TMS320C27xx Translation Assistant User's Guide

Literature Number: SPRU278
March 1998

Texas Instruments

Preface

Read This First

About This Manual

This book explains how the TMS320C27xx translation assistant utility fits in with the rest of the code development tools for the Texas Instruments TMS320C27xx devices. It tells you how to use the translation assistant utility to translate code you already have for TMS320C2xx devices into code that will run on TMS320C27xx devices.

How to Use This Manual

This book helps you learn how to use the Texas Instruments translation assistant utility specifically designed for the TMS320C27xx devices. This book is divided into four parts:

- **Introductory information**, consisting of Chapter 1, gives you an overview of the translation assistant and associated assembly language development tools.
- **Translation assistant description**, consisting of Chapter 2, contains detailed information about using the translation assistant. This portion explains how to invoke the translation assistant utility and generate online help files to help translate instructions that are not fully translated by the translation assistant. This chapter also describes how the translation assistant works with macros.
- **Differences,** consisting of Chapter 3, describes in detail the differences between TMS320C2xx and TMS320C27xx instructions.
- **Reference material**, consisting of appendixes A–D, provides supplementary information, including detailed conversion guidelines for nontranslatable TMS320C2xx instructions in Appendix A, conversion examples in Appendix B, error messages in Appendix C, and a glossary in Appendix D.

Notational Conventions

This document uses the following conventions:

- The TMS320C27xx core is also referred to as TMS320C27xx or ’C27xx. The TMS320C2xx is also referred to as ’C2xx.

- Program listings, program examples, and interactive displays are shown in a `special typeface`. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

```
1 000000          .data
2 000000 002F       .byte  47
3 000001 0032       .byte  50
4 000002 D903        ADDB AR1 , #3
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered. Syntax that is entered on a command line is centered. Syntax that is used in a text file is left-justified. Here is an example of command-line syntax:

  ta27 *[options] filename*

  The **ta27** command invokes the translation assistant utility and has two parameters. The first parameter, *options*, is optional (see the next bullet for details). The second parameter, *filename*, is required and you can enter more than one.

- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves. This is an example of a command that has an optional parameter:

  ta27 [*options*] *filename*

  The **ta27** command has two parameters. The second parameter, *filename*, is required. The first parameter, *options*, is optional. Since options is plural, you can select several options.

- In assembler syntax statements, column 1 is reserved for the first character of a label or symbol. If the label or symbol is optional, it is usually not shown. If it is a required parameter, it is shown starting against the left margin of the shaded box, as in the example below. No instruction, command, directive, or parameter, other than a symbol or label, can begin in column 1.

  symbol **.usect** **"***section name***",** *size in bytes* [**,** *alignment*]

  The *symbol* is required for the .usect directive and must begin in column 1. The *section name* must be enclosed in quotes and the parameter *size in bytes* must be separated from the *section name* by a comma. The *alignment* is optional and, if used, must be separated from the size in bytes, by a comma.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte *value*$_1$ [, ... , *value*$_n$]

  Note that **.byte** does not begin in column one.

  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, each separated from the previous one by a comma.

- Following are other symbols and abbreviations used throughout this document:

Symbol	Definition	Symbol	Definition
B,b	Suffix — binary integer	MSB	Most significant bit
H,h	Suffix — hexadecimal integer	0x	Prefix — hexadecimal integer
LSB	Least significant bit	Q,q	Suffix — octal integer

Related Documentation From Texas Instruments

The following books describe the TMS320C2x, 'C2xx, and 'C27xx devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C2xx User's Guide (literature number SPRU127) discusses the hardware aspects of the 'C2xx 16-bit, fixed-point digital signal processors. It describes the architecture, the instruction set, and the on-chip peripherals.

TMS320C27xx DSP CPU and Instruction Set Reference Guide (literature number SPRU220) describes the central processing unit (CPU) and the assembly language instructions of the TMS320C27xx 16-bit fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.

TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide (literature number SPRU018) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C1x, 'C2x, 'C2xx, and 'C5x generations of devices.

TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide (literature number SPRU024) describes the 'C2x/C2xx/C5x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C2x, 'C2xx, and 'C5x generations of devices.

TMS320C27xx Assembly Language Tools User's Guide (literature number SPRU211) describes the assembly language tools (assembler and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C27xx device.

TMS320C27xx Optimizing C Compiler User's Guide (literature number SPRU212) describes the TMS320C27xx C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the TMS320C27xx device.

TMS320C27xx Simulator Getting Started (literature number SPRU216) describes how to install the simulator and the C source debugger for the TMS320C27xx device. The installation for MS-DOS™, SunOS™, and HP-UX™ systems are covered.

TMS320C27xx Emulator Getting Started (literature number SPRU215) describes how to install the emulator software and the C source debugger for the TMS320C27xx device. The installation for MS-DOS™, SunOS™, and HP-UX™ systems are covered.

TMS320C27xx C Source Debugger User's Guide (literature number SPRU214) tells you how to invoke the TMS320C27xx emulator and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C2xx Emulator Getting Started Guide (literature number SPRU209) tells you how to install the Windows™ 3.1 and Windows™ 95 versions of the 'C2xx emulator and C source debugger interface.

TMS320C2xx Simulator Getting Started (literature number SPRU137) describes how to install the TMS320C2xx simulator and the C source debugger for the 'C2xx. The installation for MS-DOS™, PC-DOS™, SunOS™, Solaris™, and HP-UX™ systems is covered.

Trademarks

To Help Us Improve Our Documentation . . .

If you would like to make suggestions or report errors in documentation, please send us mail or email. Be sure to include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251–1443

Email: comments@books.sc.ti.com

If You Need Assistance . . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32

Email: epic@ti.com

Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68	
English	+33 1 30 70 11 65	
Francais	+33 1 30 70 11 64	
Italiano	+33 1 30 70 11 67	
EPIC Modem BBS	+33 1 30 70 11 99	
European Factory Repair	+33 4 93 22 25 40	
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type "Go TIASP"	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: dsph@ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Figures

Tables

Examples

Notes

Chapter 1

Introduction

The TMS320C27xx translation assistant utility is one of the several code development tools available for TMS320C27xx devices. It converts existing assembly code written with the TMS320C2xx instruction set to assembly code that you can run on ’C27xx devices. This chapter shows how the translation assistant utility fits into the general software tools development flow and gives a brief description of each tool in the code development process.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1–1 shows the TMS320C27xx software development flow. The shaded portion highlights the most common development path; the other portions are optional. They are peripheral functions that enhance the development process.

1.2 Tools Descriptions

The following list describes the tools that are shown in Figure 1–1:

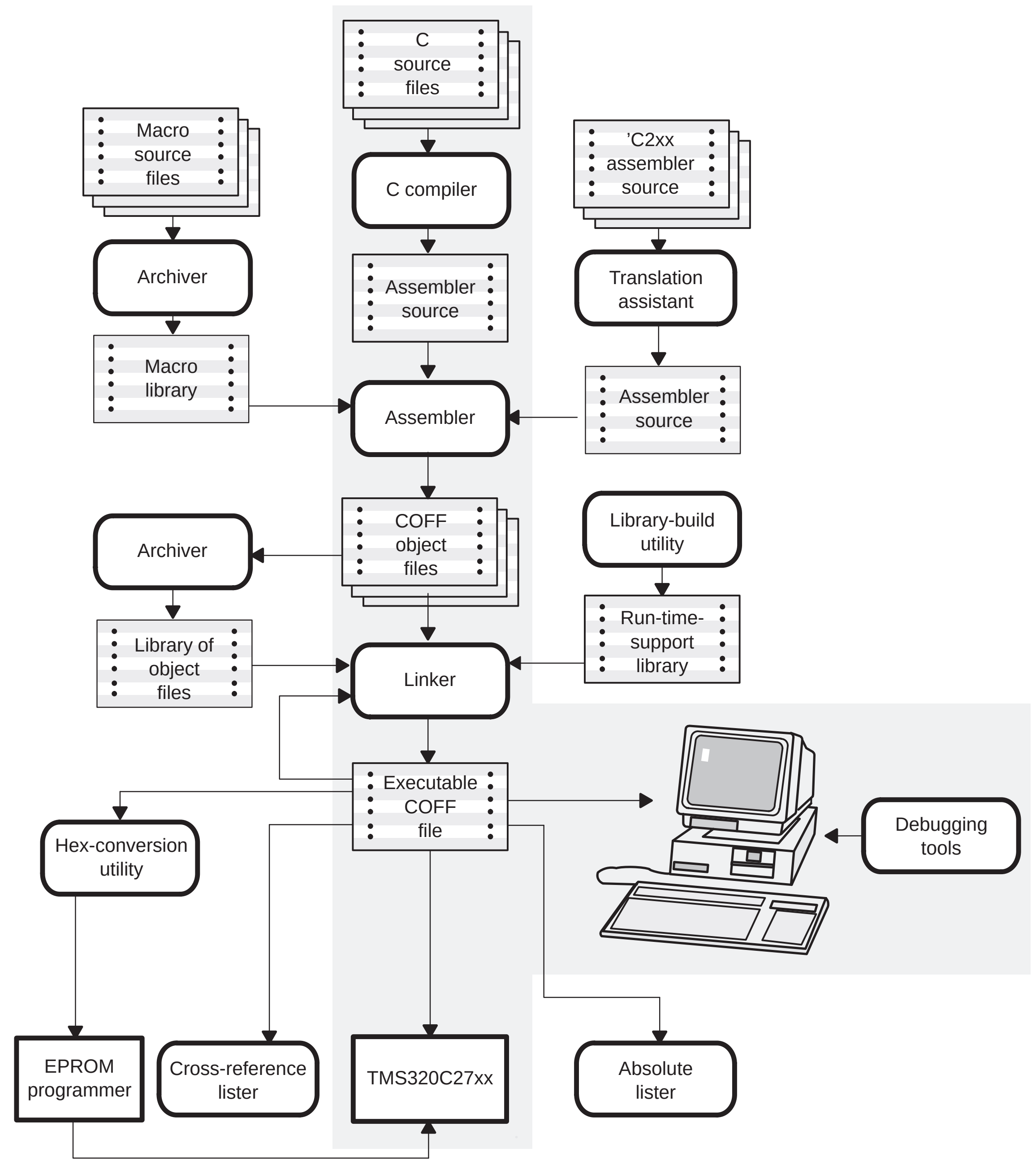
- The **C compiler** accepts C source code and produces TMS320C27xx assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package:
  - The shell program enables you to compile, assemble, and link source modules in one step.
  - The optimizer modifies code to improve the efficiency of C programs.
  - The interlist utility interlists C source statements with assembly language output to correlate code produced by the compiler with your source code.

  See the *TMS320C27xx Optimizing C Compiler User's Guide* for more information.

- The **assembler** translates assembly language source files into machine language COFF object files. Source files can contain instructions, assembler directives, and macro directives. You can use assembler directives to control various aspects of the assembly process, such as the source listing format, data alignment, and section content. See the *TMS320C27xx Assembly Language Tools User's Guide* for more information about using the assembler. See the *TMS320C27xx CPU and Instruction Set Reference Guide* for detailed information on the assembly language instruction set.

- The **linker** combines object files into a single executable COFF object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files (created by the assembler) as input. It also accepts archiver library members and output modules created by a previous linker run. Linker directives allow you to combine object file sections, bind sections or symbols to addresses or within memory ranges, and define or redefine global symbols. See the *TMS320C27xx Assembly Language Tools User's Guide* for more information.

Figure 1–1. TMS320C27xx Software Development Flow



❑ The **archiver** allows you to collect a group of files into a single archive file, called a library. For example, you can collect several macros into a macro library. The assembler searches the library and uses the members that are called as macros by the source file. You can also use the archiver to collect a group of object files into an object library. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members. See the *TMS320C27xx Assembly Language Tools User's Guide* for more information.

❑ You can use the **library-build utility** to build your own customized runtime-support library. See the *TMS320C27xx Optimizing C Compiler User's Guide* for more information.

❑ The **hex conversion utility** converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S™, or Tektronix™ object format. The converted file can be downloaded to an EPROM programmer. See the *TMS320C27xx Assembly Language Tools User's Guide* for more information.

❑ The **absolute lister** accepts linked object files as input and creates .abs files as output. You assemble the .abs files to produce a listing that contains absolute addresses rather than relative addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations.

❑ The **translation assistant** accepts 'C2xx assembly source input and produces a 'C27xx assembly source file. Instructions that are not directly translatable are flagged with warning messages to help you complete the translation.

❑ The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definition, and their references in the linked source files. See the *TMS320C27xx Assembly Language Tools User's Guide* for more information.

❑ The main product of this development process is a module that can be executed in a **TMS320C27xx** device. You can use one of several debugging tools to refine and correct your code. Available products include:

  ■ An instruction-accurate and clock-accurate software simulator
  ■ An XDS™ emulator
  ■ An evaluation module (EVM)

  For information about these debugging tools, see the *TMS320C27xx C Source Debugger User's Guide.*

1.3 Translation Assistant Overview

The translation assistant automates and simplifies the task of converting a working ’C2xx application to a ’C27xx application. The translation assistant directly translates about 75% of the ’C2xx instruction set, creating a simple instruction-to-instruction mapping for these instructions to ’C27xx instructions. Figure 1–2 shows the translation assistant’s role in the assembly language development process.

In the instruction mix of a typical ’C27xx application, approximately 90% of the actual instructions used can be automatically translated from the ’C2xx form. Only 10% of the instructions require hand-coded conversion and familiarity with the original ’C2xx application assembly code. This significantly reduces the effort necessary to make the translation.

The translation assistant utility executes a first-pass translation of the ’C2xx instructions. This process does not take full advantage of significant ’C27xx architecture features that optimize code execution speed, reduce code cycle/word counts, and minimize overall code size. Therefore, while the resulting translation is not optimal, it is an excellent first draft and saves considerable time.

Note: Effective Use of the Translation Assistant

Effective use of this translation assistant requires an understanding of the ’C2xx assembly application being translated.

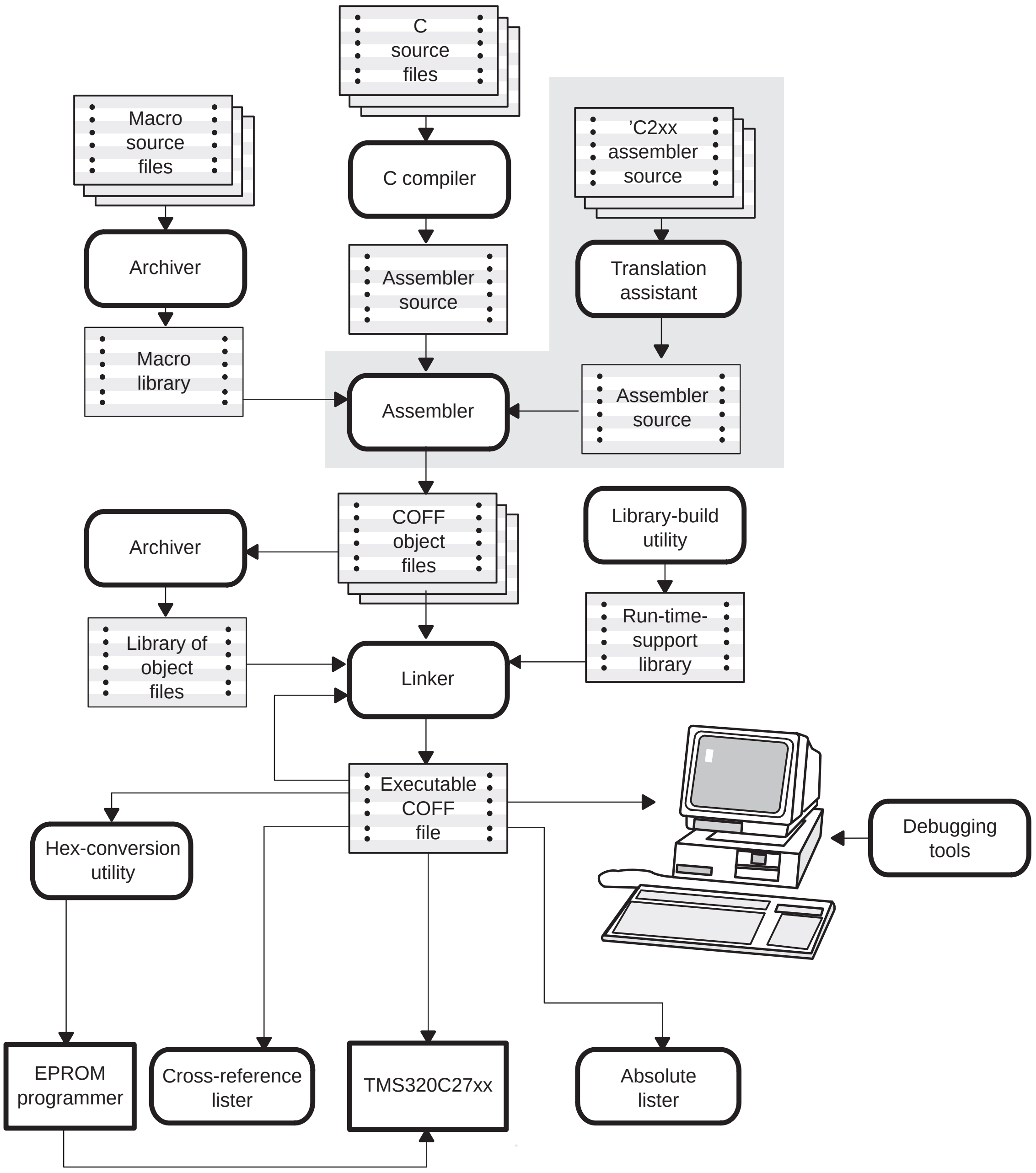
1.3.1 Translation Assistant Presumptions

The translation assistant presumes that any input used already assembles correctly using the ’C1x/’C2x/’C5x assembler run with the –v2xx switch.

1.3.2 Translation Assistant Limitations

The translation assistant cannot convert macro definitions. It ignores them. Optionally, the translation assistant replaces macro invocations with the expanded macro, replacing the formal parameters with the actual arguments used in invocation.

Figure 1–2. Translation Assistant in the Software Development Flow



Chapter 2

Using the Translation Assistant

The TMS320C27xx translation assistant converts assembly code that uses the ’C2xx instruction set to ’C27xx assembly code. The translation assistant uses options to define and control its operation. This chapter explains what each option does and how to use the translation assistant effectively. Topics in this chapter include:

Topic **Page**

2.1 Invoking the Translation Assistant

To invoke the translation assistant, enter:

