Manual Update Sheet

DATE: June 1, 1998

Document Being Updated: ***TMS320C3x/C4x Optimizing C Compiler User's Guide***

Literature Number Being Updated: SPRU034G

Manual Included in a Kit: Yes

This Manual Update Sheet (SPRZ130) ships with the *TMS320C3x/C4x Optimizing C Compiler User's Guide*.

Updates within paragraphs appear in a **bold typeface**.

Page: **Change or Add:**

2–6 Add one row (-msrev*x*) s to the General Shell Option section of Table 2–1, *Compiler Options Summary Table*.

2–6 Add note (see also -msrev) to the -v*xx* option

General Shell Options	Option	Effect
These options control the overall operation of the cl30 shell. For more information, see page 2-11.	–@*filename*	causes the compiler shell to read options from the specified file
	–c	no linking (negates –z)
	–d*name*[=def]	predefine a constant
	–g	enable symbolic debugging
	–i*dir*	define #include search path
	–k	keep .asm file
	–msrev*x*	specifies the silicon revision of the target chip to enable additional functionality for more recent silicon releases. Used in conjunction with –v flag.
		Example: –msrev6 –v31 : enable new (1996) 'C31 parallel instructions –msrev2 – v32 : enable new (1996) 'C32 parallel instructions
	–n	compile only (create .asm file)
	–q	suppress progress messages (quiet)
	–qq	suppress all messages (super quiet)
	–s	interlist optimizer comments and assembly source statements if available; otherwise interlist C and assembly source statements
	–ss	interlist C and assembly source statements
	–u*name*	undefine a constant
	–v*xx*	specify processor; *xx* = 30, 31, 32, 40, or 44 (default is –v30) (see also –msrev)
	–z	enable linking (all options following are passed to linker)

Page:	Change or Add:
2–62	Specify silicon revision (see -msrev in shell)

3.1. Keywords

Page:	Change or Add:
3–1	Add Section 3.3, Keywords, to the mini table of contents.
3–7	Add Section 3.3, Keywords, with subsection cregister Keyword and subsection near and far keywords.

3.1.1 The cregister Keyword

The TMS320C3x/C4x compiler extends the C language by adding the cregister keyword to allow high level language access to control registers.

When you use the cregister keyword on an object, the compiler compares the name of the object to a list of standard control registers for the ’C3x/C4x (see Table 3–2). If the name matches, the compiler generates the code to reference the control register. If the name does not match, the compiler issues an error.

Table 3–2. Valid Control Registers

Register	Description
IE	CPU/DMS interrupt enable register
IF	CPU interrupt flag register
IOF	I/O flags
ST	Status register

The cregister keyword can only be used in file scope. The cregister keyword is not allowed on any declaration within the boundaries of a function. It can only be used on objects of type integer or pointer. The cregister keyword is not allowed on objects of any floating-point type or on any structure or union object.

The cregister keyword does not imply that the object is volatile. If the control register being referenced is volatile (that is, can be modified by some external control), then the object should be declared with the volatile keyword also.

To use a control register in Table 3-2, you must declare each register (with or without volatile) as follows:

```
extern cregsiter volatile unsigned int register;
```

Once you have declared the register, you can use the register name directly. See T*MS320C3x/C4x CPU and Instruction Set Reference Guide* for detailed information on the control registers.

Example 3–1. Define and Use Control Registers

```
extern cregisger volatile unsigned int IE;
extern cregister volatile unsigned int IF;
extern cregisger volatile unsigned int IOF;
extern cregister volatile unsigned int ST;


unsigned myfunc (unsigned int mask)
{
       while (ST & mask == 0)
              /* Do nothing;  wait */;
       return IOF;
}
```

3.3.2 The near and far Keywords

The ’C3x/C4x C compiler extends the C language with the near and far keywords to specify how global and static variables are accessed and how functions are called.

