ROMing Software Components: An RTS Use Case

Alan Campbell *Software Development Org Applications*

ABSTRACT

Placing code and constant data in ROM can lead to a big benefit in system cost. Since ROM area is typically one quarter the size of equivalent RAM, the DSP silicon cost is lower. Performance is also improved, since accessing ROM is faster than accessing external memory.

There are, however, software considerations when ROMing components. For example, 'C6000 direct accesses to .bss global variables in a ROMed function can severely inhibit user flexibility in data placement. The C initialization records are another concern, since they are often constructed as one large section, which can impact the boot-time of devices. (That is, consumers don't want to wait 5 seconds for their device to come out of sleep mode!)

This document discusses such considerations. It focuses on automating the ROMing process to eliminate the possibility of errors. Patching, the process of replacing a faulty or poorly performing function, is also discussed.

The use case is the TI Run-Time Support Library. However, the scripts and guidelines are applicable to almost any software component.

This document references sample code and batch files to run script commands. These can be downloaded using this link: http://www-s.ti.com/sc/techlit/spraan5.zip.

Note that this document does not provide coding guidelines for ROM such as function pointer indirections. The References section lists several excellent application notes and presentations that address this well.

Contents

1 Introduction

Several TI devices have ROM. One example is the TMS320C6727 floating-point device with 384 KB of ROM. The gains in system cost, performance boost versus external memory, and the reduction in boot-time all make ROM an attractive option.

This application note outlines the preferred process for ROM creation: several techniques have been used in the past but some, such as partial linking, have little-known side-effects. Once you have a ROM executable, you need to ensure that the application uses the ROMed functions at the precise addresses where they were placed. This process needs to be automatic—the user should not need to concern themselves with whether the ROMed function is truly being used or if instead a RAM flavor is mistakenly being linked in.

The user *does* need to get involved if patches are required. That is, if a function in ROM has a bug or there's a new optimized version, the process also needs to be painless. It should be obvious how to reference the new, patched RAM version, and the tools (linker, etc.) should help.

Automation is essential throughout this process. Scripts are used to determine which functions are good ROM candidates from a library. If you already know what you wish to ROM, then you can still leverage scripts at the application ROM usage stage to make absolutely certain that you reference the ROM functions from the exact addresses at which they were masked.

The use-case example for this work is the TI 'C6000 Run-Time Support Library. This is a common candidate for ROMing since it is stable, functions are small, and the call-tree is minimal (most functions are "leafs").

This document references sample code and batch files to run script commands. These can be downloaded using this link: http://www-s.ti.com/sc/techlit/spraan5.zip.

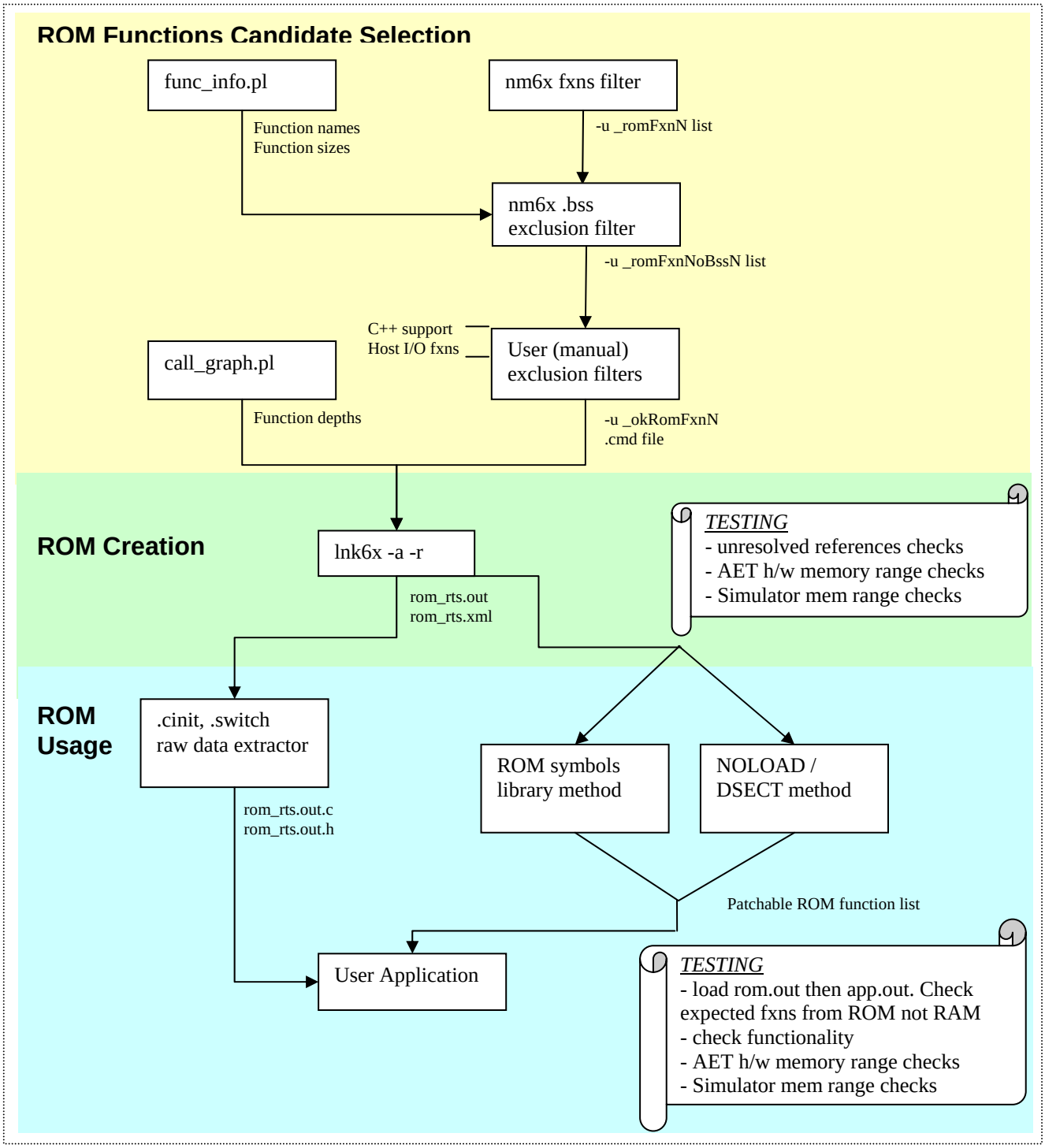
Pre-requisites to run the attached scripts and examples include:

- ActivePerl >= version 5.8.3. Alternatively, if you build on cygwin or Linux you can download dependent packages via the method outlined in this FAQ.
- Code Composer Studio >= version 3.3.

