Because you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 6-9, int is used. You can reference linker-defined symbols in a similar manner.

6.5.3 Sharing C/C++ Header Files With Assembly Source

You can use the .cdecls assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code. For more information, see the C/C++ header files chapter in the *TMS320C55x Assembly Language Tools User's Guide*.

6.5.4 Using Inline Assembly Language

Within a C/C++ program, you can use the asm statement to insert a single line of assembly language into the assembly language file created by the compiler. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code. For more information, see Section 5.8.

The asm statement is useful for inserting comments in the compiler output. Simply start the assembly code string with a semicolon (;) as shown below:

```
asm(";*** this is an assembly language comment");
```

NOTE: Using the asm Statement

Keep the following in mind when using the asm statement:

- Be extremely careful not to disrupt the C/C++ environment. The compiler does not check or analyze the inserted instructions.
- Avoid inserting jumps or labels into C/C++ code because they can produce unpredictable results by confusing the register-tracking algorithms that the code generator uses.
- Do not change the value of a C/C++ variable when using an asm statement. This is because the compiler does not verify such statements. They are inserted as is into the assembly code, and potentially can cause problems if you are not sure of their effect.
- Do not use the asm statement to insert assembler directives that change the assembly environment.
- Avoid creating assembly macros in C code and compiling with the --symdebug:dwarf (or -g) option. The C environment's debug information and the assembly macro expansion are not compatible.

6.5.5 Using Intrinsics to Access Assembly Language Statements

The C55x compiler recognizes a number of intrinsic operators. Intrinsics allow you to express the meaning of certain assembly statements that would otherwise be cumbersome or inexpressible in C/C++. Intrinsics are used like functions; you can use C/C++ variables with these intrinsics, just as you would with any normal function.

The intrinsics are specified with a leading underscore, and are accessed by calling them as you do a function. For example:

```
int x1, x2, y;
y = _sadd(x1, x2);
```

Many of the intrinsic operators support saturation. During saturating arithmetic, every expression which overflows is given a reasonable extremum value, either the maximum or the minimum value the expression can hold. For instance, in the above example, if x1==x2==INT_MAX, the expression overflows and saturates, and y is given the value INT_MAX. Saturation is controlled by setting the saturation bit, ST1_SATD, by using these instructions:

```
BSET ST1_SATD
BCLR ST1_SATD
```

The compiler must turn this bit on and off to mix saturating and non-saturating arithmetic; however, it minimizes the number of such bit changing instructions by recognizing blocks of instructions with the same behavior. For maximum efficiency, use saturating intrinsic operators for exactly those operations where you need saturated values in case of overflow, and where overflow can occur. Do not use them for loop iteration counters.

The compiler supports associative versions for some of the addition and multiply-and-accumulate intrinsics. These associative intrinsics are prefixed with _a_. The compiler is able to reorder arithmetic computations involving associative intrinsics, which may produce more efficient code.

For example:

```
int x1, x2, x3, y;
y = _a_sadd(x1, _a_sadd(x2, x3)); /* version 1 */
```

can be reordered inside the compiler as:

```
y = _a_sadd(_a_sadd(x1, x2), x3); /* version 2 */
```

However, this reordering may affect the value of the expression if saturation occurs at different points in the new ordering. For instance, if x1==INT_MAX, x2==INT_MAX, and x3==INT_MIN, version 1 of the expression will not saturate, and y will be equal to (INT_MAX-1); however, version 2 will saturate, and y will be equal to -1. A rule of thumb is that if all your data have the same sign, you may safely use associative intrinsics.

Most of the multiplicative intrinsic operators operate in fractional-mode arithmetic. Conceptually, the operands are Q15 fixed-point values, and the result is a Q31 value. Operationally, this means that the result of the normal multiplication is left shifted by one to normalize to a Q31 value. This mode is controlled by the fractional mode bit, ST1_FRCT.

