

## ***Open Watcom Code Generator Interface***

# ***Table of Contents***

Introduction .....	1
General .....	3
Segments .....	9
Labels .....	11
Back Handles .....	13
Type definitions .....	15
Procedure Declarations .....	19
Expressions .....	21
Leaf Nodes .....	25
Assignment Operations .....	27
Arithmetic/logical operations .....	29
Procedure calls .....	31
Comparison/short-circuit operations .....	33
Control flow operations .....	35
Select and Switch statements. ....	37
Other .....	41
Data Generation .....	45
Front End Routines .....	49
Debugging Information .....	59
Registers .....	67
Miscellaneous .....	69
A. Pre-defined macros .....	73
B. Register constants .....	77
C. Debugging Open Watcom Code Generator .....	79

---

# ***Introduction***

The code generator (back end) interface is a set of procedure calls. These are divided into following category of routines.

- Code Generation (CG)
- Data Generation (DG)
- Miscellaneous Back End (BE)
- Front end supplied (FE)
- Debugger information (DB)



---

# General

***cg\_init\_info BEInit( cg\_switches switches, cg\_target\_switches targ\_switches, uint  
optsize, proc\_revision proc )***

Initialize the code generator. This must be the first routine to be called.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>switches</i>	Select code generation options. The options are bits, so may be combined with the bit-wise operator  . Options apply to the entire compilation unit. The bit values are defined below.
-----------------	--

<i>targ_switches</i>	Target specific switches. The bit values are defined below.
----------------------	---

<i>optsize</i>	A number between 0 and 100. 0 means optimize for speed, 100 means optimize for size. Anything in between selects a compromise between speed and size.
----------------	---

<i>proc</i>	The target hardware configuration, defined below.
-------------	---

<i>Returns</i>	Information about the code generator revision in a <i>cg_init_info</i> structure, defined below.
----------------	--

<b><i>Switch</i></b>	<b><i>Definition</i></b>
----------------------	--------------------------

<i>NO_OPTIMIZATION</i>	Turn off optimizations.
------------------------	-------------------------

<i>DBG_NUMBERS</i>	Generate line number debugging information.
--------------------	---

<i>FORTTRAN_ALIASING</i>	Assume pointers are only used for parameter passing.
--------------------------	--

<i>DBG_DF</i>	Generate debugging information in DWARF format.
---------------	---

<i>DBG_CV</i>	Generate debugging information in CodeView format. If neither <i>DBG_DF</i> nor <i>DBG_CV</i> is set, debugging information (if any) is generated in the Watcom format.
---------------	---

<i>RELAX_ALIAS</i>	Assume that a static/extern variable and a pointer to that same variable are not used within the same routine.
--------------------	--

<i>DBG_LOCALS</i>	Generate local symbol information for use by a debugger.
-------------------	--

<i>DBG_TYPES</i>	Generate typing information for use by a debugger.
------------------	--

<i>LOOP_UNROLLING</i>	Turn on loop unrolling.
-----------------------	-------------------------

<i>LOOP_OPTIMIZATION</i>	Turn on loop optimizations.
--------------------------	-----------------------------

<i>INS_SCHEDULING</i>	Turn on instruction scheduling.
-----------------------	---------------------------------

<i>MEMORY_LOW_FAILS</i>	Allow the code generator to run out of memory without being able to generate object code (allows the 386 compiler to use EBP as a cache register).
<i>FP_UNSTABLE_OPTIMIZATION</i>	Allow the code generator to perform optimizations that are mathematically correct, but are numerically unstable. E.g. converting division by a constant to a multiplication by the reciprocal.
<i>NULL_DEREF_OK</i>	NULL points to valid memory and may be dereferenced.
<i>FPU_ROUNDING_INLINE</i>	Inline floating-point value rounding (actually truncation) routine when converting floating-point values to integers.
<i>FPU_ROUNDING_OMIT</i>	Omit floating-point value rounding entirely and use FPU default. Results will not be ISO C compliant.
<i>ECHO_API_CALLS</i>	Log each call to the code generator with its arguments and return value. Only available in debug builds.
<i>OBJ_ELF</i>	Emit ELF object files.
<i>OBJ_COFF</i>	Emit COFF object files. For Intel compilers, OMF object files will be emitted in the absence of either switch.
<i>OBJ_ENDIAN_BIG</i>	Emit big-endian object files (COFF or ELF). If OBJ_ENDIAN_BIG is not set, little-endian objects will be generated.
<i>x86 Switch</i>	<b>Definition</b>
<i>I_MATH_INLINE</i>	Do not check arguments for operators like O_SQRT. This allows the compiler to use some specialty x87 instructions.
<i>EZ_OMF</i>	Generate Phar Lap EZ-OMF object files.
<i>BIG_DATA</i>	Use segmented pointers (16:16 or 16:32). This defines TY_POINTER to be equivalent to TY_HUGE_POINTER.
<i>BIG_CODE</i>	Use inter segment (far) call and return instructions.
<i>CHEAP_POINTER</i>	Assume far objects are addressable by one segment value. This must be used in conjunction with BIG_DATA. It defines TY_POINTER to be equivalent to TY_FAR_POINTER.
<i>FLAT_MODEL</i>	Assume all segment registers address the same base memory.
<i>FLOATING_FS</i>	Does FS float (or is it pegged to DGROUP).
<i>FLOATING_GS</i>	Does GS float (or is it pegged to DGROUP).
<i>FLOATING_ES</i>	Does ES float (or is it pegged to DGROUP).
<i>FLOATING_SS</i>	Does SS float (or is it pegged to DGROUP).
<i>FLOATING_DS</i>	Does DS float (or is it pegged to DGROUP).

<i>USE_32</i>	Generate code into a use32 segment (versus use16).
<i>INDEXED_GLOBALS</i>	Generate all global and static variable references as an offset past EBX.
<i>WINDOWS</i>	Generate windows prolog/epilog sequences for all routines.
<i>CHEAP_WINDOWS</i>	Generate windows prolog/epilog sequences assuming that call backs functions are defined as <code>__export</code> .
<i>NO_CALL_RET_TRANSFORM</i>	Do not change a <code>CALL</code> followed by a <code>RET</code> into a <code>JMP</code> . This is used for some older overlay managers that cannot handle a <code>JMP</code> to an overlay.
<i>CONST_IN_CODE</i>	Generate all constant data into the code segment. This only applies to the internal code generator data, such as floating point constants. The front end decides where its data goes using <code>BESetSeg()</code> .
<i>NEED_STACK_FRAME</i>	Generate a traceable stack frame. The first instructions will be <b>INC BP</b> if the routine uses a far return instruction, followed by <b>PUSH BP</b> and <b>MOV BP,SP</b> . (ESP and EBP for 386 targets).
<i>LOAD_DS_DIRECTLY</i>	Generate code to load DS directly. By default, a call to <code>__GETDS</code> routine is generated.
<i>GEN_FWAIT_386</i>	Generate <code>FWAIT</code> instructions on 386 and later CPUs. The 386 never needs <code>FWAIT</code> for data synchronization, but <code>FWAIT</code> may still be needed for accurate exception reporting.
<b><i>RISC Switch</i></b>	<b><i>Definition</i></b>
<i>ASM_OUTPUT</i>	Print final pseudo-assembly on the console. Debug builds only.
<i>OWL_LOGGING</i>	Log calls to the Object Writer Library
<i>STACK_INIT</i>	Pre-initialize stack variables to a known bit pattern.
<i>EXCEPT_FILTER_USED</i>	Set when SEH (Structured Exception Handling) is used.

The supported `proc_revision` CPU values are:

CPU\_86  
CPU\_186  
CPU\_286  
CPU\_386  
CPU\_486  
CPU\_586

The supported `proc_revision` FPU values are:

FPU\_NONE  
FPU\_87  
FPU\_387  
FPU\_586  
FPU\_EMU  
FPU\_E87

FPU\_E387  
FPU\_E586

The supported proc\_revision WEITEK values are:

WTK\_NONE  
WTK\_1167  
WTK\_3167  
WTK\_4167

The following example sets the processor revision information to indicate a 386 with 387 and Weitek 3167.

```
proc_revision proc;  
  
SET_CPU( p, CPU_386 );  
SET_FPU( p, FPU_387 );  
SET_WTK( p, WTK_3167 );
```

The return value structure is defined as follows:

```
typedef union cg_init_info {  
    struct {  
        unsigned revision    : 10; /* contains II_REVISION */  
        unsigned target      : 5;  /* has II_TARG_??? */  
        unsigned is_large    : 1;  /* 1 if 16 bit host */  
    } version;  
    int success;  
} cg_init_info;  
  
enum {  
    II_TARG_8086,  
    II_TARG_80386,  
    II_TARG_STUB,  
    II_TARG_CHECK,  
    II_TARG_370,  
    II_TARG_AXP,  
    II_TARG_PPC,  
    II_TARG_MIPS  
};
```

### ***void BEStart()***

Start the code generator. Must be called immediately after all calls to BEDefSeg have been made. This restriction is relaxed somewhat for the 80(x)86 code generator. See BEDefSeg for details.

### ***void BEMStop()***

Normal termination of code generator. This must be the second last routine called.



***void BEAbort()***

Abnormal termination of code generator. This must be the second last routine called.

***void BEFini()***

Finalize the code generator. This must be the last routine called.

***patch\_handle BEPatch()***

Allocate a patch handle which can be used to create a patchable integer (an integer which will have a constant value provided sometime while the codegen is handling the CGDone call). See CGPatchNode.

***void BEPatchInteger( patch\_handle hdl, signed\_32 value )***

Patch the integer corresponding to the given handle to have the given value. This may be called repeatedly with different values, providing CGPatchNode has been called and BEFiniPatch has not been called.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>hdl</i>	A patch_handle returned from an earlier invocation of BEPatch which has had a node allocated for it via CGPatchNode. If CGPatchNode has not been called with the handle given, the behaviour is undefined.
------------	--

<i>value</i>	A signed 32-bit integer value. This will be the new value of the node which has been associated with the patch handle.
--------------	--

***cg\_name BEFiniPatch( patch\_handle hdl )***

This must be called to free up resources used by the given handle. After this, the handle must not be used again.



---

# Segments

The object file produced by the code generator is composed of various segments. These are defined by the front end. A program may have as many data and code segments as required by the front end. Each segment may be regarded as an individual file of objects, and may be created simultaneously. There is a current segment, selected by `BESetSeg()`, into which all DG routines generate their data. The code for each routine is generated into the segment returned by the `FESegID()` call when it is passed the `cg_sym_handle` for the routine. It is illegal to write data to the code segment for a routine in between the `CGProcDecl` call and the `CGReturn` call.

The following routines are used for initializing, finalizing, defining and selecting segments.

## ***void BEDefSeg( segment\_id segid, seg\_attr attr, char \*str, uint align )***

Define a segment. This must be called after `BEInit` and before `BESetSeg`. For the 80(x)86 code generator, you are allowed to define additional segments after `BESetSeg` if they are:

1. Code Segments
2. PRIVATE data segments.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>segid</i>	A non-negative integer used as an identifier for the segment. It is arbitrarily picked by the front end.
<i>attr</i>	Segment attribute bits, defined below.
<i>str</i>	The name given to the segment.
<i>align</i>	The segment alignment requirements. The code generator will pick the next larger alignment allowed by the object module format. For example, 9 would select paragraph alignment.

<b><i>Attribute</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>EXEC</i>	This is a code segment.
<i>GLOBAL</i>	The segment is accessible to other modules. (versus PRIVATE).
<i>INIT</i>	The segment is statically initialized.
<i>ROM</i>	The segment is read only.
<i>BACK</i>	The code generator may put its data here. One segment must be marked with this attribute. It may not be a COMMON, PRIVATE or EXEC segment. If the front end requires code in the EXEC segment, the <code>CONST_IN_CODE</code> switch must be passed to <code>BEInit()</code> .
<i>COMMON</i>	All occurrences of this segment will be overlayed. This is used for FORTRAN common blocks.