2 ROM Design Flow

Figure 1 shows the approach this document takes:

Figure 1. Flow of ROM Creation and Usage



There are three basic phases shown in Figure 1: ROM function candidate selection (Section 3), ROM executable creation (Section 4), and ROM usage by the application (Section 5). Testing occurs throughout the latter two stages. Scripting is employed heavily throughout.

During ROM candidate selection, we use scripts to narrow the ROM function list to those most suitable. Criteria include:

- No .bss references (at least for ‘C6000 and TMS470)
- Function call depth, size, and stack usage not too large
- Host I/O functions and C++ language support (new, delete, etc.)
- Additional user criteria

The result of this first phase is a linker command file (rom_chosen_fxns.cmd) with –u _romFuncN references that cause the linker to bring in the function and all its sub-references (children).

Next we feed the cmd file into the ROM creation stage and create an executable, relocatable output module. We can hand the .out file to the silicon team for ROM masking directly (after testing!). We use the exact same bits for testing in the simulator as well (independent pre-load of rts_rom.out).

Finally we have an application that uses the ROM. Patching on a function-by-function basis is enabled via one of two scripted techniques available to us. Nuances such as how to bring forward the raw data from initialized sections like .switch and .cinit are also covered.

3 Automatically Determining What Functions to Put in ROM

Note: If you already know what library functions you want to place in ROM and are aware of ROM limitations imposed by .cinit, .bss, etc., you can skip this section.

Imagine you are the system integrator and you need to ROM a component. How do you know what functions to ROM? Should you just ROM everything?

The answer is typically no.

3.1 Getting a List of Functions from a Library

The first step is to find out what's in the library. Do this with the nm6x utility in cgtools\bin as follows:

```
 [>] C:\CCStudio_v3.3\C6000\cgtools\bin\nm6x –fg rts6400.lib
```

This produces a listing of global symbols with filenames pre-pended as follows:

```
...
sinhf.obj: 00000000 U _expf
sinhf.obj: 00000000 T _sinhf
sinit.obj: 00000000 T ___add_dtor
sinit.obj: 00000000 U __dtors_ptr
sinit.obj: 00000000 T _call_dtors
sinit.obj: 00000000 B _dtors
snprintf.obj: 00000000 U __printfi
snprintf.obj: 00000000 U _memcpy
snprintf.obj: 00000000 T _snprintf
sprintf.obj: 00000000 U __printfi
sprintf.obj: 00000000 U _memcpy
sprintf.obj: 00000000 T _sprintf
...
```

"T" stands for text (code), "B" is for bss and "U" denotes "unresolved" (for example, a call to a function as opposed to the function definition). Not shown above is "D" for initialized data that is const data, which is a very good candidate for ROMing.

We can filter this output to list only code symbols as follows:

```
[>] nm6x -fg rts6400.lib | perl -w -n -e "print if (/\s+T\s+/)" | perl -w -n -a
-e "s/$F[0]\s+[0-9a-fA-F]+/   \/\* $F[0] \*\/ \t/g; s/ T /-u /g; print"
```

This is a complicated command, so a batch file is provided to help. It produces output like this:

```
...
/* sinhf.obj: */     -u _sinhf
/* sinit.obj: */     -u ___add_dtor
/* sinit.obj: */     -u _call_dtors
/* snprintf.obj: */  -u _snprintf
/* sprintf.obj: */   -u _sprintf
/* sqrt.obj: */      -u _sqrt
/* sqrtf.obj: */     -u _sqrtf
/* sscanf.obj: */    -u _sscanf
/* strasg.obj: */    -u __strasg
...
```

Why do we use Perl for this as opposed to grep and/or sed and/or awk? The answer is simply that later stages in this process require Perl, hence we wanted to keep dependencies to a minimum and avoid the need for the cygwin or MKS toolkit.

Note: The –e Perl option makes it obvious that the job can be done from the command line. If you are sed-savvy, you can rewrite the above command to avoid Perl.

Dissecting the above command we have:

- **`perl –w –n –e`**
  –w enables warnings. –n assumes a "while (<>) { ... }" loop around program letting us do line by line search and replace. –e invokes the Perl command line.

- **`"print if (/\s+T\s+/)"`**
  Print any lines if they match 1 or more whitespace characters followed by the ‘T’ text/code key, followed again by 1 or more whitespace characters. This will match, for example,

```
sinhf.obj: 00000000 T _sinhf
```

- **`"s/$F[0]\s+[0-9a-fA-F]+/   \/\* $F[0] \*\/ \t/g; s/ T /-u /g; print"`**
  The matched code/text lines are piped thru the above filter. The –a Perl switch tokenizes the remaining lines by default on white-space separators. So sinhf.obj: gets stored in $F[0]. The s/something/somethingElse/ syntax is used for find and replace. We replace sinhf.obj: followed by whitespace (\s+) followed by a set of alphanumeric digits, with the C comment characters. Hence…

```
sinhf.obj: 00000000
```

  becomes…

```
/* sinhf.obj: */
```

  Finally we match the code/text qualifier “ T “ and replace it with “–u” to create the finished line (the –u linker qualifier is explained in later sections.)

```
/* sinhf.obj: */              -u _sinhf
```

Once you’re happy with the output, pipe the results to a file by appending > all_fxns.cmd (or any other output filename) to the long nm6x command.

Now we’ve condensed the list to contain only code sections. At the same time we’ve created the syntax the linker expects to see for forced inclusion of a function and all its children.