The intrinsics in Table 6-14 are special in that they accept pointers and references to values; the arguments are passed by reference rather than by value. These values must be modifiable values (for example, variables but not constants, nor arithmetic expressions). These intrinsics do not return a value; they create results by modifying the values that were passed by reference. These intrinsics depend on the C++ reference syntax, but are still available in C code with the C++ semantics.

No declaration of the intrinsic functions is necessary, but declarations are provided in the header file, c55x.h, included with the compiler.

Many of the intrinsic operators are useful for implementing basic DSP functions described in the Global System for Mobile Communications (GSM) standard of the European Telecommunications Standards Institute (ETSI). These functions have been implemented in the header file, gsm.h, included with the compiler. Additional support for ETSI GSM functions is described in Section 6.5.5.2.

6.5.5.1 Descriptions of C55x Intrinsics

Table 6-9 through Table 6-15 list all of the intrinsic operators in the TMS320C55x C/C++ compiler. A *function* prototype is presented for each intrinsic that shows the expected type for each parameter. If the argument type does not match the parameter, type conversions are performed on the argument. Where argument order matters, the order of the intrinsic's input arguments matches that of the underlying hardware instruction. The resulting assembly language mnemonic is given for each instruction; for some instructions, such as MPY, an alternate instruction such as SQR (which is a specialized MPY) may be generated if it is more efficient. A brief description is provided for each intrinsic. For a precise definition of the underlying instruction, see the *TMS320C55x DSP Mnemonic Instruction Set Reference Guide* and *TMS320C55x DSP Algebraic Instruction Set Reference Guide*.

Table 6-8. C55x Circular Addressing Intrinsics

Compiler Intrinsic	Description
int _circ_incr(int index, int incr, unsigned int size)	Returns the circular increment of index+incr relative to size when these preconditions are met: 0 <= index < size and incr <= size.

Table 6-9. C55x C/C++ Compiler Intrinsics (Addition, Subtraction, Negation, Absolute Value)

Compiler Intrinsic		Assembly Instruction	Description
int	_sadd(int src1, int src2)	ADD	Returns the saturated sum of its operands.
int	_a_sadd(int src1, int src2)		
long	_lsadd(long src1, long src2)		
long	_a_lsadd(long src1, long src2)		
long long	_llsadd(long long src1, long long src2)		
long long	_a_llsadd(long long src1, long long src2)		
int	_ssub(int src1, int src2)	SUB	Returns the saturated value of the expression (src1 − src2).
long	_lssub(long src1, long src2)		
long long	_llssub(long long src1, long long src2)		
int	_sneg(int src)	NEG	Returns the saturated value of the expression (0 − src).
long	_lsneg(long src)		
long long	_llsneg(long long src)		
int	_abss(int src)	ABS	Returns the saturated absolute value of its operands.
long	_labss(long src)		
long long	_llabss(long src)		

Table 6-10. TMS320C55x C/C++ Compiler Intrinsics (Multiplication)