**PRIVATE** The segment is non combinable. This is used for far data items.

**GIVEN\_NAME** Normally, the back end feels free to prepend or append strings to the segment name passed in by the front end. This allows a naive front end to specify a constant set of segment names, and have the code generator mangle them in such a manner that they work properly in concert with the set of `cg_switches` that have been specified (e.g. prepending the module name to the code segments when `BIG_CODE` is specified on the x86). When `GIVEN_NAME` is specified, the back end outputs the segment name to the object file exactly as given.

**THREAD\_LOCAL** Segment contains thread local data. Such segments may need special handling in executable modules.

### ***segment\_id BESetSeg( segment\_id segid )***

Select the current segment for data generation routines. Code for a routine is always output into the segment returned by `FESegID` when it is passed the routine symbol handle.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>segid</i>	Selects the current segment.
--------------	------------------------------

<i>Returns</i>	The previous current segment.
----------------	-------------------------------

**Notes:** When emitting data into an EXEC or BACK segment, be aware that the code generator is at liberty to emit code and/or back end data into that segment anytime you make a call to a code generation routine (CG\*). Do NOT expect data items to be contiguous in the segment if you have made an intervening CG\* call.

### ***segment\_id BEGetSeg( void )***

Return the current segment for generation routines.

<i>Returns</i>	The current segment.
----------------	----------------------

### ***void BEFlushSeg( segment\_id segid )***

`BEFlushSeg` informs the back end that no more code/data will be generated in the specified segment. For code segments, it must be called after the `CGReturn()` for the final function which is placed in the segment. This causes the code generator to flush all pending information associated with the segment and allows the front end to free all the back handles for symbols which were referenced by the code going into the segment. (The FORTRAN compiler uses this since each function has its own symbol table which is thrown out at the end of the function).

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>segid</i>	The code segment id.
--------------	----------------------

---

# Labels

The back end uses a **label\_handle** for flow of control. Each **label\_handle** is a unique code label. These labels may only be used for flow of control. In order to define a label in a data segment, a **back\_handle** must be used.

## ***label\_handle* BENewLabel()**

Allocate a new control flow label.

*Returns*            A new label\_handle.

## ***void* BEFiniLabel( *label\_handle* lbl )**

Indicate that a label\_handle will not be used by the front end anymore. This allows the back end to free some memory at some later stage.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>lbl</i>	A label_handle
------------	----------------



---

# Back Handles

A **back\_handle** is the front end's handle for a code generator symbol table entry. A **cg\_sym\_handle** is the code generator's handle for a front end symbol table entry. The back end may call FEBack, passing in any cg\_sym\_handle that has been passed to it. The front end must allocate a back\_handle via BNewBack if one does not exist. Subsequent calls to FEBack should return the same back\_handle. This mechanism is used so that the back end does not have to do symbol table searches. For example:

```
back_handle FEBack( SYMPOINTER sym )
{
    if( sym->back == NULL ) {
        sym->back = BNewBack( sym );
    }
    return( sym->back );
}
```

It is the responsibility of the front end to free each back\_handle, via BEFreeBack, when it frees the corresponding cg\_sym\_handle entry.

A back\_handle for a symbol having automatic or register storage duration (auto back\_handle) may not be freed until CGReturn is called. A back\_handle for a symbol having static storage duration, (static back\_handle) may not be freed until BStop is called or until after a BEFlushSeg is done for a segment and the back\_handle will never be referenced by any other function.

The code generator will not require a back handle for symbols which are not defined in the current compilation unit.

The front end must define the location of all symbols with static storage duration by passing the appropriate back\_handle to DGLabel. It must also reserve the correct amount of space for that variable using DGBytes or DGUBytes.

The front end may also allocate an back\_handle with static storage duration that has no cg\_sym\_handle associated with it (anonymous back\_handle) by calling BNewBack(NULL). These are useful for literal strings. These must also be freed after calling BStop.

## ***back\_handle BNewBack( cg\_sym\_handle sym )***

Allocate a new back\_handle.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>sym</i>	The front end symbol handle to be associated with the back_handle. It may be NULL.
------------	--

<b><i>Returns</i></b>	A new back_handle.
-----------------------	--------------------

### ***void BEFinBack( back\_handle bck )***

Indicate that **bck** will never be passed to the back end again, except to BEFreeBack. This allows the code generator to free some memory at some later stage.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>bck</i>	A back_handle.
------------	----------------

### ***void BEFreeBack( back\_handle bck )***

Free the back\_handle **bck**. See the preamble in this section for restrictions on freeing a back\_handle.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>bck</i>	A back_handle.
------------	----------------



---

# Type definitions

Base types are defined as constants. All other types (structures, arrays, unions, etc) are simply defined by their length. The base types are:

<i>Type</i>	<i>C type</i>
<i>TY_UINT_1</i>	unsigned char
<i>TY_INT_1</i>	signed char
<i>TY_UINT_2</i>	unsigned short
<i>TY_INT_2</i>	signed short
<i>TY_UINT_4</i>	unsigned long
<i>TY_INT_4</i>	signed long
<i>TY_UINT_8</i>	unsigned long long
<i>TY_INT_8</i>	signed long long
<i>TY_LONG_POINTER</i>	far *
<i>TY_HUGE_POINTER</i>	huge *
<i>TY_NEAR_POINTER</i>	near *
<i>TY_LONG_CODE_PTR</i>	(far *)()
<i>TY_NEAR_CODE_PTR</i>	(near *)()
<i>TY_SINGLE</i>	float
<i>TY_DOUBLE</i>	double
<i>TY_LONG_DOUBLE</i>	long double
<i>TY_INTEGER</i>	int
<i>TY_UNSIGNED</i>	unsigned int
<i>TY_POINTER</i>	*
<i>TY_CODE_PTR</i>	(*)()
<i>TY_BOOLEAN</i>	The result of a comparison or flow operator. May also be used as an integer.

<i>TY_DEFAULT</i>	Used to indicate default conversion
<i>TY_NEAR_INTEGER</i>	The result of subtracting 2 near pointers
<i>TY_LONG_INTEGER</i>	The result of subtracting 2 far pointers
<i>TY_HUGE_INTEGER</i>	The result of subtracting 2 huge pointers

There are two special constants.

*TY\_FIRST\_FREE* The first user definable type

*TY\_LAST\_FREE* The last user definable type.

### ***void BEDefType( cg\_type what, uint align, unsigned\_32 len )***

Define a new type to the code generator.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>what</i>	An integral value greater than or equal to <i>TY_FIRST_FREE</i> and less then or equal to <i>TY_LAST_FREE</i> , used as the type identifier.
<i>align</i>	Currently ignored.
<i>len</i>	The length of the new type.

### ***void BEAliasType( cg\_type what, cg\_type to )***

Define a type to be an alias for an existing type.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>what</i>	Will become an alias for an existing type.
<i>to</i>	An existing type.

### ***unsigned\_32 BETypeLength( cg\_type type )***

Return the length of a previously defined type, or a base type.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>type</i>	A previously defined type.
<i>Returns</i>	The length associated with the type.

***uint BTypeAlign( cg\_type type )***

Return the alignment requirements of a type. This is always 1 for x86 and 370 machines.

***Parameter      Definition***

*type*              A previously defined type.

*Returns*              The alignment requirements of **type** as declared in BEDefType, or for a base type, as defined by the machine architecture.



---

# Procedure Declarations

## ***void CGProcDecl( cg\_sym\_handle name, cg\_type type )***

Declare a new procedure. This must be the first routine to be called when generating each procedure.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>name</i>	The front end symbol table entry for the procedure. A back_handle will be requested.
-------------	--

<i>type</i>	The return type of the procedure. Use TY_INTEGER for void functions.
-------------	--

## ***void CGParmDecl( cg\_sym\_handle name, cg\_type type )***

Declare a new parameter to the current function. The calls to this function define the order of the parameters. This function must be called immediately after calling CGProcDecl. Parameters are defined in left to right order, as defined by the procedure prototype.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>name</i>	The symbol table entry for the parameter.
-------------	---

<i>type</i>	The type of the parameter.
-------------	----------------------------

## ***label\_handle CGLastParm()***

End a parameter declaration section. This function must be called after the last parameter has been declared. Prior to this function, the only calls the front-end is allowed to make are CGParmDecl and CGAutoDecl.

## ***void CGAutoDecl( cg\_sym\_handle name, cg\_type type )***

Declare an automatic variable.

This routine may be called at any point in the generation of a function between the calls to CGProcDecl and CGReturn, but must be called before **name** is passed to CGFName.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>name</i>	The symbol table entry for the variable.
-------------	--

<i>type</i>	The type of the variable.
-------------	---------------------------

### ***temp\_handle CGTemp( cg\_type type )***

Yields a temporary with procedure scope. This can be used for things such as iteration counts for FORTRAN do loops, or a variable in which to store the return value of a function. This routine should be used **only if necessary**. It should be used when the front end requires a temporary which persists across a flow of control boundary. Other temporary results are handled by the expression trees.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>type</i>	The type of the new temporary.
-------------	--------------------------------

<i>Returns</i>	A temp_handle which may be passed to CGTempName. This will be freed and invalidated by the back end when CGReturn is called.
----------------	--

---

# Expressions

Expression processing involves building an expression tree in the back end, using calls to CG routines. There are routines to generate leaf nodes, binary and unary nodes, and others. These routines return a handle for a node in a back end tree structure, called a **cg\_name**. This handle must be exactly once in a subsequent call to a CG routine. A tree may be built in any order, but a cg\_name is invalidated by a call to any CG routine with return type void. The exception to this rule is CGTrash.

There is no equivalent of the C address of operator. All leaf nodes generated for symbols, via CGFEName, CGBackName and CGTempName, yield the address of that symbol, and it is the responsibility of the front end to use an indirection operator to get its value. The following operators are available:

<i>0-ary Operator</i>	<i>C equivalent</i>
-----------------------	---------------------

<i>O_NOP</i>	N/A
--------------	-----

<i>Unary Operator</i>	<i>C equivalent</i>
-----------------------	---------------------

<i>O_UMINUS</i>	-x
-----------------	----

<i>O_COMPLEMENT</i>	x
---------------------	---

<i>O_POINTS</i>	(*x)
-----------------	------

<i>O_CONVERT</i>	x=y
------------------	-----

<i>O_ROUND</i>	Do not use!
----------------	-------------

<i>O_LOG</i>	log(x)
--------------	--------

<i>O_COS</i>	cos(x)
--------------	--------

<i>O_SIN</i>	sin(x)
--------------	--------

<i>O_TAN</i>	tan(x)
--------------	--------

<i>O_SQRT</i>	sqrt(x)
---------------	---------

<i>O_FABS</i>	fabs(x)
---------------	---------

<i>O_ACOS</i>	acos(x)
---------------	---------

<i>O_ASIN</i>	asin(x)
---------------	---------

<i>O_ATAN</i>	atan(x)
---------------	---------

<i>O_COSH</i>	cosh(x)
---------------	---------

<i>O_SINH</i>	sinh(x)
---------------	---------

<i>O_TANH</i>	tanh(x)
<i>O_EXP</i>	exp(x)
<i>O_LOG10</i>	log10(x)
<i>O_PARENTHESIS</i>	This operator represents the "strong" parentheses of FORTRAN and C. It prevents the back end from performing certain mathematically correct, but floating point incorrect optimizations. E.g. in the expression "(a*2.4)/2.0", the back end is not allowed constant fold the expression into "a*1.2".

