Open Watcom Code Generator Interface

Table of Contents

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.

Parameter	***Definition***
switches	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.
targ_switches	Target specific switches. The bit values are defined below.
optsize	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.
proc	The target hardware configuration, defined below.
Returns	Information about the code generator revision in a cg_init_info structure, defined below.

Switch	***Definition***
NO_OPTIMIZATION	Turn off optimizations.
DBG_NUMBERS	Generate line number debugging information.
FORTRAN_ALIASING	Assume pointers are only used for parameter passing.
DBG_DF	Generate debugging information in DWARF format.
DBG_CV	Generate debugging information in CodeView format. If neither DBG_DF nor DBG_CV is set, debugging information (if any) is generated in the Watcom format.
RELAX_ALIAS	Assume that a static/extern variable and a pointer to that same variable are not used within the same routine.
DBG_LOCALS	Generate local symbol information for use by a debugger.
DBG_TYPES	Generate typing information for use by a debugger.
LOOP_UNROLLING	Turn on loop unrolling.
LOOP_OPTIMIZATION	Turn on loop optimizations.
INS_SCHEDULING	Turn on instruction scheduling.

MEMORY_LOW_FAILS	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).
FP_UNSTABLE_OPTIMIZATION	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.
NULL_DEREF_OK	NULL points to valid memory and may be dereferenced.
FPU_ROUNDING_INLINE	Inline floating-point value rounding (actually truncation) routine when converting floating-point values to integers.
FPU_ROUNDING_OMIT	Omit floating-point value rounding entirely and use FPU default. Results will not be ISO C compliant.
ECHO_API_CALLS	Log each call to the code generator with its arguments and return value. Only available in debug builds.
OBJ_ELF	Emit ELF object files.
OBJ_COFF	Emit COFF object files. For Intel compilers, OMF object files will be emitted in the absence of either switch.
OBJ_ENDIAN_BIG	Emit big-endian object files (COFF or ELF). If OBJ_ENDIAN_BIG is not set, little-endian objects will be generated.

x86 Switch	***Definition***
I_MATH_INLINE	Do not check arguments for operators like O_SQRT. This allows the compiler to use some specialty x87 instructions.
EZ_OMF	Generate Phar Lap EZ-OMF object files.
BIG_DATA	Use segmented pointers (16:16 or 16:32). This defines TY_POINTER to be equivalent to TY_HUGE_POINTER.
BIG_CODE	Use inter segment (far) call and return instructions.
CHEAP_POINTER	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.
FLAT_MODEL	Assume all segment registers address the same base memory.
FLOATING_FS	Does FS float (or is it pegged to DGROUP).
FLOATING_GS	Does GS float (or is it pegged to DGROUP).
FLOATING_ES	Does ES float (or is it pegged to DGROUP).
FLOATING_SS	Does SS float (or is it pegged to DGROUP).
FLOATING_DS	Does DS float (or is it pegged to DGROUP).

4 General

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

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.

Parameter	*Definition*
hdl	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.
value	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 algn)

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

1. Code Segments
2. PRIVATE data segments.

Parameter	***Definition***
segid	A non-negative integer used as an identifier for the segment. It is arbitrarily picked by the front end.
attr	Segment attribute bits, defined below.
str	The name given to the segment.
algn	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.

Attribute	***Definition***
EXEC	This is a code segment.
GLOBAL	The segment is accessible to other modules. (versus PRIVATE).
INIT	The segment is statically initialized.
ROM	The segment is read only.
BACK	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 CONST_IN_CODE switch must be passed to BEInit().
COMMON	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.

Parameter ***Definition***

segid Selects the current segment.

Returns 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.

Returns 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).

Parameter ***Definition***

segid 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.

Parameter	***Definition***
lbl	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 BENewBack 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 = BENewBack( 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 BEStop 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 BENewBack(NULL). These are useful for literal strings. These must also be freed after calling BEStop.

back_handle BENewBack(cg_sym_handle sym)

Allocate a new back_handle.

Parameter	***Definition***
sym	The front end symbol handle to be associated with the back_handle. It may be NULL.
Returns	A new back_handle.

void BEFiniBack(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.

Parameter	*Definition*
bck	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.

Parameter	*Definition*
bck	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:

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

TY_DEFAULT	Used to indicate default conversion
TY_NEAR_INTEGER	The result of subtracting 2 near pointers
TY_LONG_INTEGER	The result of subtracting 2 far pointers
TY_HUGE_INTEGER	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.