3.2 Determining What Functions to Eliminate

On ‘C6000 the .bss section is problematic for ROMing. Global variables are defined in the .bss section. The register DP (B14) points to the beginning of the .bss section, and global variables are accessed at an offset from the DP. The .bss section, then, cannot be split. If it were split, some DP + offset global variable accesses would work, and others would not.

References to globals need to be at a fixed address in RAM. This is true even for .far or functions in ROM calling RAM. Since .bss is a single un-splittable section, there is a high risk that the application’s .bss may inadvertently move the ROM-component-referenced .bss simply as a result of linker placement.

The following technique would work in the user's linker command file:

```
.bss: {
   romImage.obj(.bss)    /* Place the ROMed .bss section first */
   *(.bss)               /* then place remaining application .bss sections */
} > 0x00801000           /* starting at fixed address used in romImage ROMing */
```

But what does a second ROM component do? How does it know the end of romImage.obj's .bss contribution?

Since .bss presents a problem, we eliminate all files containing .bss from the ROM candidate list. This is a bit of a sledgehammer approach; only the functions that reference .bss are relevant, but this is tricky to determine without advanced scripts.

Some component developers opt to build 'C6000 content with --mem_model:data=far. This makes all data references far (and .far is splittable). Unless you have many global scalars, this shouldn't have a big performance impact. See this FAQ for more on 'C6000 memory models.

The following complex command gathers a list of all the files in the RTS that reference .bss whose functions must ultimately be removed from the ROM list:

```
 [>] nm6x -fg rts6400.lib | perl -w -n -a -e "print \"$F[0]\n\" if (/\s+B\s+/)"
```

This command matches " B " lines such as:

```
sinit.obj: 00000000 B _dtors
```

The –a tokenizing then isolates the strings and grabs the first word in $F[0], which yields:

```
sinit.obj:
```

Next we need to cross-reference the .bss files list with our code/text functions list, eliminating all functions from any of the .bss-referencing files. That's too difficult on the command line, so a "real" Perl script is used (elim_bss.pl).

```
[>] perl elim_bss.pl all_fxns.cmd bss_files.txt
```

In this example, all_fxns.cmd is the result of the code/text filter and bss_files.txt is the result of the bss files filter. The result is as follows:

```
/* divu.obj: */      -u __divu
/* dtos.obj: */      -u _dtos
/* ecvt.obj: */      -u _ecvt

/* exp.obj: */       -u _exp
/* exp10.obj: */     -u _exp10
/* exp10f.obj: */    -u _exp10f
/* exp2.obj: */      -u _exp2
/* exp2f.obj: */     -u _exp2f
```

Functions in files that reference .bss are replaced with a blank line for easy diff-checking against the original list.

Now we've condensed the list to be only those code sections not referencing .bss.

3.3 Removing C++ and Host I/O Functions

The RTS library contains a mixture of function categories, including:

- Core functions embedded programs need, such as memset, memcpy, and math functions.
- C++ functions such as new and delete.
- Host I/O operations such as clock(), printf, and friends.

Typically you only need the core routines and would like to automatically eliminate everything else. Unfortunately this is quite difficult to script.

We can determine the C++ functions by running a Perl script, func_info.pl, from the CGT XML utilities package (login required) as follows. (Several scripts from the CGT XML utilities package are used throughout this doc.)

```
[>] ofd6x -xg rts6400.lib | perl c:\temp\cg_xml\ofd\func_info.pl -n
```

If you have a recent OFD version with command-line options to trim the XML size and therefore speedup the processing, you can use the following instead:

```
[>] ofd6x -xg --xml_indent=0 --obj_display=none --dwarf_display=none,dinfo
rts6400.lib | perl c:\temp\cg_xml\ofd\func_info.pl -n
```

This yields:

```
...
"operator delete[]","EDGRTS/array_nodel.cpp",0x0,0x8,8
"operator new[]","EDGRTS/array_nonew.cpp",0x0,0x8,8
"operator delete[]","EDGRTS/array_pdel.cpp",0x0,0x4,4
"operator new[]","EDGRTS/array_pnew.cpp",0x0,0x8,8
"operator delete","EDGRTS/delete.cpp",0x0,0x18,24
"operator delete","EDGRTS/delnothrow.cpp",0x0,0x8,8
"__std__needed_destruction_list","EDGRTS/dtor_list.cpp",0x0,0x8,8
"__already_marked_for_destruction","EDGRTS/dtor_list.cpp",0x0,0xc,12
...
```

If you are sure the end application will be in C, then any function in a .cpp file can be eliminated from the ROM candidates list. However this is not a robust technique since it relies on file extensions. We therefore do not provide any automation to eliminate such functions.

Host I/O is not much better; there's nothing to key on to differentiate printf() from a core routine like memset(). So we simply do a manual elimination.

The end-result is in rom_chosen_fxns.cmd:

```
...
   /* abs.obj: */    -u _abs
   /* abs.obj: */    -u _labs
   /* abs.obj: */    -u _llabs
   /* absd.obj: */   -u __absd
   /* absf.obj: */   -u __absf
   /* acos.obj: */   -u _acos
...
```

You can further trim the list if you know that certain functions will not be used in the final system.

3.4 Using a Call-Graph

ROMing leaf functions at the end of the call-tree is ideal. Patching is simple—you can just replace that ROM function with a RAM version.

As you go farther up the call-tree, if A calls B and you ROM both, then a bug in B means you also need to replace A. Or, indirection (function pointers) can be employed when A calls B. In either case there's more to consider.

So if you can fill up the ROM with stable leaf functions, that's ideal.