<i>Binary Operator</i>	<i>C equivalent</i>
------------------------	---------------------

<i>O_PLUS</i>	+
<i>O_MINUS</i>	-
<i>O_TIMES</i>	*
<i>O_DIV</i>	/
<i>O_MOD</i>	%
<i>O_AND</i>	&
<i>O_OR</i>	
<i>O_XOR</i>	^
<i>O_RSHIFT</i>	>>
<i>O_LSHIFT</i>	<<
<i>O_COMMA</i>	,
<i>O_TEST_TRUE</i>	( x & y ) != 0
<i>O_TEST_FALSE</i>	( x & y ) == 0
<i>O_EQ</i>	==
<i>O_NE</i>	!=
<i>O_GT</i>	>
<i>O_LE</i>	<=
<i>O_LT</i>	<
<i>O_GE</i>	>=
<i>O_POW</i>	pow( x, y )
<i>O_ATAN2</i>	atan2( x, y )



*O\_FMOD*                      fmod( x, y )

*O\_CONVERT*                  See below.

The binary *O\_CONVERT* operator is only available on the x86 code generator. It is used for based pointer operations (the result type of the *CGBinary* call must be a far pointer type). It effectively performs a *MK\_FP* operation with the left hand side providing the offset portion of the address, and the right hand side providing the segment value. If the right hand side expression is the address of a symbol, or the type of the expression is a far pointer, then the segment value for the symbol, or the segment value of the expression is used as the segment value after the *O\_CONVERT* operation.

**Short circuit operators    C equivalent**

*O\_FLOW\_AND*                &&

*O\_FLOW\_OR*                  ||

*O\_FLOW\_NOT*                !

**Control flow operators    C equivalent**

*O\_GOTO*                      goto label;

*O\_LABEL*                    label;;

*O\_IF\_TRUE*                  if( x ) goto label;

*O\_IF\_FALSE*                if( !(x) ) goto label;

*O\_INVOKE\_LABEL*          GOSUB (Basic)

*O\_LABEL\_RETURN*        RETURN (Basic)

The type passed into a CG routine is used by the back end as the type for the resulting node. If the node is an operator node (*CGBinary*, *CGUnary*) the back end will convert the operands to the result type before performing the operation. If the type *TY\_DEFAULT* is passed, the code generator will use default conversion rules to determine the resulting type of the node. These rules are the same as the ANSI C value preserving rules, with the exception that characters are not promoted to integers before doing arithmetic operations.

For example, if a node of type *TY\_UINT\_2* and a node of type *TY\_INT\_4* are to be added, the back end will automatically convert the operands to *TY\_INT\_4* before performing the addition. The resulting node will have type *TY\_INT\_4*.



---

# Leaf Nodes

## ***cg\_name CGInteger( signed\_32 val, cg\_type type )***

Create an integer constant leaf node.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>val</i>	The integral value.
<i>type</i>	An integral type.

## ***cg\_name CGInt64( signed\_64 val, cg\_type type )***

Create an 64-bit integer constant leaf node.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>val</i>	The 64-bit integer value.
<i>type</i>	An integral type.

## ***cg\_name CGFloat( char \*num, cg\_type type )***

Create a floating-point constant leaf node.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>num</i>	A NULL terminated E format string. (-1.23456E-102)
<i>type</i>	A floating point type.

## ***cg\_name CGFName( cg\_sym\_handle sym, cg\_type type )***

Create a leaf node representing the address of the back\_handle associated with **sym**. If sym represents an automatic variable or a parameter, CGAutoDecl or CGParmDecl must be called before this routine is first used.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>sym</i>	The front end symbol.
<i>type</i>	The type to be associated with the value of the symbol.

### ***cg\_name CGBackName( back\_handle bck, cg\_type type )***

Create a leaf node which represents the address of the back\_handle.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>bck</i>	A back handle.
------------	----------------

<i>type</i>	The type to be associated with the <b>value</b> of the symbol.
-------------	--

### ***cg\_name CGTempName( temp\_handle temp, cg\_type type )***

Create a leaf node which yields the address of the temp\_handle.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>temp</i>	A temp_handle.
-------------	----------------

<i>type</i>	The type to be associated with the <b>value</b> of the symbol.
-------------	--

---

# Assignment Operations

***cg\_name CGAssign( cg\_name dest, cg\_name src, cg\_type type )***

Create an assignment node.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>dest</i>	The destination address.
-------------	--------------------------

<i>src</i>	The source value.
------------	-------------------

<i>type</i>	The type to which the destination address points.
-------------	---

<i>Returns</i>	The value of the right hand side.
----------------	-----------------------------------

***cg\_name CGLVAssign( cg\_name dest, cg\_name src, cg\_type type )***

Like CGAssign, but yields the address of the destination.

***cg\_name CGPreGets( cg\_op op, cg\_name dest, cg\_name src, cg\_type type )***

Used for the C expressions  $a += b$ ,  $a /= b$ .

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>op</i>	The arithmetic operator to be used.
-----------	-------------------------------------

<i>dest</i>	The address of the destination.
-------------	---------------------------------

<i>src</i>	The value of the right hand side.
------------	-----------------------------------

<i>type</i>	The type to which the destination address points.
-------------	---

<i>Returns</i>	The value of the left hand side.
----------------	----------------------------------

***cg\_name CGLVPreGets( cg\_op op, cg\_name dest, cg\_name src, cg\_type type )***

Like CGPreGets, but yields the address of the destination.

### ***cg\_name CGPostGets( cg\_op op, cg\_name dest, cg\_name src, cg\_type type )***

Used for the C expressions a++, a--. No automatic scaling is done for pointers.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>op</i>	The operator.
-----------	---------------

<i>dest</i>	The address of the destination
-------------	--------------------------------

<i>src</i>	The value of the increment.
------------	-----------------------------

<i>type</i>	The type of the destination.
-------------	------------------------------

<i>Returns</i>	The value of the left hand side before the operation occurs.
----------------	--

---

# Arithmetic/logical operations

***cg\_name CGBinary( cg\_op op, cg\_name left, cg\_name right, cg\_type type )***

Binary operations. No automatic scaling is done for pointer operations.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>op</i>	The operator.
<i>left</i>	The value of the left hand side.
<i>right</i>	The value of the right hand side.
<i>type</i>	The result type.
<i>Returns</i>	The value of the result.

***cg\_name CGUnary( cg\_op op, cg\_name name, cg\_type type )***

Unary operations.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>op</i>	The operator.
<i>name</i>	The value of operand.
<i>type</i>	The result type.
<i>Returns</i>	The value of the result.

***cg\_name CGIndex( cg\_name name, cg\_name by, cg\_type type, cg\_type ptype )***

Obsolete. Do not use.





---

# Procedure calls

***call\_handle CGInitCall( cg\_name name, cg\_type type, cg\_sym\_handle aux\_info )***

Initiate a procedure call.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>name</i>	The address of the routine to call.
-------------	-------------------------------------

<i>type</i>	The return type of the routine.
-------------	---------------------------------

<i>aux_info</i>	A handle which the back end may passed to FEAuxInfo to determine the attributes of the call.
-----------------	--

<b><i>Returns</i></b>	A <b>call_handle</b> to be passed to the following routines.
-----------------------	--

***void CGAddParm( call\_handle call, cg\_name name, cg\_type type )***

Add a parameter to a call\_handle. The order of parameters is defined by the order in which they are passed to this routine. Parameters should be added in right to left order, as defined by the procedure call.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>call</i>	A call_handle.
-------------	----------------

<i>name</i>	The value of the parameter.
-------------	-----------------------------

<i>type</i>	The type of the parameter. This type will be passed to FEParamType to determine the actual type to be used when passing the parameter. For instance, characters are usually passes as integers in C.
-------------	--

***cg\_name CGCall( call\_handle call )***

Turn a call\_handle into a cg\_name by performing the call. This may be immediately followed by an optional addition operation, to reference a field in a structure return value. An indirection operator must immediately follow, even if the function has no return value.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>call</i>	A call_handle.
-------------	----------------

<b><i>Returns</i></b>	The address of the function return value.
-----------------------	---



---

# Comparison/short-circuit operations

***cg\_name CGCompare( cg\_op op, cg\_name left, cg\_name right, cg\_type type )***

Compare two values.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>op</i>	The comparison operator.
-----------	--------------------------

<i>left</i>	The value of the left hand side.
-------------	----------------------------------

<i>right</i>	The value of the right hand side.
--------------	-----------------------------------

<i>type</i>	The type to which to convert the operands to before performing comparison.
-------------	--

<b><i>Returns</i></b>	A TY_BOOLEAN cg_name, which may be passed to a control flow CG routine, or used in an expression as an integral value.
-----------------------	--



---

# Control flow operations

## ***cg\_name CGFlow( cg\_op op, cg\_name left, cg\_name right )***

Perform short-circuit boolean operations.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>op</i>	An operator.
-----------	--------------

<i>left</i>	A TY_BOOLEAN or integral cg_name.
-------------	-----------------------------------

<i>right</i>	A TY_BOOLEAN or integral cg_name, or NULL if op is O_FLOW_NOT.
--------------	--

<i>Returns</i>	A TY_BOOLEAN cg_name.
----------------	-----------------------

## ***cg\_name CGChoose( cg\_name sel, cg\_name n1, cg\_name n2, cg\_type type )***

Used for the C expression **sel** ? **n1** : **n2**.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>sel</i>	A TY_BOOLEAN or integral cg_name used as the selector.
------------	--

<i>n1</i>	The value to return if <b>sel</b> is non-zero.
-----------	--

<i>n2</i>	The value to return if <b>sel</b> is zero.
-----------	--

<i>type</i>	The type to which convert the result.
-------------	---------------------------------------

<i>Returns</i>	The value of <b>n1</b> or <b>n2</b> depending upon the truth of <b>sel</b> .
----------------	--

## ***cg\_name CGWarp( cg\_name before, label\_handle label, cg\_name after )***

To be used for FORTRAN statement functions.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>before</i>	An arbitrary expression tree to be evaluated before <b>label</b> is called. This is used to assign values to statement function arguments, which are usually temporaries allocated with CGTemp.
---------------	---

<i>label</i>	A label_handle to invoke via O_CALL_LABEL.
--------------	--

<i>after</i>	An arbitrary expression tree to be evaluated after <b>label</b> is called. This is used to retrieve the statement function return value.
--------------	--

**Returns**            The value of **after**. This can be passed to CGEval, to guarantee that nested statement functions are fully evaluated before their parameter variables are reassigned, as in f(1,f(2,3,4),5).

### **void CG3WayControl( cg\_name expr, label\_handle lt, label\_handle eq, label\_handle gt )**

Used for the FORTRAN arithmetic if statement. Go to label **lt**, **eq** or **gt** depending on whether **expr** is less than, equal to, or greater than zero.

<b>Parameter</b>	<b>Definition</b>
<i>expr</i>	The selector value.
<i>lt</i>	A label_handle.
<i>eq</i>	A label_handle.
<i>gt</i>	A label_handle.

### **void CGControl( cg\_op op, cg\_name expr, label\_handle lbl )**

Generate conditional and unconditional flow of control.

<b>Parameter</b>	<b>Definition</b>
<i>op</i>	a control flow operator.
<i>expr</i>	A TY_BOOLEAN expression if op is O_IF_TRUE or O_IF_FALSE. NULL otherwise.
<i>lbl</i>	The target label.

### **void CGBigLabel( back\_handle lbl )**

Generate a label which may be branched to from a nested procedure or used in NT structured exception handling. Don't use this call unless you *\*really\**, *\*really\** need to. It kills a lot of optimizations.

<b>Parameter</b>	<b>Definition</b>
<i>lbl</i>	A back_handle. There must be a front end symbol associated with this back handle.

### **void CGBigGoto( back\_handle value, int level )**

Generate a branch to a label in an outer procedure.

<b>Parameter</b>	<b>Definition</b>
<i>lbl</i>	A back_handle. There must be a front end symbol associated with this back handle.
<i>level</i>	The lexical level of the target label.

