CHAPTER 2

WHIRL Symbol Table Specification

2.1 Introduction and Overview

This document describes the symbol table portion of the WHIRL file produced and used by the SGI Pro64™ compiler. A separate document describes the WHIRL intermediate program representation.

The WHIRL symbol table is made up of a series of tables. They are designed for compilation, optimization and storage efficiency. The way the tables are organized closely corresponds to the compiler's view of the symbol table. The model also enhances locality in references to the tables.

The WHIRL symbol table is divided into the global part and the local part. The local part is organized by program units (PUs). Figure 1 gives a pictorial overview of the WHIRL symbol table as produced by the front-ends. There are different kinds of tables. The tables that can appear in both the global and local part of the symbol table are:

1. ST_TAB — This is the fundamental building block of the symbol table. In general, any symbol with a name occupies an entry in this table. Any constant value that reside in memory (floating point and string constants) also occupies an entry in this table.
2. INITO_TAB — Each entry specifies the initial value(s) of an initialized data object. It in turn refers to one or more entries in the INITV_TAB for initial values of each individual component of the data object.
3. ST_ATTR_TAB — Each entry associates some miscellaneous attributes with an entry in the ST_TAB.

The tables that can only appear in the global part of the symbol table are:

1. PU_TAB — Each entry represents a procedure that appears in the source file as either function prototype or definition.
2. TY_TAB — Each entry represents a distinct type in the program. It in turn refers to the FLD_TAB, TYLIST_TAB, ARB_TAB, or PU_TAB to specify the full structure of each type.
3. FLD_TAB — Each entry specifies a field in a struct type.
4. TYLIST_TAB — Each entry specifies a parameter type in a function prototype declaration.
5. ARB_TAB — Each entry gives information about a dimension of an array type.

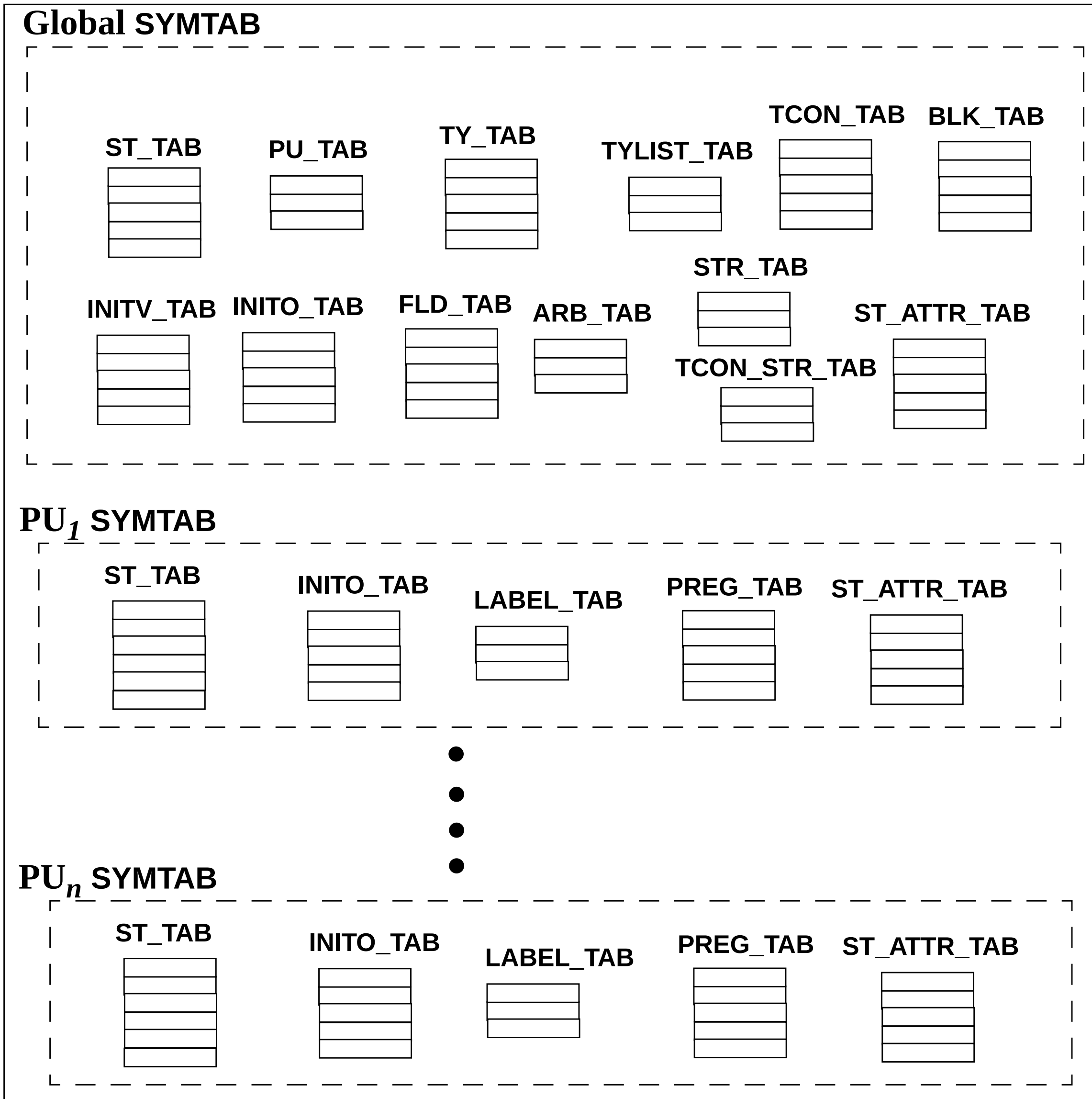

Figure 1 WHIRL Symbol Table produced by the front-ends

6. TCON_TAB — The values of any non-integer constants are stored here. For string constants, it in turn refers to the TCON_STR_TAB.
7. BLK_TAB — Each entry specifies layout information of a block of data.
8. INITV_TAB — Each entry describes the initial value of a scalar component of an initialized data object.

9. STR_TAB — All strings are stored here. They include names of variable, types, labels, etc.
10. TCON_STR_TAB — All string literals defined in the user program are stored in this table.

The tables that can only appear in the local part of the symbol table are:

1. LABEL_TAB —Information associated with each WHIRL label used in the PU is stored here.
2. PREG_TAB —Information associated with each pseudo-register used in the PU is stored here.

Apart from the above tables, each compiler component is free to allocate additional tables for its own internal use in storing extra information. The additional tables are to have the same number of entries and be referred to by the same type of index as one of the above tables. As a general rule, the first entry of each table has index 1; index 0 is reserved to stand for uninitialized index value. The design also assumes that any table will never grow to more than 16 million entries, so that only 24 bits are needed to contain a table index. An exception is STR_TAB, in which the index is really a byte offset.

The tables listed so far mainly serves the purpose of communicating information gathered by the front-ends to the back-end phases during compilation. The back-end optimization phases may create more information, and the new information can reside in additional tables created for the purpose of passing information to the other back-end components. These tables will be prefixed by the name of the component that creates the information in the table, e.g. IPA_ST_TAB, WOPT_ST_TAB, etc. In particular, BE_ST_TAB (Section 2.19.1) serves to communicate information among the back-end components, including IPA.