```
ta27 [inputfile[outputfile][options] ]
```

ta27	is the command that invokes the translation assistant.
inputfile	names the assembly source file you want to translate. If you do not specify an extension, *.asm* is assumed.
outputfile	names a translated file with the default extension *.trn* if the output file is not specified.
options	identifies the assembler options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen. Single-letter options without parameters can be combined: for example, –la is equivalent to –l –a. Options that have parameters, such as –i, must be specified separately.

–a	instructions with absolute memory references are translated, but the operands still need programmer verification.
–d	**–d***name* **[=***value***]** or **–d***name* **[:***value***]** sets the *name* symbol to *value*.
–h	displays a summary of the translation assistant command-line options on your monitor.
–i	specifies a directory where the translation assistant can find files named by the .copy, .include, or .mlib directive.
–k	keeps the original assembly code line as a comment line in the translated output.
–l	creates a .htm file.
–me	expands the macro definition and then translates the ’C2xx instructions within the macro definition.
–w	overwrites the existing .trn files.

2.2 Naming Alternate Directories for Input

The .copy, .include, and .mlib directives tell the translation assistant to use code from external files. The .copy and .include directives tell the translation assistant to read source statements from another file. The .mlib directive names a library that contains macro functions. Chapter 4, *Assembler Directives,* of the *TMS320C27xx Assembly Language Tools User's Guide* contains examples of the .copy, .include, and .mlib directives. The syntaxes for these directives are:

.copy "*filename*"

.include "*filename*"

.mlib "*filename*"

The *filename* parameter names a .copy/.include file. The translation assistant uses the .copy/.include file to read statements from a macro library that contains macro definitions. The filename can be a complete pathname, a partial pathname, or a filename with no path information. The translation assistant searches for the file in these locations in the order given here:

- ❑ The directory that contains the current source file. (The current source file is the file being translated when the .copy, .include, or .mlib directive is encountered.)
- ❑ Any directory named with the –i translation assistant option.
- ❑ Any directory set with the environment variable A_DIR. (See section 2.2.3 on page 2-5 for more information)

Note: Augmenting the Search Algorithm

You can augment the translation assistant's directory search algorithm by using the –i option or the environment variable.

2.2.1 Using the –I Option

The –i option names an alternate directory that contains .copy/.include files or macro libraries. The format of the –i option is as follows:

ta27 –i *pathname* *source filename*

You can use up to ten –i options per invocation; each –i option names one pathname. In assembly source, you can use the .copy, .include, or .mlib directive without specifying path information. If the translation assistant does not find the file in the directory that contains the current source file, it searches the paths designated by the –i options.

Assume that a file called source.asm in the current directory contains the following directive statement:

```
.copy "copy.asm"
```

The pathname for the copy.asm invocation command is as follows:

Operating System	Pathname	Invocation Command
UNIX™	`/'C27xx/files/copy.asm`	`ta27 –i/'C27xx/files source.asm`
Windows™	`c:\'C27xx\files\copy.asm`	`ta27 –ic:\'C27xx\files source.asm`

The translation assistant first searches for copy.asm in the current directory because source.asm is in the current directory. Then the translation assistant searches in the directory named with the –i option.

2.2.2 Using the H_DIR Environment Variable

The translation assistant contains three online help files for use with the –i option. The translation assistant uses the environment variable H_DIR to reference the HTML based online help files when you use the –l option to generate a .htm file for your input file. The three HTML based online help files (help.htm, error.htm, and diff.htm) must be saved in the directory specified by H_DIR. Otherwise, the translation assistant generates incorrect information in the .htm file.

The CD-ROM installation utility automatically assigns the default pathname for the H_DIR environment variable. The command syntax for assigning H_DIR is as follows:

Operating System	Enter
UNIX	**setenv H_DIR "***pathname***"**
Windows	**set H_DIR=** *pathname*

Only one pathname is allowed for H_DIR.

The environment variable remains set until you reboot the system or reset the variable by entering one of these commands:

Operating System	Enter
UNIX	**setenv H_DIR ""**
Windows	**set H_DIR=**

2.2.3 Using the A_DIR Environment Variable

The translation assistant uses the environment variable A_DIR to name alternate directories that contain .copy/.include files or macro libraries. The CD-ROM installation utility automatically assigns the default pathname for the A_DIR environment variable. The command syntax for assigning the environment variable is as follows:

Operating System	**Enter**
UNIX	**setenv A_DIR “”**
Windows	**set A_DIR=**

The environment variable remains set until you reboot the system or reset the variable by entering one of these commands:

Operating System	**Enter**
UNIX	**setenv A_DIR “”**
Windows	**set A_DIR=**

2.3 Using the HTML Option (–l)

The –l option in the translation assistant utility generates a help file (in addition to the translated file) based on your input file. This file is a hypertext markup language (HTML) file that includes hypertext links to the help files included with the translation assistant utility (help.htm, error.htm, and diff.htm). These files (help.htm, error.htm, and diff.htm) must be collected together into one directory indicated by the environment variable H_DIR. This environment variable is set for you by the installation utility. Section 2.2.2, *Using the H_DIR Environment Variable*, page 2-4, explains how the environment variable is set.

Presume the following information:

- The ’C2xx assembly file is named sample.asm, and it is located in a working directory named D:\projects\c27xx.
- H_DIR is set to C:\tools\ta, and the help files are located in C:\tools\ta.

In the working directory, issue the command:

```
ta27 sample.asm
```

If you get no error or warning messages, sample.trn is the final translated output file. Otherwise, issue the following command:

```
ta27 sample.asm –l
```

This command produces two output files (sample.trn and sample.htm) in your working directory.

View the sample.htm file in your web browser by typing the following location in the location or URL box.

file:///D:\projects\c27xx\sample.htm

The sample.htm file has links associated with the instuction that did not translate correctly. Use these links to access the help.htm, error.htm, and diff.htm files that provide the information you need to translate the source file successfully.

Note: Using the –l Option

If you have a problem using the –l option, ensure that the H_DIR environment variable is set.

2.4 Using the Absolute Memory Reference Option (–a)

The translation assistant does not translate instructions with absolute memory references. You can use the –a option to force the translation assistant to translate ’C2xx instructions, but you must verify the translated ’C27xx code.

Example 2–1. Absolute Memory References

(a) ’C2xx source code

```
data1  .set   32
       ADD    data1,3
```

(b) Translation assistant output (without –a)

```
data1  .set   32
       ADD    data1,3

***WARNING!  [W2000]:  Cannot translate absolute
                       memory reference–Op 1

; No Errors, 2 Warnings, 2 Instructions

; 0.00 % instructions translated
```

(c) Translation assistant output (with –a)

```
data1  .set   32

       MOV    DP,#data1

       ADD    ACC, @data1 << 3


; No errors, No Warnings, 1 Instruction

; 100.00 % instructions translated
```

Note: Absolute Memory Reference

In this example, you need to verify that the data page pointer (DP) holds the correct page number.

2.5 Using the Symbolic Constant Option (–d)

The ’C2xx assembler has a –d option that equates a constant value with a symbol. The symbol can then be used in place of a value in assembly source. Within the assembler source, you can test the symbol with the following directives:

Table 2–1. Testing TMS320C2xx Symbols Defined with the (–d) Option

Type of Test	Directives Usage
Existence	.if $isdefed (”name”)
Nonexistence	.if $isdefed (”name”) =0
Equal to value	.if name = value
Not equal to value	.if name != value

To be consistent with the ’C2xx assembler, the translation assistant also provides the –d option. The format of the –d option is as follows:

ta27 –d*name***[=***value***]**

or

ta27 –d*name***[:***value***]**

The *name* is the name of the symbol you want to define. The *value* is the value you want to assign to the symbol. If you omit the value, the symbol is set to 1. The translation assistant does not do any symbol substitution or literal interpretation when the –d option is used. The translation assistant checks only the expression syntax that invokes symbolic constants.

For more information about how to define symbolic constants, see the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User’s Guide* and the *TMS320C27xx Assembly Language Tools User’s Guide*.

2.6 Using the Comment Option (–k)

In ’C2xx and ’C27xx assembly code, a comment can begin in any column and extends to the end of the source line. If it begins in column 1, it can start with a semicolon (;) or an asterisk (*). Comments that begin anywhere else on the line must begin with a semicolon. The asterisk identifies a comment only if it appears in column 1.

For the translation assistant, a source statement that contains only a comment is valid. The translation assistant does not translate a comment, it simply writes the comment to the translated output.

If you use the –k option, the translation assistant keeps the original line of assembly code as a commented line in the translated output.

Example 2–2. Comment Line

(a) ’C2xx source code

```
***************************
* Always start from reset*
***************************
       .sect  “reset”
       B      ALU1      ; Initialize Data Memory.
ALU1:  setc   INTM      ; Disable all interrupts
```

(b) Translation assistant output (with –k)

```
****************************
* Always start from reset.*
****************************
       .sect  “reset”
;      B      ALU1      ; Initialize Data Memory.
       LB     ALU1      ; Initialize Data Memory.
;ALU1: setc   INTM      ; Disable all interrupts.
ALU1:  SETC   INTM      ; Disable all interrupts.