If not, a call graph helps determine what *is* the depth of function A? Here's a sample of the call_graph for the RTS 'C6400 library:

```
__neqd : wcs = 0 : fn = cmpd.c : sz = 184

__neqf : wcs = 0 : fn = cmpf.c : sz = 68

__remul : wcs = 0 : fn = imath40.c : sz = 92
|  __divul : wcs = 0 : fn = imath40.c : sz = 424

__remull : wcs = 72 : fn = imath64.c : sz = 100
|  __divull : wcs = 48 : fn = imath64.c : sz = 640
|  |  <repeat ...>
|  __mpyll : wcs = 0 : fn = mpyll.c : sz = 136

_acos : wcs = 168 : fn = acos.c : sz = 1008
|  __addd : wcs = 56 : fn = addd.c : sz = 344
|  |  <repeat ...>
|  __cmpd : wcs = 0 : fn = cmpd.c : sz = 140
|  __mpyd : wcs = 48 : fn = mpyd.c : sz = 428
|  |  <repeat ...>
|  __negd : wcs = 16 : fn = negd.c : sz = 44
|  __subd : wcs = 72 : fn = subd.c : sz = 52
|  |  <repeat ...>
|  _sqrt : wcs = 120 : fn = sqrt.c : sz = 632
|  |  __addd : wcs = 56 : fn = addd.c : sz = 344
|  |  |  <repeat ...>
|  |  __cmpd : wcs = 0 : fn = cmpd.c : sz = 140
|  |  __mpyd : wcs = 48 : fn = mpyd.c : sz = 428
|  |  |  <repeat ...>
|  |  __subd : wcs = 72 : fn = subd.c : sz = 52
|  |  |  <repeat ...>
|  |  _frexp : wcs = 88 : fn = frexp.c : sz = 440
|  |  |  __cmpd : wcs = 0 : fn = cmpd.c : sz = 140
|  |  |  __mpyd : wcs = 48 : fn = mpyd.c : sz = 428
|  |  |  |  <repeat ...>
|  |  _ldexp : wcs = 80 : fn = ldexp.c : sz = 596
|  |  |  __cmpd : wcs = 0 : fn = cmpd.c : sz = 140
|  |  |  __mpyd : wcs = 48 : fn = mpyd.c : sz = 428
|  |  |  |  <repeat ...>
```

Based on this information we might deduce:

- _neqd, _neqf, _remul are solid candidates for ROMing. The first two have a depth of 0 and _remul has a depth of 1. All of these functions are fairly small.
- The total size of _remull, including its children, is > 876 bytes. Backtracking child function _divull, it calls three other functions with combined size ~ 300 bytes. So the depth is 2 and total size is > 1 KB. Perhaps this should be considered for exclusion.
- acos() has at least a depth of 4, is quite sizeable in itself, and brings in other quite large functions such as ldexp(). ROMing _acos() may not be wise.

This call graph was obtained via the call_graph.pl script in the cg_xml package.

```
 [>] ofd6x -xg rts6400.lib | perl c:\temp\cg_xml\ofd\call_graph.pl --func_info
```

Or, if you have a recent OFD version with command-line options to trim the XML size and therefore speedup the processing, you can use this command:

```
 [>] ofd6x -xg --xml_indent=0 --obj_display=none,header,optheader
--dwarf_display=none,dinfo rts6400.lib | perl c:\temp\cg_xml\ofd\call_graph.pl
--func_info
```

4 ROM Creation

Once the list of functions to ROM has been determined, the second phase of the process is ROM creation.

4.1 No Partial Links

In the past partial linking was a technique used in ROM image creation. Partial linking (lnk6x –r) was designed to partition large applications, linking each part separately, and then linking all the parts together to create the final executable program.

Partial linking can have several undesired side-effects:

- **No trampolines.** If the total PC address reach of your ROM code is > 21 bits you're out of luck.
- **No conditional linking.** Several CGT issues have been fixed, so this may be a non-issue, but some users have seen additional functions being linked in even when not referenced.
- **More prone to errors.** With just –r, unresolved ROM symbols could be mistakenly left in the partially linked object file.

Instead we use lnk6x –a –r. This produces an executable, relocatable output module. As noted previously, we can hand that .out to the silicon team for ROM masking directly. We use the exact same bits for testing in the simulator as well (independent pre-load of rts_rom.out).

4.2 ROM Creation Command Line

To generate the ROMable rts_rom.out we do the following:

```
lnk6x -w -a -r rom_chosen_fxns.cmd rom_rts.cmd -o=rom_rts.out -m=rom_rts.map --
xml_link_info=rom_rts.xml -l rts6400.lib
```

The rom_chosen_fxns.cmd file is our symbols command file generated in Section 3. At this point, the –u _symbolName syntax needs explanation. The –u option introduces an unresolved symbol into the output module's symbol table. If we didn't do this, then nothing would get linked in to rom_rts.out because no references to the functions were made! This option also forces any dependent child functions to be brought into the ROM image.

As noted before, using –a with –r produces an *executable, relocatable* object module. The output file contains the special linker symbols, an optional header, and all resolved symbol references. However, the relocation information is retained. In theory we don't need –r, however by keeping it we enable future scripts to take advantage of this information. More importantly –r is essential if you choose the monolithic .rom_rts path outlined in Section 5.2, whereby you link against the ROM image and NOLOAD the whole section.

The upside of producing an executable object module in conjunction with –u _symbolName is that we can't "forget" functions. If function A calls function B, and we do –u _A, then B gets brought in automatically. Even then we have a second error-check by virtue of –a since the linker warns us if A calls B but B, for unknown reasons, didn't get included in the ROM build. The downside is that if function A is in g729.lib and it calls memcpy() from the RTS lib, then you'd need to eliminate A from the g729 component ROM build since memcpy() wouldn't be resolved in that component alone. For this problem it's more natural to ROM a set of components into a single ROM image.