Syntactically, the near and far keywords are treated as storage class modifiers. They can appear before, after, or in between the storage class specifiers and types. Two storage class modifiers cannot be used together in a single declaration. For example, the following are legal uses:

```
far static int x;
static near int x;
static int far x;
far int foo ();
static far int foo ()
```

Near and far data objects

Global and static data objects can be accessed in the following two ways:

near keyword — The compiler assumes that the data item can be accessed relative to the data page pointer. For example:

```
sti   ro, @_var
```

far keyword — The compiler cannot access the data item via the dp. This can be required if the total amount of program data is larger than the offset allowed (32K) from the DP. For example:

```
ldiu  @cl1, aro
sti  ro, *aro
```

where cl1 in the .bss holds the address of the variable.

By default, the compiler generates small-memory model code, which means that every data object is handled as if it were declared near, unless it is actually declared far. If an object is declared near, it is normally loaded using relative offset addressing from the data page pointer (DP, which is B14). DP points to the beginning of the .bss section. If the options –ml, –mb, or –mf are used, the default assumptions change. See those options for how their usage changes the defaults of the standard memory model.

If you use the DATA_SECTION pragma, the object is indicated as a far variable, and this cannot be overridden. This ensures access to the variable, since the variable might not be on the same data page as the .bss section. See section 3.5.2, *The DATA_SECTION pragma*, on page 3-13.

Near and far function calls

Function calls can be invoked in one of two ways:

near keyword — The compiler assumes that the destination of the call is within 2^{24} bytes of the caller. Here, the compiler uses the PC-relative call instruction.

```
call _func
```

far keyword — The compiler is told by the user that the call is not within 2^{24} bytes of the caller, and the compiler uses the register addressing mode of the call instruction.

```
ldiu  @cl1, ro
callu  ro
.
.
.
```

where constant cl1 in the .bss holds the address of the function.

By default, the compiler generates small-memory model code, which means that every function call is handled as if it were declared near, unless it is actually declared far. The default changes, however, when options –ml, –mf, or –mb are used.

Page:	Change or Add:

4–31 Change the last four lines of the EXIT code example.

Exit

```
POP   AR4   ; restore AR4
POPF  R4    ; restore upper 32 bits of R4
POP   R4    ; restore lower 32 bits of R4
POPF  R3    ; restore upper 32 bits of R3
POP   R3    ; restore lower 32 bits of R3
SUBI  2,SP  ; deallocate local frame
POP   FP    ; restore Frame Pointer
POP   ST    ; restore STatus register
reti
```

B–2 Cut the second bullet for the *Using the I/O Functions* section.

Page: **Change or Add:**

The following is the list of known bugs fixed for this release, R5.10. Please refer to the appropriate contact on the If You Need Assistance page of the *TMS320C3x/4x Optimizing C Compiler User's Guide* for more information.

Bug number	Description
SDSsq02943	How should the SET COND bit be set in the C4x ST reg for the C Compiler?
SDSsq03255	Passing –g to asm30 does not work with current Debuggers
SDSsq03762	Command file for lnk30 does not allow embedded "–" in file name
SDSsq03767	Archiver does not allow a default .obj extension
SDSsq03797	.sym debug information contains wrong offset
SDSsq03799	get incorrect results for long doubles divide operation
SDSsq03814	TMS320C3x User's Guide documents –gsrev6 switch
SDSsq03879	Codegen creates illegal parallel instruction subf3 || stf
SDSsq03888	Compiler generate "decomposition error" on sample code
SDSsq03890	Logical AND operator bug in v. 5.00, works ok in v. 4.70
SDSsq03897	optimization level 2 is not as effective as optimization level 1.
SDSsq03908	Optimization appears to use direct instead of indirect addressing
SDSsq03910	Hex3– –byte option, doesn't work with 5.0, worked fine with 4.7 see detail file's.
SDSsq03921	C Compiler generates wrong Assembly code for the #line Statement
SDSsq03982	COMPILER ERROR: Unsupported constant operand for LDP
SDSsq03985	optimizer generates wrong starting address for structure element
SDSsq03988	When linking a debugger MemBlk Save .obj file, >> out of memory, aborting
SDSsq03993	Assembler Generates incorrect "P" parallel-addressing mode 01 instead of 11; MPYF3||SUBF3
SDSsq04019	Far function call generates invalid addresses in big memory configuration
SDSsq04024	Compiler removes circular buffer references from asm (" ") statements
SDSsq04084	pragma DATA_SECTION generates code incorrectly for v5.0 when small mem model
SDSsq04151	The year, tm_year, in ret.src will print as 3 digits after year 2000
SDSsq04155	Compiler generates incorrect code using register passing model.
SDSsq04164	PRTS30 macros for SWW control are incorrect
SDSsq04166	Compiler allocating RS/RC/RE inside c_intxx()'s
SDSsq04172	C4x 32-Bit FP Divide Routine DIV_F40 has error
SDSsq04373	Codegen adds errant adds into assembly code
SDSsq04202	With –mf, compiler generates bad shifts
SDSsq04215	Incorrect code generated for calls to ISR and functions; ST not popped last