; No Errors, No Warnings, 2 Instructions
; 100.00 % instructions translated
```

2.7 Working With Macros (–me Option)

The translation assistant does not translate macro definitions. Macro definitions are passed through to the output file unchanged. By default, macro invocations are also passed through unchanged. However, you can use the –me option to expand macros inline. The resulting ’C2xx assembly code is translated (see Example 2–3).

Example 2–3. Macro Expansion

(a) ’C2xx source code

```
    ADD3   .macro x, y, z
           ADD    x
           ADD    y
           ADD    z
           .endm

           ADD3   *+, *+, *
```

(b) Translation assistant output (with –me)

```
    ADD3   .macro x, y, z
           ADD    x
           ADD    y
           ADD    z
           .endm

;          ADD3   *+, *+, *
           ADD    ACC, *++
           ADD    ACC, *++
           ADD    ACC, *

;   No Errors, No Warnings, 3 Instructions

;   100.00 % Instructions translated
```

2.7.1 Directives in Macros

When macro invocations are expanded, the macro is inlined and the code is no longer in a macro environment. The following source code generates the output code as shown.

Example 2–4. Directives in Macros

(a) ’C2xx source code

```
mymac  .macro    x,y,z
       .word     x
       .eval     y,z
       .endm

       mymac     4,0,temp
```

(b) Translation assistant output (with –me)

```
mymac  .macro    x,y,z
       .word     x
       .eval     y,z
       .endm

;      mymac     4,0,temp
       .word     4
       .aval     0,temp

;   No Errors, No Warnings, No Instructions
```

2.7.2 Macro Local Variables

When macro local variables are encountered, they are changed so that repeated calls to the macro do not generate identical labels. The following source code generates the output code as shown:

Example 2–5. Macro Local Variables

(a) 'C2xx source code

```
    mac2   .macro arg
    lab?   .word arg
           .endm

           mac2 4
           mac2 40
           mac2 400
```

(b) Translation assistant output (with –me)

```
    mac2      .macro arg
    lab?      .word arg
              .endm

    ;         mac2 4
    lab$3$    .word 4
    ;         mac2 40
    lab$4$    .word 40
    ;         mac2 400
    lab$5$    .word 400
;   No Errors, No Warnings, No Instructions
```

The local label name is appended with $N$, where N is the number of macro invocations. Ensure that there are no other labels that could be identical to a generated macro local label.

2.7.3 Defining Labels When Invoking A Macro

If there is a label associated with a macro invocation, that label is not used after expansion and translation. This is because the label is commented out with the macro invocation. The following source code generates the output code as shown:

Example 2–6. Defining Labels

(a) 'C2xx source code:

```
   mac3   .macro
          .word 0F403h
          .endm
   label  mac3
```

(b) Translation assistant output (with –me)

```
   mac3   .macro
          .word 0F403h
          .endm
   ;label  mac3
          .word 0F403h
;   No Errors, No Warnings, No Instructions
```

Label is not defined when the code is assembled. Ensure that label definitions do not appear on the same line as the macro invocations. Rewrite the source code as shown in (c).

(c) Rewritten 'C2xx source code:

```
   mac3   .macro arg
          .word 0F403h
          .endm
   label
          mac3
```

(d) Translation assistant output (with –me)

```
   mac3   .macro arg
          .word 0F403h
          .endm
label
;         mac3
          .word 0F403h
;   No Errors, No Warnings, No Instructions
```

Chapter 3

Operational Differences

This chapter describes the operational differences between ’C2xx and ’C27xx devices. These differences are described so that you can convert instructions manually that cannot be directly translated by the translation assistant. See Appendix A for help with manual conversions.

Topic **Page**

3.1 Direct Addressing Modes

In the ’C2xx’s direct addressing mode, data memory is addressed in 128-word blocks called pages. The entire 64K-byte area of data memory is segmented into 512 data pages. The current data page is defined by the value in the 9-bit data page pointer (DP), which is found in status register ST0. The particular word on the page is identified by a 7-bit offset. Thus, a 16-bit address can be separated into a 9-bit value, which identifies the data page number, and a 7-bit value, which identifies the word on the page. For ’C2xx instructions using the direct addressing mode, the 7-bit offset specifies the location of the operand in data memory.

Data pages in the ’C27xx contain 64 words. In the ’C27xx’s DP direct addressing mode, a 6-bit value is used to identify the operand’s location on the current page. The data page is identified by the 16 most significant bits of the 22-bit address. The page value is stored in the 16-bit DP register.

The translation assistant does not directly translate from the ’C2xx’s 16-bit address to the ’C27xx’s 22-bit address. Therefore, the conversion must be done manually. You can choose bit-by-bit conversion or use two simple mathematical/logical formulas to do the conversion. There is one formula for the DP value and one for the offset:

’C27xx’s DP value = (’C2xx’s DP $\times$ 2) + ((’C2xx’s 7-bit offset & (40h)) >> 6)

’C27xx’s offset = (’C2xx’s 7-bit offset & (3Fh))

where & and >> 6 represent a bit-wise logical AND operation and a logical right shift by six bit positions, respectively.

For example, the 16-bit ’C2xx address 0CB74h is represented in binary as 1100 1011 0111 0100b. The ’C2xx’s DP register contains 1 1001 0110b (196h). The 7-bit offset is 111 0100b (74h). The address can be represented for the ’C27xx using 22 bits by inserting six leading zeros: 00 0000 1100 1011 0111 0100b. The ’C27xx’s DP register would contain 0000 0011 0010 1101b (32Dh). The 6-bit offset would be 11 0100b (34h). Alternatively, using the formulas:

’C27xx’s DP value = (196h $\times$ 2) + ((74h & (40h)) >> 6)

= 32Ch + 1h

= 32Dh

’C27xx’s offset = (74h & (3Fh))

= 34h

3.2 Indirect Addressing Modes

Table 3–1 lists the indirect addressing options and operands supported by the ’C2xx with their corresponding ’C27xx options and operands. Most ’C2xx options are directly supported by ’C27xx options. As shown in the table, the increment using the reverse carry propagation (also known as bit-reversed addressing) option is not directly supported by the ’C27xx, but can be implemented using a look-up table to store and access the bit-reversed address (see Table 3–1). The decrement using the bit-reversed addressing option is not available with the ’C27xx.

Table 3–1. Comparing Indirect Addressing Operands

Option	**’C2xx Operand**	**’C27xx Operand**
No increment or decrement	*	*
Increment by 1	*+	*++
Decrement by 1	*–	*––
Increment by index amount	*0+	*0++
Decrement by index amount	*0–	*0––
Increment by index amount using reverse carry propagation during the add	*BR0+	*+ARx[AR0] or *+XARn[AR0] (in conjunction with a look-up table)
Decrement by index amount using reverse carry propagation during the subtraction	*BR0–	Not supported

The bit-reversed addressing mode is part of the indirect addressing implemented with the auxillary registers and the associated arithmetic unit. In this mode, a value (index) contained in INDX is either added to or subtracted from the auxiliary register pointed to by the ARP. However, the carry bit is not propagated in the forward direction; instead, it is propagated in the reverse direction. The result is a change of the address location (see Table 3–2).

Table 3–2. Bit-Reversed Indices

Step	Bit Pattern	Bit-Reversed Pattern	Bit-Reversed Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Suppose you want to copy data from one data array (Data1) to a second data array (Data2), using bit-reversed addressing to determine each word's new location in Data2 (see Table 3–3).

Table 3–3. Data Move Using Bit-Reversed Addressing

Data1 Address Offset	Data1 Data	Data2 Address Offset	Data2 Data
0	4215	0	4215
1	68	1	21
2	374	2	374
3	95	3	5632
4	21	4	68
5	753	5	753
6	5632	6	95
7	14	7	14

To implement the copy with the ’C2xx instruction set, you can use the auxiliary registers to store each array’s starting address, the index value (one-half of the array size), and the counter value (one less than the array size). Because each data array is eight words long (SIZE = 8), the initial value of the counter is 7 (SIZE – 1). Arbitrarily choose AR3 to store the counter. The index value, in AR0, is 4 (one-half of SIZE). AR1 and AR2 can be set to point to the first locations of Data1 and Data2, respectively. The following ’C2xx code segment can be used to implement the copy operation:

Example 3–1. TMS320C2xx Copy Operation

```
        MAR     AR1
LOOP    LACL    *+, AR2
        SACL    *BR0+, AR3
        BANZ    LOOP, *–,AR1
```

To implement the copy using the ’C27xx instruction set, you need a bit-reversed table of indices to find each word’s new location in Data2. The table of indices for an 8-word array is shown in Table 3–2. From the table, you can see that the values in Data1 at offsets 0 and 6 from the beginning of array Data1 should be copied to offsets 0 and 3, respectively, from the beginning of array Data2. AR1, AR2, and AR3 contain the address of Data1, the address of Data2, and the counter, respectively. AR4 points to the first location in the table of bit-reversed indices.

The table of indices is generated by you and referenced directly by the code. (See Table 3–2, *Bit-Reversed Indices*, page 3-3, for information that will help you generate a table of indices.)

AR0 is still used as the index value, except now the value is obtained from the table pointed to by AR4. The following ’C27xx code segment can be used to implement the copy operation:

Example 3–2. TMS320C27xx Copy Operation

```
LOOP    MOV     AL, *AR1++
        MOV     AR0, *AR4++
        MOV     *+AR2[AR0], AL
        BANZ    LOOP, AR3––
```

The ’C27xx code in Example 3–2 copies a word from Data1 (pointed to by AR1) into the low word of the accumulator and increments AR1; then copies the index value from the bit-reversed table to AR0 and increments AR4. The index value (in AR0) is added to the base address of Data2 (given by AR2) to determine the destination for the contents of the low word of the accumulator. Finally, the contents of AR3 are tested for end-of-loop conditions and AR3 is decremented.

3.3 Program Memory Addresses

The ’C2xx’s BCND, BLPD, and CC instructions use a 16-bit operand as the program memory address (pma). Similar ’C27xx instructions (LB, PREAD, and CALL) use a 22-bit program memory address. In general, no additional steps are needed to convert the ’C2xx 16-bit address to a 22-bit address for the ’C27xx.

3.4 Accumulator Instructions

The ’C2xx’s BACC, CALA, TBLR, and TBLW instructions use the contents of the lower half of the 32-bit accumulator (ACC) as the program memory address (pma). Thus, the specified address is 16 bits. The higher half of the ACC plays no role in determining the pma. Similar ’C27xx instructions (LB, CALL, PREAD, and PWRITE) use the contents of the 22-bit XAR7 register as the pma and are affected by the higher half of the ACC: ACC(21:0) is copied to XAR7 (see Appendix A for examples). You must ensure the program memory address is correct.

For example, suppose the ACC contains 0F5AC 9344h. Execution of a ’C2xx BACC instruction would cause program execution to resume at location 09344h in program space. Replacing BACC with the suggested combination of ’C27x instructions MOV, MOVL, LB, and POPL would cause XAR7 to be loaded with 02C 9344h. Program execution would resume with the instruction at location 02C 9344h in program space. This error occurs because the least significant six bits of the high word of the ACC make up the most significant six bits of the 22-bit address stored in XAR7.

If the correct pma is in the lower 64K words of memory (a maximum of 16 bits are needed to form the address), the contents of the 22-bit XAR7 register can be modified to force the upper six bits to 0. This provides the correct 16-bit address. Example 3–3 shows one method of clearing XAR7(21:16).

Example 3–3. Clearing XAR7(21:16)

```
MOV    tempaddr, XAR7
MOV    tempaddr+1, #0
MOV    XAR7, tempaddr
```

The instructions in Example 3–3 must be included before the LB, CALL, PREAD, or PWRITE instruction that uses the contents of XAR7 as its operand. The address represented by tempaddr must be even to ensure proper alignment for 32-bit accesses.

The entire contents of XAR7 are stored to two words in memory. Because each memory word is 16 bits, 2 words are required to store the 22-bit quanity. The lower word is stored in the first memory location and the higher word is stored in the second memory location. The contents of the upper word are replaced with 0 using the MOV instruction. The contents of the 2-word area are restored to XAR7 using the MOV instruction. The end result is that the upper six bits of XAR7 contain 0, and the lower 16 bits contain the value from the lower 16 bits of the accumulator, ACC(15:0).

3.5 BANZ Instructions

If the current auxiliary register’s value is nonzero, the ’C2xx BANZ instruction causes the PC to be loaded with the address given by the pma operand. The ’C27xx BANZ tests the auxiliary register specified (as an instruction operand) for a nonzero value. If the register contains a nonzero value, the new PC value is set to the sum of the current PC and the 16-bit signed offset that is specified as an operand. The 16-bit signed offset is sign-extended to 22 bits before the addition.

You must determine how many words fall between the BANZ instruction and the instruction located at pma. This value is the 16-bit signed offset. It is positive if pma follows the location of BANZ, negative if pma precedes the location of BANZ, or 0 if pma is the address of the BANZ instruction. Table 3–4 shows additional differences in the way the ’C2xx BANZ and the ’C27xx BANZ instructions work with indirect addressing.

Table 3–4. Additional Differences Between BANZ Instructions

’C2xx	**’C27x**
Supports all the standard options	Supports only the decrement-by-1 option
The next auxiliary register can be specified by using the ARn operand in indirect addressing mode	An additional instruction (i.e., NOP *ARPn) is needed to specify the next auxiliary register
Tests the *current* auxiliary register; the ARP must be changed to point to the AR containing the value to be tested *before* the BANZ	Tests and decrements the auxiliary register specified as an operand; no need to change the ARP

The ’C2xx and ’C27xx code segments in Example 3–4 highlight differences in the ways the new execution address is determined and the auxiliary register AR1 (which is used as a counter) is tested and decremented.

The ’C2xx SACL instruction causes the lower 16 bits of the ACC to be written to the location pointed to by AR2, AR2 to be incremented, and the ARP to be changed to AR1. The ’C27xx MOV instruction writes the lower 16 bits of the ACC into the location directed by AR2 and increments AR2. (The ARP is not changed.)

The ’C2xx BANZ instruction branches to the address specified by LOOP if AR1 is nonzero, decrements AR1 by one, and changes the ARP to 2. The ’C27xx BANZ instruction branches to the address given by LOOP. The assembler calculates the value of the 16-bit offset by determining LOOP’s address relative to the address of the BANZ instruction. Because LOOP appears one word before BANZ, the offset should be –1, which is represented as a signed hexadecimal number (0FFFFh). Program execution would resume at the previous instruction (i.e., the MOV instruction at LOOP). There is no need to change the ARP to test and decrement AR1 when using the ’C27xx version of BANZ.

Example 3–4. Using BANZ Instructions

(a) ’C2xx code segment