The remaining sections of this chapter describe the symbol table structures in more details and the interfaces to them.

2.2 SCOPE

Depending on the context, a different set of symbol tables might become visible. For example, in a nested procedure, three ST_TABs are visible — its own local ST_TAB, the parent PU’s ST_TAB, and the global ST_TAB. Associated with each PU, a SCOPE array is defined for specifying the list of visible tables. The index to this array is the lexical scope. Index 0 is reserved. Index 1 refers to the global symbol tables, and index 2 refers to the local symbol tables. A nested procedure will have an index starting at 3, depending on the level of nesting. The type of the SCOPE array index is SYMTAB_IDX, which is an unsigned 8-bit integer.

Strictly speaking, SCOPE arrays are not part of the symbol table, and they are never written out to a WHIRL file.Tables that can only appear in the global part of the symbol table are always visible. So they are not explicitly described by the SCOPE array.

Each element of a SCOPE array has the following structure, size 24 bytes:

Table 1 Layout of a SCOPE Array Element

Offset	Field	Type	Description	Field size
byte 0	pool	MEM_POOL *	pointer to the memory pool for local tables	1 word
byte 4	st	ST *	pointer to the ST for this PU	1 word
byte 8	st_tab	ST_TAB *	pointer to the table of ST entries	1 word
byte 12	label_tab	LABEL_TAB *	pointer to the table of labels	1 word
byte 16	preg_tab	PREG_TAB *	pointer to the table of pseudo registers	1 word
byte 20	inito_tab	INITO_TAB *	pointer to the table of INITO entries.	1 word
byte 24	st_attr_tab	ST_ATTR_TAB *	pointer to the table of ST_ATTR entries.	1 word

For the global scope (i.e., index 1of the SCOPE array), the fields pool, st, label_tab, and preg_tab are not used, and contain the NULL pointer.

2.3 ST_TAB

Each entry of this table is an ST. A symbol in the program is uniquely identified by a value of type ST_IDX.

2.3.1 ST_IDX

ST_IDX is of size 32 bits, and is composed of two parts:

Table 2 Layout of ST_IDX

Field	Description	Field position and size
level	lexical level	least significant 8 bits
index	index to ST_TAB	most significant 24 bits

The low order 8 bits are used to index into the SCOPE array in order to get to the ST_TAB.

2.3.2 ST Entry

The ST entry has the following structure, size 32 bytes:

Table 3 Layout of ST

Offset	Field	Description	Field size
byte 0 byte 0	name_idx tcon	STR_IDX to the name string TCON_IDX of the constant value	1 word 1 word
byte 4	flags	misc. attributes of this entry	1 word
byte 8	flags_ext	more flags for future extension	1 byte
byte 9	sym_class	class of symbol	1 byte
byte 10	storage_class	storage class of symbol	1 byte
byte 11	export	export class of the symbol	1 byte
byte 12 byte 12 byte 12	type pu blk	TY_IDX of the high-level type PU_IDX if program unit BLK_IDX if CLASS_BLOCK	1 word 1 word 1 word
byte 16	offset	offset from base	2 words
byte 24	base_idx	ST_IDX of the base of the allocated block	1 word
byte 28	st_idx	ST_IDX for this entry	1 word

name_idx/tcon: If sym_class is CLASS_CONST, the tcon field holds the index to the TCON_TAB. For all other sym_class values, the name_idx field holds the index to the STR_TAB. If the export class is EXPORT_LOCAL or

EXPORT_LOCAL_INTERNAL, the name is optional. And when there is no name, name_idx should be zero.

flags/flags_ext: Miscellaneous attributes, See Section 2.3.5.

sym_class: The class of symbol, see Table 4.

storage_class: The storage class of symbol, see Table 5.

export: The export class of symbol, see Section 2.3.4.

type/pu/blk: If sym_class is CLASS_FUNC, then the pu field holds the index to the PU_TAB. If sym_class is CLASS_BLOCK, this field holds the BLK_IDX. If sym_class is CLASS_NAME, this field must be zero. For all other valid sym_class values, the type field holds the TY_IDX that describes the type of this symbol.

One exception is a CLASS_NAME symbol that has the ST_ASM_FUNCTION_ST bit set, in which case the pu field holds the index to the PU_TAB.

offset: The byte offset from base_idx. If base_idx is equal to st_idx, then offset must be zero.

base_idx: This is the ST_IDX for the ST that describes the base address (i.e., this symbol is an alias of the specified symbol). If it is equal to it's own st_idx, then the address of this symbol is independently assigned. If ST_IS_WEAK_ALIAS is set, base_idx is overloaded to specify the corresponding strong definition (see Table 9 and Section 2.3.7). If ST_IS_SPLIT_COMMON is set, base_idx is overloaded to be the full common definition. It is illegal to set both ST_IS_WEAK_ALIAS and ST_IS_SPLIT_COMMON.

The following rules apply when setting the base address of a symbol. If a symbol A is based on symbol B (i.e. base_idx of A is equal to st_idx of B), then:

i. storage_class of A must be the same as storage_class of B, except when the sym_class of B is CLASS_BLOCK and storage_class of B is SCLASS_UNKNOWN.

ii. if sym_class of A is CLASS_BLOCK, sym_class of B must be CLASS_BLOCK.

iii. offset of A plus the size of A must not be larger than the size of B.

st_idx: ST_IDX of this symbol. This is used mainly for fast conversion from a pointer to a given ST to the corresponding ST_IDX.

2.3.3 Symbol Class and Storage Class

There is a symbol class and a storage class associated with each ST entry. Both of which are enumeration type:

Table 4 Symbol Class

Name	Value	Description
CLASS_UNK	0	uninitialized
CLASS_VAR	1	data variable
CLASS_FUNC	2	function
CLASS_CONST	3	constant, a TCON holds the real value
CLASS_PREG	4	pseudo register
CLASS_BLOCK	5	base address for a block of data.
CLASS_NAME	6	placeholder for a named ST entry

Table 5 Storage Class

Name	Value	Description
SCLASS_UNKNOWN	0	no specific storage class (e.g., a block of data of mixed storage classes)
SCLASS_AUTO	1	local stack variable
SCLASS_FORMAL	2	formal parameter
SCLASS_FORMAL_REF	3	reference parameter
SCLASS_PSTATIC	4	PU scope static data
SCLASS_FSTATIC	5	file scope static data
SCLASS_COMMON	6	common block (linker allocated)
SCLASS_EXTERN	7	unallocated external data or text
SCLASS_UGLOBAL	8	uninitialized global data

Table 5 Storage Class

Name	Value	Description
SCLASS_DGLOBAL	9	initialized global data
SCLASS_TEXT	10	executable code
SCLASS_REG	11	register variable
SCLASS_CPLINIT	12	special data object describing initialization of static/global C++ classes.
SCLASS_EH_REGION	13	special table describing C++ exception handling (See Section 2.3.6)
SCLASS_EH_REGION_SUPP	14	supplemental data structure for C++ exception handling (See Section 2.3.6)
SCLASS_DISTR_ARRAY	15	data object that is placed in the special Elf section _MIPS_distr_array
SCLASS_COMMENT	16	names of such symbols are to be placed in the special Elf section .comment.
SCLASS_THREAD_PRIVATE_FUNCS	17	data object that is placed in the special Elf section _MIPS_thread_private_funcs