Parameter	***Definition***
what	An integral value greater than or equal to TY_FIRST_FREE and less then or equal to TY_LAST_FREE, used as the type identifier.
align	Currently ignored.
len	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.

Parameter	***Definition***
what	Will become an alias for an existing type.
to	An existing type.

unsigned_32 BETypeLength(cg_type type)

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

Parameter	***Definition***
type	A previously defined type.
Returns	The length associated with the type.

uint BETypeAlign(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.

18 Type definitions

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.

Parameter	*Definition*
name	The front end symbol table entry for the procedure. A back_handle will be requested.
type	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.

Parameter	*Definition*
name	The symbol table entry for the parameter.
type	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 CGFEName.

Parameter	*Definition*
name	The symbol table entry for the variable.
type	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.

Parameter	*Definition*
type	The type of the new temporary.
Returns	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:

0-ary Operator	***C equivalent***
O_NOP	N/A
Unary Operator	***C equivalent***
O_UMINUS	-x
O_COMPLEMENT	x
O_POINTS	(*x)
O_CONVERT	x=y
O_ROUND	Do not use!
O_LOG	log(x)
O_COS	cos(x)
O_SIN	sin(x)
O_TAN	tan(x)
O_SQRT	sqrt(x)
O_FABS	fabs(x)
O_ACOS	acos(x)
O_ASIN	asin(x)
O_ATAN	atan(x)
O_COSH	cosh(x)
O_SINH	sinh(x)

O_TANH	tanh(x)
O_EXP	exp(x)
O_LOG10	log10(x)
O_PARENTHESIS	This operator represents the "strong" parenthesises 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".
Binary Operator	***C equivalent***
O_PLUS	+
O_MINUS	-
O_TIMES	*
O_DIV	/
O_MOD	%
O_AND	&
O_OR	|
O_XOR	^
O_RSHIFT	>>
O_LSHIFT	<<
O_COMMA	,
O_TEST_TRUE	(x & y) != 0
O_TEST_FALSE	(x & y) == 0
O_EQ	==
O_NE	!=
O_GT	>
O_LE	<=
O_LT	<
O_GE	>=
O_POW	pow(x, y)
O_ATAN2	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.

Parameter	*Definition*
val	The integral value.
type	An integral type.

cg_name CGInt64(signed_64 val, cg_type type)

Create an 64-bit integer constant leaf node.

Parameter	*Definition*
val	The 64-bit integer value.
type	An integral type.

cg_name CGFloat(char *num, cg_type type)

Create a floating-point constant leaf node.

Parameter	*Definition*
num	A NULL terminated E format string. (-1.23456E-102)
type	A floating point type.

cg_name CGFEName(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.

Parameter	*Definition*
sym	The front end symbol.
type	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.

Parameter	*Definition*
bck	A back handle.
type	The type to be associated with the **value** of the symbol.

cg_name CGTempName(temp_handle temp, cg_type type)

Create a leaf node which yields the address of the temp_handle.

Parameter	*Definition*
temp	A temp_handle.
type	The type to be associated with the **value** of the symbol.

Assignment Operations

cg_name CGAssign(cg_name dest, cg_name src, cg_type type)

Create an assignment node.

Parameter	*Definition*
dest	The destination address.
src	The source value.
type	The type to which the destination address points.
Returns	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.

Parameter	*Definition*
op	The arithmetic operator to be used.
dest	The address of the destination.
src	The value of the right hand side.
type	The type to which the destination address points.
Returns	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.

Parameter	*Definition*
op	The operator.
dest	The address of the destination
src	The value of the increment.
type	The type of the destination.
Returns	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.

Parameter	*Definition*
op	The operator.
left	The value of the left hand side.
right	The value of the right hand side.
type	The result type.
Returns	The value of the result.

cg_name CGUnary(cg_op op, cg_name name, cg_type type)

Unary operations.

Parameter	*Definition*
op	The operator.
name	The value of operand.
type	The result type.
Returns	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.

Parameter	*Definition*
name	The address of the routine to call.
type	The return type of the routine.
aux_info	A handle which the back end may passed to FEAuxInfo to determine the attributes of the call.
Returns	A **call_handle** 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.

Parameter	*Definition*
call	A call_handle.
name	The value of the parameter.
type	The type of the parameter. This type will be passed to FEParmType 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.

Parameter	*Definition*
call	A call_handle.
Returns	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.

Parameter	*Definition*
op	The comparison operator.
left	The value of the left hand side.
right	The value of the right hand side.
type	The type to which to convert the operands to before performing comparison.
Returns	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.