Now analyzing rom_rts.cmd as follows:

```
MEMORY {
        RTS_RAM:     o = 0x00018000, l = 0x00008000   /* uninitialized RAM data */
        RTS_ROM:     o = 0x00030000, l = 0x00020000   /* ROM code */
        RTS_RAM_I:   o = 0x00050000, l = 0x00010000   /* initialized RAM data */
        DUMMY:       o = 0x00060000, l = 0x00010000   /* 'fake' section */
}

SECTIONS {
        .rom_rts: {
                *(.text)
               *(.const)
        } > RTS_ROM

        .ram_rts: {
                *(.bss)
                *(.far)
        } > RTS_RAM

        .switch {} > RTS_RAM_I

        .cinit  {} > DUMMY
        .pinit  {} > DUMMY

}
```

All code and const data goes into ROM. The bundling into a single .rom_rts section assumes that the memory architecture is unified, that is it can be used as both program and data memory. On ISAs with dedicated memory for program vs. data you'll need separate output sections. For example:

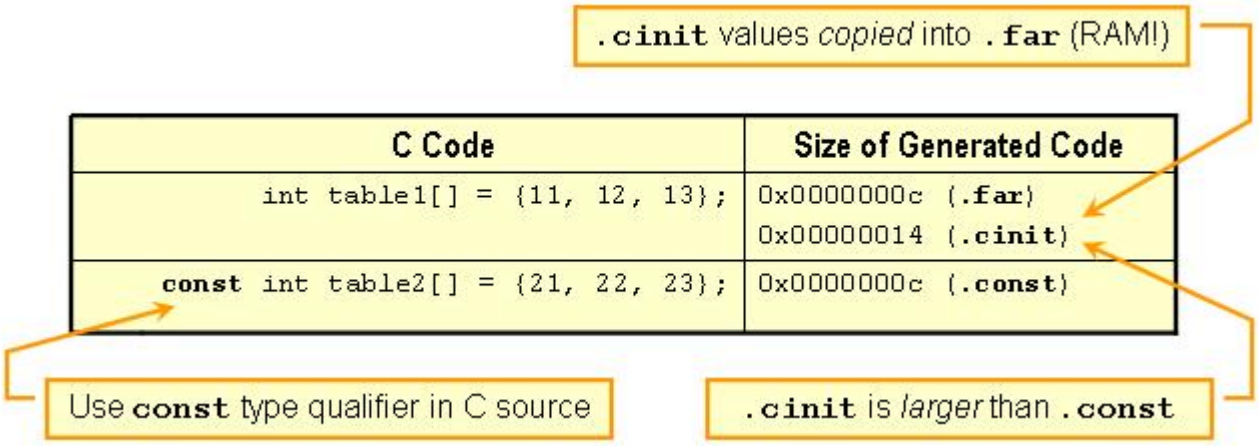
```
        .rom_rts_code: { *(.text) } > RTS_ROM_CODE
        .rom_rts_code: { *(.const) } > RTS_ROM_CONST
```

Any .far data referenced by the code goes into RAM.

The .cinit and .pinit sections are placed in a "DUMMY" memory section. The next section explains how to handle these two initialized data sections.

A general recommendation is that if you have access to the source code, you should use const aggressively. Forgetting const can incur more than double the RAM footprint hit since the data goes into .far and the (slightly larger) C initialization records also often go into RAM (unless you have a good boot routine splitting up ROM and RAM .cinit). A bigger .cinit also means longer boot-times.

Figure 2. Footprint Benefits of Using const Keyword



Returning to rom_rts.cmd, notice that we place all text sections in .rom_rts. If you have additional code sections to be ROMed that are not subsections of .text, you should also place them in ROM output section(s). However, good advice is to rely on the compiler –mo flag (-ms on TMS470) to provide function-level granularity allowing each function to be linked independently. Note that function-level linking is also useful for optimal cache placement purposes on 'C64x/'C64x+. Code in files to be ROMed may not necessarily contain all functions in the "active" code of a module and so may cause L1P cache conflict misses if ROMed at random locations. If a module's active ROM code is less than the cache size (32 KB), it is best to lump together all such code in ROM (the order does not matter). When actually using the ROM, the associated RAM active code needs to be placed in memory in a way that it does not conflict with the ROM area, otherwise there will be unnecessary conflict miss penalties.

Major performance gains can be made on ‘C6000 with clever cache placement, so using –mo upfront enables this later in the process. The –mo option also aids the patching process since we can simply replace a leaf function without having to replace any others that may have been in the same source file.

Also on the ROM command line we have:

```
--xml_link_info=rom_rts.xml
```

This is essential. It produces an XML representation of the traditional linker map file. Many of you have written scripts in the past to parse the text-based map file, and many of you got upset when the format changed as features were added, resulting in broken scripts! The XML equivalent suffers no such problems—all content is reached via a tree of tag-names that will never change.

There are hundreds of XML parsers available. In this case, we used ActivePerl and the XML::Simple parser module.

The end-result of the ROM build is rom_rts.out. We can confirm it is indeed an executable by running ofd6x on the .out file.

```
OBJECT FILE:  rom_rts.out

 Object File Information

    File Name:               rom_rts.out
    Format:                  TI-COFF Version 2
    File Type:               executable file
    Time Stamp:              Fri Mar 30 15:13:16 2007
    Machine:                 TI C6x
    Machine Endian:          little endian
    Entry Point:             0x00000000
    Vendor:                  Texas Instruments, Inc.
    Vendor Version:          6000
    Number of Sections:      7
    File Length:             566661
    TI-COFF f_flags:         0x000011a2
    TI-COFF f_flag:          F_SYMMERGE (debug type information merged)
    CPU Generation:          0xa
    Control Data Endian:     little endian
...
```

Now we can use the ROM image and the XML file we produced along with it, but first a word about some quirks we need to be aware of with certain Codegen sections.

4.3 Issues with the .cinit Section

The .cinit section is a compiler-generated section that contains data for initializing global and static variables. The linker combines each input library and object file's .cinit into a single .cinit section. The boot routine or a loader uses this table to initialize all the system variables in .bss and .far on 'C6000.

We've already eliminated files with .bss from the ROM. So what's peculiar?

The .cinit section is an initialized data section, but we can't easily ROM it because it's a single block with special NULL-record termination requirements. If we ROMed separate .cinit component contributions, we'd need a more complex bootloader.

Hence, let's assume we're not ROMing .cinit. The peculiarity then, is that because it's an initialized section, we must "bring forward" the raw data it contains to the final application build.

Thankfully, unlike .far, variables and pointers that are contained in .cinit are not referenced/called directly from the ROM. That removes one burden since we only need to bring the data forward; we don't need to preserve its "location" from the ROM creation stage.