---

## Select and Switch statements.

The select routines are used as follows. CGSelOther should always be used even if there is no otherwise/default case.

```
end_label = BNewLabel();

sel_label = BNewLabel();
CGControl( O_GOTO, NULL, sel_label );
sel_handle = CGSelInit();

case_label = BNewLabel();
CGControl( O_LABEL, NULL, case_label );
CGSelCase( sel_handle, case_label, case_value );

    ... generate code associated with "case_value" here.

CGControl( O_GOTO, NULL, end_label ); // or else, fall through
other_label = BNewLabel();
CGControl( O_LABEL, NULL, other_label );
CGSelOther( sel_handle, other_label );

    ... generate "otherwise" code here

CGControl( O_GOTO, NULL, end_label ); // or else, fall through

CGControl( O_LABEL, NULL, sel_label );
CGSelect( sel_handle );

CGControl( O_LABEL, NULL, end_label );
```

### ***sel\_handle CGSelInit()***

Create a sel\_handle.

*Returns*            A sel\_handle to be passed to the following routines.

### ***void CGSelCase( sel\_handle sel, label\_handle lbl, signed\_32 val )***

Add a single value case to a select or switch.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>sel</i>	A sel_handle obtained from CGSelInit().
------------	---

<i>lbl</i>	The label to be associated with the case value.
------------	---

<i>val</i>	The case value.
------------	-----------------

### ***void CGSelRange( sel\_handle s, signed\_32 lo, signed\_32 hi, label\_handle lbl )***

Add a range of values to a select. All values are eventually converted into unsigned types to generate the switch code, so lo and hi must have the same sign.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>s</i>	A sel_handle obtained from CGSelInit().
<i>lo</i>	The lower bound of the case range.
<i>hi</i>	The upper bound of the case range.
<i>lbl</i>	The label to be associated with the case value.

### ***void CGSelOther( sel\_handle s, label\_handle lbl )***

Add the otherwise case to a select.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>s</i>	A sel_handle.
<i>lbl</i>	The label to be associated with the otherwise case.

### ***void CGSelect( sel\_handle s, cg\_name expr )***

Add the select expression to a select statement and generate code. This must be the last routine called for a given select statement. It invalidates the sel\_handle.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>s</i>	A sel_handle.
<i>expr</i>	The value we are selecting.

### ***void CGSelectRestricted( sel\_handle s, cg\_name expr, cg\_switch\_type allowed )***

Identical to CGSelect, except that only switch generation techniques corresponding to the set of allowed methods will be considered when determining how to produce code.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
<i>s</i>	A sel_handle.
<i>expr</i>	The value we are selecting.
<i>allowed</i>	The allowed methods of generating code. Must be a combination (non-empty) of the following bits:

CG\_SWITCH\_SCAN  
CG\_SWITCH\_BSEARCH



CG\_SWITCH\_TABLE



---

# Other

## ***void CGReturn( cg\_name name, cg\_type type )***

Return from a function. This is the last routine that may be called in any routine. Multiple return statements must be implemented with assignments to a temporary variable (CGTemp) and a branch to a label generated just before this routine call.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>name</i>	The value of the return value, or NULL.
-------------	---

<i>type</i>	The type of the return value. Use TY_INTEGER for void functions.
-------------	--

## ***cg\_name CGEval( cg\_name name )***

Evaluate this expression tree now and assign its value to a leaf node. Used to force the order of operations. This should only be used if necessary. Normally, the expression trees adequately define the order of operations. This usually used to force the order of parameter evaluation.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>name</i>	The tree to be evaluated.
-------------	---------------------------

<i>Returns</i>	A leaf node containing the value of the tree.
----------------	---

## ***void CGDone( cg\_name name )***

Generate the tree and throw away the resultant value. For example, CGAssign yields a value which may not be needed, but must be passed to this routine to cause the tree to be generated. This routine invalidates all cg\_name handles. After this routine has returned, any pending inline function expansions will have been performed.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>name</i>	The cg_name to be generated/discarded.
-------------	--

## ***void CGTrash( cg\_name name )***

Like CGDone, but used for partial expression trees. This routine does not cause all existing cg\_names to become invalid.

### ***cg\_type CGType( cg\_name name )***

Returns the type of the given cg\_name.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>name</i>	A cg_name.
-------------	------------

<i>Returns</i>	The type of the cg_name.
----------------	--------------------------

### ***cg\_name \*CGDuplicate( cg\_name name )***

Create two copies of a cg\_name.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>name</i>	The cg_name to be duplicated.
-------------	-------------------------------

<i>Returns</i>	A pointer to an array of two new cg_names, each representing the same value as the original. These should be copied out of the array immediately since subsequent calls to CGDuplicate will overwrite the array.
----------------	--

### ***cg\_name CGBitMask( cg\_name name, byte start, byte len, cg\_type type )***

Yields the address of a bit field. This address may not really be used except with an indirection operator or as the destination of an assignment operation.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>name</i>	The address of the integral variable containing the bit field.
-------------	--

<i>start</i>	The position of the least significant bit of the bit field. 0 indicates the least significant bit of the host data type.
--------------	--

<i>len</i>	The length of the bit field in bits.
------------	--------------------------------------

<i>type</i>	The integral type of the value containing the bit field.
-------------	--

<i>Returns</i>	The address of the bit field. To reference field2 in the following C structure for a little endian target, use start=4, len=5, and type=TY_INT_2. For a big endian target, start=7.
----------------	---

```
typedef struct {  
    short field1 : 4;  
    short field2 : 5;  
    short field3 : 7;  
}
```

***cg\_name CGVolatile( cg\_name name )***

Indicate that the given address points to a volatile location. This back end does not remember this information beyond this node in the expression tree. If an address points to a volatile location, the front end must call this routine each time that address is used.

***Parameter      Definition***

*name*              The address of the volatile location.

*Returns*           A new *cg\_name* representing the same value as *name*.

***cg\_name CGCallback( cg\_callback func, void \*ptr )***

When a callback node is inserted into the tree, the code generator will call the given function with the pointer as a parameter when it turns the node into an instruction. This can be used to retrieve order information about the placement of nodes in the instruction stream.

***Parameter      Definition***

*func*              This is a pointer to a function which is compatible with the C type "void (\*)(void \*)". This function will be called with the second parameter to this function as it's only parameter sometime during the execution of the CGDone call.

*ptr*                This will be a parameter to the function given as the first parameter.

***cg\_name CGPatchNode( patch\_handle hdl, cg\_type type )***

This prepares a leaf node to hold an integer constant which will be provided sometime during the execution of the CGDone call by means of a BEPatchInteger() call. It is an error to insert a patch node into the tree and not call BEPatchInteger().

***Parameter      Definition***

*hdl*                A handle for a patch allocated with BEPatch().

*type*              The actual type of the node. Must be an integer type.



---

# Data Generation

The following routines generate a data item described at the current location in the current segment, and increment the current location by the size of the generated object.

## ***void DGLabel( back\_handle bck )***

Generate the label for a given back\_handle.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>bck</i>	A back_handle.
------------	----------------

## ***void DGBackPtr( back\_handle bck, segment\_id segid, signed\_32 offset, cg\_type type )***

Generate a pointer to the label defined by the back\_handle.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>bck</i>	A back_handle.
------------	----------------

<i>segid</i>	The segment_id of the segment in which the label for <b>bck</b> will be defined if it has not already been passed to DGLabel.
--------------	---

<i>offset</i>	A value to be added to the generated pointer value.
---------------	---

<i>type</i>	The pointer type to be used.
-------------	------------------------------

## ***void DGFEPtr( cg\_sym\_handle sym, cg\_type type, signed\_32 offset )***

Generate a pointer to the label associated with **sym**.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>sym</i>	A cg_sym_handle.
------------	------------------

<i>type</i>	The pointer type to be used.
-------------	------------------------------

<i>offset</i>	A value to be added to the generated pointer value.
---------------	---

### ***void DGInteger( unsigned\_32 value, cg\_type type )***

Generate an integer.

<i>Parameter</i>	<i>Definition</i>
<i>value</i>	An integral value.
<i>type</i>	The integral type to be used.

### ***void DGInteger64( unsigned\_64 value, cg\_type type )***

Generate an 64-bit integer.

<i>Parameter</i>	<i>Definition</i>
<i>value</i>	An 64-bit integer value.
<i>type</i>	The integral type to be used.

### ***void DGFloat( char \*value, cg\_type type )***

Generate a floating-point constant.

<i>Parameter</i>	<i>Definition</i>
<i>value</i>	An E format string (ie: 1.2345e-134)
<i>type</i>	The floating point type to be used.

### ***void DGChar( char value )***

Generate a character constant. Will be translated if cross compiling.

<i>Parameter</i>	<i>Definition</i>
<i>value</i>	A character value.

### ***void DGString( char \*value, uint len )***

Generate a character string. Will be translated if cross compiling.

<i>Parameter</i>	<i>Definition</i>
<i>value</i>	Pointer to the characters to put into the segment. It is not necessarily a null terminated string.
<i>len</i>	The length of the string.



**`void DGBytes( unsigned_32 len, byte *src )`**

Generate raw binary data.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>src</i>	Pointer to the data.
------------	----------------------

<i>len</i>	The length of the byte stream.
------------	--------------------------------

**`void DGIBytes( unsigned_32 len, byte pat )`**

Generate the byte **pat**, **len** times.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>pat</i>	The pattern byte.
------------	-------------------

<i>len</i>	The number of times to repeat the byte.
------------	---

**`void DGUBytes( unsigned_32 len )`**

Generate **len** undefined bytes.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>len</i>	The size by which to increase the segment.
------------	--

**`void DGAlign( uint align )`**

Align the segment to an **align** byte boundary. Any slack bytes will have an undefined value.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>align</i>	The desired alignment boundary.
--------------	---------------------------------

**`unsigned_32 DGSeek( unsigned_32 where )`**

Seek to a location within a segment.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>where</i>	The location within the segment.
--------------	----------------------------------

<i>Returns</i>	The current location in the segment before the seek takes place.
----------------	--

### ***unsigned long DGTell()***

*Returns*            The current location within the segment.

### ***unsigned long DGBackTell( back\_handle bck )***

*Returns*            The location of the label within its segment. The label must have been previously generated via DGLabel.

---

# Front End Routines

## ***void FEGenProc( cg\_sym\_handle sym )***

This routine will be called when the back end is generating a tree and encounters a function call having the **call\_class** MAKE\_CALL\_INLINE. The front end must save its current state and start generating code for **sym**. FEGenProc calls may be nested if the code generator encounters an inline within the code for an inline function. The front end should maintain a state stack. It is up to the front end to prevent infinite recursion.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>sym</i>	The cg_sym_handle of the function to be generated.
------------	--

## ***back\_handle FEBack( cg\_sym\_handle sym )***

Return, and possibly allocate using BENewBack, a back handle for sym. See the example under "Back Handles" on page 13

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>sym</i>	
------------	--

<i>Returns</i>	A back_handle.
----------------	----------------

## ***segment\_id FESegID( cg\_sym\_handle sym )***

Return the segment\_id for symbol **sym**. A negative value may be returned to indicate that the symbol is defined in an unknown PRIVATE segment which has been defined in another module. If two symbols have the same negative value returned, the back end assumes that they are both defined in the same (unknown) segment.

<i>Parameter</i>	<i>Definition</i>
------------------	-------------------

<i>sym</i>	A cg_sym_handle.
------------	------------------

<i>Returns</i>	A segment_id.
----------------	---------------

## ***char \*FEModuleName()***

<i>Returns</i>	A null terminated string which is the name of the module being compiled. This is usually the file name with path and extension information stripped.
----------------	--

### ***char FEStackCheck( cg\_sym\_handle sym )***

*Returns*            1 if stack checking required for this routine

### ***unsigned FELexLevel( cg\_sym\_handle sym )***

*Returns*            The lexical level of routine **sym**. This must be zero for all languages except Pascal. In Pascal, 1 indicates the level of the main program. Each nested procedures adds an additional level.

### ***char \*FENAME( cg\_sym\_handle sym )***

*Returns*            A NULL terminated character string which is the name of sym. A null string should be returned if the symbol has no name. NULL should never be returned.

### ***char \*FEExtName( cg\_sym\_handle sym, int request )***

*Returns*            A various kind in dependency on request parameter.