Parameter	*Definition*
op	An operator.
left	A TY_BOOLEAN or integral cg_name.
right	A TY_BOOLEAN or integral cg_name, or NULL if op is O_FLOW_NOT.
Returns	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**.

Parameter	*Definition*
sel	A TY_BOOLEAN or integral cg_name used as the selector.
n1	The value to return if **sel** is non-zero.
n2	The value to return if **sel** is zero.
type	The type to which convert the result.
Returns	The value of **n1** or **n2** depending upon the truth of **sel**.

cg_name CGWarp(cg_name before, label_handle label, cg_name after)

To be used for FORTRAN statement functions.

Parameter	*Definition*
before	An arbitrary expression tree to be evaluated before **label** is called. This is used to assign values to statement function arguments, which are usually temporaries allocated with CGTemp.
label	A label_handle to invoke via O_CALL_LABEL.
after	An arbitrary expression tree to be evaluated after **label** 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.

Parameter	*Definition*
expr	The selector value.
lt	A label_handle.
eq	A label_handle.
gt	A label_handle.

void CGControl(cg_op op, cg_name expr, label_handle lbl)

Generate conditional and unconditional flow of control.

Parameter	*Definition*
op	a control flow operator.
expr	A TY_BOOLEAN expression if op is O_IF_TRUE or O_IF_FALSE. NULL otherwise.
lbl	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.

Parameter	*Definition*
lbl	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.

Parameter	*Definition*
lbl	A back_handle. There must be a front end symbol associated with this back handle.
level	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 = BENewLabel();

sel_ label = BENewLabel();
CGControl( O_ GOTO, NULL, sel_ label );
sel_ handle = CGSelInit();

case_ label = BENewLabel();
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 = BENewLabel();
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.

Parameter	***Definition***
sel	A sel_handle obtained from CGSelInit().
lbl	The label to be associated with the case value.
val	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.

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

void CGSelOther(sel_handle s, label_handle lbl)

Add the otherwise case to a select.

Parameter	*Definition*
s	A sel_handle.
lbl	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.

Parameter	*Definition*
s	A sel_handle.
expr	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.

Parameter	*Definition*
s	A sel_handle.
expr	The value we are selecting.
allowed	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.

Parameter	*Definition*
name	The value of the return value, or NULL.
type	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.

Parameter	*Definition*
name	The tree to be evaluated.
Returns	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.

Parameter	*Definition*
name	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.

Parameter	*Definition*
name	A cg_name.
Returns	The type of the cg_name.

cg_name *CGDuplicate(cg_name name)

Create two copies of a cg_name.

Parameter	*Definition*
name	The cg_name to be duplicated.
Returns	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.

Parameter	*Definition*
name	The address of the integral variable containing the bit field.
start	The position of the least significant bit of the bit field. 0 indicates the least significant bit of the host data type.
len	The length of the bit field in bits.
type	The integral type of the value containing the bit field.
Returns	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.

Parameter	*Definition*
bck	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.

Parameter	*Definition*
bck	A back_handle.
segid	The segment_id of the segment in which the label for **bck** will be defined if it has not already been passed to DGLabel.
offset	A value to be added to the generated pointer value.
type	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**.

Parameter	*Definition*
sym	A cg_sym_handle.
type	The pointer type to be used.
offset	A value to be added to the generated pointer value.

void DGInteger(unsigned_32 value, cg_type type)

Generate an integer.

Parameter	*Definition*
value	An integral value.
type	The integral type to be used.

void DGInteger64(unsigned_64 value, cg_type type)

Generate an 64-bit integer.

Parameter	*Definition*
value	An 64-bit integer value.
type	The integral type to be used.

void DGFloat(char *value, cg_type type)

Generate a floating-point constant.

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

void DGChar(char value)

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

Parameter	*Definition*
value	A character value.

void DGString(char *value, uint len)

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

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

void DGBytes(unsigned_32 len, byte *src)

Generate raw binary data.

Parameter	*Definition*
src	Pointer to the data.
len	The length of the byte stream.

void DGIBytes(unsigned_32 len, byte pat)

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

Parameter	*Definition*
pat	The pattern byte.
len	The number of times to repeat the byte.

void DGUBytes(unsigned_32 len)

Generate **len** undefined bytes.

Parameter	*Definition*
len	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.

Parameter	*Definition*
align	The desired alignment boundary.

unsigned_32 DGSeek(unsigned_32 where)

Seek to a location within a segment.

Parameter	*Definition*
where	The location within the segment.
Returns	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.