So how do we bring the data forward? Our solution is to extract the raw data from the ROM executable using the cg_xml scripts package and the OFD tool again.

```
[>] ofd6x -x ..\rom_image\rom_rts.out | perl c:\temp\cg_xml\ofd\extract_sections.pl
-s=.cinit -p=.cinit:rom_rts
```

This produces two files, rom_rts.out.c and rom_rts.out.h. The .h file contains the extern declarations for the variables and the C file contains the raw data as follows:

```
#pragma DATA_SECTION(_cinit, ".cinit:rom_rts")
/*******************************************************
** _cinit[0x7fc]: paddr = 0x00060000
*******************************************************/
const unsigned char _cinit[0x7fc] = {
0x88, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xf0, 0x3f, 0xda, 0x90, 0xa4, 0xa2, 0xaf, 0xa4, 0xee, 0x3f, 0x87, 0xa4,
0xbe, 0x08, 0x00, 0x00, 0x00, 0x40, 0x06, 0x01, 0x00, 0x01, 0x00, 0x00, 0x00,
...
0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x9c, 0x01, 0x01, 0x00, 0x00,
0x00, 0x00, 0x00, };
```

The extract_sections.pl file simply takes the sections you pass on the command line (--sect or –s parameter), looks up their location in the COFF file, and peels off the raw data associated with that initialized section. Optionally you can tag the section with the --pragma or –p parameter to put it in its own section or subsection. Using subsections such as .cinit:rom_rts means we have no extra linker work to do in the final application—once we add rom_rts.out.c to the application build, the RTS ROM's .cinit contribution is automatically folded into the final .cinit tables.

See Reference 8 for a full explanation of this process, or run perldoc extract_sections.pl.

4.4 Issues with the .switch Section

The .switch section is a little easier to handle than .cinit but still warrants some explanation.

The .switch section contains data tables related to “case” statements in C. It is an initialized data section. It’s easier than .cinit because it can be freely split up and there are no special section termination requirements.

But it’s still tricky because it’s a compiler-generated section with only object-file granularity. Let’s say you have a file foo.c with 4 functions A, B, C, and D built with –mo for conditional linkage. A and B are to be ROMed, but not C and D. Now imagine that A has a .switch contribution and so does C. If you ROM .switch then it’s the *whole file’s* .switch without function granularity. The .switch section is somewhat like .far in that address references made from ROM must be preserved in the final application. Now we have a situation where C’s .switch references *must* be to foo.obj’s switch section—this is a tough restriction to place on the application.

The safest approach is to not have anything in ROM that references the .switch section.

If we do ROM functions referencing .switch then we should ROM all functions in the file, or at least all the functions with a .switch contribution. Like .cinit, we need to “bring forward” that initialized data in such cases (presuming we put .switch in RAM), using the aforementioned extract_sections.pl technique.

```
ofd6x -x ..\rom_image\rom_rts.out | perl c:\temp\cg_xml\ofd\extract_sections.pl
-s=.cinit -p=.cinit:rom_rts -s=.switch -p=.switch:rom_rts
```

Here though, we also have some extra linker work to do at the final link since address references to .switch need to be preserved.

```
/* ROM RTS command file */
MEMORY {
...
        RTS_RAM_I:   o = 0x00050000, l = 0x00010000
...
}

SECTIONS {
...
        .switch {} > RTS_RAM_I
...
}
```

```
/* application command file – keep ROM RTS’s switch references at same MEMORY */
        .switch_rts { rom_rts.out.obj(.switch) } > RTS_RAM_I
```

5 ROM Usage

The key at this stage is to make sure we use the ROM functions and constant data from the addresses they were masked at.

How do we ensure that we're referencing a function from ROM and not from RAM? One way is to compile out the ROM functions from the source code before linking so that double or ambiguous linking is prevented. Ambiguous linking may result in either ROM functions getting bypassed (causing a space penalty), or RAM functions linking to ROM functions at the wrong addresses. The compile out for ROM however requires changes in several sources to #ifdef out specific functions that were ROMed, making the sources non-natural and cumbersome to maintain. Another technique would be to strip out ROM objects at the linker-level before linking the release image. Unfortunately this feature is not supported in the linker.

Two working methods are available to us. Furthermore we can automate both to eliminate the possibility of errors.

5.1 ROM Symbols Library

Here we build a ROM symbols library that may be added to the final link in the same fashion as a normal library. Basically, for each symbol in the ROM that we want to expose, we produce a temporary assembly file as shown below:

```
; This file is *auto-generated*. Do not edit. Mods risk being overwritten
; Assembly file : rom_rts_out___divull_tmp.asm

        .global     __divull
__divull              .set        0x38b40
```

Multiple files like this (1 per symbol) are assembled and archived to produce the ROM symbols library. How do we achieve this? A Perl script in the CGT XML package parses the XML map file produced in Section 4.2. It takes as input the linker XML and the name of any ROM and RAM memory sections used in rom_rts.cmd.

```
 [>] perl c:\temp\cg_xml\map\get_rom_symbols.pl rom_rts.xml -s=.rom_rts –s=.ram_rts
```

It goes through the XML file looking for symbols within the memory ranges specified. Each time it finds one it opens a new .asm file, makes the symbol global, sets its value to where it was ROMed (or the RAM location referenced) then closes the file.

Why one symbol per file? Why not put all the symbol .global and .set directives in a single file then archive just that single assembled file? If we did this, it would defeat per-function patching. We need to be able to patch functions individually. A single .asm file would reduce the granularity back to file-level and give an all or nothing usage of ROM code.