Not all combinations of symbol class and storage class are valid. Only those listed below are allowed:

Table 6 Valid Symbol Class and Storage Class Combinations

Symbol class	Storage class	Description
CLASS_UNK	SCLASS_UNKNOWN	uninitialized
CLASS_VAR	SCLASS_AUTO	stack variable
CLASS_VAR	SCLASS_FORMAL	formal parameter
CLASS_VAR	SCLASS_FORMAL_REF	reference parameter
CLASS_VAR	SCLASS_PSTATIC	PU scope static variable
CLASS_VAR	SCLASS_FSTATIC	file scope variable
CLASS_VAR	SCLASS_COMMON	common block
CLASS_VAR	SCLASS_EXTERN	unallocated external variable
CLASS_VAR	SCLASS_UGLOBAL	uninitialized global variable
CLASS_VAR	SCLASS_DGLOBAL	initialized global variable

Table 6 Valid Symbol Class and Storage Class Combinations

Symbol class	Storage class	Description
CLASS_VAR	SCLASS_CPLINIT	special data object describing initialization of static/ global C++ classes.
CLASS_VAR	SCLASS_EH_REGION	special table describing C++ exception handling
CLASS_VAR	SCLASS_EH_REGION_SUPP	supplemental data structure for C++ exception handling
CLASS_VAR	SCLASS_DISTR_ARRAY	data object that is placed in the special Elf section _MIPS_distr_array
CLASS_VAR	SCLASS_THREAD_PRIVATE_FUNCS	data object that is placed in the special Elf section _MIPS_thread_private_funcs
CLASS_FUNC	SCLASS_EXTERN	undefined function
CLASS_FUNC	SCLASS_TEXT	defined function
CLASS_CONST	SCLASS_FSTATIC	constant
CLASS_CONST	SCLASS_EXTERN	constant symbol defined in another file (e.g. in IPA-generated symbol table)
CLASS_PREG	SCLASS_REG	pseudo register
CLASS_BLOCK	all storage classes except SCLASS_UNKNOWN and SCLASS_REG	a block of data or text of the specified storage class
CLASS_BLOCK	SCLASS_UNKNOWN	a block of data or text of unspecified storage class (e.g., a block of mixed storage classes)
CLASS_NAME	SCLASS_UNKNOWN	an ST entry that only has a name and nothing else, usually used as a place holder for special symbols that are passed to the linker
CLASS_NAME	SCLASS_COMMENT	an ST entry whose name is to be placed in the Elf section .comment

2.3.4 Export Scopes

This enumeration describes the possible scopes that symbols exported from a file may map into, i.e., linker globals for DSO (dynamically shared object)-related components.

Table 7 Export Scopes

Export Scope	Value	Description
EXPORT_LOCAL	0	not exported, must be defined in current file (e.g. C static data), address can be exported from DSO using a pointer
EXPORT_LOCAL_INTERNAL	1	not exported, must be defined in current file, only visible within current file, only used within the DSO or executable
EXPORT_INTERNAL	2	exported, only visible and used within the DSO or executable, must be defined in current DSO or executable
EXPORT_HIDDEN	3	exported, name is hidden within DSO or executable, address can be exported from DSO using a pointer, must be defined in current DSO or executable
EXPORT_PROTECTED	4	exported, non-preemptible, must be defined in current DSO or executable
EXPORT_PREEMPTIBLE	5	exported, preemptible
EXPORT_OPTIONAL	6	correspond to STO_OPTIONAL in Elf symbol table (see <sys/elf.h>)

Only an EXPORT_LOCAL or EXPORT_LOCAL_INTERNAL symbol must be defined in the file being compiled. All others can be either defined or undefined. All symbols except EXPORT_PREEMTIBLE must be defined in the current DSO or executable.

Only EXPORT_LOCAL and EXPORT_LOCAL_INTERNAL symbols are allowed in a local ST_TAB. Symbols with all other export scopes must be placed in the global ST_TAB. Furthermore, the ST entries of all functions, regardless of export scope, must be placed in the global ST_TAB.

Valid combinations of export scopes and storage classes are listed in the following table:.

Table 8 Valid Combinations of Storage Class and Export Scopes

Storage class	Export scopes	Description
SCLASS_UNKNOWN SCLASS_AUTO SCLASS_FORMAL SCLASS_FORMAL_REF SCLASS_PSTATIC SCLASS_FSTATIC SCLASS_CPLINIT SCLASS_EH_REGION SCLASS_EH_REGION_SUPP SCLASS_DISTR_ARRAY SCLASS_THREAD_PRIVATE_FUNCS SCLASS_COMMENT	EXPORT_LOCAL EXPORT_LOCAL_INTERNAL	file or PU scope variables
SCLASS_COMMON SCLASS_EXTERN SCLASS_UGLOBAL SCLASS_DGLOBAL	EXPORT_INTERNAL EXPORT_HIDDEN EXPORT_PROTECTED EXPORT_PREEMPTIBLE	DSO scope data or text symbols
SCLASS_COMMON SCLASS_DGLOBAL	EXPORT_LOCAL EXPORT_LOCAL_INTERNAL	member of a common or data block; these symbols must have base_idx pointing to an ST entry with the same storage class
SCLASS_EXTERN	EXPORT_LOCAL EXPORT_LOCAL_INTERNAL	local symbols that are not defined in the current file; use in IPA-generated file where a CLASS_CONST symbol is defined in a separate file.
SCLASS_TEXT	EXPORT_LOCAL EXPORT_LOCAL_INTERNAL	static functions
SCLASS_TEXT	EXPORT_INTERNAL EXPORT_HIDDEN EXPORT_PROTECTED EXPORT_PREEMPTIBLE	global functions
SCLASS_REG	EXPORT_LOCAL EXPORT_LOCAL_INTERNAL	registers

2.3.5 ST Flags

Associated with each ST entry are one or more attributes that describe specific property of it. Some of them are mutually exclusive and some of them are related. They are described in the following table:

Table 9 Miscellaneous Attributes of an ST Entry

Flag/Value	Description
ST_IS_WEAK_SYMBOL 0x00000001	weak name • not valid for EXPORT_LOCAL or EXPORT_LOCAL_INTERNAL • see Section 2.3.7 for semantics of weak symbols
ST_IS_SPLIT_COMMON 0x00000002	part of a split common • base_idx gives the ST_IDX of the corresponding complete common definition • ST_IS_WEAK_SYMBOL must not be set
ST_IS_NOT_USED 0x00000004	symbol is not referenced
ST_IS_INITIALIZED 0x00000008	initialized static or global variable • only valid for CLASS_VAR, CLASS_CONST, and CLASS_BLOCK • only valid for SCLASS_PSTATIC, SCLASS_FSTATIC, SCLASS_EXTERN, SCLASS_DGLOBAL, SCLASS_UGLOBAL, SCLASS_CPLINIT, SCLASS_EH_REGION, SCLASS_EH_RGION_SUPP, SCLASS_DIST_ARRAY, and SCLASS_THREAD_PRIVATE_FUNCS. • also valid for SCLASS_UNKNOWN if symbol class is CLASS_BLOCK • for SCLASS_UGLOBAL, ST_INIT_VALUE_ZERO must be set (uninitialized globals and globals explicitly initialized to zero are equivalent) • must be set for SCLASS_DGLOBAL • for CLASS_VAR, if ST_INIT_VALUE_ZERO is not set, there must be a corresponding INITO entry
ST_IS_RETURN_VAR 0x00000010	return value for Fortran function • only valid for SCLASS_AUTO
ST_IS_VALUE_PARM 0x00000020	parameter is passed by value • only valid for SCLASS_FORMAL
ST_PROMOTE_PARM 0x00000040	parameter has been promoted from chars/short to int or from float to double • only valid for C/C++