```
      MAR    *,AR2
      LAR    AR1,#4
      LAR    AR2,#60h
LOOP  SACL   *+,AR1
      BANZ   LOOP,*–,AR2
```

(b) ’C27xx code segment

```
      NOP    *ARP2
      MOVB   AR1,#4
      MOVB   AR2,#60h
LOOP  MOV    *++, AL
      BANZ   LOOP AR1––
```

3.6 General-Purpose Input Pins

The ’C2xx has pins that you can use to supply input signals from an external device or output signals to an external device. The $\overline{\text{BIO}}$ pin is a general-purpose input pin that can be used as an alternative to an interrupt pin. This allows time-critical code to execute without being disturbed by interrupts. Code execution can be controlled based on the state of the $\overline{\text{BIO}}$ pin with the use of three instructions: BCND (conditional branch), CC (conditional call), and RETC (conditional return).

Because there is no $\overline{\text{BIO}}$ pin on the ’C27xx, you must use an alternative method for implementing the conditional branch, call, or return instruction. The ’C27xx implementation uses a maskable interrupt pin and polls the interrupt flag register (IFR) to determine whether the corresponding interrupt bit has been set (i.e., an interrupt has occurred). To test for the presence of the interrupt, the IFR is pushed onto the stack. Then, the TBIT instruction tests the specified bit in the IFR and fills the TC bit with the former bit’s value. If TC is 0, the IFR is again pushed onto the stack and the TBIT instruction is used to test the IFR bit. If TC is 1, the IFR bit is cleared by performing a bit-wise AND of *mask* with the contents of IFR, where *mask* contains a 1 in every bit position except the bit that corresponds to the interrupt pin being used to emulate the $\overline{\text{BIO}}$ pin; this bit is 0. After clearing the appropriate bit in the IFR, the branch, call, or return is executed (see Example 3–5).

In the code segment shown, the assumption is that the $\overline{\text{INT6}}$ will be used to emulate the ’C2xx’s $\overline{\text{BIO}}$ pin. The bit position of $\overline{\text{INT6}}$ in the IFR is 5. Therefore, *bit num* is 5 and *mask* is 1111 1111 1101 1111b (0FFDFh). The program memory address (i.e., 10000h) was chosen arbitrarily.

Example 3–5. TMS320C27xx $\overline{\text{BIO}}$ Pin Emulation

```
LOOP  PUSH  IFR
      TBIT  *––SP, #5
      SB    LOOP, NTC
      AND   IFR, #0FFDFh
      CALL  10000
```

3.7 General-Purpose Output Pins

As noted in the previous section, the ’C2xx has pins that you can use to supply input signals from an external device or output signals to an external device. The XF output pin can be used as a flag to signal an external device that an event within the ’C2xx device has occurred. Status register ST1 contains the most recent XF value in bit 4. XF can be set to 1 or cleared to 0 using ’C2xx SETC XF and CLRC XF instructions, respectively.

There is no XF output pin and no corresponding bit in the status register on the ’C27xx. A read-back latch (in custom logic) and the IACK instruction can be used with the ’C27x instruction to emulate the operation of the ’C2xx XF output pin. The latch should be connected to the data write data bus (DWDB). Any one of the lower 16 bits of the DWDB can be selected to emulate the ’C2xx XF pin. Executing IACK causes the immediate operand to be placed on DWDB(15:0). Thus, specifying 0h as the operand for IACK causes 0 to be placed on each of the 16 bits of the data bus. This operation is similar to the CLRC XF instruction. Writing a nonzero value can have the same effect as SETC XF (see Example 3–6).

Assume DWDB bit 4 (from the read-back latch) will be used to emulate the ’C2xx’s XF pin. The first ’C27xx IACK instruction causes each of the lower 16 bits of the DWDB to be driven high (with 1s); this is similar to the ’C2xx SETC XF instruction. The second IACK drives DWDB bit 4 low (clears XF). DWDB(15:5 and 3:0) remain high for the second IACK.

Example 3–6. Setting and Clearing an External Flag

(a) ’C2xx code segment

```
SETC   XF
CLRC   XF
```

(b) ’C27xx code segment

```
IACK   #0FFFFh
IACK   #0FFEFh
```

3.8 Status Register Bits

Though bit maps for ’C2xx status registers ST0 and ST1 differ significantly from bit maps for ’C27xx status registers ST0 and ST1, for the most part, ’C27xx status bits operate the same as like-named ’C2xx bits (see Figure 3–1, *TMS320C2xx Status Register Bits*, and Figure 3–2, *TMS320C27xx Status Register Bits*, page 3-13). Specifically, the ARP, OVM, INTM, TC, SXM, and C bits are present in ’C27xx status registers and operate the same as the ’C2xx versions. However, there are some differences between the operation of the ’C2xx’s PM (product shift mode) and OV (overflow flag) bits and the ’C27xx’s PM and V (overflow condition) bits.

The ’C2xx has two PM bits to support no shift, left shift by one bit, left shift by four bits, and right shift by six bits; the ’C27xx has three PM bits to support no shift, left shift by one bit, and right shift by up to six bits. The ’C27xx does not support the ’C2xx’s left-shift-by-four product shift mode.

The difference between the ’C2xx’s OV bit and the ’C27xx V bit is more subtle than the PM difference. The OV flag is set if the ACC overflows; the ’C27xx’s V bit is set if an overflow occurs in any register that holds the result of an operation.

Some bits that were available in ’C2xx status registers are not provided in ’C27xx status registers. Specifically, the DP (data page pointer), ARB (auxiliary register pointer buffer), CNF (DARAM configuration), and XF (XF pin status) bits from the ’C2xx have no equivalent ’C27xx status bits. The DP has been replaced by the 16-bit data page register (DP) in the ’C27xx. In the ’C2xx, the CNF bit is used to allow DARAM block B0 to be mapped to either data space (if CNF = 0) or program space (if CNF = 1). In the ’C27xx, B0 is mapped to data space and program space by default, so there is no need for the CNF bit. The ’C2xx’s XF bit is used to control the XF output pin. There is no XF output pin on the ’C27xx, but software control of an output pin can be implemented (to emulate XF) using custom logic. (See section 3.7, *General-Purpose Output Pins*, page 3-11, for more information about emulating the XF pin.)

The ’C27xx also has eight status bits that were not provided in ’C2xx status registers. Six bits compose the accumulator overflow counter (OVC) and increase the dynamic range of the ACC to 38 bits. The remaining two bits, Z (zero condition) and N (negative condition), indicate when the result of an operation is 0 or negative, respectively.

Figure 3–1. TMS320C2xx Status Register Bits

(a) Status register ST0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	ARP		OV	OVM	1	INTM					DP				

(b) Status register ST1

15–13	12	11	10	9	8–5	4	3–2	1–0
ARB	CNF	TC	SXM	C	1	XF	1	PM

Figure 3–2. TMS320C27xx Status Register Bits

(a) Status register ST0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		OVC					PM		V	N	Z	C	TC	OVM	SXM

(b) Status register ST1

15–13	12–8	7	6	5	4	3	2	1	0
ARP	Reserved	IDLESTAT	EALLOW	LOOP	SPA	VMAP	PAGE0	DBGM	INTM

3.9 Repeatable Instructions

The RPT instruction causes the CPU to repeat the next instruction (the one following the RPT) a specified number of times. Though the RPT instruction is supported in both the ’C2xx and ’C27xx instruction sets, there are some repeatable ’C2xx instructions which have no equivalent repeatable ’C27xx instruction. You must translate these ’C2xx instructions manually. Consider the code segments in Example 3–7.

Example 3–7. Repeating an Unsigned ADD Instruction

(a) ’C2xx code segment

```
LACC   #0            ; load ACC with zero
LAR    AR1, #1000h   ; point AR1 to beginning of data
LAR    AR2, #200h    ; point AR2 to result
MAR    *, AR1        ; set current AR to 1
RPT    0Fh           ; repeat next instruction 16 times
ADDS   *+            ; performed unsigned add of data
MAR    *, AR2        ; set current AR to 2
SACL   *+            ; store low word of result
SACH   *             ; store high word of result
```

(b) Translation assistant output

```
MOV    ACC,#0        ; load ACC with zero
MOV    AR1,#1000h    ; point AR1 to beginning of data
MOV    AR2,#200h     ; point AR2 to result
NOP    *ARP1         ; set current AR to 1
RPT    #0Fh ||       ; repeat next instruction 16 times
*** WARNING! line 9: [W2000]: Instruction not repeatable on
                              ’C27xx
       ADDS*+        ; performed unsigned add of data
NOP    *ARP2         ; set current AR to 2
MOV    *++,ACC       ; store low word of result
MOVH   *,ACC         ; store high word of result

; No Errors, 1 Warning, 9 Instructions
;  88.89 % instructions translated
```

(c) Manually generated ’C27xx code segment

```
       MOV    ACC, #0       ; load ACC with zero
       MOV    AR1, #1000h   ; point AR1 to beginning of data
       MOV    AR2, #200h    ; point AR2 to result
       NOP    *ARP1         ; set current AR to 1
;      RPT    0Fh           ; repeat next instruction 16times
       MOV    AR3, #0Fh     ; set AR3 (counter) to 15
;      ADDS   *+            ; performed unsigned add of data
LOOP   ADDU   ACC, *++      ; performed unsigned add of data
       BANZ   LOOP, AR3––   ; test counter for zero
       NOP    *ARP2         ; set current AR to 2
       MOV    *++,ACC       ; store low word of result
       MOVH   *,ACC         ; store high word of result
```

The code in Example 3–7(a) calculates the check sum of the contents of 16 locations, from address 1000h through address 100Fh, and stores the resulting 32-bit value in locations 0200h (the low word) and 0201h (the high word). Though the ’C2xx ADDS instruction is directly translatable to ’C27xx ADDU, ADDU is not repeatable. Part (b) shows the code and the warning message output by the translation assistant when it attempted to translate a repeated ’C2xx ADDS instruction to a repeatable ’C27xx instruction. Part (c) shows the manually generated version of the code. AR3 is used as the counter. This eliminates the need to use the ’C27xx RPT instruction. The ’C27xx BANZ instruction tests AR3 to determine whether the count has reached 0 and decrements AR3 if it is not yet 0.

In Example 3–8 (a), the ’C2xx RPT instruction is followed by the SFR instruction with an immediate operand of 3. The RPT instruction causes the CPU to execute the SFR instruction a total of four times. The contents of the accumulator instruction (ACC) is shifted to the right by four bits.

Example 3–8. Repeating a Right Shift Instruction

(a) ’C2xx code segment

```
    RPT    #3     ; repeat next instruction 4 times
    SFR           ; right–shift ACC contents 4 times
```

(b) Translation assistant output)

```
    RPT    #3 ||     ; repeat next instruction 4 times
*** WARNING! line 5: [W2000]: Instruction not repeatable on
                              ’C27xx
    SFR              ; right–shift ACC contents 4 times

; No Errors, 1 Warning, 2 Instructions
;  50.00 % instructions translated
```

(c) Manually generated ’C27xx code segment