Bug number	Description
SDSsq04219	compiler code calls MOD_I40 instead of MOD_U40
SDSsq04227	Incorrect code generated for array reference using –o2
SDSsq04261	INCOMPATIBLE ADDRESSING MODES generated with –02 switch used
SDSsq04296	Assembler swaps operands for MPYF3 || ADDF3
SDSsq04298	RTS has general errors/nuisances
SDSsq04301	Compiler inserts unneeded .aligns into .const section, wastes space
SDSsq04257	Compiler uses inefficient FP divide vs 4.70
SDSsq04338	Compiler is generating illegal code
SDSsq04378	Compiler does not account for second word of long double when optimizing

Page: **Change or Add:**

The following is the list of bugs closed for this release.

Bugs	Description
SDSsq02750	Enhancement request for error and warning summary
SDSsq02751	Request to change the function of the block keyword
SDSsq02917	Documentation incomplete
SDSsq02633	Problem with STF; Need option to do RND before STF
SDSsq02964	ISR's should use more ARx's than Rx's to reduce push/pops
SDSsq02965	switch (var & 7) generates useless default case
SDSsq02966	data & prog on same page = direct addressing
SDSsq02967	Allow assembly DATA in non-standard sections
SDSsq02969	Assembler should warn of possible parallelism
SDSsq03152	Add capabilities to support quoted strings in packed .string ASM directv
SDSsq03153	Add capability for packed char/int arrays to save memory
SDSsq03247	Sometimes the map and stdout do not report eh .cinit section.
SDSsq03257	Wants to be able to use a C header to make assembly .struct directives
SDSsq03258	Optimize the c_intxx() functions
SDSsq03333	customer has problem compiling following code with v4.70
SDSsq03492	strtod() function does not give accurate results
SDSsq03570	Invalid Divide Routine
SDSsq03644	Request for compiler to be able to place init data in named sections
SDSsq03645	Request for compiler to allow packed bytes
SDSsq03788	Symbol "pinit" at address FFFFFFFF appears in link map
SDSsq03798	Encoding of MPYF||SUBF changed from v4.7
SDSsq03892	preprocessing does not work correctly for conditional compilations
SDSsq03896	asm(" ") statement with circular addressing not compiled corrected by C compiler
SDSsq03909	When converting to ascii-hex format, sometimes places entry point in output file
SDSsq03911	Error when compiling code with #line preprocessor command
SDSsq04011	enhancement request – 40bit precision in register variable calls
SDSsq04090	v.5.00 ASM30 generates an INDIRECT ADDRESS REQUIRED error that v.4.70 does not.
SDSsq04165	Differences from V4.70 to V5.00
SDSsq04167	Inline memcpy being generated in c_intxx()'s
SDSsq04220	Optimizer ignores –onx from shell command line and –z from opt30 command line.
SDSsq04347	The #pragma CODE_SECTION doesn't work

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

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

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.