Table 9 Miscellaneous Attributes of an ST Entry

Flag/Value	Description
ST_KEEP_NAME_W2F 0x00000080	whirl2f should neither declare nor rename this symbol • only valid for CLASS_VAR
ST_IS_DATAPOOL 0x00000100	Fortran data pools
ST_IS_RESHAPED 0x00000200	symbol has a distribute_reshape pragma supplied for it • only valid for CLASS_VAR
ST_EMIT_SYMBOL 0x00000400	must appear in the symbol table of the Elf object file • only valid for CLASS_VAR, CLASS_NAME, and CLASS_FUNC, • used by C++ to force certain local symbols to be written out to the Elf object file
ST_HAS_NESTED_REF 0x00000800	symbol is referenced by a PU nested in the current PU • only valid for SCLASS_AUTO, SCLASS_PSTATIC, SCLASS_FORMAL, and SCLASS_FORMAL_REF.
ST_INIT_VALUE_ZERO 0x00001000	uninitialized global or static symbol • only valid for CLASS_VAR • only valid for SCLASS_EXTERN, SCLASS_UGLOBAL, SCLASS_FSTATIC, and SCLASS_PSTATIC • ST_IS_INITIALIZED must be set • also valid for symbol explicitly initialized to zero
ST_GPREL 0x00002000	can be accessed via an offset from the global pointer • only valid for CLASS_VAR and CLASS_CONST • not valid for SCLASS_AUTO, SCLASS_FORMAL, and SCLASS_FORMAL_REF
ST_NOT_GPREL 0x00004000	can not be accessed via an offset from the global pointer • only valid for CLASS_VAR and CLASS_CONST • not valid for SCLASS_AUTO, SCLASS_FORMAL, and SCLASS_FORMAL_REF
ST_IS_NAMELIST 0x00008000	special symbol for namelists • only valid for CLASS_VAR • used by whirl2f to identify namelist symbols
ST_IS_F90_TARGET 0x00010000	symbol may be accessed by dereferencing an F90 pointer • only valid for CLASS_VAR • if not set, no direct load or store to this symbol can alias with any load or store through an F90 pointer • if not set, no indirect load or store through an F90 pointer can access this item

Table 9 Miscellaneous Attributes of an ST Entry

Flag/Value	Description
ST_DECLARED_STATIC 0x00020000	VMS formals declared static • only valid for CLASS_VAR
ST_IS_EQUIVALENCED 0x00040000	part of an Fortran equivalence • only valid for CLASS_VAR
ST_IS_FILL_ALIGN 0x00080000	symbol has a fill_symbol or align_symbol pragma supplied • only valid for CLASS_VAR
ST_IS_OPTIONAL_ARGUMENT 0x00100000	formal parameter is optional • only valid for SCLASS_FORMAL and SCLASS_FORMAL_REF • it is illegal to speculate loads/stores of this symbol
ST_PT_TO_UNIQUE_MEM 0x00200000	memory location pointed to by this symbol cannot be accessed via any other way • only valid for SCLASS_VAR • only valid for pointer, or non-scalar type that contains pointers • only valid for compiler-generated symbols • for non-scalar type, such as a struct that contains a pointer or an array of pointers, this flag applies to all pointers within the structure • a pointer with this bit set refers to a memory location that is never accessed indirectly via any other pointer or directly via any local or global variable in the entire program • the compiler phase that sets this bit must guarantee that the above property holds even through inlining or other code motion • copying such pointers to another pointers is allowed, as long as these other pointers are never derefereced
ST_IS_TEMP_VAR 0x00400000	compiler generated temporary variable or formal parameters • only valid for SCLASS_AUTO, SCLASS_FORMAL, and SCLASS_FORMAL_REF
ST_IS_CONST_VAR 0x00800000	read-only static or global variable • only valid for CLASS_VAR • not valid for SCLASS_AUTO, SCLASS_FORMAL, and SCLASS_FORMAL_REF • compiler can allocate this symbol in read-only data segment

Table 9 Miscellaneous Attributes of an ST Entry

Flag/Value	Description
ST_ADDR_SAVED 0x01000000	the address of this symbol is saved to another variable • not valid for SCLASS_REG
ST_ADDR_PASSED 0x02000000	the address of this symbol is passed to another PU as actual parameter • not valid for SCLASS_REG • this flag is now re-computed by the compiler backend and is not set by the frontend
ST_IS_THREAD_PRIVATE 0x04000000	symbol is a private data object of an MP program • storage of this symbol is not shared by the threads of an MP program
ST_PT_TO_COMPILER_GENERATED_MEM 0x08000000	symbol is a pointer to compiler-allocated memory space • only valid for pointer type • only valid for compiler-generated symbols • pragmas or other data object attributes specified by users do not apply to this memory location because it is not visible to them
ST_IS_SHARED_AUTO 0x10000000	an automatic variable that is accessed within a parallel region and has shared scope • only valid for SCLASS_AUTO
ST_ASSIGNED_TO_DEDICATED_PREG 0x20000000	symbol is associated to a dedicated (hardware) register • compiler should always keep this symbol's value in the specified register • only valid for CLASS_VAR • must be volatile type
ST_ASM_FUNCTION_ST 0x40000000	name of this symbol is an assembly language code corresponding to a program unit • only valid for symbols in the global symbol table • only valid for CLASS_NAME, SCLASS_UNKNOWN • only valid for EXPORT_LOCAL • not valid for nested PU • must have valid PU_IDX, and the corresponding PU entry must have PU_NO_DELETE and PU_NO_INLINE bits set, with a 0 TY_IDX.

2.3.6 Exception Handling Region

Symbols of storage class SCLASS_EH_REGION are allocated by the code generator for the tables that control exception-handling. These tables are allocated in a special section created by the linker; they never correspond directly to program entities. They have no existence before code generation, so they are never referred to in the WHIRL.

Symbols of storage class SCLASS_EH_REGION_SUPP represent initialized variables created by the frontend to provide supplementary information about exception-handling actions to be taken by the exception-handling runtimes when an exception is thrown. They are allocated in a second special section created by the linker. They appear in the ereg_supp field of the WHIRL, but only the exception-handling part of the code generator should ever look at them.

The data in the sections corresponding to the storage classes SCLASS_EH_REGION and SCLASS_EH_REGION_SUPP should be read only by the exception-handling runtimes and should never be modified once it is generated.