```
;   RPT    #3        ; repeat next instruction 4 times
    SFR    ACC,4     ; right–shift ACC contents 4 times
```

The SFR instruction is supported by the ’C27xx instruction set, but it is not repeatable. When the translation assistant encounters an SFR following an RPT instruction, it generates a warning, as in Example 3–8 (b).

The manually generated ’C27xx code shown in part (c) uses SFR with operands ACC and an immediate value of 4 to eliminate the use of the RPT instruction. The number of instructions required to perform the right shift by four bits was also reduced from two (for ’C2xx) to one (for ’C27xx).

The following syntaxes of ’C27xx instructions are repeatable.

MAC	P, loc, 0:pmem
MOV	*(0:16bit), loc
MOV	loc, #0
MOV	loc, #16bit
MOV	loc, *(0:16bit)
NOP	
NOP	*ind
NORM	ACC, aux++
NORM	ACC, aux--
PREAD	loc, *XAR7
PWRITE	*XAR7, loc
ROL	ACC
ROR	ACC
SUBCU	ACC, loc

3.10 Assembler Directives

Asssembler directives supply program data and control the assembly process. To accomodate the ’C2xx and ’C27xx architecture/definition differences, the translation assistant deletes or converts some ’C2xx assembler directives. The translation assistant can translate most ’C2xx specific assembler directives; however, the ’C2xx directives that are not translated require no manual conversion and are passed through to the output file unchanged. The following table lists the ’C2xx directives that are translated by the translation assistant and the corresponding ’C27xx translation assistant output.

Table 3–5. Directive Changes

’C2xx Directives	’C27xx Directives
`.align`	`.align 64`
`.blong` value$_1$ [,...,value$_n$]	`.long` value$_1$ [,..., value$_n$]
`.bfloat` value$_1$ [,..., value$_n$]	`.float` value$_1$ [,..., value$_n$]
`.even`	`.align 1`
`.mmregs`	(deleted)
`.port`	(deleted)
`.string` string$_1$ [,...,string$_n$]	`.pstring` string$_1$ [,..., string$_n$]
`.version`	(deleted)

See the *TMS320C2x/C2xx/C5x Assembly Language Tools User’s Guide* and *TMS320C27xx Assembly Language Tools User’s Guide* for more information about assembler directives.

Appendix A

Converting TMS320C2xx Instructions

This appendix contains guidelines for converting TMS320C2xx instructions that cannot be directly translated by the translation assistant utility to a set of TMS320C27xx instructions that perform similar functions.

These guidelines are arranged alphabetically by ’C2xx instruction. The left column lists the nontranslatable ’C2xx instruction; the middle column lists the corresponding ’C27xx instructions. Notes on differences you should consider when converting ’C2xx code to ’C27xx code are provided in the rightmost column.

Table A–1. ADDT Instructions

’C2xx Instruction(s)	**’C27xx Instruction(s)**	**Comments**
ADDT dma	MOVL *SP++, ACC MOV ACC, @dma LSL ACC, T ADDL ACC, *––SP	❑ ’C2xx’s dma is a 7-bit offset from the page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from the page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes,* page 3-2.) ❑ ’C27xx’s ADDT is affected by the state of the SXM bit; LSL is not. Thus, the data may not retain the correct sign after the shift.
LACL dma1 ADDT dma2	MOV ACC, dma2 LSL ACC, T ADDL ACC, @dma	❑ ADDT affects the OV (overflow flag); ADDL affects the V and OVC bits. (See section 3.8, *Status Register Bits,* page 3-12.) ❑ ADDT is repeatable; ADDL is not. (See section 3.9, *Repeatable Instructions,* page 3-14.)
ADDT *ind[,ARn]	MOVL *SP++, ACC If no ARn specified: MOV ACC, *ind If ARn specified and *ind=*: MOV ACC, *ARPn If ARn specified and *ind !=*: MOV ACC, *ind NOP *ARPn LSL ACC, T ADDL ACC,*––SP	❑ ’C2xx’s ADDT is affected by the state of the SXM bit; LSL is not. Thus, data may not retain the correct sign after the shift. ❑ ADDT supports multiple indirect operands (inc, dec, add AR0, etc.); ’C7xx’s instructions do not support the same operand types. (See section 3.1, *Direct Addressing Modes,* page 3-2.) ❑ ADDT affects the OV bit; ADDL affects the V and OVC bits. (See section 3.8, *Status Register Bits,* page 3-12.) ❑ ADDT is repeatable; ADDL is not. (See section 3.9, *Repeatable Instructions,* page 3-14.)

Table A–2. BACC Instructions

’C2xx Instruction(s)	**’C27xx Instruction(s)**	**Comments**
BACC	MOV *SP++,XAR7 MOVL @XAR7, ACC LB *XAR7 The following instruction is to be done in destination routine: MOV XAR7, *––SP	❑ BACC branches to the 16-bit address in ACC; LB branches to the 22-bit address in XAR7. (See section 3.4, *Accumulator Instructions*, page 3-7.) ❑ The only auxiliary register available for use by the LB instruction is XAR7. Thus, using the LB instruction may make it difficult to use XAR7 as a stack pointer.

Table A–3. BANZ Instructions

’C2xx Instruction(s)	**’C27xx Instruction(s)**	**Comments**
BANZ pma[,*ind [,ARn]]	BANZ 16bitoffset, ARx–– If ARn specified: NOP *ARPn	❑ ’C2xx’s BANZ assigns an absolute address of pma to the PC; ’C27xx’s BANZ uses a signed 16-bit offset from the current PC as the operand. ❑ ’C2xx’s BANZ supports multiple indirect operands (inc, dec, add, AR0, etc.); ’C27xx’s BANZ supports decrement only (ARx––). (See section 3.2, *Indirect Addressing Modes,* page 3-3.)

Table A–4. BCND Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
BCND pma, BIO	Loop PUSH IFR TBIT *––SP, #bit num SB Loop, NTC AND IFR, #mask LB pma	❑ ’C2xx instruction tests the state of $\overline{\text{BIO}}$. There is no $\overline{\text{BIO}}$ pin on the C27xx; an interrupt line is used instead. (See section 3.6, *General-Purpose Input Pins,* page 3-10.) ❑ BCND has no effect on the TC bit; ’C27xx instructions modify and test TC bit to control flow of code execution.
Multiple conditions (all must be true): BCND pma, BIO, EQ, C	Multiple conditions (all must be true): Loop PUSH IFR TBIT *––SP, #bit num SB Loop, NTC TEST ACC SB Loop, NEQ SB Loop, NC AND IFR, #mask LB pma	❑ BCND branches to 16-bit address pma; LB uses a 22-bit address. (See section 3.3, *Program Memory Address*, page 3-6.) ❑ BCND supports a branch based on multiple conditions; ’C27xx requires additional instructions to support multiple conditions. ❑ If any branch instruction tests for EQ, NEQ, LT, LEQ, GT, and/or GEQ, the TEST ACC instruction must precede the first of the branch instructions.

Table A–5. BITT Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
BITT dma	PUSH @AL MOV AL, @dma LSL AL, T TBIT @AL, #15 POP @AL	❑ ’C2xx’s dma is a 7-bit offset from the page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from the page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes,* page 3-2.) ❑ BITT affects TC only; ’C27xx instructions affect the C, N, Z and TC bits. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ BITT is repeatable; TBIT is not. (See section 3.9, *Repeatable Instructions*, page 3-14.)
BITT *ind [,ARn]	PUSH @AL If no ARn specified: MOV AL, *ind If ARn specified and *ind=*: MOV AL, *ARPn If ARn specified and *ind!=*: MOV AL, *ind NOP *ARPn LSL AL,T TBIT @AL,#15 POP @AL	❑ BITT affects TC only; ’C27xx instructions affect the C, N, Z, and TC bits. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ BITT is repeatable; TBIT is not. (See section 3.9, *Repeatable Instructions,* page 3-14.) ❑ ’C2xx’s BITT supports multiple indirect operands (inc, dec, add, AR0, etc.); ’C27xx’s instructions do not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–6. BLPD Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
BLPD #pma, dma	MOV *SP++,XAR7 MOV XAR7, #pma PREAD @dma, *XAR7 MOV XAR7, *––SP	❑ ’C2xx’s pma is a 16-bit address; ’C27xx’s pma is a 22-bit address. (See section 3.3, *Program Memory Addresses*, page 3-6.) ❑ ’C2xx’s dma is a 7-bit offset from page number given by a 9-bit DP value; ’C27xx’s dma is a 6-bit offset from page number given by a 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.) ❑ BLPD uses pma as addressed location in program space; PREAD uses XAR7 to access location in program space.
BLPD #pma, *ind[,ARn]	MOV *SP++,XAR7 MOV XAR7, #pma If no ARn specified: PREAD @ind, *XAR7 If no ARn specified and *ind=*: PREAD *ARPn, *XAR7 If no ARn specified and *ind!=*: PREAD *ind, *XAR7 NOP *ARPn MOV XAR7, *––SP	❑ ’C2xx’s pma is a 16-bit address; ’C27xx’s pma is a 22-bit address. (See section 3.3, *Program Memory Addresses*, page 3-6.) ❑ ’C2xx’s BLPD supports multiple indirect operands (inc, dec, add, AR0, etc.); ’C27xx’s instructions do not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.) ❑ BLPD uses pma as addressed location in program space; PREAD uses XAR7 to access location in program space.

Table A–7. CALA Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
CALA	MOV *SP++,XAR7 MOVL @XAR7, ACC CALL *XAR7 The following instruction is to be done in destination routine: MOV XAR7, *––SP	❑ CALA uses ACC contents as 16-bit branch address; CALL uses the 22-bit address from XAR7. (See section 3.4, *Accumulator Instructions*, page 3-7.) ❑ The only auxiliary register available for use by the CALL instruction is XAR7. Thus, using the CALL instruction can make it difficult to use XAR7 as a stack pointer.

Table A–8. CC Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
CC pma, BIO	Loop PUSH IFR TBIT *––SP, #bit num SB Loop, NTC AND IFR, #mask CALL pma	❑ ’C2xx instruction tests the state of $\overline{\text{BIO}}$ pin. There is no $\overline{\text{BIO}}$ pin on the ’C27xx; an interrupt line is used instead. (See section 3.6, *General-Purpose Input Pins*, page 3-10.) ❑ CC has no effect on TC bit; ’C27xx instructions modify and test the TC bit to control flow of code execution. ❑ CC branches to 16-bit address pma; LB uses a 22-bit address. (See section 3.3, *Program Memory Addresses*, page 3-6.) ❑ CC supports a subroutine call based on multiple conditions; ’C27xx requires additional instructions to support multiple conditions. ❑ If the ’C2xx conditional call instruction tests for EQ, NEQ, LT, LEQ, GT and/or GEQ, the TEST ACC instruction must precede the first ’C27xx branch instruction that tests for any of these conditions.
Multiple conditions (all must be true):	Multiple conditions (all must be true):	
CC pma, BIO, EQ, C	Loop PUSH IFR TBIT *––SP, #bit num SB Loop, NTC TEST ACC SB Loop, NEQ SB Loop, NC AND IFR, #mask CALL pma	

Table A–9. CLRC, CNFD, and EINT Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
CLRC CNF or CNFD	None	❑ There is no CNF status bit on the ’C27xx. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ CNFD is equivalent to the ’C2xx CLRC CNF instruction. There is no equivalent ’C27xx instruction.
CLRC INTM or EINT	CLRC INTM	❑ The CLRC INTM instruction clears the interrupt mode bit (’C2xx) or the interrupt enable mask bit (’C27xx), globally enabling maskable interrupts. The ’C27xx version of this instruction requires two cycles to execute. Therefore, the ’C27xx instruction following CLRC is not protected from interrupts. If protection from interrupts is required and the ’C2xx instruction following the CLRC INTM is a RET, use a ’C27xx RETE instead of {CLRC INTM;RET} to return and enable interrupts simultaneously. ❑ EINT is equivalent to the ’C2xx CLRC INTM instruction.
CLRC XF	IACK #vector-value where vector-value is a 16-bit number with at least one bit cleared.	❑ The XF output pin does not exist on the ’C27xx. (See section 3.7, *General-Purpose Output Pins*, page 3-11.)

Table A–10. CMPR Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
CMPR constant If constant is: 0, test ARx = AR0 1, test ARx < AR0 2, test ARx > AR0 3, test ARx ≠ AR0	PUSH @AL MOV AL, @AR0 CMP AL, @ARx POP @AL Where ARx is the current auxiliary register.	❑ CMPR affects TC only; ’C27xx instructions effect C, N, and Z, but have no affect on TC. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ CMPR is repeatable; CMP is not repeatable. (See section 3.9, *Repeatable Instructions*, page 3-14.)

Table A–11. DMOV Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
DMOV dma	MOV *(0:dest), @dma where dest = 16 LSBs of destination address. or MOV T,@dma MOV @(dma+1),T or MOV *SP++, *(0:dma) MOV *(0:dma), *––SP	❑ ’C2xx’s dma is a 7-bit offset from page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from the page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.) ❑ DMOV is affected by the CNF bit; there is no CNF bit in the ’C27xx. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ If you use the second group of ’C27xx instructions, the addressed memory location must be between offsets 0 and 62 (inclusive) from the top of the page because the MOV combination does not work across page boundaries. The first and third groups work across page boundaries because the full 22-bit destination address is provided. The second group also corrupts the T register. ❑ The ’C2xx instruction is repeatable.
DMOV *ind [,ARn]	If *ind=*: MOV T, *++ MOV *––, T If ind=*+: MOV T, *++ MOV *, T If *ind=*–: MOV T, *++ MOV *––, T NOP *–– If ARn specified: NOP *ARPn	❑ DMOV is affected by the CNF bit; there is no CNF bit in the ’C27xx. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ ’C27xx instructions corrupt the T register. ❑ DMOV is repeatable; MOV instructions involving the T register are not. Circular addressing can be used with the ’C27xx to emulate repeated data moves. ❑ The translations given are for the most likely occurrences of DMOV with indirect addressing operands. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–12. IN Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
IN dma, PA	MOV @dma, *(0:PA)	❑ The ’C2xx IN instruction accesses I/O space; there is no I/O space in the ’C27xx, so the address is in data space. ❑ ’C2xx’s dma is a 7-bit offset from the page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from the page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.)
IN *ind, PA[,ARn]	If no ARn specified: MOV *ind, *(0:PA) If ARn specified and *ind=*: MOV *ARPn, *(0:PA) If ARn specified and *ind!=*: MOV *ind, *(0:PA) NOP *ARPn	❑ The ’C2xx IN instruction accesses I/O space; there is no I/O space in the ’C27xx, so the address is in data space. ❑ IN is repeatable only if one of the two ’C27xx instructions is repeatable. (See section 3.9, *Repeatable Instructions*, page 3-14.)

Table A–13. INTR Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
INTR K	For K = 1 to 14: INTR INTK For K = 0 to 31: TRAP #K	❑ The ’C2xx INTR instruction does not automatically clear the IFR bit; additional code is required. INTR for the ’C27xx clears the bit automatically; the TRAP instruction does not.

Table A–14. LACT Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
LACT dma	MOV ACC, @dma LSL ACC, T	❑ ’C2xx’s dma is a 7-bit offset from page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.) ❑ LACT is affected by the SXM bit; LSL is not. Thus, data may not retain the correct sign after the shift.
LACT *ind [,ARn]	If no ARn specified: MOV ACC, *ind If ARn specified and *ind=*: MOV ACC, *ARPn If ARn specified and *ind!=*: MOV ACC, *ind NOP *ARPn LSL ACC, T	❑ LACT is affected by the SXM bit; LSL is not. Thus, data may not retain the correct sign after the shift. ❑ ’C2xx’s LACT supports multiple indirect operands (inc, dec, add, AR0, etc.); ’C27xx’s instructions do not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–15. LST Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