Parameter	***Definition***
sym	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

Parameter	***Definition***
sym	
Returns	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.

Parameter	***Definition***
sym	A cg_sym_handle.
Returns	A segment_id.

char *FEModuleName()

Returns	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 FETrue()

Returns The value of TRUE. This is normally 1.

char FEMoreMem(size_t size)

Release memory for the back end to use.

Parameter	***Definition***
size	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 necessarally 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.

Parameter	*Definition*
msg	Defined below.
extra	Extra information. The type and meaning depends on the value of **msg** and is indicated below.

MSG_INFO_FILE	Informational message about file. extra (void) is ignored.
MSG_CODE_SIZE	Code size. Extra (int) is the size of the generated code.
MSG_DATA_SIZE	Data size. Extra (int) is the size of the generated data.
MSG_ERROR	A back end error message. Extra (char *) is the error message.
MSG_FATAL	A fatal code generator error. Extra (char *) is the reason for the fatal error. The front end should issue this message and exit immediately to the system.
MSG_INFO_PROC	Informational message about current procedure. Extra (char *) is a message.
MSG_BAD_PARM_REGISTER	Invalid parameter register returned from FEAuxInfo. Extra (int) is position of the offending parameter.
MSG_BAD_RETURN_REGISTER	Invalid return register returned from FEAuxInfo. Extra (aux_handle) is the offending aux_handle.
MSG_REGALLOC_DIED	The register alloc ran out of memory. Extra (cg_sym_handle) is the procedure which was not fully optimized.
MSG_SCOREBOARD_DIED	The register scoreboard ran out of memory. Extra (cg_sym_handle) is the procedure which was not fully optimized.
MSG_PEEPHOLE_FLUSHED	Peep hole optimizer flushed due to lack of memory. (void)
MSG_BACK_END_ERROR	BAD NEWS! Internal compiler error. Extra (int) is an internal error number.
MSG_BAD_SAVE	Invalid register modification information return from FEAuxInfo. Extra (aux_handle) is the offending aux_handle.
MSG_WANT_MORE_DATA	The back end wants more data space. Extra (int) is amount of additional memory needed to run. (DOS real mode hosts only).
MSG_BLIP	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)
MSG_BAD_LINKAGE	Cannot resolve linkage conventions. 370 only. (sym)

MSG_SCHEDULER_DIED	Instruction scheduler ran out of memory. Extra (cg_sym_handle) is the procedure which was not fully optimized.
MSG_NO_SEG_REGS	(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.
MSG_SYMBOL_TOO_LONG	Given symbol is too long and is truncated to maximum permited 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***
extra	Extra information. Its type and meaning is determined by the value of class.
class	Defined below.

Parameters	***Return Value***
(cg_sym_handle, AUX_LOOKUP)	aux_handle - given a cg_sym_handle, return an aux_handle.

(aux_handle, CALL_BYTES)	byte_seq * - A pointer to bytes to be generated instead of a call, or NULL if a call is to be generated.
	`typedef struct byte_ seq {`
	`  char  length;`
	`  char  data[ 1 ];`
	`} byte_ seq;`

(aux_handle, CALL_CLASS)	call_class * - returns call_class of the given aux_handle. See definitions below.
(short, FREE_SEGMENT)	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)
(NULL, OBJECT_FILE_NAME)	char * - The name of the object file to be generated.
(aux_handle, PARM_REGS)	hw_reg_set[] - The set of register to be used as parameters.
(aux_handle, RETURN_REG)	hw_reg_set * - The return register. This is only called if the routine is declared to have the SPECIAL_RETURN call_class.
(NULL, REVISION_NUMBER)	int - Front end revision number. Must return II_REVISION.
(aux_handle, SAVE_REGS)	hw_reg_set * - Registers which are preserved by the routine.
(cg_sym_handle, SHADOW_SYMBOL)	cg_sym_handle - An alternate handle for a symbol. Required for

FORTRAN. 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).