Symbols of storage class SCLASS_EH_REGION or SCLASS_EH_REGION_SUPP have a very unique semantic with respect to storage and scope. They are local to the PU in terms of scope, meaning that they can only be referenced from within the defining PU. Their storage is not allocated form the stack, but from the global storage area. Hence, multiple instances of the same PU (e.g., recursive calls) share the same memory locations and values of these symbols. However, they differ from SCLASS_PSTATIC symbols in that when the defining PU is cloned or inlined, new copies of these symbols need to be created.

2.3.7 Semantics of Weak Symbols

The semantics of a weak symbol depends on its storage_class and base_idx, which is summarized in the following table:

Table 10 Semantics of Weak Symbols

storage_class	base_idx != st_idx	base_idx == st_idx
SCLASS_TEXT SCLASS_UGLOBAL SCLASS_DGLOBAL	weak symbol that has storage allocated[1]	weak definition before data layout[2]
SCLASS_EXTERN	weak symbol with an alias to a strong definition[3]	undefined weak symbol[4]

1. This refers to defined variables or functions that are marked weak. After layout, they can be based on other symbols. The weak flag means that they can be preempted by a strong definition. When they are preempted, their associated storage is either wasted or can be deleted.

2. Similar to (1), with the exception that storage of this symbol has not been laid out.

Basically, treat (1) and (2) as regular variable or function definitions, with the exception that they might be preempted by a strong definition. Once preempted, they corresponding storage cannot be referenced via this symbol name.

3. This is a weak alias to a strong definition. The name of this symbol is bound with the storage owned by the corresponding strong definition (specified by base_idx). The weak attribute makes this binding preemptible.

4. This symbol has no storage of its own and is not associated with any other symbol. The linker should not complain when no definition can be found, and should assign 0 as its address.

2.4 PU_TAB

Each entry of this table gives information about each PU that appears in the source file either as procedure declaration or function prototype. The index to this table, PU_IDX, can be used as a PU identifier.

The PU entry has the following structure, size 24 bytes:

Table 11 Layout of PU

Offset	Field	Description	Field size
byte 0	target_idx	TARGET_INFO_IDX to the target-specific info.	1 word
byte 4	prototype	TY_IDX to give the prototype type information	1 word
byte 8	lexical_level	lexical level (scope) of symbols in this PU	1 byte
byte 9	gp_group	gp-group number of this PU	1 byte
byte 10	src_lang	source language of this PU	1 byte
byte 11	unused	unused, must be filled with zeros.	5 bytes
byte 16	flags	flags associated with this function prototype	2 words

target_idx: Index to TARGET_INFO_TAB, which contains the target-specific information about this PU such as register usage information, etc. The TARGET_INFO_TAB is current undefined and is reserved for future expansion. In the current release, target_idx must be zero.

prototype: The TY_IDX for the type of the function.

lexical_level: Lexical level of symbols defined in this PU (i.e. index to the SCOPE array, see Section 2.2). It is always greater than 1.

gp_group: Gp-group id for this PU; used in multi-got program. Single GOT programs have gp_group zero.

src_lang: Source language of this PU, see Table 13.

unused: For alignment of flags, must be filled with zeros.

flags: Miscellaneous attributes, see Table 12.

Table 12 Miscellaneous Attributes of an PU Entry

Flag/Value	Description
PU_IS_PURE 0x00000001	pure function • does not modify the global state • does not make reference to the global state
PU_NO_SIDE_EFFECTS 0x00000002	does not modify the global state
PU_IS_INLINE_FUNCTION 0x00000004	inline keyword specified • function may be inlined
PU_NO_INLINE 0x00000008	function must not be inlined • mutually exclusive with PU_MUST_INLINE
PU_MUST_INLINE 0x00000010	function must be inlined • mutually exclusive with PU_NO_INLINE
PU_NO_DELETE 0x00000020	function must never be deleted
PU_HAS_EXC_SCOPES 0x00000040	has C++ exception handling region, or would have if exceptions were enabled. • PU_CXX_LANG must be set
PU_IS_NESTED_FUNC 0x00000080	a nested function • lexical_level must be larger than 2

Table 12 Miscellaneous Attributes of an PU Entry

Flag/Value	Description
PU_HAS_NON_MANGLED_CALL 0x00000100	function is called with non-reshaped array as actual parameter • must keep a copy of the function with non-mangled name
PU_ARGS_ALIASED 0x00000200	parameters might point to same or overlapping memory location • PU_F77_LANG or PU_F90_LANG must be set
PU_NEEDS_FILL_ALIGN_LOWERING 0x00000400	contains symbols that have the fill_symbol or align_symbol pragma specified
PU_NEEDS_T9 0x00000800	register $t9 must contain the lowest address of the PU
PU_HAS_VERY_HIGH_WHIRL 0x00001000	PU has very high WHIRL
PU_HAS_ALTENTRY 0x00002000	PU contains alternate entry points • PU_F77_LANG or PU_F90_LANG must be set
PU_RECURSIVE 0x00004000	PU is self-recursive, or is part of a multi-function recursion
PU_IS_MAINPU 0x00008000	main entry point of a program
PU_UPLEVEL 0x00010000	other PU nested in this one
PU_MP_NEEDS_LNO 0x00020000	must invoke LNO on this PU, regardless of compilation options
PU_HAS_ALLOCA 0x00040000	contains calls to alloca
PU_IN_ELF_SECTION 0x00080000	the code generator must put this PU in its own Elf section
PU_HAS_MP 0x00100000	contains a MP construct
PU_MP 0x00200000	a PU created by the MP lowerer
PU_HAS_NAMELIST 0x00400000	has namelist declaration • PU_F77_LANG or PU_F90_LANG must be set
PU_HAS_RETURN_ADDRESS 0x00800000	contain references to the special symbol __return_address

Table 12 Miscellaneous Attributes of an PU Entry

Flag/Value	Description
PU_HAS_REGION 0x01000000	PU has regions in it
PU_HAS_INLINES 0x02000000	PU has inlined code in it
PU_CALLS_SETJMP 0x04000000	PU contains calls to setjmp.
PU_CALLS_LONGJMP 0x08000000	PU contains calls to longjmp.
PU_IPA_ADDR_ANALYSIS 0x10000000	the ST_ADDR_SAVED bits for all symbols referenced in this PU are set by IPA's address analysis • the compiler backend should trust the (more accurate) results of IPA and need not recompute the ST_ADDR_SAVED bits for this PU
PU_SMART_ADDR_ANALYSIS 0x20000000	suppress the conservative address-taken validation • do not perform conservative address-taken verification, which might set the ST_ADDR_SAVED bit unnecessarily • set when more accurately address analysis has been performed.
0x40000000	obsolete
PU_HAS_GLOBAL_PRAGMAS 0x80000000	a dummy PU that contains global pragmas • a place holder for all global scope pragmas
PU_HAS_USER_ALLOCA 0x0000000100000000	PU contains user-specified call to alloca() • if this pu is inlined, an explicitly deallocation needs to be generated
PU_HAS_UNKNOWN_CONTROL_FLOW 0x0000000200000000	PU has control flow going in or out of the pu scope that do not following calling convention • tail-call optimization should be disabled

Table 13 Source Language of a PU

Flag/Value	Description
PU_UNKNOWN_LANG 0x00	Source language unknown
PU_MIXED_LANG 0x01	PU contains code from multiple source languages • resulted from cross-file inlining
PU_C_LANG 0x02	Source language is C
PU_CXX_LANG 0x04	Source language is C++
PU_F77_LANG 0x08	Source language is Fortran 77
PU_F90_LANG 0x10	Source language is Fortran 90
PU_JAVA_LANG 0x20	Source language is Java

2.5 TY_TAB

Each entry of this table is a TY. Any high level type in the program is uniquely identified by a value of type TY_IDX.