LST #m, dma where m = 0 or 1	PUSH dma POP STm where m = 0 or 1	❑ ’C2xx’s dma is a 7-bit offset from page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.) ❑ Status register bits for ’C2xx are defined differently from status bits for ’C27xx. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ LST is repeatable; the PUSH/POP combination is not. (See section 3.9, *Repeatable Instructions*, page 3-14.) ❑ For m = 0, the ’C2xx LST instruction does not affect bits 9 (INTM) and 10 (Reserved). Bit 9 is unchanged from its original value, and bit 10 is always read as 1. The value popped off the stack with the ’C27xx POP instruction replaces all 16 bits of ST0. ❑ For m = 1, the ’C2xx LST instruction does not affect bits 2, 3, and 5–8 (Reserved) of ST1. The value popped off the stack with the ’C27xx POP instruction does not replace bits 5 (LOOP), 7 (IDLESTAT), and 8–12 (reserved) in ST1. ❑ For m = 1, the ’C2xx LST instruction loads a value into the ARB (bits 13–15) in ST1. This 3-bit value is also copied into the ARP (bits 13–15) in ST0. The ’C27xx’s ST0 and ST1 do not include the ARB. Thus, a new value for ARP can be loaded directly into the ’C27xx’s ARP (bits 13–15) in ST1.

Table A–15. LST Instructions (Continued)

’C2xx Instruction(s)	**’C27xx Instruction(s)**	**Comments**
LST #m, *ind [,ARn] where m = 0 or 1	If no ARn specified: PUSH *ind If ARn specified and *ind=*: PUSH *ARPn If ARn specified and *ind!=*: PUSH *ind NOP *ARPn POP STm where m = 0 or 1	❑ Status register bits for ’C2xx are defined differently from status bits for ’C27xx. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ LST is repeatable; the PUSH/POP combination is not. (See section 3.9, *Repeatable Instructions*, page 3-14.) ❑ For m = 0, with the ’C2xx LST instruction, the ARP (bits 13–15) gets loaded with the value accessed by the indirect addressing operand. Any specified ARn (the optional third operand in the instruction) is ignored. The ’C27xx instruction (PUSH or NOP) should have *ARPn as its operand, where n is the value of the most significant three bits of data from the addressed location. ❑ ’C2xx’s LST supports multiple indirect operands (inc, dec, add AR0, etc.); ’C27xx’s instructions do not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–16. LTD Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
LTD dma	MOVA T, @dma MOV *(0:dest), @dma where dest = 16 LSBs of destination address. or MOV T, @dma MOV @(dma+1), T or MOV *SP++, *(0:dma) MOV (0:dma), *––SP	❑ ’C2xx’s dma is a 7-bit offset from page number given by 9-bit DP value; ’C27xx’s dma is a 6-bit offset from page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page3-2.) ❑ If the second group of ’C27xx instructions is used, the addressed memory location must be between offsets 0 and 62 (inclusive) from the top of the page because the MOV combination does not work across page boundaries. The first and third groups work across page boundaries because the full 22-bit destination address is provided. The first and second groups also corrupt the T register. ❑ ’C2xx LTD instruction is repeatable; ’C27xx MOVA is not. (See section 3.9, *Repeatable Instructions*, page 3-14.)
LTD *ind [,ARn]	If*ind=*: MOVA T, *++ MOV *––, T If *ind=*+: MOVA T, *++ MOV *, T If *ind=*–: MOVA T, *++ MOV *––, T NOP *–– If ARn specified: NOP *ARPn	❑ LTD affects the OV and C bits; MOVA affects the OVC, C, N, V, and Z bits. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ ’C27xx instructions corrupt the T register. ❑ LTD is repeatable; MOVA is not. (See section 3.9, *Repeatable Instructions*, page 3-14.) ❑ The translations given are for the most likely occurrences of LTD with indirect addressing operands. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–17. MACD Instructions

’C2xx Instruction(s)	**’C27xx Instruction(s)**	**Comments**
MACD pma, dma	MAC P, @dma, 0:pma MOV *(0:dest), @dma where dest = 16 LSBs of destination address or MAC P, @dma, 0:pma MOV @(dma+1), T	❑ ’C2xx’s dma is a 7-bit offset from the page number given by the 9-bit DP value. The first group of ’C27xx instructions uses the full 22-bit destination address; the second group of ’C27xx instructions uses dma as a 6-bit offset from the page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.) ❑ If you use the second group of ’C27xx instructions, the addressed memory location must be between offsets 0 and 62 (inclusive) from the top of the page because the MOV combination does not work across page boundaries. The first group works across page boundaries because the full 22-bit destination address is provided. The first group also corrupts the T register. ❑ MACD affects C and OV; MAC affects C, N, V, Z, and OVC. ❑ ’C2xx MACD instruction is repeatable; ’C27xx combinations of instructions are not. (See section 3.9, *Repeatable Instructions*, page 3-14.)

Table A–17. MACD Instructions (Continued)

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
MACD pma, *ind [,ARn]	if *ind=*: MAC P, *++, 0:pma MOV *––, T If *ind=*+: MAC T, *++, 0:pma MOV *,T If *ind=*–: MAC T, *++, 0:pma MOV *––, T NOP *–– If *ind=*0+: MAC T, *++, 0:pma MOV *––,T NOP *0++ If *ind=*0–: MAC T,*++,0:pma MOV *––, T NOP *0–– If ARn specified: NOP *ARPn	❑ MACD affects C and OV; MAC affects C, N, V, Z, and OVC. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ ’C2xx MACD instruction is repeatable; ’C27xx combinations of instructions are not. (See section 3.9, *Repeatable Instructions*, page 3-14.) ❑ ’C2xx’s MACD supports multiple indirect operands (inc, dec, add AR0, etc.); ’C27xx’s instructions do not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–18. MPY Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
MPY #k	For k = 0 to 255: MPYB P, T, #k For k < 0 and k > 255: MOV temp, ACC MPYA P, @T, #k MOVL ACC, temp Where temp is a reference to a 32-bit data-memory location used to store ACC contents.	❑ ’C2xx MPY uses a 13-bit signed value to perform a signed multiplication. The first group of ’C27xx instructions uses an 8-bit unsigned operand; the second group uses a 16-bit operand. ❑ MPY affects no status bits. The first group of ’C27xx instruction affects no status bits; the second group affects C, V, and OVC. ❑ MPY is repeatable; neither group of ’C27xx instructions is repeatable. (See section 3.9, *Repeatable Instructions*, page 3-14.)

Table A–19. NORM Instructions

’C2xx Instruction(s)	**’C27xx Instruction(s)**	**Comments**
NORM *ind	For AR0–AR5, use either: NORM ACC, ARx++ or NORM ACC, ARx–– where x = 0, 1, 2, 3, 4, or 5 For XAR6 and XAR7, use either: NORM ACC, XARy++ or NORM ACC, XARy –– where y = 6 or 7	❑ ’C2xx NORM affects TC bit; ’C27xx NORM affects TC, N, and Z bits. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ When ’C2xx NORM is repeated, no operations are performed after normalization is complete; the ARx register can be modified after normalization is complete when repeating the ’C27xx NORM instruction. ❑ ’C2xx’s NORM supports multiple indirect operands (inc, dec, add AR0, etc.); ’C27xx’s instructions do not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–20. OUT Instructions

’C2xx Instruction(s)	**’C27xx Instruction(s)**	**Comments**
OUT dma, PA	MOV *(0:PA), @dma	❑ The ’C2xx OUT instruction accesses I/O space; there is no I/O space in the ’C27xx. MOV writes data to program space. ❑ ’C2xx’s dma is a 7-bit offset from the page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.) ❑ ’C2xx OUT causes the $\overline{\text{IS}}$ line to go low as a hardware indication that an I/O access is occurring; ’C27xx MOV has no similar function—there is no I/O space in the ’C27xx.
OUT *ind, PA [,ARn]	If ARn specified: MOV *(0:PA), *ind If ARn specified and *ind=*: MOV (0:PA), *ARPn If ARn specified and *ind!=*: MOV *(0:PA), *ind NOP *ARPn	❑ The ’C2xx OUT instruction accesses I/O space; there is no I/O space in the ’C27xx. MOV writes data to program space. ❑ ’C2xx OUT causes the $\overline{\text{IS}}$ line to go low as a hardware indication that an I/O access is occurring; ’C27xx MOV has no similar function—there is no I/O space in the ’C27xx. ❑ OUT is repeatable; the combination of ’C27xx instructions is not. (See section 3.9, *Repeatable Instructions*, page 3-14.) ❑ ’C2xx’s OUT supports multiple indirect operands (inc, dec, add, AR0, etc.); ’C27xx’s instructions do not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–21. RETC Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
RETC BIO	Loop PUSH IFR TBIT *––SP, #bit num SB Loop, NTC AND IFR, #mask RET	❑ The ’C2xx RETC instruction tests the state of $\overline{\text{BIO}}$. There is no $\overline{\text{BIO}}$ pin on the ’C27xx; an interrupt line is used instead. (See section 3.6, *General-Purpose Input Pins*, page 3-10.) ❑ RETC has no effect on the TC bit; ’C27xx instructions modify and test TC to control the flow of code execution.
Multiple conditions (all must be true): RETC pma, BIO, EQ, C	Multiple conditions (all must be true): Loop PUSH IFR TBIT *––SP, #bit num SB Loop, NTC TEST ACC SB Loop, NEQ SB Loop, NC AND IFR, #mask RET	❑ RETC supports a branch based on multiple conditions; ’C27xx requires additional instructions to support multiple conditions. If any branch instruction tests for EQ, NEQ, LT, LEQ, GT, and/or GEQ, the TEST ACC instructions must precede the first of the branch instructions.

Table A–22. RPT Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
RPT *ind[,ARn]	If ARn specified, no similiar operation; otherwise: RPT *ind	❑ The ’C2xx RPT instruction supports modifications to ARP; the ’C27xx RPT does not. ❑ ’C2xx’s RPT supports multiple indirect operands (inc, dec, add AR0, etc.); ’C27xx’s instructions do not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–23. SETC Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
SETC CNF	None	❑ There is no CNF status bit on the ’C27xx. (See section 3.8, *Status Register Bits*, page 3-12.)
SETC XF	IACK #vector_value where vector_value is a 16-bit number with at least one bit set.	❑ The XF output pin does not exist on the ’C27xx. (See section 3.7, *General-Purpose Output Pins*, page 3-11.)

Table A–24. SPM Instructions

’C2xx Instruction(s)	C27xx Instruction(s)	Comments
SPM 2 (for 4-bit left shift)	None	❑ The ’C27xx does not support 4-bit left shifts. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ To remove redundant sign bits when multiplying a 16-bit number and a 13-bit number, the ’C27xx’s MPYA instruction can be used in the following way: MOVL temp, ACC MPYA P, @T, #(13-bit-operand<<4) MOVL ACC, temp

Table A–25. SST Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
SST #m, dma	PUSH STm POP dma	❑ ’C2xx’s dma is a 7-bit offset from the page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from the page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.) ❑ Status register bits for ’C2xx are defined differently from status bits for ’C27xx. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ SST is repeatable; the PUSH/POP combination is not. (See section 3.9, *Repeatable Instructions*, page 3-14.)
SST #m, *ind[,ARn]	PUSH STm If no ARn specified: POP *ind If ARn specified and *ind=*: POP *ARPn If ARn specified and *ind!=*: POP *ind NOP *ARPn	❑ Status register bits for ’C2xx are defined differently from status bits for ’C27xx. (See section 3.8, *Status Register Bits*, page 3-12.) ❑ LST is repeatable; the PUSH/POP combination is not. (See section 3.9, *Repeatable Instructions*, page 3-14.) ❑ ’C2xx’s SST supports multiple indirect operands (inc, dec, add AR0, etc.); the ’C27xx instructions do not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–26. SUBT Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
SUBT dma	MOVL *SP++, ACC MOV ACC, @dma LSL ACC, T SUBL ACC, *––SP NEG ACC	❑ ’C2xx’s dma is a 7-bit offset from the page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from the page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.) ❑ SUBT is affected by SXM bit; the ’C27xx combination is not. ❑ SUBT is repeatable; ’C27xx combination is not. (See section 3.9, *Repeatable Instructions*, page 3-14.)
SUBT *ind[,ARn]	MOVL *SP++, ACC If no ARn specified: MOV ACC, *ind If ARn specified and *ind=*: MOV ACC, *ARPn If ARn specified and *ind!=*: MOV ACC, *ind NOP *ARPn LSL ACC, T SUBL ACC, *––SP NEG ACC	❑ SUBT is affected by SXM bit; the ’C27xx combination is not. ❑ SUBT is repeatable; The ’C27xx combination is not. (See section 3.9, *Repeatable Instructions*, page 3-14.) ❑ ’C2xx’s SUBT supports multiple indirect operands (inc, dec, add AR0, etc.); ’C27xx’s instructions do not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–27. TBLR Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
TBLR dma	MOV *SP++,XAR7 MOVL @XAR7, ACC PREAD @dma, *XAR7 MOV XAR7,*––SP	❑ ’C2xx’s dma is a 7-bit offset from the page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from the page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.) ❑ TBLR uses a 16-bit address from AL (the lower half of ACC); PREAD uses a 22-bit address from XAR7. (See section 3.4, *Accumulator Instructions*, page 3-7.)
TBLR *ind[,ARn]	MOV *SP++,XAR7 MOVL @XAR7, ACC If no ARn specified: PREAD *ind, *XAR7 If ARn specified and *ind=*: PREAD *ARPn, *XAR7 If ARn specified and *ind!=*: PREAD *ind, *XAR7 NOP *ARPn MOV XAR7,*––SP	❑ TBLR uses a 16-bit address from AL (the lower half of ACC); PREAD uses a 22-bit address from XAR7. (See section 3.4, *Accumulator Instructions*, page 3-7.) ❑ ’C2xx’s TBLR supports multiple indirect operands (inc, dec, add AR0, etc.); ’C27xx’s PREAD does not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–28. TBLW Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
TBLW dma	MOV *SP++,XAR7 MOVL @XAR7, ACC PWRITE *XAR7, @dma MOV XAR7,*––SP	❑ ’C2xx’s dma is a 7-bit offset from the page number given by the 9-bit DP value; ’C27xx’s dma is a 6-bit offset from the page number given by the 16-bit DP register. (See section 3.1, *Direct Addressing Modes*, page 3-2.) ❑ TBLW uses a 16-bit address from AL (the lower half of ACC); PWRITE uses a 22-bit address from XAR7. (See section 3.4, *Accumulator Instructions*, page 3-7.)
TBLW *ind[,ARn]	MOV *SP++,XAR7 MOVL @XAR7, ACC If no ARn specified: PWRITE *XAR7, *ind If ARn specified and *ind=*: PWRITE *XAR7, *ARPn If ARn specified and *ind!=*: PWRITE *XAR7, *ind NOP *ARPn MOV XAR7,*––SP	❑ TBLW uses a 16-bit address from AL (the lower half of ACC); PWRITE uses a 22-bit address from XAR7. (See section 3.4, *Accumulator Instructions*, page 3-7.) ❑ ’C2xx’s TBLW supports multiple indirect operands (inc, dec, add AR0, etc.); ’C27xx’s PWRITE does not support the same operand types. (See section 3.2, *Indirect Addressing Modes*, page 3-3.)

Table A–29. TRAP Instructions

’C2xx Instruction(s)	’C27xx Instruction(s)	Comments
TRAP	TRAP #VectorNumber where VectorNumber = 0 to 31	❑ The ’C2xx TRAP instruction always vectors to location 022h; the ’C27xx TRAP instruction can vector to any one of 32 locations as defined by VectorNumber. ❑ The ’C2xx TRAP instruction does not disable maskable interrupts; the ’C27xx TRAP instruction disables maskable interrupts.

Appendix B

Conversion Examples

This appendix provides programming examples for ’C2xx instructions that cannot be directly translated by the translation assistant utility. The left column of the tables shows the ’C2xx operation including the nontranslatable ’C2xx instruction. The right column shows the equivalent ’C27xx operation.

Topic **Page**

B.1 Group 1 (BCND BIO, CC BIO, and RETC BIO Instructions)