#### ***Request parameter Returns***

*EXTN\_BASENAME* NULL terminated character string which is the name of sym. A null string should be returned if the symbol has no name. NULL should never be returned.

*EXTN\_PATTERN* NULL terminated character string which is the pattern for symbol name decoration. '\*' is replaced by symbol name. '^' is replaced by its upper case equivalent. '!' is replaced by its lower case equivalent. '#' is replaced by '@nnn' where nnn is decimal number representing total size of all function parameters. If an '\' is present, the character following is used literally.

*EXTN\_PRMSIZE* Returns int value which represent size of all parameters when symbol is function.

### ***cg\_type FEParmType( cg\_sym\_handle func, cg\_sym\_handle parm, cg\_type type )***

*Returns*            The type to which to promote an argument with a given type before passing it to a procedure. Type will be a dealiased type.

### ***int FETTrue()***

*Returns*            The value of TRUE. This is normally 1.

### ***char FEMoreMem( size\_t size )***

Release memory for the back end to use.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>size</i>	is the amount of memory required
-------------	----------------------------------

*Returns*            1 if at least **size** bytes were released. May always return 0 if memory is not a scarce resource in the host environment.

### ***dbg\_type FEDbgType( cg\_sym\_handle sym )***

*Returns*            The dbg\_type handle for the symbol **sym**.

### ***fe\_attr FEAttr( cg\_sym\_handle sym )***

Return symbol attributes for **sym**. These are bits combinable with the bit-wise or operator |.

#### ***Parameter      Definition***

*sym*                A cg\_sym\_handle.

#### ***Return value      Definition***

*FE\_PROC*            A procedure.

*FE\_STATIC*        A static or external symbol.

*FE\_GLOBAL*        Is a global (extern) symbol.

*FE\_IMPORT*        Needs to be imported.

*FE\_CONSTANT*    The symbol is read only.

*FE\_MEMORY*        This automatic variable needs a memory location.

*FE\_VISIBLE*       Accessible outside this procedure?

*FE\_NOALIAS*      No pointers point to this symbol.

*FE\_UNIQUE*        This symbol should have an address which is different from all other symbols with the FE\_UNIQUE attribute.

*FE\_COMMON*        There might be multiple definitions of this symbol in a program, and it should be generated in such a way that all versions of the symbol are merged into one copy by the linker.

*FE\_ADDR\_TAKEN*    The symbol has had it's address taken somewhere in the program (not necessarily visible to the code generator).

*FE\_VOLATILE*       The symbol is "volatile" (in the C language sense).

*FE\_INTERNAL*      The symbol is not at file scope.

### ***void FEMessage( msg\_class msg, void \*extra )***

Relays information to the front end.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>msg</i>	Defined below.
------------	----------------

<i>extra</i>	Extra information. The type and meaning depends on the value of <b>msg</b> and is indicated below.
--------------	--

<i>MSG_INFO_FILE</i>	Informational message about file. <i>extra</i> (void) is ignored.
----------------------	---

<i>MSG_CODE_SIZE</i>	Code size. <i>Extra</i> (int) is the size of the generated code.
----------------------	--

<i>MSG_DATA_SIZE</i>	Data size. <i>Extra</i> (int) is the size of the generated data.
----------------------	--

<i>MSG_ERROR</i>	A back end error message. <i>Extra</i> (char *) is the error message.
------------------	---

<i>MSG_FATAL</i>	A fatal code generator error. <i>Extra</i> (char *) is the reason for the fatal error. The front end should issue this message and exit immediately to the system.
------------------	--

<i>MSG_INFO_PROC</i>	Informational message about current procedure. <i>Extra</i> (char *) is a message.
----------------------	--

<i>MSG_BAD_PARM_REGISTER</i>	Invalid parameter register returned from FEAuxInfo. <i>Extra</i> (int) is position of the offending parameter.
------------------------------	--

<i>MSG_BAD_RETURN_REGISTER</i>	Invalid return register returned from FEAuxInfo. <i>Extra</i> (aux_handle) is the offending aux_handle.
--------------------------------	---

<i>MSG_REGALLOC_DIED</i>	The register alloc ran out of memory. <i>Extra</i> (cg_sym_handle) is the procedure which was not fully optimized.
--------------------------	--

<i>MSG_SCOREBOARD_DIED</i>	The register scoreboard ran out of memory. <i>Extra</i> (cg_sym_handle) is the procedure which was not fully optimized.
----------------------------	---

<i>MSG_PEEPHOLE_FLUSHED</i>	Peep hole optimizer flushed due to lack of memory. (void)
-----------------------------	---

<i>MSG_BACK_END_ERROR</i>	BAD NEWS! Internal compiler error. <i>Extra</i> (int) is an internal error number.
---------------------------	--

<i>MSG_BAD_SAVE</i>	Invalid register modification information return from FEAuxInfo. <i>Extra</i> (aux_handle) is the offending aux_handle.
---------------------	---

<i>MSG_WANT_MORE_DATA</i>	The back end wants more data space. <i>Extra</i> (int) is amount of additional memory needed to run. (DOS real mode hosts only).
---------------------------	--

<i>MSG_BLIP</i>	Blip. Let the world know we're still alive by printing a dot on the screen. This is called approximately every 4 seconds during code generation. (void)
-----------------	---

<i>MSG_BAD_LINKAGE</i>	Cannot resolve linkage conventions. 370 only. (sym)
------------------------	---

<i>MSG_SCHEDULER_DIED</i>	Instruction scheduler ran out of memory. Extra (cg_sym_handle) is the procedure which was not fully optimized.
<i>MSG_NO_SEG_REGS</i>	(Only occurs in the x86 version). The cg_switches did not allow any segment registers to float, but the user has requested a far pointer indirection. Extra (cg_sym_handle) is the procedure which contained the far pointer usage.
<i>MSG_SYMBOL_TOO_LONG</i>	Given symbol is too long and is truncated to maximum permitted length for current module output format. Extra (cg_sym_handle) is the symbol which was truncated.

## **void \*FEAuxInfo( void \*extra, aux\_class class )**

relay information to back end

### **Parameter      Definition**

<i>extra</i>	Extra information. Its type and meaning is determined by the value of class.
<i>class</i>	Defined below.

### **Parameters**

### **Return Value**

( <i>cg_sym_handle</i> , <i>AUX_LOOKUP</i> )	aux_handle - given a cg_sym_handle, return an aux_handle.
( <i>aux_handle</i> , <i>CALL_BYTES</i> )	byte_seq * - A pointer to bytes to be generated instead of a call, or NULL if a call is to be generated. <pre>typedef struct byte_seq {     char    length;     char    data[ 1 ]; } byte_seq;</pre>
( <i>aux_handle</i> , <i>CALL_CLASS</i> )	call_class * - returns call_class of the given aux_handle. See definitions below.
( <i>short</i> , <i>FREE_SEGMENT</i> )	short - A free segment value which is free memory for the code generator to use. The first word at segment:0 is the size of the free memory in bytes. (DOS real mode host only)
( <i>NULL</i> , <i>OBJECT_FILE_NAME</i> )	char * - The name of the object file to be generated.
( <i>aux_handle</i> , <i>PARAM_REGS</i> )	hw_reg_set[] - The set of register to be used as parameters.
( <i>aux_handle</i> , <i>RETURN_REG</i> )	hw_reg_set * - The return register. This is only called if the routine is declared to have the SPECIAL_RETURN call_class.
( <i>NULL</i> , <i>REVISION_NUMBER</i> )	int - Front end revision number. Must return II_REVISION.
( <i>aux_handle</i> , <i>SAVE_REGS</i> )	hw_reg_set * - Registers which are preserved by the routine.
( <i>cg_sym_handle</i> , <i>SHADOW_SYMBOL</i> )	cg_sym_handle - An alternate handle for a symbol. Required for

FORTTRAN. Usually implemented by turning on the LSB of a pointer or MSB of an integer.

- ( *NULL*, *SOURCE\_NAME* )      char \* - The name of the source file to be put into the object file.
  
- ( *cg\_sym\_handle*, *TEMP\_LOC\_NAME* )      Return one of TEMP\_LOC\_NO, TEMP\_LOC\_YES, TEMP\_LOC\_QUIT. After the back end has assigned stack locations to those temporaries which were not placed in registers, it begins to call FEAuxInfo with this request and passes in the *cg\_sym\_handle* for each of those temporaries. If the front end responds with TEMP\_LOC\_QUIT the back end will stop making TEMP\_LOC\_NAME requests. If the front end responds with TEMP\_LOC\_YES the back end will then perform a TEMP\_LOC\_TELL request (see next). If the front end returns TEMP\_LOC\_NO the back end moves onto the next *cg\_sym\_handle* in its list.
  
- ( *int*, *TEMP\_LOC\_TELL* )      Returns nothing. The 'int' value passed in is the relative position on the stack for the temporary identified by the *cg\_sym\_handle* passed in from the previous TEMP\_LOC\_NAME. The value for an individual temporary has no meaning, but the difference between two of the values is the number of bytes between the addresses of the temporaries on the stack.
  
- ( *void \**, *NEXT\_DEPENDENCY* )      Returns the handle of the next dependency file for which information is available. To start the list off, the back end passes in NULL for the dependency file handle.
  
- ( *void \**, *DEPENDENCY\_TIMESTAMP* )      Given the dependency file handle from the last NEXT\_DEPENDENCY request, return pointer to an unsigned long containing a timestamp value for the dependency file.
  
- ( *void \**, *DEPENDENCY\_NAME* )      Given the dependency file handle from the last NEXT\_DEPENDENCY request, return a pointer to a string containing the name for the dependency file.
  
- ( *NULL*, *SOURCE\_LANGUAGE* )      Returns a pointer to a string which identifies the source language of the pointer. E.g. "C" for C, "FORTRAN" for FORTRAN, "CPP" for C++.
  
- ( *cg\_sym\_handle*, *DEFAULT\_IMPORT\_RESOLVE* )      Only called for imported symbols. Returns a *cg\_sym\_handle* for another imported symbol which the reference should be resolved to if certain conditions are met (see IMPORT\_TYPE request). If NULL or the original *cg\_sym\_handle* is returned, there is no default import resolution symbol.
  
- ( *int*, *UNROLL\_COUNT* )      Returns a user-specified unroll count, or 0 if the user did not specify an unroll count. The parameter is the nesting level of the loop for which the request is being made. Loops which are not contained inside of other loops are nesting level 1. If this function returns a non-zero value, the loop in question will be unrolled that many times (there will be (count + 1) copies of the body).



<b><i>x86 Parameters</i></b>	<b><i>Return value</i></b>
( <i>NULL</i> , <i>CODE_GROUP</i> )	char * - The name of the code group.
( <i>aux_handle</i> , <i>STRETURN_REG</i> )	hw_reg_set * - The register which points to a structure return value. Only called if the routine has the SPECIAL_STRUCT_RETURN attribute.
( void *, <i>NEXT_IMPORT</i> )	void * (See notes at end) - A handle for the next symbol to generate a reference to in the object file.
( void*, <i>IMPORT_NAME</i> )	char * - The EXTDEF name to generate given a handle
( void *, <i>NEXT_IMPORT_S</i> )	void * (See notes at end) - A handle for the next symbol to generate a reference to in the object file.
( void*, <i>IMPORT_NAME_S</i> )	Returns a cg_sym_handle. The EXTDEF name symbol reference to generate given a handle.
( void*, <i>NEXT_LIBRARY</i> )	void * (See notes at end) - Handle for the next library required
( void*, <i>LIBRARY_NAME</i> )	char * - The library name to generate given a handle
( <i>NULL</i> , <i>DATA_GROUP</i> )	char * - Used to name DGROUP exactly. NULL means use no group at all.
( <i>segment_id</i> , <i>CLASS_NAME</i> )	NULL - Used to name the class of a segment.
( <i>NULL</i> , <i>USED_8087</i> )	NULL - Indicate that 8087 instructions were generated.
( <i>NULL</i> , <i>STACK_SIZE_8087</i> )	int - How many 8087 registers are reserved for stack.
( <i>NULL</i> , <i>CODE_LABEL_ALIGNMENT</i> )	char * - An array x, such that x[i] is the label alignment requirements for labels nested within i loops.
( <i>NULL</i> , <i>PROEPI_DATA_SIZE</i> )	int - How much stack is reserved for the prolog hook routine.
( <i>cg_sym_handle</i> , <i>IMPORT_TYPE</i> )	Returns IMPORT_IS_WEAK, IMPORT_IS_LAZY, IMPORT_IS_CONDITIONAL. If the DEFAULT_IMPORT_RESOLVE request returned a default resolution symbol the back end then performs an IMPORT_TYPE request to determine the type of the resolution. IMPORT_IS_WEAK generates a weak import (the symbol is not searched for in libraries). IMPORT_IS_LAZY generates a lazy import (the symbol is searched for in libraries). IMPORT_IS_CONDITIONAL is used for eliminating unused virtual functions. The default symbol resolution is used if none of the conditional symbols are referenced/defined by the program. The back end is informed of the list of conditional symbols by the following three aux requests. IMPORT_IS_CONDITIONAL_PURE is used for eliminating unused pure virtual functions.