2.5.1 TY_IDX

TY_IDX is of size 32 bits, and is composed of two parts. The high order 24 bits is the index to TY_TAB. The low order 8 bits contains information that qualifies the type. Among the low order 8 bits is the alignment information. The actual alignment is given by 2^{align}.

Table 14 Layout of TY_IDX

Offset[1]	Field	Description	Field size
bit 0	align	alignment	5 bits
bit 5	const	const type qualifier	1 bit
bit 6	volatile	volatile type qualifier	1 bit
bit 7	restrict	restrict type qualifier	1 bit
bit 8	index	index to TY_TAB	24 bits

[1] Bit offsets assume big Endian bit ordering. For example, the index field is always the most significant 24 bits, regardless of the Endianess of the machine.

TY_IDX appears appear in many different places:

1. in WHIRL nodes that access data objects.
2. in ST entries.
3. in components for type specification: TY, FLD, TYLIST.

Each TY has a natural (and maximum) alignment, which can be determined by analysis of the details of the type. Thus, we omit the natural alignment information from the TY. The alignment of a TY directly affects the alignment in the TY_IDX of an object that encloses or refers to it, unless the object's own alignment is modified by pragmas or type casts. An optimization phase may also improve the alignment of an object by forcing better placement during data layout, in which case it only needs to fix up the alignment of the ST's TY_IDX. Whenever the alignment in the TY_IDX of the WHIRL node and the TY_IDX of the ST being accessed by the WHIRL node do not agree, code generation picks the more efficient (better) alignment of the two. Thus, if a phase worsens the alignment of an object, it has to fix the TY_IDX in all the WHIRL references to it, which is normally impossible.

The above rule dealing with alignment also applies to the other type qualifying bits: whenever a type qualifying bit is different between the TY_IDX of the WHIRL node and the TY_IDX of the ST being accessed by the WHIRL node, code generation picks the more efficient of the two.

2.5.2 TY entry

The TY entry has the following structure, size 24 bytes:

Table 15 Layout of TY

Offset	Field	Description	Field size
byte 0	size	size of the type in bytes	2 words
byte 8	kind	kind of type	1 byte
byte 9	mtype	corresponding WHIRL data type	1 byte
byte 10	flags	TY flags	2 bytes
byte 12	fld	FLD_IDX for struct/class field information	1 word
byte 12	tylist	TYLIST_IDX for function prototype	1 word
byte 12	arb	ARB_IDX for array bound description	1 word
byte 16	name_idx	STR_IDX to the name string	1 word
byte 20	etype	TY_IDX of array element (array only)	1 word
byte 20	pointed	TY_IDX of the pointed-to type (pointers only)	1 word
byte 20	pu_flags	function-specific attributes	1 word

size: The size of the type in bytes. For KIND_FUNCTION and KIND_VOID, the size is zero. For KIND_ARRAY, this is the size of the entire array, except when for variable length arrays, the size is zero.

kind: Field describing if the type is a scalar, structure, etc. See Table 16.

mtype: WHIRL data type, see Table 17. See Table 20 for valid combinations of mtype and kind.

flags: Miscellaneous attributes, see Table 18.

fld/tylist/arb: Index to one of the tables that provide additional type information, depending on the value of kind (see Table

16). For KIND_SCALAR, KIND_POINTER and KIND_VOID, this field is zero.

name_idx: The name of the type. For anonymous types, this field should be zero.

etype/pointed/pu_flags:For KIND_ARRAY, etype gives the type of the array element. For KIND_POINTER, pointed gives the type that it points to. For KIND_FUNCTION, pu_flags contains attributes of the function. For all other values of kind, this field is zero.

Types that are structurally identical can share common TY entries in order to minimize the size of TY_TAB.

Table 16 Kinds of TY

Name	Value	Description
KIND_INVALID	0	invalid or uninitialized
KIND_SCALAR	1	integer or floating point, no kids
KIND_ARRAY	2	array, arb_idx points to array bound description, etype gives the type of the array element
KIND_STRUCT	3	structure or union, fld_idx points to the field description
KIND_POINTER	4	pointers, pointed gives the type that it points to
KIND_FUNCTION	5	function or procedure, tylist_idx points to the list of TY_IDX for the return type and parameter types.
KIND_VOID	6	C void type, no kids

Table 17 WHIRL Basic Data Type

Flag	Value	Description
MTYPE_UNKNOWN	0	unknown type
MTYPE_B	1	boolean
MTYPE_I1	2	8-bit signed integer
MTYPE_I2	3	16-bit signed integer
MTYPE_I4	4	32-bit signed integer
MTYPE_I8	5	64-bit signed integer
MTYPE_U1	6	8-bit unsigned integer
MTYPE_U2	7	16-bit unsigned integer
MTYPE_U4	8	32-bit unsigned integer
MTYPE_U8	9	64-bit unsigned integer
MTYPE_F4	10	32-bit IEEE floating point
MTYPE_F8	11	64-bit IEEE floating point
MTYPE_F10	12	80-bit IEEE floating point
MTYPE_F16	13	128-bit IEEE floating point
MTYPE_STR MTYPE_STRING	14	character string
MTYPE_FQ	15	SGI long double

Table 17 WHIRL Basic Data Type

Flag	Value	Description
MTYPE_M	16	memory chunk, for structures
MTYPE_C4	17	32-bit complex
MTYPE_C8	18	64-bit complex
MTYPE_CQ	19	128-bit complex
MTYPE_V	20	void type
MTYPE_BS	21	bits
MTYPE_A4	22	32-bit address
MTYPE_A8	23	64-bit address
MTYPE_C10	24	80-bit IEEE complex
MTYPE_C16	25	128-bit IEEE complex
MTYPE_I16	26	128-bit signed integer
MTYPE_U16	27	128-bit unsigned integer

Table 18 Miscellaneous Attributes of a TY Entry

Flag/Value	Description
TY_IS_CHARACTER 0x0001	Fortran character type
TY_IS_LOGICAL 0x0002	Fortran logical type
TY_IS_UNION 0x0004	type is a union • only valid for KIND_STRUCT
TY_IS_PACKED 0x0008	struct or class is packed
TY_PTR_AS_ARRAY 0x0010	treat pointer as array (used by whirl2c/whirl2f)
TY_ANONYMOUS 0x0020	anonymous struct/class/union • only valid for KIND_STRUCT
TY_SPLIT 0x0040	split from a larger common block
TY_IS_F90_POINTER 0x0080	pointer is subject to F90 alias rules
TY_NOT_IN_UNION 0x0100	type cannot be part of a union

Table 18 Miscellaneous Attributes of a TY Entry

Flag/Value	Description
TY_NO_ANSI_ALIAS 0x0200	ANSI alias rules do not apply
TY_IS_NON_POD 0x0400	a C++ non-pod structure • constructor/destructor calls must be generated when creating a temp. variable of this type (usually done by the frontend)

Table 19 Attributes of a Function

Flag/Value	Description
TY_RETURN_TO_PARAM 0x00000001	a function returning a struct that is larger than twice the size of the largest integer type • an additional argument (first) is passed which contains the address where the return value is to be placed
TY_IS_VARARGS 0x00000002	allows variable number of arguments • the last formal parameter is a descriptor of the variable part of the parameter list
TY_HAS_PROTOTYPE 0x00000004	function has ANSI-style prototype defined.