These ’C2xx instructions are similar in that the state of the $\overline{\text{BIO}}$ pin determines whether the branch, call, or return instruction is executed. Because the ’C27xx does not have a $\overline{\text{BIO}}$ pin, you must have an alternative method for implementing the conditional branch, call, or return instruction.

The equivalent ’C27xx operation uses a maskable interrupt pin and polls the interrupt flag register (IFR). The TBIT instruction tests the IFR bit that corresponds to the interrupt and fills the TC bit with the value of that IFR bit. If TC = 1, the interrupt has been detected, and the branch, call, or return instruction is executed. The following table shows an example of the BCND ’C2xx operation and its equivalent ’C27xx operation.

Table B–1. BCND Equivalent Operation

’C2xx Operation	’C27xx Equivalent Operation
`BCND   pma,BIO`	`PUSH   IFR`
	`TBIT   *––SP,#13`
	`SB     –2,NTC`
	`AND    IFR,#0xdfff`
	`LB     pma`

B.2 Group 2 (SETC XF and CLRC XF Instructions)

These ’C2xx instructions are common in that the ’C27xx does not have an XF or CNF bit, so the signal is emulated using IACK and a latch.

The IACK #16bit instruction behaves like a normal write operation. The 16-bit immediate value is available on the data bus lines DWDB(15:0), and simultaneously the IACK strobe is driven low. The IACK signal can be used to latch DWDB15. The emulated XF signal follows writes of 1 or 0 to DWDB15. Table B–2 shows an example of the SETC XF ’C2xx instruction and its equivalent ’C27xx operation.

Table B–2. SETC XF Equivalent Operations

’C2xx Operation	’C27xx Equivalent Operation
`CLRC SXM`	`CLRC SXM`
`CLRC OVM`	`CLRC OVM`
`SETC XF`	`IACK #8000H`

B.3 Group 3 (LST #0/#1 and SST #0/#1 Instructions)

The LST and SST instructions are grouped together because they both operate on the status registers (ST0 and ST1). Equivalent operations on the ’C27xx status registers can be performed using the PUSH and POP instructions. Table B–3 shows an example of the ’C2xx SST #0/#1 operation and its equivalent ’C27xx operation.

Table B–3. SST #0/#1 Equivalent Operations

’C2xx Operation	’C27xx Equivalent Operation
`DINT`	`SETC   INTM`
`SST    #1,*+`	`PUSH   ST1`
`SST    #0,*+`	`POP    *++`
`EINT`	`PUSH   ST0`
	`POP    *++`
	`CLRC   INTM`

B.4 Group 4 (BACC and CALA Instructions)

These ’C2xx instructions get a 16-bit destination address from the low word of ACC. The ’C27xx LB (or CALL) instruction gets a 22-bit destination address from the extended auxiliary register XAR7. Table B–4 shows an example of the BACC ’C2xx operation and its equivalent ’C27xx operation.

Table B–4. BACC Equivalent Operations

’C2xx Operation	’C27xx Equivalent Operation
`ADD    #TABLE`	`ADD    ACC,#TABLE`
`TBLR   TEMP`	`MOVL   @XAR7,ACC`
`LACL   TEMP`	`PREAD  @TEMP,*XAR7`
`BACC`	`MOV    AL,#TEMP`
	`MOVL   @XAR7,ACC`
	`LB     *XAR7`

B.5 Group 5 (BITT, LACT, ADDT, and SUBT Instructions)

These instructions all have a T register shift. Their similar ’C27xx operation is shown in Table B–5, using LACT as an example.

Table B–5. LACT Equivalent Operations

’C2xx Operation	’C27xx Equivalent Operation
`MAR    *,AR0`	`NOP *ARP0`
`SPLK   #1111H,*`	`MOV *1111H`
`LT     *,AR1`	`MOV T,*ARP1`
`LDP    #4H`	`MOV ACC, @DMA`
`LACT   DMA`	`LSL ACC, T`
`SACL   *`	`MOV*, ACC`

B.6 Group 6 (DMOV, LTD, and MACD Instructions)

These instructions all use T as a temporary storage register; however, you must ensure the locations referenced by dma and (dma+1) are on the same data page. dma should be a value from 0 to 62. Table B–6 shows an example of the DMOV ’C2xx operation and its equivalent ’C27xx operation.

Table B–6. DMOV Equivalent Operations

’C2xx Operation		’C27xx Equivalent Operation	
`BLDD`	`*,#B1DATA,AR2`	`MOV`	`*(0:B1DATA),*ARP2`
`DMOV`	`*,AR3`	`MOV`	`T,*++`
`BLDD`	`#(B1DATA+1),*`	`MOV`	`*––,T`
`LACT`	`*,AR4`	`MOV`	`*,*(0:(B1DATA+1))`
`SUB`	`#4444H`	`MOV`	`ACC,*`
`BCND`	`erdata4,NEQ`	`NOP`	`ARP4`
		`LSL`	`ACC,T`
		`SUB`	`ACC,#4444H`
		`TEST`	`ACC`
		`B`	`erdata4,NEQ`

B.7 Group 7 (IN and OUT Instructions)

In the ’C2xx, the programming address (PA) is an address in I/O space. The ’C27xx does not contain I/O space; on the ’C27xx, the PA is the 16 LSBs of a data memory address. Table B–7 shows an example of the OUT ’C2xx operation and its equivalent ’C27xx operation.

Table B–7. OUT Equivalent Operations

’C2xx Operation		’C27xx Equivalent Operation	
SPM	0	SPM	#0
LAR	AR1,#0300h	MOV	SP,#0x0300
MAR	*,AR1	NOP	*ARP1
SPLK	#0001h,*	MOV	*,#0x0001
OUT	*,ffffh	MOV	*(0:0xffff),*

B.8 Group 8 (TBLR and TBLW Instructions)

The TBLR ’C2xx instruction gets a 16-bit program memory address from the low word of ACC. The ’C27xx PREAD/PWRITE instruction gets a 22-bit program memory address from the extended auxiliary register XAR7. Table B–8 shows an example of the TBLR ’C2xx operation and its equivalent ’C27xx operation.

Table B–8. TBLR Equivalent Operations

’C2xx Operation	’C27xx Equivalent Operation
`LACC #202`	`MOV ACC,#202`
`LACC AR2,#100`	`MOV AR2,#100`
`TBLR TMP15`	`MOVL . @XAR7,ACC`
`ADD #1`	`PREAD @TMP15,*XAR7`
`SACL *+`	`ADD ACC,#1`
	`MOV *++,ACC`

Appendix C

Translation Assistant Error Messages

When the translation assistant completes its second pass, it reports any errors that it encountered during the translation. It also prints these errors in the listing file (if one is created); an error is printed following the source line that incurred it. You should attempt to correct the first error that occurs in your code first; a single error condition can cause a cascade of spurious errors.

If you have received a translation assistant error message, use this appendix to find possible solutions to the problem that you encountered. First, locate the error message class number. (The class numbers are listed in numerical order in this appendix.) Then, locate the error message that you encountered within that class. (Each class number has an alphabetical list of error messages that are associated with it.) Each class has a *Description* of the problem and an *Action* that suggests possible remedies. Many of the errors involve incorrect use of, or syntax for, an instruction or a directive. See the *TMS320C27xx Assembly Language Tools User's Guide* or the *TMS320C27xx Optimizing C Compiler User's Guide*, as appropriate.

E0000

Comma required to separate arguments
Comma required to separate parameters
Left parenthesis expected
Left parenthesis is missing
Matching right parenthesis is missing
Missing matching right bracket for condition
Missing right quote of string constant
No matching right parenthesis
Right parenthesis expected
Syntax error
Unrecognized character type
Unrecognized special character

Description	These are errors about general syntax. The required syntax is not present.
Action	Correct the source per the error message text.

E0002

Illegal mnemonic specified
Invalid mnemonic specification

Description	These are errors about invalid mnemonics. The specified instruction, macro, or directive was not recognized.
Action	Check the directive or instruction used, then correct the source.

E0003

Cluttered string constant operand encountered
Constant out of range
Illegal conditional operand
Illegal memaddr specification
Illegal register for conditional
Illegal register pair specification
Invalid binary constant specified
Invalid constant specification
Invalid decimal constant specified
Invalid float constant specified
Invalid hex constant specified
Invalid octal constant specified
Memory operand missing offset amount

Description	These are errors about invalid operands. The instruction, parameter, or other operand specified was not recognized.
Action	Correct the source per the error message text.

E0004

Absolute, well-defined integer value expected
Cannot use A side register for dest
Conditional not allowed
Identifier expected
Identifier operand expected
IFR illegal as destination register
IN illegal as destination register
Illegal character argument specified
Illegal offset mode for 15-bit const
Illegal operand
Illegal register for branch
Illegal string constant operand specified
Illegal structure reference
Instruction cannot use control register
Invalid data size for relocation
Invalid float constant specified
Invalid identifier, *%s*, specified
Invalid macro parameter specified
Invalid operand, *%c*
Must have one control register
No parameters available for macro arguments
Operand must be register indirect
PC illegal as destination register
Register expected
Single character operand expected
String constant or substitution symbol expected
String operand expected
Structure/Union tag symbol expected
Substitution symbol operand expected

Description These are errors about illegal operands. The instruction, parameter or other operand specified was not legal for this syntax.

Action Correct the source per the error message text.

E0005

field value operand
Missing operand
Missing operand(s)
Operand missing

Description These are errors about missing operands; a required operand is not supplied.

Action Correct the source so that all required operands are declared.

E0006

.break must occur within a loop
Conditional assembly mismatch
Matching .endloop missing
No matching .endif specified
No matching .endloop specified
No matching .if specified
No matching .loop specified
Open block(s) inside macro
Unmatched .endloop directive
Unmatched .if directive

Description	These are errors about unmatched conditional assembly directives. A directive was encountered that requires a matching directive, but the assembler could not find the matching directive.
Action	Correct the source per the error message text.

E0007

Conditional nesting is too deep
Loop count out of range

Description	These are errors about conditional assembly loops. Conditional block nesting cannot exceed 32 levels.
Action	Correct the .macro/.endmacro, .if/.elseif/.else/.endif, or .loop/.break/.endloop source.

E0008

Bad use of .access directive
Matching .struct directive is not present
Matching .union directive is not present

Description	This is an error about unmatched structure definition directives. In a .struct/.endstruct sequence, a directive was encountered that requires a matching directive, but the assembler could not find the matching directive.
Action	Check the source for mismatched structure definition directives and correct.

E0009

B14 or B15 required as long displacement base register
Base address register expected
Base register and index register must be from same file
Base register expected
Can't use relocatable expression in scaled addressing mode
Cannot apply bitwise NOT to floats
Cannot use register offset in unscaled addressing mode
Constant out of range
Illegal struct/union reference dot operator
Matching right bracket is missing
Missing structure/union member or tag
Structure or union tag symbol expected
Structure or union tag symbol not found
Unary operator must be applied to a constant

Description	These are errors about an illegally used operator. The operator specified was not legal for the given operands.
Action	Correct the source per the error message text so that all required operands are declared.

E0100

.setsym requires a label
Label missing
Label required

Description	These are errors about required labels. The given directive requires a label, but none is specified.
Action	Correct the source by specifying the required label.

E0101

Stand-alone labels not permitted in structure/union defs

Description	This is an error about an invalid labels. Structure and union definitions do not permit a label, but one is specified.
Action	Remove the invalid label.

E0102

Local label *%d* defined differently in each pass
Local label *%d* is multiply defined
Local label *%d* is not defined in this section
Local labels can't be used with directives

Description	These are errors about the illegal use of local labels.
Action	Correct the source per the error message text. Use .newblock to reuse local labels.

E0200

Bad term in expression
Binary operator can't be applied
Difference between segment symbols not permitted
Divide by zero
Operation cannot be performed on given operands
Unary operator can't be applied
Well-defined expression required

Description	These are errors about general expressions. An illegal operand combination was used, or an arithmetic type is required but not present.
Action	Correct the source per the error message text.

E0201

Absolute operands required for FP operations!
Floating-point divide by zero
Floating-point expression required
Floating-point overflow
Floating-point underflow
Illegal floating-point expression
Invalid floating-point operation

Description	These are errors about floating-point expressions. A floating-point expression was used where an integer expression is required, an integer expression was used where a floating-point expression is required, or a floating-point value is invalid.
Action	Correct the source per the error message text.

E0300

%s **is not defined in this source file**
%s **is operand to both .ref and .def**
Can’t tag an undefined symbol
Cannot equate an external symbol to an external symbol
Cannot redefine this section name
Empty structure or union definition
Illegal structure or union tag
Missing closing ’}’ for repeat block
Redefinition of ***%s*** **attempted**
Structure tag can’t be global
Structure/union member, ***%s*****, not found**
Symbol ***%s*** **has already been defined**
Symbol can’t be defined in terms of itself
Symbol expected in label field
Symbol expected
Symbol, ***%s*****, has already been defined**
The following symbols are undefined:
Union member previously defined
Union tag can’t be global

Description	These are errors about general symbols. An attempt was made to redefine a symbol or to define a symbol illegally.
Action	Correct the source per the error message text.

E0301

Cannot redefine local substitution symbol
Substitution stack overflow
Substitution symbol not found

Description	These are errors about general substitution symbols. An attempt was made to redefine a symbol or to define a symbol illegally.
Action	Correct the source per the error message text. Make sure that the operand of a substitution symbol is defined either as a macro parameter or with a .asg or .eval directive.

E0400

Symbol table entry is not balanced

Description	A symbolic debugging directive does not have a complementing directive (for example, a .block without a .endblock).
Action	Check the source for mismatched conditional assembly directives and correct.

E0500

Macro argument string is too long
Missing macro name
Too many variables declared in macro

Description These are errors about general macros.

Action Correct the source per the error message text.

E0501

.mexit directive outside macro definition
Macro definition not terminated with .endm
Matching .endm missing
Matching .macro missing
No active macro definition

Description These are errors about macro definition directives. A macro directive does not have a complementing directive (that is, a .macro is used without a .endm).

Action Correct the source per the error message text.

E0600

***%s* is not in archive format**
***%s* macro library not found**
Bad archive entry for *%s*
Bad archive name
Can't read a line from archive entry
Macro library is not in archive format