Compiler Intrinsic		Assembly Instruction	Description
int	_smpy(int src1, int src2)	MPY	Returns the saturated fractional-mode product of its operands.
long	_lsmpy(int src1, int src2)		
long	_lsmpyu(unsigned src1, unsigned src2)		
long	_lsmpysu(int src1, unsigned src2)		
long long	_llsmpy(int sr1, int src2)		
long long	_llsmpyu(unsigned src1, unsigned src2)		
long long	_llsmpysu(int sr1, unsigned src2)		
long	_lmpy(int src1, int src2)	MPY	Returns the unsaturated integer-mode (non-fractional-mode) product of its operands.
long	_lmpyu(unsigned src1, unsigned src2)		
long	_lmpysu(int src1, unsigned src2)		
long long	_llmpy(int src1, int src2)		
long	_lsmpyi(int src1, int src2)	MPY	Returns the saturated integer-mode product of its operands.
long	_lsmpyui(unsigned src1, unsigned src2)		
long	_lsmpysui(int src1, unsigned src2)		
long long	_llsmpyi(int src1, int src2)		
long long	_llsmpyui(unsigned src1, unsigned src2)		
long long	_llsmpysui(int src1, unsigned src2)		
long	_lsmpyr(int src1, int src2)	MPYR	Returns the saturated fractional-mode product of its operands, rounded as if the intrinsic _sround were used.
long	_smac(long src1, int src2, int src3)	MAC	Returns the saturated sum of src1 and the fractional-mode product of src2 and src3.
long	_a_smac(long src1, int src2, int src3)		
long	_smacsu(long src1, int src2, unsigned src3)		
long long	_llsmac(long long src1, int src2, int src3)		
long long	_llsmacu(long long src1, unsigned src2, unsigned src3)		
long long	_llsmacsu(long long src1, int src2, unsigned src3)		
long	_smaci(long src1, int src2, int src3)	MAC	Returns the saturated sum of src1 and the integer-mode product of src2 and src3.
long	_smacsui(long src1, int src2, unsigned src3)		
long long	_llsmaci(long long src1, int src2, int src3)		
long long	_llsmacui(long long src1, unsigned src2, unsigned src3)		
long long	_llsmacsui(long long src1, int src2, unsigned src3)		
long	_smacr(long src1, int src2, int src3)	MACR	Returns the saturated sum of src1 and the fractional-mode product of src2 and src3. The sum is rounded as if the intrinsic _sround were used.
long	_a_smacr(long src1, int src2, int src3)		

Table 6-10. TMS320C55x C/C++ Compiler Intrinsics (Multiplication) (continued)

Compiler Intrinsic		Assembly Instruction	Description
long long long long long long long long long 	_smas(long src1, int src2, int src3) _a_smas(long src1, int src2, int src3) _smassu(long src1, int src2, unsigned src3) _llsmas(long long src1, int src2, int src3) _llsmasu(long long src1, unsigned src2, unsigned src3) _llsmassu(long long src1, int src2, unsigned src3)	MAS	Returns the saturated difference of src1 and the fractional-mode product of src2 and src3.
long long long long long long long long 	_smasi(long src1, int src2, int src3) _smassui(long src1, int src2, unsigned src3) _llsmasi(long long src1, int src2, int src3) _llsmasui(long long src1, unsigned src2, unsigned src3) _llsmassui(long long src1, int src2, unsigned src3)	MAS	Returns the saturated difference of src1 and the integer-mode product of src2 and src3.
long long	_smasr(long src1, int src2, int src3) a_smasr(long src1, int src2, int src3)	MASR	Returns the saturated difference of src1 and the fractional-mode product of src2 and src3. The sum is rounded as if the intrinsic _sround were used.

Table 6-11. TMS320C55x C/C++ Compiler Intrinsics (Shifting)

Compiler Intrinsic		Assembly Instruction	Description
int long long long	_sshl(int src1, int src2) _lsshl(long src1, int src2) _llsshl(long long, int)	SFTS	Returns the saturated value of the expression (src1<<src2). If src2 is negative, a right shift is performed instead.
int long	_shrs(int src1, int src2) _lshrs(long src1, int src2)	SFTS	Returns the saturated value of the expression (src1>>src2). If src2 is negative, a left shift is performed instead.
int long long long	_shl(int src1, int src2) _lshl(long src1, int src2) _llshl(long long src1, int src2)	SFTS	Returns the expression (src1<<src2). If src2 is negative, a right shift is performed instead. No saturation is performed.