Table 20 Valid Combinations of TY Kinds and WHIRL Data Types

Kind	Valid WHIRL data type
KIND_SCALAR	all mtypes except MTYPE_UNKNOWN and MTYPE_V
KIND_ARRAY	MTYPE_UNKNOWN and MTYPE_M
KIND_STRUCT	MTYPE_M
KIND_POINTER	MTYPE_U4 or MTYPE_U8 (for MIPS) MTYPE_A4 or MTYPE_A8 (for Merced)
KIND_FUNCTION	MTYPE_UNKNOWN
KIND_VOID	MTYPE_V

2.6 FLD_TAB

Each entry of this table gives information about a field in a struct or union. The TY of the struct type points to the FLD entry for the first field.

The remaining fields follow in consecutive FLD_TAB entries until a flag indicates it is the last field.

The FLD entry has the following structure, size 24 bytes:

Table 21 Layout of FLD

Offset	Field	Description	Field size
byte 0	name_idx	STR_IDX to the name string	1 word
byte 4	type	TY_IDX of field	1 word
byte 8	ofst	offset within struct in bytes	2 words
byte 16	bsize	bit field size in bits	1 byte
byte 17	bofst	bit field offset starting at byte specified by	1 byte
byte 18	flags	FLD flags	2 bytes
byte 20	st	ST_IDX to the ST entry, if any.	4 bytes

Table 22 Miscellaneous Attributes of an FLD Entry

Flag/Value	Description
FLD_LAST_FIELD 0x0001	indicate the last field in a struct
FLD_EQUIVALENCE 0x0002	this field belongs to an equivalence of a common block (i.e., overlaps in memory with other common block element(s))
FLD_BEGIN_UNION 0x0004	beginning of a union in a Fortran record
FLD_END_UNION 0x0008	end of a union in a Fortran record
FLD_BEGIN_MAP 0x0010	beginning of a map in a Fortran record
FLD_END_MAP 0x0020	end of a map in a Fortran record
FLD_IS_BIT_FIELD 0x0040	indicate a bit field • bsize and bofst are valid only if this flag is set

name_idx: STR_IDX to the name string, 0 if anonymous.

type: The TY_IDX of this field. If ofst is equal to the total size of the struct, the size of the type pointed to by type must be zero.

ofst: The byte offset of this field within the struct. This must be less than or equal to the total size of the struct. When the offset is equal to the size of the struct, type must be an TY_IDX of a type with zero size.

bsize: The size of the bit field in number of bits. Valid only if FLD_IS_BIT_FIELD is set; must be zero otherwise.

bofst: The bit field offset starting at the byte specified by ofst. Valid only if FLD_IS_BIT_FIELD is set; must be zero otherwise.

flags: Miscellaneous attributes, see Table 22.

st: ST_IDX to the (optional) ST entry corresponding to this field.

- typically used for common block elements where each element has a separate ST entry.
- the ST entry must be one in the global symbol table.
- when not set, must be zero.

2.7 TYLIST_TAB

Each entry of this table gives the type of each parameter in a function prototype. The TY of the function prototype points to the TYLIST entry that gives the return type. The ensuing entries give the types of the parameters. A TY_IDX value of 0 specifies the end of the parameter list.

The TYLIST entry has the following structure:

Table 23 Layout of TYLIST

Offset	Field	Description	Field size
byte 0	type	TY_IDX to the type	1 word

2.8 ARB_TAB

Each entry of this table gives information about a dimension of an array. The TY of the array type points to the ARB entry for the first dimension, indicated by ARB_FIRST_DIMEN. For C/C++ arrays, this corresponds to the leftmost dimension. For Fortran arrays, this corresponds to the rightmost dimension. The remaining dimensions follow in consecutive ARB_TAB entries until an entry with ARB_LAST_DIMEN set. The dimension of the array must be specified in dimension of every entry.

The ARB entry has the following structure, size 32 bytes:

Table 24 Layout of ARB

Offset	Field	Description	Field size
byte 0	flags	misc. attributes	2 bytes
byte 2	dimension	dimension of the array	2 bytes
byte 4	unused	unused, must be filled with zeros	1 word
byte 8	lbnd_val	constant lower bound value	2 words
byte 8	lbnd_var	ST_IDX of variable that stores the non-constant lower bound	1 word
byte 12	lbnd_unused	filler for lbnd_var, must be zero	1 word
byte 16	ubnd_val	constant upper bound value	2 words
byte 16	ubnd_var	ST_IDX of variable that stores the non-constant upper bound	1 word
byte 20	ubnd_unused	filler for ubnd_var, must be zero	1 word
byte 24	stride_val	constant stride	2 words
byte 24	stride_var	ST_IDX of variable that stores the non-constant stride	1 word
byte 28	stride_unused	filler for stride_var, must be zero	1 word

Table 25 Miscellaneous Attributes of an ARB Entry

Flags/Value	Description
ARB_CONST_LBND 0x0001	lower bound is constant
ARB_CONST_UBND 0x0002	upper bound is constant
ARB_CONST_STRIDE 0x0004	stride is constant
ARB_FIRST_DIMEN 0x0008	current dimension is first
ARB_LAST_DIMEN 0x0010	current dimension is last

2.9 TCON_TAB

Each entry of this table is the TCON for storing integer, floating point or string constant values. The first three entries of this table are reserved. The first entry (index 0) is reserved for uninitialized index value. The second entry (index 1) always contains 4-byte floating point value 0.0. the third entry (index 2) always contains 8-byte floating point value 0.0. These entries are shared. All other values are entered independently without checking for duplicates.

The TCON entry has the following structure, size 40 bytes:

Table 26 Layout of TCON

Offset	Field	Description	Field size
byte 0	ty	WHIRL data type, see Table 17	1 word
byte 4	flags	misc. attributes	1 word
byte 8	ival	signed integer (MTYPE_I1, MTYPE_I2, and MTYPE_I4)	1 word
byte 8	uval	unsigned integer (MTYPE_U1, MTYPE_U2, and MTYPE_U4)	1 word
byte 8	i0	64-bit signed integer (MTYPE_I8)	2 words
byte 8	k0	64-bit unsigned integer (MTYPE_U8)	2 words
byte 8	fval	32-bit floating point (MTYPE_F4) real part for 32-bit complex (MTYPE_C4)	1 word
byte 8	dval	64-bit floating point (MTYPE_F8) real part for 64-bit complex (MTYPE_C8)	2 words
byte 8	qval	128-bit floating point (MTYPE_FQ) real part for 128-bit complex (MTYPE_CQ)	4 words
byte 8	sval	string literal (MTYPE_STR/MTYPE_STRING) • byte 8 holds a character pointer (1 word) • byte 12 holds the number of bytes of the string (1 word)	3 words
byte 24	fival	imaginary part for 32-bit complex (MTYPE_C4)	1 word
byte 24	dival	imaginary part for 64-bit complex (MTYPE_C8)	2 words
byte 24	qival	imaginary part for 128-bit complex (MTYPE_CQ)	4 words

2.10 INITO_TAB

Each entry of this table connects an initialized global or static data object with an INITV entry (see Section 2.11), which describes the initial values. Each entry of this table is an INITO, which is identified by a value of type INITO_IDX.