- ( *cg\_sym\_handle*, *CONDITIONAL\_IMPORT* )  
Returns void \*. Once the back end determines that it has a conditional import, it performs this request to get a conditional list handle which is the head of the list of conditional symbols.
- ( void \*, *CONDITIONAL\_SYMBOL* )  
Returns a *cg\_sym\_handle*. Give an conditional list handle, return the front end symbol associated with it.
- ( void \*, *NEXT\_CONDITIONAL* ) Given an conditional list handle, return the next conditional list handle. Return NULL at the end of the list.
- ( *aux\_handle*, *VIRT\_FUNC\_REFERENCE* )  
Returns void \*. When performing an indirect function call, the back end invokes FEAuxInfo passing the *aux\_handle* supplied with the CGInitCall. If the indirect call is referencing a C++ virtual function, the front end should return a magic cookie which is the head of a list of virtual functions that might be invoked by this call. If it is not a virtual function invocation, return NULL.
- ( void \*, *VIRT\_FUNC\_NEXT\_REFERENCE* )  
Returns void \*. Given the magic cookie returned by the *VIRT\_FUNC\_REFERENCE* or a previous *VIRT\_FUNC\_NEXT\_REFERENCE*, return the next magic cookie in the list of virtual functions that might be referenced from this indirect call. Return NULL if at the end of the list.
- ( void \*, *VIRT\_FUNC\_SYM* )  
Returns *cg\_sym\_handle*. Given a magic cookie from a *VIRT\_FUNC\_REFERENCE* or *VIRT\_FUNC\_NEXT\_REFERENCE*, return the *cg\_sym\_handle* for that entry in the list of virtual functions that might be invoked.
- ( *segment\_id*, *PEGGED\_REGISTER* )  
Returns a pointer at a *hw\_reg\_set* or NULL. If the pointer is non-NULL and the *hw\_reg\_set* is not EMPTY, the *hw\_reg\_set* will indicate a segment register that is pegged (pointing) to the given *segment\_id*. The code generator will use this segment register in any references to objects in the segment. If the pointer is NULL or the *hw\_reg\_set* is EMPTY, the code generator uses the *cg\_switches* to determine if a segment register is pointing at the segment or if it will have to load one.

### ***Call Class***

*REVERSE\_PARMS*

### ***Meaning***

Reverse the parameter list.

*SUICIDAL*

Routine never returns.

*PARMS\_BY\_ADDRESS*

Pass parameters by reference.

*MAKE\_CALL\_INLINE*

Call should be inline. FEGenProc will be called for code sequence when required.

### ***x86 Call Class***

### ***Meaning***

<i>FAR</i>	Does routine require a far call/return.
<i>LOAD_DS_ON_CALL</i>	Load DS from DGROUP prior to call.
<i>CALLER_POPS</i>	Caller pops/removes parms from the stack.
<i>ROUTINE_RETURN</i>	Routine allocates structure return memory.
<i>SPECIAL_RETURN</i>	Routine has non-default return register.
<i>NO_MEMORY_CHANGED</i>	Routine modifies no visible statics.
<i>NO_MEMORY_READ</i>	Routine reads no visible statics.
<i>MODIFY_EXACT</i>	Routine modifies no parameter registers.
<i>SPECIAL_STRUCT_RETURN</i>	Routine has special struct return register.
<i>NO_STRUCT_REG_RETURNS</i>	Pass 2/4/8 byte structs on stack, as opposed to registers.
<i>NO_FLOAT_REG_RETURNS</i>	Return floats as structs.
<i>INTERRUPT</i>	Routine is an interrupt routine.
<i>NO_8087_RETURNS</i>	No return values in the 8087.
<i>LOAD_DS_ON_ENTRY</i>	Load ds with dgroup on entry.
<i>DLL_EXPORT</i>	Is routine an OS/2 export symbol?
<i>FAT_WINDOWS_PROLOG</i>	Generate the real mode windows prolog code.
<i>GENERATE_STACK_FRAME</i>	Always generate a traceable prolog.
<i>EMIT_FUNCTION_NAME</i>	Emit the function name in front of the function in the code segment.
<i>GROW_STACK</i>	Emit a call to grow the stack on entry
<i>PROLOG_HOOKS</i>	Generate a prolog hook call.
<i>EPILOG_HOOKS</i>	Generate an epilog hook call.
<i>THUNK_PROLOG</i>	Generate a thunking prolog for routines calling 16 bit code.
<i>FAR16_CALL</i>	Performs a 16:16 call in the 386 compiler.
<i>TOUCH_STACK</i>	Certain people (who shall remain nameless) have implemented an operating system (which shall remain nameless) that can't be bothered figuring out whether a page reference is in the stack or not. This attribute forces the first reference to the stack (after a routine prologue has grown it) to be through the SS register.



---

# Debugging Information

These routines generate information about types, symbols, etc.

## ***void DBLineNum( uint no )***

Set the current source line number.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>no</i>	Is the current source line number.
-----------	------------------------------------

## ***void DBModSym( cg\_sym\_handle sym, cg\_type indirect )***

Define a symbol within the module (file scope).

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>sym</i>	is a front end symbol handle.
------------	-------------------------------

<i>indirect</i>	is the type of indirection needed to obtain the value
-----------------	---

## ***void DBObject( dbg\_type tipe, dbg\_loc loc )***

Define a function as being a member function of a C++ class, and identify the type of the class and the location of the object being manipulated. This function may only be done after the DBModSym for the function.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>tipe</i>	is the debug type of the class that the function is a member of.
-------------	--

<i>loc</i>	is a location expression that evaluates to the address of the object being manipulated by the function (the contents of the 'this' pointer in C++). This parameter is NULL if the routine is a static member function.
------------	--

## ***void DBLocalSym( cg\_sym\_handle sym, cg\_type indirect )***

As DBModSym but for local (routine scope) symbols.

### ***void DBGenSym( cg\_sym\_handle sym, dbg\_loc loc, int scoped )***

Define a symbol either with module scope ('scoped' == 0) or within the current block ('scoped' != 0). This routine supersedes both DBLocalSym and DBModuleSym. The 'loc' parameter is a location expression (explained later) which allows an arbitrary sequence of operations to locate the storage for the symbol.

<b><i>Parameter</i></b>	<b><i>Definition</i></b>
-------------------------	--------------------------

<i>sym</i>	is a front end symbol handle.
------------	-------------------------------

<i>loc</i>	the location expression which is evaluated by the debugger to locate the lvalue of the symbol.
------------	--

<i>scoped</i>	whether the symbol is file scoped or not.
---------------	---

### ***void DBBegBlock()***

Open a new scope level.

### ***void DBEndBlock()***

Close the current scope level.

### ***dbg\_type DBScalar( char \*name, cg\_type tipe )***

Defines the string **name** to have type **tipe**.

### ***dbg\_type DBScope( char \*name )***

define a symbol which "scopes" subsequent symbols. In C, the keywords **enum**, **union**, **struct** may perform this function as in **struct foo**.

### ***dbg\_name DBBegName( const char \*name, dbg\_type scope )***

start a type name whose type is yet undetermined

### ***dbg\_type DBForward( dbg\_name name )***

declare a type to be a forward reference

### ***dbg\_type DBEndName( dbg\_name name, dbg\_type tipe )***

complete the definition of a type name.

***dbg\_type DBArray( dbg\_type index, dbg\_type base )***

define a C array type

***dbg\_type DBIntArray( unsigned\_32 hi, dbg\_type base )***

define a C array type

***dbg\_type DBSubRange( signed\_32 lo, signed\_32 hi, dbg\_type base )***

define an integer range type

***dbg\_type DBPtr( cg\_type ptr\_type, dbg\_type base )***

declare a pointer type

***dbg\_type DBBasedPtr( cg\_type ptr\_type, dbg\_type base, dbg\_loc seg\_loc )***

declare a based pointer type. The 'seg\_loc' parameter is a location expression which evaluates to the base address for the pointer after the indirection has been performed. Before the location expression is evaluated, the current lvalue of the pointer symbol associated with this type is pushed onto the expression stack (needed for based on self pointers).

***dbg\_struct DBBegStruct()***

start a structure type definition

***void DBAddField( dbg\_struct st, unsigned\_32 off, char \*nm, dbg\_type base )***

add a field to a structure

***void DBAddBitField( dbg\_struct st, unsigned\_32 off, byte strt, byte len, char \*nm, dbg\_type base )***

add a bit field to a structure

***void DBAddLocField( dbg\_struct st, dbg\_loc loc, uint attr, byte strt, byte len, char \*nm, dbg\_type base )***

Add a field or bit field to a structure with a generalized location expression 'loc'. The location expression should assume the the address of the base of the structure has already been pushed onto the debugger's evaluation stack. The 'attr' parameter contains a zero or more of the following attributes or'd together:

<i>Attribute</i>	<i>Definition</i>
------------------	-------------------

<i>FIELD_ATTR_INTERNAL</i>	the field is internally generated by the compiler and would not be normally visible to the user.
----------------------------	--

*FIELD\_ATTR\_PUBLIC* the field has the C++ 'public' attribute.

*FIELD\_ATTR\_PROTECTED* the field has the C++ 'protected' attribute.

*FIELD\_ATTR\_PRIVATE* the field has the C++ 'private' attribute.

If the field being described is *\_not\_* a bit field, the 'len' parameter should be set to zero.

### ***void DBAddInheritance( dbg\_struct st, dbg\_type inherit, dbg\_loc adjust )***

Add the fields of an inherited structure to the current structure being defined.

<i>Parameter</i>	<i>Definition</i>
<i>st</i>	the <i>dbg_struct</i> handle for the structure currently being defined.
<i>inherit</i>	the <i>dbg_type</i> of a previously defined structure which is being inherited.
<i>adjust</i>	a location expression which evaluates to a value which is the amount to adjust the field offsets by in the inherited structure to access them in the current structure. The base address of the symbol associated with the structure type is pushed onto the location expression stack before the expression is evaluated.

