to detect this - the result of stack overflow is a program crash.

Multiple stacks are used with uC/OS-II, each task has its own stack. This makes more than 4K of total stack space available.

Separate I&D Space

Separate Instruction & Data space is a memory mapping achieved by setting a Memory Mapping Unit (MMU) register bit that causes inversion of address lines when the CPU accesses data and/or root segment memory. What this means is that for the normal Rabbit/Dynamic C separate I&D space memory mapping, an instruction like this:

```
LD HL,(0x0250)
```

can be located in root code space, and the Rabbit will fetch the instruction normally, but when the memory at address 0x0250 is read to load to the HL register, the Rabbit inverts address line 16, which means it will really read two bytes from memory at physical 0x10250. This is a physical address in flash separate from root code. This is where the compiler puts root constants in this mode.

And an instruction like this:

```
LD HL,(0xB000)
```

inverts address line 19 when the address at 0xB000 is read, which means it really reads the two bytes at physical address 0x8B000 (the DATASEG register value is typically 0 in this memory mapping)

This doubles the root memory available because code and data can share the same logical addresses and but map to different actual physical addresses.

Appendix B: Xmem Keywords and API Functions

This section describes keywords that place code or data in xmem and functions that can be used to access this extended memory.

Keywords

xmem, root

These keywords are for use with functions only. They force functions to be compiled to root or xmem. The keyword should be placed before both the prototype and the body of the function in C:

```
xmem int foo();      // prototype
xmem int foo()       // body
{
   // some code
}
```

Or after the `#asm` directive for pure assembly functions:

```
#asm xmem
   foo::
   // some code
#endasm
```

For assembly blocks encapsulated in C, the encapsulating function will override the keyword used on the assembly block

```
xmem foo(){
   #asm root
   // some code that will not be in root!
   #endasm
}
```

xstring, xdata

See the keyword descriptions in the *Dynamic C User's Manual* and the samples programs in `\SAMPLES\XMEM`. These are for defining xmem constants that normally go into flash.

API Functions for xmem

See the Dynamic C on-line function lookup for full details.

paddr

`unsigned long paddr(void* pointer)`

Convert a 16-bit logical pointer into its physical address. Use caution when converting address in the E000-FFFF range. This function will return the address based on the XPC on entry.

paddrDS

`unsigned long paddrDS(void* pointer)`

Convert a logical pointer into its physical address. This function assumes the pointer points to data in the data segment, which eliminates some runtime testing compared with the more general function, `paddr()`. The data segment is used to store statically-allocated data items. This function will generate incorrect results if used for addresses in the root code (i.e., program code or constants), stack (i.e., auto variables), or xmem segments.

paddrSS

`unsigned long paddrSS(void* pointer)`

Convert a logical pointer into its physical address. This function assumes the pointer points to data in the stack segment, which eliminates some runtime testing compared with the more general function, `paddr()`. The stack segment is used to store "auto" data items. This function will generate incorrect results if used for addresses in the root code (i.e., program code or constants), data (i.e., statically allocated variables), or xmem segments.

root2xmem

`int root2xmem(unsigned long dest, void *src, unsigned len)`

Stores `len` characters from logical address `src` to physical address `dest`.

xgetint

`int xgetint(long src)`

Return the integer pointed to by `src`. This is the most efficient function for obtaining 2 bytes from xmem.

xgetfloat

`float xgetfloat(long src)`

Return the float pointed to by `src`.

xgetlong

`long xgetlong(long src)`

Return the long word pointed to by `src`.

xmem2root

`int xmem2root(void *dest, unsigned long src, unsigned len)`

Stores `len` characters from physical address `src` to logical address `dest`.

xmem2xmem

`int xmem2xmem(unsigned long dest, unsigned long src, unsigned len)`

Stores `len` characters from physical address `src` to physical address `dest`.

xmemchr

`long xmemchr(long src, char ch, unsigned short n)`

Search for the first occurrence of character `ch` in the xmem area pointed to by `src`.

xsetint

`void xsetint(long dst, int val)`

Set the integer pointed to by `dst`. This is the most efficient function for writing 2 bytes to xmem.

xsetfloat

`void xsetfloat(long dst, float val)`

Set the float pointed to by dst. This is the most efficient function for writing 4 bytes to xmem.

xsetlong

`void xsetlong(long dst, long val)`

Set the long integer pointed to by `dst`. This is the most efficient function for writing 4 bytes to xmem.

xstrlen

`unsigned int xstrlen(long src)`

Return the length of the string in xmem pointed to by `src`. If there is no null terminator within the first 65536 bytes of the string, then the return value will be meaningless.

_LIN2SEG

`_LIN2SEG (Assembly language macro)`

A macro to be used in assembler code. On entry, A:HL contains linear (20-bit only) address. Converts this to segmented form with XPC value in A, and logical part in HL (in range 0xE000-0xEFFF). Uses no other registers except trashes flags.

Appendix C. Version Differences in Dynamic C and Rabbit Processors

Separate I&D Space Support

Separate I&D space is supported on all Rabbit processor versions except the original, which is marked **IQ2T**. Dynamic C will generate a compiler error if an attempt is made to compile to a board with this chip with separate I&D space enabled.

Dynamic C support for separate I&D space was introduced in version 7.30. For more details on this feature, see the Designer's Handbook; e.g., the *Rabbit 3000 Designer's Handbook*.

Default Storage Class

Prior to Dynamic C 8.01, the default storage class for local variables was `static`. With Dynamic C 8.01, it was changed to `auto`. The `#class auto` directive could not be used to change the storage class until DC version 7.02P3. It can be changed back to static with `#class static`.

In-lining of Internal I/O

This size and speed optimization was introduced in DC 8.02.

File Compression Support

`#zimport` was introduced in DC 8.01.