Table 6-12. TMS320C55x C/C++ Compiler Intrinsics (Shifting and Storing)

Compiler Intrinsic		Description
void	_llsshlstore(long long src, int cnt, int *dst)	Stores bits 16-31 of the result of a saturating (based on bit 31) shift of src by cnt into dst using: MOV HI(saturate(src << cnt)) , dst
void void	_llsshlstorer(long long, int, int*) _llsshlstorern(long long, int, int*)	Stores bits 16-31 of the rounded result of a saturating (based on bit 31) shift of src by cnt into dst using: MOV rnd(HI(saturate(src << cnt))), dst _llsshlstorer rounds by adding 2^{15} using saturating arithmetic (biased round to positive infinity). _llsshlstorern rounds to nearest multiple of 2^{16} using saturating arithmetic. Ties are broken by rounding to even.
void void	_llshlstorer(long long, int, int*) _llshlstorern(long long, int, int*)	Stores bits 16-31 of the rounded result of shifting src by cnt using: MOV rnd(HI(src << cnt)), dst _llshlstorer rounds by adding 2^{15} using unsaturating arithmetic (biased round to positive infinity). _llshlstorern rounds to nearest multiple of 2^{16} using unsaturating arithmetic. Ties are broken by rounding to even.

Table 6-13. TMS320C55x C/C++ Compiler Intrinsics (Rounding, Saturation, Bitcount, Extremum)

Compiler Intrinsic		Assembly Instruction	Description
long long long	_round(long src) _llround(long long src)	ROUND	Uses unsaturating arithmetic (biased round to positive infinity) and clears the lower 16 bits. The upper 16 bits of the Q31 result can be treated as a Q15 value.
long long long	_sround(long src) _llsround(long long src)	ROUND	Returns the value src rounded by adding 2^15 using saturating arithmetic (biased round to positive infinity) and clearing the lower 16 bits. The upper 16 bits of the Q31 result can be treated as a Q15 value.
long long long	_roundn(long src) _llroundn(long long src)	ROUND	Returns the value src rounded to the nearest multiple of 2^16 using unsaturating arithmetic and clearing the lower 16 bits. Ties are broken by rounding to even. The upper 16 bits of the Q31 result can be treated as a Q15 value.
long long long	_sroundn(long src) _llsroundn(long long src)	ROUND	Returns the value src rounded to the nearest multiple of 2^16 using saturating arithmetic and clearing the lower 16 bits. Ties are broken by rounding to even. The upper 16 bits of the Q31 result can be treated as a Q15 value.
int int int	_norm(int src) _lnorm(long src) _llnorm(long long src)	EXP	Returns the left shift count needed to normalize src to a 32-bit long value. This count may be negative.
long	_lsat(long long src)	SAT	Returns src saturated to a 32-bit long value. If src was already within the range allowed by long, the value does not change; otherwise, the value returned is either LONG_MIN or LONG_MAX.
int	_count(unsigned long long src1, unsigned long long src2)	BCNT	Returns the number of bits set in the expression (src1 & src2).
int long long long	_max(int src1, int src2) _lmax(long src1, long src2) _llmax(long long src1, long long src2)	MAX	Returns the maximum of src1 and src2.
int long long long	_min(int src1, int src2) _lmin(long src1, long src2) _llmin(long long src1, long long src2)	MIN	Returns the minimum of src1 and src2.

Table 6-14. Compiler Intrinsics (Arithmetic With Side Effects)

Compiler Intrinsic	Assembly Instruction	Description

void _firs(int *, int *, int *, int&, long&)
void _firsn(int *, int *, int *, int&, long&)
	FIRSADD
	FIRSSUB
		Performs the corresponding instruction as follows:
		int *p1, *p2, *p3, srcdst1;
		long srcdst2;
		...
		_firs(p1, p2, p3, srcdst1, srcdst2);
		_firsn(p1, p2, p3, srcdst1, srcdst2);
		Which becomes (respectively):
		FIRSADD *p1, *p2, *p3, srcdst1, srcdst2
		FIRSSUB *p1, *p2, *p3, srcdst1, srcdst2
		Mode bits SATD, FRCT, and M40 are 0.

void _lms(int *, int *, int&, long&)	LMS	Performs the LMS instruction as follows: Where *type* is long or long long int *p1, *p2, srcdst1; *type* srcdst2; ... _lms (p1, p2, srcdst1, srcdst2); Which becomes: LMS *p1, *p2, srcdst1, srcdst2 For _llslms and _llslmsi saturation is enabled. For _llslms fractional mode is enabled