### ***dbg\_type DBEndStruct( dbg\_struct st )***

end a structure definition

### ***dbg\_enum DBBegEnum( cg\_type tipe )***

begin defining an enumerated type

### ***void DBAddConst( dbg\_enum en, const char \*nm, signed\_32 val )***

add a symbolic constant to an enumerated type

### ***void DBAddConst64( dbg\_enum en, const char \*nm, signed\_64 val )***

add a symbolic 64-bit integer constant to an enumerated type

### ***dbg\_type DBEndEnum( dbg\_enum en )***

finish declaring an enumerated type



***dbg\_proc DBBegProc( cg\_type call\_type, dbg\_type ret )***

begin the a current procedure

***void DBAddParm( dbg\_proc pr, dbg\_type tipe )***

declare a parameter to the procedure

***dbg\_type DBEndProc( proc\_list \*pr )***

end the current procedure

***dbg\_type DBFtnType( char \*name, dbg\_ftn\_type tipe )***

declare a fortran COMPLEX type

***dbg\_type DBCharBlock( unsigned\_32 len )***

declare a type to be a block of length **len** characters

***dbg\_type DBIndCharBlock( back\_handle len, cg\_type len\_type, int off )***

declare a type to be a block of characters. The length is found at run-time at back\_handle **len** + offset **off**. The integral type of the back\_handle location is **len\_type**

***dbg\_type DBLocCharBlock( dbg\_loc loc, cg\_type len\_type )***

declare a type to be a block of characters. The length is found at run-time at the address specified by the location expression **loc**. The integral type of the location is **len\_type**

***dbg\_type DBFtnArray( back\_handle dims, cg\_type lo\_bound\_tipe, cg\_type num\_elts\_tipe, int off, dbg\_type base )***

define a FORTRAN array dimension slice. **dims** is a back handle + offset **off** which will point to a structure at run-time. The structure contains the array low bound (type **lo\_bound\_tipe**) followed by the number of elements (type **num\_elts\_tipe**). **base** is the element type of the array.

***dbg\_type DBDereference( cg\_type ptr\_type, dbg\_type base )***

declare a type to need an implicit de-reference to retrieve the value (for FORTRAN parameters)

**Notes:** This routine has been superceded by the use of location expressions.

### ***dbg\_loc DBLocInit( void )***

create an initial empty location expression

### ***dbg\_loc DBLocSym( dbg\_loc loc, cg\_sym\_handle sym )***

push the address of 'sym' on to the expression stack

### ***dbg\_loc DBLocTemp( dbg\_loc loc, temp\_handle tmp )***

push the address of 'tmp' on to the expression stack

### ***dbg\_loc DBLocConst( dbg\_loc loc, unsigned\_32 val )***

push the constant 'val' on to the expression stack

### ***dbg\_loc DBLocOp( dbg\_loc loc, dbg\_loc\_op op, unsigned other )***

perform the following list of operations on the expression stack

<i>Operation</i>	<i>Definition</i>
------------------	-------------------

<i>DB_OP_POINTS</i>	take the top of the expression stack and use it as the address in an indirection operation. The result type of the operation is given by the 'other' parameter which must be a <i>cg_type</i> which resolves to either an <i>unsigned_16</i> , <i>unsigned_32</i> , a 16-bit far pointer, or a 32-bit far pointer.
---------------------	--

<i>DB_OP_ZEX</i>	zero extend the top of the stack. The 'other' parameter is a <i>cg_type</i> which is either 1 byte in size or 2 bytes in size. That size determines how much of the original top of stack value to leave untouched.
------------------	---

<i>DB_OP_XCHG</i>	exchange the top of stack value with the stack entry indexed by 'other'.
-------------------	--

<i>DB_OP_MK_FP</i>	take the top two entries on the stack. Make the second entry the segment value and the first entry the offset value of an address.
--------------------	--

<i>DB_OP_ADD</i>	add the top two stack entries together.
------------------	---

<i>DB_OP_DUP</i>	duplicate the top stack entry.
------------------	--------------------------------

<i>DB_OP_POP</i>	pop off (throw away) the top stack entry.
------------------	---

### ***void DBLocFini( dbg\_loc loc )***

the given location expression will not be used anymore.

### ***unsigned DBSrcFile( char \*fname )***

add the file name into the list of source files for positon info, return handle to this name

**Notes:** Handle 0 is reserved for base source file name and is added by BE automatically during initialization.

***void DBSrcCue( unsigned fno, unsigned line, unsigned col )***  
add source position info for the appropriate source file



---

# Registers

The `hw_reg_set` type is an abstract data type capable of representing any combination of machine registers. It must be manipulated using the following macros. A parameter `c`, `c1`, `c2`, etc. indicate a register constant such as `HW_EAX` must be used. Anything else must be a variable of type `hw_reg_set`.

The following are used for static initialization.

```
HW_D_1( c1 )
HW_NotD_1( c1 )
HW_D_2( c1, c2 )
HW_NotD_2( c1, c2 )
HW_D_3( c1, c2, c3 )
HW_NotD_3( c1, c2, c3 )
HW_D_4( c1, c2, c3, c4 )
HW_NotD_4( c1, c2, c3, c4 )
HW_D_5( c1, c2, c3, c4, c5 )
HW_NotD_5( c1, c2, c3, c4, c5 )
HW_D( c1 )
HW_NotD( c1 )
hw_reg_set regs[] = {
    /* the EAX register */
    HW_D( HW_EAX ),
    /* all registers except EDX and EBX */
    HW_NotD_2( HW_EDX, HW_EBX )
};
```

The following are to build registers dynamically.

<i>Macro</i>	<i>Usage</i>
<i>HW_CEqual( a, c )</i>	Is <b>a</b> equal to <b>c</b>
<i>HW_COvlap( a, c )</i>	Does <b>a</b> overlap with <b>c</b>
<i>HW_CSubset( a, c )</i>	Is <b>a</b> subset of <b>c</b>
<i>HW_CAsgn( dst, c )</i>	Assign <b>c</b> to <b>dst</b>
<i>HW_CTurnOn( dst, c )</i>	Turn on registers <b>c</b> in <b>dst</b> .
<i>HW_CTurnOff( dst, c )</i>	Turn off registers <b>c</b> in <b>dst</b> .
<i>HW_COnlyOn( a, c )</i>	Turn off all registers except <b>c</b> in <b>dst</b> .
<i>HW_Equal( a, b )</i>	Is <b>a</b> equal to <b>b</b>
<i>HW_Ovlap( a, b )</i>	Does <b>a</b> overlap with <b>b</b>
<i>HW_Subset( a, b )</i>	Is <b>a</b> subset of <b>b</b>

<i>HW_Asgn( dst, b )</i>	Assign <b>b</b> to <b>dst</b>
<i>HW_TurnOn( dst, b )</i>	Turn on registers <b>b</b> in <b>dst</b> .
<i>HW_TurnOff( dst, b )</i>	Turn off registers <b>b</b> in <b>dst</b> .
<i>HW_OnlyOn( dst, b )</i>	Turn off all registers except <b>b</b> in <b>dst</b> .

The following example selects the low order 16 bits of any register. that has a low part.

```
hw_reg_set low16( hw_reg_set reg )
{
    hw_reg_set low;

    HW_CAsgn( low, HW_EMPTY );
    HW_CTurnOn( low, HW_AX );
    HW_CTurnOn( low, HW_BX );
    HW_CTurnOn( low, HW_CX );
    HW_CTurnOn( low, HW_DX );
    if( HW_Ovlap( reg, low ) ) {
        HW_OnlyOn( reg, low );
    }
}
```

The following register constants are defined for all targets.

*HW\_EMPTY* The null register set.

*HW\_UNUSED* The set of unused register entries.

*HW\_FULL* All possible registers.

The following example yields the set of all valid machine registers.

```
hw_reg_set reg;

HW_CAsgn( reg, HW_FULL );
HW_CTurnOff( reg, HW_UNUSED );
```

---

# Miscellaneous

I apologize for my lack of consistency in this document. I use the terms function, routine, procedure interchangeably, as well as index, subscript - select, switch - parameter, argument - etc. I come from a multiple language background and will always be hopelessly confused.

The NEXT\_IMPORT/NEXT\_IMPORT\_S/NEXT\_LIBRARY are used as follows.

```
handle = NULL;
for( ;; ) {
    handle = FEAuxInfo( handle, NEXT_IMPORT );
    if( handle == NULL )
        break;
    do_something( FEAuxInfo( handle, IMPORT_NAME ) );
}
```

The FREE\_SEGMENT request is used as follows.

```
segment = 0;
for( ;; ) {
    segment = FEAuxInfo( segment, FREE_SEGMENT );
    if( segment == NULL )
        break;
    segment_size = *(short *)MK_FP( segment, 0 ) * 16;
    this_is_my_memory_now( MK_FP( segment, 0 ), segment_size );
}
```

The main line in Pascal is defined to be lexical level 1. Add 1 for each nested subroutine level. C style routines are defined to be lexical level 0.

The following types are defined by the code generator header files:

<i>Utility type</i>	<i>Definition</i>
<i>bool</i>	(unsigned char) 0 = false, non-0 = true.
<i>byte</i>	(unsigned char)
<i>int_8</i>	(signed char)
<i>int_16</i>	(signed short)
<i>int_32</i>	(signed long)
<i>signed_8</i>	(signed char)
<i>signed_16</i>	(signed short)
<i>signed_32</i>	(signed long)
<i>uint</i>	(unsigned)
<i>uint_8</i>	(unsigned char)

*uint\_16* (unsigned short)

*uint\_32* (unsigned long)

*unsigned\_8* (unsigned char)

*unsigned\_16* (unsigned short)

*unsigned\_32* (unsigned long)

*real* (float)

*reallong* (double)

*pointer* (void\*)

**Type**      **Definition**

*aux\_class* (enum) Passed as 2nd parameter to FEAuxInfo.

*aux\_handle* (void\*) A handle used as 1st parameter to FEAuxInfo.

*back\_handle* (void\*) A handle for a back end symbol table entry.

*byte\_seq* (struct) Passed to back end in response to CALL\_BYTES FEAuxInfo request.

*call\_class* (unsigned long) A set of combinable bits indicating the call attributes for a routine.

*call\_handle* (void\*) A handle to be used in CGInitCall, CGAddParm and CGCall.

*cg\_init\_info* (union) The return value of BEInit.

*cg\_name* (void\*) A handle for a back end expression tree node.

*cg\_op* (enum) An operator to be used in building expressions.

*cg\_switches* (unsigned\_32) A set of combinable bits indicating the code generator options.

*cg\_sym\_handle* (uint) A handle for a front end symbol table entry.

*cg\_type* (unsigned short) A code generator type.

*fe\_attr* (enum) A set of combinable bits indicating symbol attributes.

*hw\_reg\_set* (struct hw\_reg\_set) A structure representing a hardware register.

*label\_handle* (void\*) A handle for a code generator code label.

*linkage\_regs* (struct) For 370 linkage conventions.

*more\_cg\_types* (enum)

*msg\_class* (enum) The 1st parameter to FEMessage.



*proc\_revision* (enum) The 3rd parameter to BEInit.

*seg\_attr* (enum) A set of combinable bits indicate the attributes of a segment.

*segment\_id* (int) A segment identifier.

*sel\_handle* (void\*) A handle to be used in the CGSel calls.

*temp\_handle* (void\*) A handle for a code generator temporary.

***Misc Type      Definition***

*HWT* hw\_reg\_part

*hw\_reg\_part* (unsigned)

*dbg\_enum* (void\*)

*dbg\_fn\_type* (enum)

*dbg\_name* (void\*)

*dbg\_proc* (void\*)

*dbg\_struct* (void\*)

*dbg\_type* (unsigned short)

*predefined\_cg\_types* (enum)



## A. Pre-defined macros

The following macros are defined by the code generator include files.