With the ROM symbols in a library we can use the linker's standard priority mechanisms such as the –priority switch.

```
-priority

/* pick up the ROM functions instead of RAM ones */
-l ..\rom_rts_eval\asm_sym_bind\rom_rts_symbols.lib

/* pick up remaining RTS fxns that weren't ROM'ed from RAM if referenced */
-l rts6400.lib
```

5.2 DSECT / NOLOAD Technique

This technique achieves the same thing as the ROM symbols library; it's just an alternative. Here we create a command file that looks as follows:

```
/* This file is *auto-generated*. Do not edit. Mods risk being overwritten */

SECTIONS {
  .dummy_rom_rts_out0 { -lrts6400.lib<imath64.obj> (.text:__remlli) } > 0x40880, type=NOLOAD
  .dummy_rom_rts_out1 { -lrts6400.lib<div.obj> (.text:_ldiv) } > 0x3e8c0, type=NOLOAD
  .dummy_rom_rts_out2 { -lrts6400.lib<fixdi.obj> (.text:__fixdi) } > 0x3f760, type=NOLOAD
  .dummy_rom_rts_out3 { -lrts6400.lib<cmpd.obj> (.text:__gtrd) } > 0x3e000, type=NOLOAD
  ...
}
```

This method requires every single function or symbol to have a separate NOLOAD or DSECT section placed at its address in ROM. This cmd file (rom_rts_noload.cmd) then gets linked into the final RAM application. The same rationale for 1 symbol per file in the ROM symbols method applies to having a separate section for each symbol—that is, per-function patchability is enabled by simply commenting out a given entry. Also if one big DSECT/NOLOAD section were used, then the release were built with new tools, it may result in wrong resolution of the ROM addresses as ROM functions may grow or shrink relative to the real code in ROM that was using a different version of tools when compiled. The separate section for each function completely prevents this situation in a legal manner; the linker generates no symbol ambiguity warnings.

Some questions still remain unanswered:

Q. What's the difference between DSECT and NOLOAD?

A. Not a great deal. Some ROM creators have used NOLOAD, some DSECT. This FAQ explains the subtle differences between them. Both do the job of ensuring that sections are not loaded and no space is allocated in target memory (since the code is already in ROM).

Q. Why call out every function? Why not just do the following in the application's cmd file?

```
rom_rts.out

SECTIONS {
      .rom_rts > RTS_ROM, type=NOLOAD  /* same location as ROM creation */
...
}
```

A. This would work but you lose the granularity for patching; ROMing becomes an "all or nothing" affair. We want to preserve the ability to patch selected functions that went into .rom_rts, hence the ROM symbols library and DSECT/NOLOAD techniques.

Q. Why did you prefix everything with archive, and object file? Why not just use .text:__remlli > 0x40880, type=NOLOAD?

A. That would indeed be cleaner. However what would happen with const contributions from each file? They'd all look the same. That is:

```
.const > 0x60108, type=NOLOAD
.const > 0x60238, type=NOLOAD
...
```

Clearly that wouldn't work. Hence we need some uniqueness qualifiers:

```
.dummy_rom_rts_out243 { -lrts6400.lib<strtod.obj> (.const) } > 0x60108, type=NOLOAD
.dummy_rom_rts_out244 { -lrts6400.lib<strerror.obj> (.const) } > 0x60238, type=NOLOAD
...
```

Q. Why mux the rom_rts.out string into the output dummy section names?

A. If we just did .dummy1, .dummy2, etc., we could conflict with other section names in the final RAM application. Muxing the ROM executable out file name string into the section ensures the script scales if > 1 component needs to be ROMed.

Q. How did you automatically generate this command file?

A. Yet another Perl script!

```
 [>] perl c:\temp\cg_xml\map\get_rom_sects.pl rom_rts.xml -s=.rom_rts -t=NOLOAD
```

(Run perldoc nameOfScript.pl to get auto-documentation.)

This script has to work a little harder since we need to find not only the symbol name and address but also the archive and object file from which they originally came.

Note: In both scripts you can pass in multiple output sections. For example, if you split code and constant data up during ROM creation you can pass in –s=.rom_rts_code –s=.rom_rts_const.

5.3 ROM Symbols vs. NOLOAD Technique

The advantages and disadvantages of the two methods are as follows:

Table 1. ROM symbols .v. NOLOAD method

	Advantages	Disadvantages
ROM Symbols	• "Natural" linker priority scheme for patching. That is, use of –priority switch.	• Generates a lot of files (but they're all temporary so a simple, provided batch file or .sh can delete them after the archive is created).
NOLOAD / DSECT	• Everything contained in a single cmd file. • User can modify the editable cmd file as necessary.	• Creates a bunch of extra output sections that make the map file harder to read.

5.4 The .far Section and RAM Functions Referred to by ROM Functions

This was covered briefly in the .cinit and .switch discussions, but it warrants further mention.

Consider the following code:

```
int tableInRAM[10];

void func3(void)
{
   volatile int x;

   func5();
   x = tableInRAM[5];
}
```

On ‘C6000, tableInRAM goes into .far initialized by a C init record. If func3 is ROMed, the address reference to tableInRAM can never change—it needs to be preserved in the final application link.

There’s no great magic in how we do this; all we do is ensure the same RAM MEMORY sections are used in the ROM creation and usage stages. For example:

```
MEMORY {
        RTS_RAM:     o = 0x00010000, l = 0x00010000 /* reserved : ROM .far accesses */
        RTS_ROM:     o = 0x00030000, l = 0x00020000
        RTS_RAM_I:   o = 0x00050000, l = 0x00010000 /* rsvd : ROM .switch accesses */
}

MEMORY {
        RAM:     o = 0x00020000, l = 0x00010000    /* application RAM */
}
```

Note that the application is free to shorten the RTS_RAM reserved space to equal the length of RTS_RAM actually used by the ROM component, although a pad may be wise.

The same theory applies to func5() in the above example if it gets explicitly placed in RAM. That is, its address needs to be preserved in the final application link. However this is unlikely to be an issue for reasons outlined in the ROM creation section.

6 Testing

The point of using scripts is to reduce the risk of error. However testing to ensure no ambiguous linking or linker warnings is still required. The documents in the References section give excellent advice on testing the ROM image. We suggest that you see these for full details.