x86 Parameters	*Return value*
(NULL, CODE_GROUP)	char * - The name of the code group.
(aux_handle, STRETURN_REG)	hw_reg_set * - The register which points to a structure return value. Only called if the routine has the SPECIAL_STRUCT_RETURN attribute.
*(void *, NEXT_IMPORT)*	void * (See notes at end) - A handle for the next symbol to generate a reference to in the object file.
(void, IMPORT_NAME)*	char * - The EXTDEF name to generate given a handle
*(void *, NEXT_IMPORT_S)*	void * (See notes at end) - A handle for the next symbol to generate a reference to in the object file.
(void, IMPORT_NAME_S)*	Returns a cg_sym_handle. The EXTDEF name symbol reference to generate given a handle.
(void, NEXT_LIBRARY)*	void * (See notes at end) - Handle for the next library required
(void, LIBRARY_NAME)*	char * - The library name to generate given a handle
(NULL, DATA_GROUP)	char * - Used to name DGROUP exactly. NULL means use no group at all.
(segment_id, CLASS_NAME)	NULL - Used to name the class of a segment.
(NULL, USED_8087)	NULL - Indicate that 8087 instructions were generated.
(NULL, STACK_SIZE_8087)	int - How many 8087 registers are reserved for stack.
(NULL, CODE_LABEL_ALIGNMENT)	char * - An array x, such that x[i] is the label alignment requirements for labels nested within i loops.
(NULL, PROEPI_DATA_SIZE)	int - How much stack is reserved for the prolog hook routine.
(cg_sym_handle, IMPORT_TYPE)	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_REFRENCE, return the next magic cookie in the list of virtual functions that might be refrenced 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	***Meaning***
REVERSE_PARMS	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***

FAR	Does routine require a far call/return.
LOAD_DS_ON_CALL	Load DS from DGROUP prior to call.
CALLER_POPS	Caller pops/removes parms from the stack.
ROUTINE_RETURN	Routine allocates structure return memory.
SPECIAL_RETURN	Routine has non-default return register.
NO_MEMORY_CHANGED	Routine modifies no visible statics.
NO_MEMORY_READ	Routine reads no visible statics.
MODIFY_EXACT	Routine modifies no parameter registers.
SPECIAL_STRUCT_RETURN	Routine has special struct return register.
NO_STRUCT_REG_RETURNS	Pass 2/4/8 byte structs on stack, as opposed to registers.
NO_FLOAT_REG_RETURNS	Return floats as structs.
INTERRUPT	Routine is an interrupt routine.
NO_8087_RETURNS	No return values in the 8087.
LOAD_DS_ON_ENTRY	Load ds with dgroup on entry.
DLL_EXPORT	Is routine an OS/2 export symbol?
FAT_WINDOWS_PROLOG	Generate the real mode windows prolog code.
GENERATE_STACK_FRAME	Always generate a traceable prolog.
EMIT_FUNCTION_NAME	Emit the function name in front of the function in the code segment.
GROW_STACK	Emit a call to grow the stack on entry
PROLOG_HOOKS	Generate a prolog hook call.
EPILOG_HOOKS	Generate an epilog hook call.
THUNK_PROLOG	Generate a thunking prolog for routines calling 16 bit code.
FAR16_CALL	Performs a 16:16 call in the 386 compiler.
TOUCH_STACK	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.

Parameter	*Definition*
no	Is the current source line number.

void DBModSym(cg_sym_handle sym, cg_type indirect)

Define a symbol within the module (file scope).

Parameter	*Definition*
sym	is a front end symbol handle.
indirect	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.

Parameter	*Definition*
tipe	is the debug type of the class that the function is a member of.
loc	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 superseeds both DBLocalSym and DBModuleSym. The 'loc' parameter is a location expression (explained later) which allows an aribitrary sequence of operations to locate the storage for the symbol.

Parameter	*Definition*
sym	is a front end symbol handle.
loc	the location expression which is evaluated by the debugger to locate the lvalue of the symbol.
scoped	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 evalated, 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:

Attribute ***Definition***

FIELD_ATTR_INTERNAL 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.

Parameter	*Definition*
st	the dbg_struct handle for the structure currently being defined.
inherit	the dbg_type of a previously defined structure which is being inherited.
adjust	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

Operation	*Definition*
DB_OP_POINTS	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 cg_type which resolves to either an unsigned_16, unsigned_32, a 16-bit far pointer, or a 32-bit far pointer.
DB_OP_ZEX	zero extend the top of the stack. The 'other' parameter is a cg_type 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.
DB_OP_XCHG	exchange the top of stack value with the stack entry indexed by 'other'.
DB_OP_MK_FP	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.
DB_OP_ADD	add the top two stack entries together.
DB_OP_DUP	duplicate the top stack entry.
DB_OP_POP	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 automaticaly 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.

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

HW_Asgn(dst, b) Assign **b** to **dst**

HW_TurnOn(dst, b) Turn on registers **b** in **dst**.

HW_TurnOff(dst, b) Turn off registers **b** in **dst**.

HW_OnlyOn(dst, b) Turn off all registers except **b** in **dst**.

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:

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

B

C

D

E

F

I

L