HW\_D  
HW\_D\_1  
HW\_D\_2  
HW\_D\_3  
HW\_D\_4  
HW\_D\_5  
BIG\_CODE  
BIG\_DATA  
CALLER\_POPS  
CHEAP\_POINTER  
CHEAP\_WINDOWS  
CONST\_IN\_CODE  
CPU\_MASK  
C\_FRONT\_END  
DBG\_FWD\_TYPE  
DBG\_LOCALS  
DBG\_NIL\_TYPE  
DBG\_NUMBERS  
DBG\_TYPES  
DLL\_EXPORT  
DO\_FLOATING\_FIXUPS  
DO\_SYM\_FIXUPS  
EMIT\_FUNCTION\_NAME  
EPILOG\_HOOKS  
EZ\_OMF  
E\_8087  
FALSE  
FAR  
FAT\_WINDOWS\_PROLOG  
FIX\_SYM\_OFFSET  
FIX\_SYM\_RELOFF  
FIX\_SYM\_SEGMENT  
FLAT\_MODEL  
FLOATING\_DS  
FLOATING\_ES  
FLOATING\_FIXUP\_BYTE  
FLOATING\_FS  
FLOATING\_GS  
FLOATING\_SS  
FORTRAN\_ALIASING  
FORTRAN\_FRONT\_END  
FPU\_MASK  
FRONT\_END\_MASK

FUNCS\_IN\_OWN\_SEGMENTS  
GENERATE\_STACK\_FRAME  
GET\_CPU  
GET\_FPU  
GET\_WTK  
GROW\_STACK  
HWREG\_INCLUDED  
HW\_0  
HW\_1  
HW\_2  
HW\_3  
HW\_64  
HW\_Asgn  
HW\_CAsgn  
HW\_CEqual  
HW\_COMMA  
HW\_COnlyOn  
HW\_COvlap  
HW\_CSubset  
HW\_CTurnOff  
HW\_CTurnOn  
HW\_DEFINE\_COMPOUND  
HW\_DEFINE\_GLOBAL\_CONST  
HW\_DEFINE\_SIMPLE  
HW\_Equal  
HW\_ITER  
HW\_NotD  
HW\_NotD\_1  
HW\_NotD\_2  
HW\_NotD\_3  
HW\_NotD\_4  
HW\_NotD\_5  
HW\_OnlyOn  
HW\_Op1  
HW\_Op2  
HW\_Op3  
HW\_Op4  
HW\_Op5  
HW\_Ovlap  
HW\_Subset  
HW\_TurnOff  
HW\_TurnOn  
II\_REVISION  
INDEXED\_GLOBALS  
INS\_SCHEDULING  
INTERNAL\_DBG\_OUTPUT  
INTERRUPT  
I\_MATH\_INLINE  
LAST\_AUX\_ATTRIBUTE  
LAST\_CGSWITCH  
LAST\_TARG\_AUX\_ATTRIBUTE  
LAST\_TARG\_CGSWITCH  
LOAD\_DS\_ON\_CALL  
LOAD\_DS\_ON\_ENTRY

LOOP\_OPTIMIZATION  
MAKE\_CALL\_INLINE  
MAX\_POSSIBLE\_REG  
MIN\_OP  
MODIFY\_EXACT  
NEED\_STACK\_FRAME  
NO\_8087\_RETURNS  
NO\_CALL\_RET\_TRANSFORM  
NO\_FLOAT\_REG\_RETURNS  
NO\_MEMORY\_CHANGED  
NO\_MEMORY\_READ  
NO\_OPTIMIZATION  
NO\_STRUCT\_REG\_RETURNS  
NULL  
NULLCHAR  
O\_FIRST\_COND  
O\_FIRST\_FLOW  
O\_LAST\_COND  
O\_LAST\_FLOW  
PARMS\_BY\_ADDRESS  
PROLOG\_HOOKS  
RELAX\_ALIAS  
REVERSE\_PARMS  
ROUTINE\_RETURN  
SEG\_EXTRN\_FAR  
SET\_CPU  
SET\_FPU  
SET\_WTK  
SPECIAL\_RETURN  
SPECIAL\_STRUCT\_RETURN  
STANDARD\_INCLUDED  
SUICIDAL  
SYM\_FIXUP\_BYTE  
THUNK\_PROLOG  
TRUE  
TY\_HUGE\_CODE\_PTR  
USE\_32  
WINDOWS  
WTK\_MASK  
\_AL  
\_AX  
\_BL  
\_BP  
\_BX  
\_CG\_H\_INCLUDED  
\_CL  
\_CMS  
\_CX  
\_DI  
\_DL  
\_DX  
\_HOST\_INTEGER  
\_OS  
\_SI

`_TARG_AUX_SHIFT`  
`_TARG_CGSWITCH_SHIFT`  
`far`  
`huge`  
`interrupt`  
`near`  
`offsetof`

## ***B. Register constants***

The following register constants are defined for x86 targets.

HW\_AH  
HW\_AL  
HW\_BH  
HW\_BL  
HW\_CH  
HW\_CL  
HW\_DH  
HW\_DL  
HW\_SI  
HW\_DI  
HW\_BP  
HW\_SP  
HW\_DS  
HW\_ES  
HW\_CS  
HW\_SS  
HW\_ST0  
HW\_ST1  
HW\_ST2  
HW\_ST3  
HW\_ST4  
HW\_ST5  
HW\_ST6  
HW\_ST7  
HW\_FS  
HW\_GS  
HW\_AX  
HW\_BX  
HW\_CX  
HW\_DX  
HW\_EAX  
HW\_EBX  
HW\_ECX  
HW\_EDX  
HW\_ESI  
HW\_EDI  
HW\_ESP  
HW\_EBP

The following registers are defined for the Alpha AXP target.

HW\_R0-HW\_R31  
HW\_D0-HW\_D31

HW\_W0-HW\_W31  
HW\_B0-HW\_B31  
HW\_F0-HW\_F31

The following registers are defined for the PowerPC target.

HW\_R0-HW\_R31  
HW\_Q3-HW\_Q29  
HW\_D0-HW\_D31  
HW\_W0-HW\_W31  
HW\_B0-HW\_B31  
HW\_F0-HW\_F31

The following registers are defined for the MIPS32 target.

HW\_R0-HW\_R31  
HW\_Q2-HW\_Q24  
HW\_D0-HW\_D31  
HW\_W0-HW\_W31  
HW\_B0-HW\_B31  
HW\_F0-HW\_F31  
HW\_FD0-HW\_FD30



## C. Debugging Open Watcom Code Generator

If you want to use `vc.dbg` command, make sure you have a `tmp` directory in root of used filesystem (see `bld/cg/dumpio.c` for details).

Note: make a `s:\tmp` to facilitate debugging in `s:\brad` :) Yeah, it's a cheap and sleazy hack...

If you need to dump something and don't know the routine to call, try **"e/s Dump"** and see what pops up...

### Instructions

You can get a dump of instructions for current function via **DumpRange** anytime between **FixEdges** and start of **GenObject**.

You can dump an individual instruction via **DumpIns**

If you need live info for a basic block, find address and call **DumpABlk( block )**.

### Symbols

If you need to see a list of symbols, use **DumpSymTab**. To look at one symbol, use **DumpSym**.

### Tree Problems

Find the line number of a piece of source near the problem. Do a **"bif { edx == LINENUMBER } DBSrcCue"** to stop near that Go to **CGDone** in order to see what resulting tree is (**DumpTree**) If there is a problem with tree, but not with API calls, do to **DBSrcCue** as above and then break on next appropriate CG API call.

### Optimization Problems (Loopopts at all)

Find the ordinal of the problem function in the file (ie 4th function) Do a **"bcnt 4 FixEdges"** in order to stop on 4th call (for example) to **FixEdges** Dump instructions (using **DumpRange**) and see if problem is in trees If not, go to **RegAlloc** and see if problem shows up yet If so, binary search between **FixEdges** and **RegAlloc** to find optimization at fault.

### Instruction Select Problems

Go to **RegAlloc** for appropriate function (called once per function when not -od) Find address of instruction which gets translated or handled improperly. (Look in results of **DumpRange** for this address). Do a **"bif { eax == address } ExpandIns"** to look at what we do to this instruction (trace through).

### ***Register Allocation Problem***

### ***Instruction Encoding Problem***

Go to *RegAlloc* invocation for routine in question. Go to *GenObject* and call *DumpRange*. Find address of instruction that gets encoded incorrectly, and do a "**bif { eax == address } *GenObjCode***" Trace into *GenObjCode* at appropriate time.

**A**

arithmetic if 36  
assignment 27-28

**B**

back handle 13-14, 45, 49  
BEAbort 7  
BEAliasType 16  
BEDefSeg 9  
BEDefType 16  
BEFin 7  
BEFinBack 14  
BEFinLabel 11  
BEFlushSeg 10  
BEFreeBack 14  
BEGetSeg 10  
BEInit 3  
BENewBack 13  
BENewLabel 11  
BESetSeg 10  
BESStart 6  
BESTop 6  
BETypeAlign 17  
BETypeLength 16  
bit fields 42  
boolean expressions. 33, 35

**C**

calling conventions 53, 56  
CG3WayControl 36  
CGAddParm 31  
CGAssign 27  
CGAutoDecl 19  
CGBackName 26  
CGBigGoto 36  
CGBigLabel 36  
CGBinary 29  
CGBitMask 42  
CGChoose 35  
CGCompare 33

CGControl 36  
CGDone 41  
CGDuplicate 42  
CGEval 41  
CGFEName 25  
CGFloat 25  
CGFlow 35  
CGIndex 29  
CGInitCall 31  
CGInt64 25  
CGInteger 25  
CGLastParm 19  
CGLVAssign 27  
CGLVPreGets 27  
CGParmDecl 19  
CGPostGets 28  
CGPreGets 27  
CGProcDecl 19  
CGReturn 41  
CGSelCase 37  
CGSelect 38  
CGSelInit 37  
CGSelOther 38  
CGSelRange 38  
CGTemp 20  
CGTempName 26  
CGTrash 41  
CGType 42  
CGUnary 29  
CGVolatile 43  
CGWarp 35  
character 46  
control flow 36-38  
conversions 23

**D**

data 45  
DBAddBitField 61  
DBAddConst 62  
DBAddConst64 62  
DBAddField 61  
DBAddInheritance 62  
DBAddLocField 61  
DBAddParm 63  
DBArray 61  
DBBasedPtr 61  
DBBegBlock 60  
DBBegEnum 62  
DBBegName 60

DBBegProc 63  
DBBegStruct 61  
DBCharBlock 63  
DBDereference 63  
DBEndBlock 60  
DBEndEnum 62  
DBEndName 60  
DBEndProc 63  
DBEndStruct 62  
DBForward 60  
DBFtnArray 63  
DBFtnType 63  
DBGenSym 60  
DBIndCharBlock 63  
DBIntArray 61  
DBLineNum 59  
DBLocalSym 59  
DBLocCharBlock 63  
DBLocConst 64  
DBLocFini 64  
DBLocInit 64  
DBLocOp 64  
DBLocSym 64  
DBLocTemp 64  
DBModSym 59  
DBObject 59  
DBPtr 61  
DBScalar 60  
DBScope 60  
DBSrcCue 65  
DBSrcFile 64  
DBSubRange 61  
DGAlign 47  
DGBackPtr 45  
DGBackTell 48  
DGBytes 47  
DGChar 46  
DGFEPtr 45  
DGFloat 46  
DGIBytes 47  
DGInteger 46  
DGInteger64 46  
DGLabel 45  
DGSeek 47  
DGString 46  
DGTell 48  
DGUBytes 47

### ***E***

error messages 52  
expressions 21, 29, 41-43

### ***F***

FEAttr 51  
FEAuxInfo 53  
FEBack 49  
FEDbgType 51  
FEExtName 50  
FEGenProc 49  
FELexLevel 50  
FEMessage 52  
FEModuleName 49  
FEMoreMem 50  
FENAME 50  
FEParmType 50  
FESegID 49  
FEStackCheck 50  
FETrue 50  
floating point constant 25, 46  
FORTRAN 35-36  
functions 19, 31, 41, 49

### ***I***

inline procedures 49  
integers 25

### ***L***

label, code 11  
label, data 13, 26, 45

volatile 43

**O**

operators 21  
options 3, 5

**P**

pascal 50  
procedures 19, 31, 41, 49

**R**

registers 67  
relocatable data item 45  
routines 19, 31, 41, 49

**S**

segments 9-10, 45, 47-49  
short circuit operations. 33, 35  
stack probes 50  
statement functions 35

**T**

Temporaries 20, 26  
types, predefined 69  
typing 15-17, 23, 42

**V**

Variables 20, 25-26