Here is a brief summary of the techniques for testing:

- **RAM test application using the ROM.** This is the most obvious suggestion. A sample application is provided, tested on CCStudio 3.3 with the 'C6416 Cycle Accurate Simulator. Do Load Program twice: once for rom_rts.out and once for the application that uses a combination of ROM and RAM functions from the Run Time Support Library.

```
    memset(buf, 0, sizeof(buf));    // memset should come from ROM
    memcpy(buf, buf2, sizeof(buf)); // memcpy should come from ROM

    // math fxns should come from ROM – good test since log10 has some cinit
    x = log10(log10_tst);
    printf("%lf\n", x);

    for (i=0; i< sizeof(buf)/sizeof(char); i++) {
  printf("buf[%d] : %d\t", i, buf[i]);   // printf & friends from RAM
    }
    printf("\n");
...
```

  Check the map file to verify that memset, memcpy, log10 all come from ROM (use a fake ROM memory area for test), while Host I/O function printf resides in RAM.

- **Ensure no data writes to ROM area.** Clearly this must not happen. In ROM creation we ensured this by placing all non-const data sections (.bss, .far, .data, .switch) in RAM. But to double-check you can do the following:
  - **Use Advanced Event Triggering.** Some DSPs have this Emulation capability. Place a data action point that halts the CPU if a data write is detected to the ROM area (use a fake ROM memory area for test).
  - **Use the Simulator.** Again, place a data action point that halts the CPU if a data write is detected to the ROM area. CCStudio 3.3, and particularly Service Pack 1, has 'C6000 support for Data Breakpoints. That is, the simulator halts if a data access within a specified memory range occurs.

- **Ensure no unresolved references from ROM functions to other ROM functions.** The methods we used in ROM creation (avoiding partial links, using –u, etc.) should ensure this. However, to be sure we can do the following:

```
 [>] nm6x -fg rom_rts.out | perl -w -n -e "print if (/\s+U\s+/)"
```

  This checks that all references to both code and data in rom_rts.out are fully resolved. That is, no "U" unresolved entries in the symbol table that relate to references from ROM. RAM-based "U" entries are OK. As an example we should **not** see anything like the following:

```
sinhf.obj: 00039040 U _expf             /* this address is in the ROM area! */
```

  Basically we're checking that all code or data going into the ROM is fully linked.

7 Conclusions

The primary goal of this application note was to automate more of the ROMing process. The scripts and (occasionally cryptic) commands ensure this.

The additional benefit is that the risk of linker warnings or errors via ambiguous linking is reduced since the scripts shouldn't miss any elements (provided they were specified and written correctly).

The use-case throughout this application note was the 'C6000 RTS library, but the techniques apply equally well to other ISAs or library components such as algorithms.

8 References

1. *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186)
2. *DSP/BIOS 5.x in ROM* (internal presentation)
3. *Telogy ROMing (*internal document*)*
4. *Software Design for SoCs with On-chip ROM* (internal presentation at TI India Developers conference, 2005)
5. *I Can't Put My Code in ROM* (internal presentation at TI Software Symposium 2007)
6. *BIOS_ROM_issues.doc* (internal document)
7. *Advanced Linker Techniques for Convenient and Efficient Memory Usage* (SPRAA46)
8. *Using OFD Utility to Create a DSP Boot Image* (SPRAA64)

Appendix A. ROMing Multiple Components

In truth, the RTS library is an easy target because it's self-contained—there are no calls to other library functions. However, ROMing a component with references to one or more libraries requires more explanation. This topic is presented in an appendix because the same techniques are used with minor changes.

As a test case, we used the VOL_TI sample algorithm supplied with TI's Reference Frameworks. Running call_graph.pl from Section 3.4 shows that VOL_TI_amplify calls memcpy(). Therefore we know that VOL_TI depends on the RTS component.

```
_VOL_TI_amplify : wcs = 0 : fn = vol_ti_ivol.c : sz = 108
|  _memcpy : wcs = ??? : fn = ??? : sz = ???
|  _scale : wcs = 0 : fn = vol_ti_ivol.c : sz = 224
|  |  __divi : wcs = ??? : fn = ??? : sz = ???
```

Note that call_graph.pl can't show memcpy's filename and size because only vol_ti.l64 was presented to it.

If we try to create a rom_vol_ti.out relocatable executable by running lnk6x as per Section 4.2, it fails as follows:

```
Undefined symbol                first referenced in file
---------                       ----------------
_IVOL_PARAMS                    vol_ti.l64
__divi                          vol_ti.l64
_memcpy                         vol_ti.l64
>> warning: output file 'rom_vol_ti.out' is not executable
```

We can fix this by presenting *both* VOL_TI and RTS components to the lnk6x step as follows:

```
[>] lnk6x -w –x -a -r vol_ti_fxns.cmd rts_fxns.cmd rom.cmd -o=rom_vol_ti_rts.out
-m=rom_vol_ti_rts.map --xml_link_info=rom_vol_ti_rts.xml -l vol_ti.l64
-l c:\CCStudio_v3.3\C6000\cgtools\lib\rts6400.lib
```

Note: Observe that we added the –x option. This forces an exhaustive search of libraries for unresolved symbols and avoids potential library ordering problems.

Finally we need to resolve _IVOL_PARAMS. This is a little trickier and may not be applicable to your ROM components. IVOL_PARAMS is a placeholder for a xDAIS parameters structure to be filled in by the final application. Hence we can't resolve that reference by presenting another library to lnk6x as we did with memcpy and _divi. If you don't have an IVOL_PARAMS implementation available and can't modify the source, then you need to eliminate any functions that reference this from the VOL_TI –u ROM function list.

```
   /* vol_ti_ialg.o64:            -u _VOL_TI_alloc */
   /* vol_ti_ialg.o64:            -u _VOL_TI_initObj */
```

VOL_TI_free() calls VOL_TI_alloc() so the exclusions above are not enough. Hence we also comment out the VOL_TI_free –u entry. In theory we should be done. However, the problem report SDSCM00016782 ("Linker is giving a reference error for a dead function") temporarily means we need to exclude all functions in the file where IVOL_PARAMS references are made, despite the fact that we recompiled VOL_TI with –mo for conditional linking.

At this point all references are satisfied and we can ROM the VOL_TI and RTS combo.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265