Description These are errors about accessing a macro library. A problem was encountered reading from or writing to a macro library archive file. It is likely that the creation of the archive file was not done properly.

Action Make sure that the macro libraries are unassembled assembler source files. Also make sure that the macro name and member name are the same and that the extension of the file is .asm.

E0700

.sym not allowed inside structure/union
Cannot use –g on assembly code with .line directives
Illegal structure/union member
No structure/union currently open

Description These are errors about the illegal use of symbolic debugging directives; a symbolic debugging directive is not used in an appropriate place.

Action Correct the source per the error message text.

E0800

A/B register file mismatch
Cannot perform operation on specified unit
Could not find a valid unit for instruction
Erroneous use of X unit
Illegal destination
Illegal form for LDDW
Illegal functional unit
Illegal memory operand register for unit
Illegal operand combination
Illegal suffix specified for branch
Illegal use of parallel operator
Instruction cannot use X unit
Instructions not permitted in structure/union definitions
Offset too large
Unit specifier disagrees with operation

Description	These are errors about illegal operands. The instruction, parameter or other operand specified was not legal for this syntax.
Action	Correct the source per the error message text.

E0801

Processor resource allocation conflict
Too many branches to labels in this packet
Too many multicycle NOPs in this packet
Too many reads from one register in this packet

Description	
Action	Correct the source per the error message text.

E0900

.var allowed only within macro definitions
Can’t include a file inside a loop or macro
Cannot change version after 1st instruction
Illegal structure definition contents
Illegal structure member
Illegal union definition contents
Illegal union member
Invalid load-time label
Invalid structure/union contents

Description	These are errors about illegally used directives. Specific directives were encountered where they are not permitted. (The directives are not permitted in that position because they will cause a corruption of the object file.) Many directives are not permitted inside of structure or union definitions.
Action	Correct the source per the error message text.

E1000

Include/Copy file not found or opened

Description	The specified filename cannot be found.
Action	Check spelling, pathname, environment variables, etc.

E1300

Copy limit has been reached
Exceeded limit for macro arguments
Macro nesting limit exceeded

Description	These errors are about general assembler limits that have been exceeded. The nesting of .copy/.include files in limited to 10 levels. Macro arguments are limited to 32 parameters. Macro nesting is limited to 32 levels.
Action	Check the source to determine how limits have been exceeded.

E2000

Illegal indirect memaddr specification
Not expecting immediate value – operand %d
Not expecting indirect operand – operand %d
Section %s is not an initialized section
Section *%s*, is not defined
Unrecognized ’C2xx instruction/directive

Description	These errors are about ’C2xx instructions.
Action	Check the source to determine what caused the problem, and correct the source.

E9999

***%s* defined differently in each pass**

Description	A symbol in the symbol table did not have the same value in pass1 and pass2. You likely have an error in a directive, macro, or label.
Action	Check the source to determine what caused the problem and correct the source.

E9999

Can’t push *%s* on expr stack
Pass conflict

Description	These are internal assembler errors. If they occur repeatedly, the assembler may be corrupt or confused.
Action	Try to assemble a smaller file. If a smaller file does not assemble, reinstall the assembler.

W0000

Delay slot count must be 1 to 9, 1 assumed
Half-word offsets must be divisible by 2, truncated
Invalid page number specified – ignored
No operands expected. Operands ignored
Specified alignment is outside accessible memory – ignored
Too many operands
Trailing operands Ignored
Word offsets must be divisible by 4, truncated

Description	These are warnings about operands. The assembler encountered operands that it did not expect.
Action	Check the source to determine what caused the problem and whether you need to correct the source.

W0001

Field value truncated to *%ld*
Field width truncated to *%d*
Maximum alignment is to 32K boundary—alignment ignored
Power of 2 required, *%ld* assumed
Section Name is limited to 8 characters
Section name, *%s*, truncated to 8 characters
String is too long—will be truncated
Value truncated to *%d*-bit width
Value truncated to byte size
Value truncated

Description	These are warnings about truncated values. The expression given was too large to fit within the instruction opcode or the required number of bits.
Action	Check the source to make sure the result is acceptable or change the source if an error has occurred.

W0002

Address expression will wrap around
Expression will overflow, value truncated

Description	These are warnings about arithmetic expressions. The assembler has done a calculation that will produce the indicated result, which may or may not be acceptable.
Action	Verify that the result is acceptable or change the source if an error has occurred.

W0003

.sym for *function name* required before .func

Description	This is a warning about problems with symbolic debugging directives. A .sym directive defining the function does not appear before the .func directive.
Action	Correct the source per the error message text.

W0004

Access only allowed in topmost structure definition
Access point has already been defined
Illegal unit specifier, ignored
Open block(s) at EOF

Description These are warnings about problems with structure definitions.

Action Correct the source per the error message text.

W1000

TMS320C27xx does not support XF
Cannot translate reference to memory-mapped register
Cannot translate absolute memory reference
Conversion requires AR0 and ARx
Conversion requires the use of the T register
Conversion uses XR7 rather than ACC
Conversion requires use of *XR7 and PREAD/PWRITE
Conversion requires the use of the P register
Conversion requires the use of the T and P registers
CLRC INTM does protect the next instruction from interrupts
Instruction not needed on TMS320C27xx
Instruction not repeatable on TMS320C27xx
Instruction is not directly translatable
IO space is removed on TMS320C27xx
Most conversions will only work if the dma offset is 0 to 62
No analogous TMS320C27xx control bit
NORM in TMS320C27xx does not work on indirect operands
Open branch delay slot at end of section *%s*
RPT instruction with next ARP not translatable
SPM operand value of 2 untranslatable
The BIO line has been removed in TMS320C27xx
There is no direct translation
Translation varies according to the value of the operands

Description These are warnings about the translation assistant.

Action Check the help file, and correct the source..

W9999

Open branch delay slot at end of section *%s*

Description This is a warning about problems with branch definitions.

Action Correct the source to remove the open branch delay slot.

Appendix D

Glossary

A

absolute address: An address that is permanently assigned to a TMS320C27xx memory location.

absolute lister: A debugging tool that accepts linked files as input and creates .abs files as output. These .abs files can be assembled to produce a listing that shows the absolute addresses of object code.

allocation: A process in which the linker calculates the final memory addresses of output sections.

archive library: A collection of individual files that have been grouped into a single file.

archiver: A software program that allows you to collect several individual files into a single file called an archive library. The archiver also allows you to delete, extract, or replace members of the archive library, as well as to add new members.

ASCII: American Standard Code for Information Interchange. A standard computer code for representing and exchanging alphanumeric information.

assembler: A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro directives. The assembler substitutes absolute operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

auxiliary register: The 'C2xx CPU provides eight 16–bit memory mapped registers (AR0–AR7) that are used for indirect data address pointers, temporary storage, or integer arithmetic processing through the auxiliary register arithmetic unit (ARAU). Each AR in 'C2xx is selected by the auxillary register pointer (ARP). The 'C27xx CPU also provides eight auxiliary registers that can be used as pointers to memory. There are six 16–bit auxiliary registers: AR0, AR1, AR2, AR3, AR4, and AR5; and two 22–bit extended auxiliary registers: XAR6 and XAR7.

B

bit-reversed addressing: Addressing in which bits of an address are reversed in order to speed the processing of algorithms, such as Fourier transform algorithms.

byte: A sequence of eight adjacent bits operated upon as a unit.

browser:A type of software that allows you to navigate information databases.

C

C compiler: A program that translates C source statements into assembly language source statements.

comment: A source statement (or portion of a source statement) that is used to document or improve readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

common object file format (COFF): A binary object file format that promotes modular programming by supporting the concept of *sections*. All COFF sections are independently relocatable in memory space; you can place any section into any allocated block of target memory.

configured memory: Memory that the linker has specified for allocation.

constant: A numeric value that does not change and that can be used as an operand.

cross-reference listing: An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

D

directives: Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).

E

emulator: A hardware development system that emulates TMS320C27xx operation.

environment variable: 1) A special system symbol that the debugger uses for finding directories or obtaining debugger options. 2) A system symbol that can be used to modify command-line input for the assembler or linker, or to modify the environment. 3) System symbols that you define and assign to a string. They are often included in batch files, for example, .cshrc.

executable module: An object file that has been linked and can be executed in a TMS320C27xx system.

expression: One or more operations in assembler programming represented by a combination of symbols, constants, and paired parentheses separated by arithmetic operators.

G

general-purpose input pins: The ’C2xx provides pins that can be used to supply input signals from an external device or output signals to an external device. The BIO pin is a general purpose input pin that can be used as an alternative to an interrupt pin.

general-purpose output pins: The ’C2xx provides pins that can be used to supply input signals from an external device or output signals to an external device. The XF pin is a general purpose output pin that can be used as a flag to signal an external device.

global symbol: A kind of symbol that is either 1) defined in the current module and accessed in another, or 2) accessed in the current module but defined in another.

H

hex-conversion utility: A program which accepts COFF files and converts them into one of several standard ASCII hexadecimal formats suitable for loading into an EPROM programmer.

high-level language debugging: The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

HTML: An abbreviation for hypertext markup language, HTML is the language used to tag various parts of a web document so browsing software can display that document’s links, text, graphics, and attached media.

L

label: A symbol that begins in column 1 of a source statement and corresponds to the address of that statement.

library-build utility: The compiler package has a utility that creates a run-time support library and installs standard header files for any configuration of compiler options that you choose.

linker: A software tool that combines object files to form an object module that can be allocated into TMS320C27xx system memory and executed by the device.

loader: A device that loads an executable module into TMS320C27xx system memory.

M

macro: A user-defined routine that can be used as an instruction.

macro call: The process of invoking a macro.

macro definition: A block of source statements that define the name and the code that make up a macro.

macro expansion: The source statements that are substituted for the macro call and are subsequently assembled.

macro invocation: The process of invoking a macro; it is known as a macro call.

macro library: An archive library composed of macros. Each file in the library must contain one macro; its name must be the same as the macro name it defines, and it must have an extension of .asm.

map file: An output file, created by the linker, that shows the memory configuration, section composition, and section allocation, as well as symbols and the addresses at which they were defined.

member: The elements or variables of a structure, union, archive, or enumeration.

memory map: A map of target system memory space, which is partitioned into functional blocks.

mnemonic: An instruction name that the assembler translates into machine code.

model statement: Instructions or assembler directives in a macro definition that are assembled each time a macro is invoked.

O

object file: A file that has been assembled or linked and contains machine-language object code.

object library: An archive library made up of individual object files.

operands: The arguments, or parameters, of an assembly language instruction, assembler directive, or macro directive.

options: Command parameters that allow you to request additional or specific functions when you invoke a software tool.

output module: A linked, executable object file that can be downloaded and executed on a target system.

R

reverse carry propagation: See bit reversed addressing.

S

section: A relocatable block of code or data that will ultimately occupy contiguous space in the TMS320C27xx memory map.

simulator: A software development system that simulates TMS320C27xx operation.

source file: A file that contains C code or assembly language code that will be compiled or assembled to form an object file.

symbol: A string of alphanumeric characters that represents an address or a value.

symbolic debugging: The ability of a software tool to retain symbolic information so that it can be used by a debugging tool, such as a simulator or an emulator.

T

target memory: Physical memory in a TMS320C27xx system into which executable object code is loaded.

translation assistant: The TMS320C27xx translation assistant converts assembly code written in the ’C2xx instruction set to code written in the ’C27xx instruction set.

W

well-defined expression: A term or group of terms that contains only symbols or assembly-time constants that have been defined before they appear in the expression.

word: A 16-bit addressable location in target memory.

Index