Table 6-14. Compiler Intrinsics (Arithmetic With Side Effects) (continued)

Compiler Intrinsic	Assembly Instruction	Description

void _abdst(int *, int *, int&, long&)
void _sqdst(int *, int *, int&, long&)
	ABDST
	SQDST
		Performs the corresponding instruction as follows:
		int *p1, *p2, srcdst1;
		long srcdst2;
		...
		_abdst(p1, p2, srcdst1, dst);
		_sqdst(p1, p2, srcdst1, dst);
		Which becomes (respectively):
		ABDST *p1, *p2, srcdst1, srcdst2
		SQDST *p1, *p2, srcdst1, srcdst2
		Mode bits SATD, FRCT, and M40 are 0.

int _exp_mant(long, long&) int _llexp_mant(long long, long long&)	MANT:: NEXP	Performs the MANT::NEXP instruction pair, as follows: int src, dst2; long dst1; ... dst2 = _exp_mant(src, dst1); Which becomes: MANT src, dst1 :: NEXP src, dst2

void _max_diff_dbl(long, long, long&, long&, unsigned &)
void _min_diff_dbl(long, long, long&, long&, unsigned &)
void _smax_diff_dbl(long, long, long&, long&, unsigned&)
void _smin_diff_dbl(long, long, long&, long&, unsigned&)
void _llmax_diff_dbl(long long, long long, long long&, long long&, unsigned&)
void _llmin_diff_dbl(long long, long long, long long&, long long&, unsigned&)
void _llsmax_diff_dbl(long long, long long, long long&, long long&, unsigned&)
void _llsmin_diff_dbl(long long, long long, long long&, long long&, unsigned&)
	DMAXDIFF
	DMINDIFF
		Performs the corresponding instruction, as follows:
		Where *type* is long or long long
		type src1, src2, dst1, dst2;
		int dst3;
		...
		_max_diff_dbl(src1, src2, dst1, dst2, dst3);
		_min_diff_dbl(src1, src2, dst1, dst2, dst3);
		Which becomes (respectively):
		DMAXDIFF src1, src2, dst1, dst2, dst3
		DMINDIFF src1, src2, dst1, dst2, dst3
		The smax and smin forms are performed with saturation enabled.

Table 6-15. C55x C/C++ Compiler Intrinsics (Non-Arithmetic)

Compiler Intrinsic	Assembly Instruction	Description
ong long _dtol(double)		Reinterpret double as long (when long is 40 bits).
long long _dtoll(double)		Reinterpret double as long long.
void _enable_interrupts(void) unsigned int _disable_interrupts(void)	BCLR ST1_INTM BSET ST1_INTM	Enables or disables interrupts and ensure enough cycles are consumed that the change takes effect before anything else happens.
void _restore_interrupts(unsigned int)		Restores interrupts to state indicated by value returned from _disable_interrupts .

6.5.5.2 Intrinsics and ETSI Functions

The functions in Table 6-16 provide additional support for ETSI GSM functions. Functions L_add_c, L_sub_c, and L_sat map to GSM inline macros. The other functions in the table are run-time functions. Additional details about these functions can be found in various ETSI documents. In particular, see Chapter 13 (BASOP: ITU-T Basic Operators) of *ITU-T Software Tool Library 2005 User's Manual* found at http://www.itu.int/rec/T-REC-G.191-200508-I/en.

Table 6-16. ETSI Support Functions

Compiler Intrinsic	Description
long L_add_c(long src1, long src2)	Adds src1, src2, and Carry bit. This function does not map to a single assembly instruction, but to an inline function.
long L_sub_c(long src1, long src2)	Subtracts src2 and logical inverse of sign bit from src1. This function does not map to a single assembly instruction, but to an inline function.