2.10.1 INITO_IDX

INITO_IDX has an identical structure as a ST_IDX. It is of size 32 bits, and is composed of two parts:

Table 27 Layout of INITO_IDX

Field	Description	Field position and size
level	lexical level	least significant 8 bits
index	index to INITO_TAB	most significant 24 bits

The low order 8 bits are used to index into the SCOPE array in order to get to the INITO_TAB.

2.10.2 INITO Entry

The INITO entry has the following structure, size 8 bytes:

Table 28 Layout of INITO

Offset	Field	Description	Field size
byte 0	st_idx	ST_IDX of the variable to be initialized	1 word
byte 4	val	INITV_IDX of the initial values description	1 word

2.11 INITV_TAB

Each entry of this table specifies the initial value of a scalar component of a data object. Initial values of complex data objects are described by a tree of INITV entries, the root of which specified by the INITV_IDX of an INITO.

The INITV entry has the following structure, size 16 bytes:

Table 29 Layout of INITV

Offset	Field	Description	Field size
byte 0	next	INITV_IDX for the value of the next array element or the field in a struct	1 word
byte 4	kind	kind of the INITV, see Table 30.	2 bytes
byte 6	repeat1	repeat factor except for INITVKIND_VAL	2 bytes
byte 8	st	ST_IDX of symbol for INITVKIND_SYMOFF	1 word
byte 8	lab	LABEL_IDX of symbol for INITVKIND_LABEL	1 word
byte 8	lab1	LABEL_IDX of label for INITVKIND_SYMDIFF(16)	1 word
byte 8	mtype	WHIRL data type for INITVKIND_ZERO and INITVKIND_ONE	1 word
byte 8	tc	TCON_IDX for INITVKIND_VAL	1 word
byte 8	blk	INITV_IDX for INITVKIND_BLOCK	1 word
byte 8	pad	padding in bytes	1 word
byte 12	ofst	byte offset from st for INITVKIND_SYMOFF	1 word
byte 12	st2	ST_IDX of symbol for INITVKIND_SYMDIFF(16)	1 word
byte 12	repeat2	repeat factor for INITVKIND_ZERO, INITVKIND_ONE, and INITVKIND_VAL	1 word
byte 12	unused	filler for INITVKIND_BLOCK, INITVKIND_PAD, and INITVKIND_LABEL, must be zero	1 word

Table 30 INITVKIND

Name	Value	Description
INITVKIND_SYMOFF	1	value is the address of the symbol (st) plus offset (ofst)
INITVKIND_ZERO	2	integer value zero
INITVKIND_ONE	3	integer value one
INITVKIND_VAL	4	an integer, floating point, or string, specified by a TCON (tc)
INITVKIND_BLOCK	5	specifies another list or tree of INITVs
INITVKIND_PAD	6	amount of padding in bytes
INITVKIND_SYMDIFF	7	value is the difference of the addresses of a label and a symbol (lab1 – st2)

Table 30 INITVKIND

Name	Value	Description
INITVKIND_SYMDIFF16	8	same as INITVKIND_SYMDIFF, except the value is 2 bytes in size
INITVKIND_LABEL	9	value is the address of the label (lab)

next/blk: The values of a data object are specified by a tree of INITVs, with the root of the tree pointed to by the INITO. INITVs specifying scalars are linked up by the next field, each of which contains an INITV_IDX. The end of a link is specified by a zero INITV_IDX. Aggregate values are grouped into a separate links headed by the blk field, which must not be the zero INITV_IDX.

kind: Kind of this INITV entry, see Table 30.

repeat1: Specifies the repeat factor of the value in this INITV entry. This cuts down the number of unnecessary duplicates. A repeat factor of one means only one instance of the value is needed. The repeat factor is never zero, except for INITVKIND_ZERO, INITVKIND_ONE, and INITVKIND_VAL, which use repeat2 instead.

st/ofst: For INITVKIND_SYMOFF, the value of this entry is equal to the address of the symbol specified by st, plus the byte offset specified by ofst.

label: For INITVKIND_LABEL, the value of this entry is equal to the address of the label specified by lab.

lab1/st2: For INITVKIND_SYMDIFF or INITVKIND_SYMDIFF16, the value of this entry is equal to the difference between the addresses of the label specified by lab1 and of the symbol specified by st2. It is a signed value equal to (lab1 – st2). For INITVKIND_SYMDIFF16, the size of the value is 2 bytes.

mtype/repeat2: For INITVKIND_ZERO and INITVKIND_ONE, this entry specifies an integral value of zero and one respectively. The WHIRL data type (signed/unsigned, size, etc.) is specified by mtype. It uses repeat2 as its repeat factor instead of repeat1.

tc/repeat2: For INITVKIND_VAL, this specifies a TCON for the scalar constant value. It uses repeat2 as its repeat factor instead of repeat1.

pad: For INITKIND_PAD, this specifies the padding in bytes. The padded value is undefined.

2.12 BLK_TAB

Each entry of this table gives information about the layout of a data block, which corresponds to a contiguous chunk of memory in the user program. Program variables are laid out with respect to data blocks. This table is created by the back end and is usually local to the back end, but can be written out to the file.

The BLK entry has the following structure, size 16 bytes:

Table 31 Layout of BLK

Offset	Field	Description	Field size
byte 0	size	size of the block	2 words
byte 8	align	alignment of the blocks: 1, 2, 4, 8	2 bytes
byte 10	flags	flags for this field, see Table 32	2 bytes
byte 12	section_idx	section index (0 if not a section) • refers to the section info in data_layout.cxx	2 bytes
byte 14	scninfo_idx	Elf scninfo_idx (0 if not a section) • refers to the Elf section info in cgemit.cxx	2 bytes

Table 32 Miscellaneous Attributes of an BLK Entry

Flag/Value	Description
BLK_SECTION 0x0001	represents an Elf section
BLK_ROOT_BASE 0x0002	block should not be merged
BLK_IS_BASEREG 0x004	block that maps into a register
BLK_DECREMENT 0x0008	grow block by decrementing
BLK_EXEC 0x0010	executable instructions (SHF_EXEC)
BLK_NOBITS 0x0020	occupies no space in file (SHT_NOBITS)
BLK_MERGE 0x0040	merge duplicates in linker (SHF_MERGE)
BLK_COMPILER_LAYOUT 0x0080	layout of all symbols within this block is done by the compiler • this implies that user's code cannot legally use address arithmetic to move from one of the symbols to another

2.13 STR_TAB

This table holds all character strings for names of symbols, types, labels, etc. This table can be viewed as a block of storage area for character strings. STR_IDX is the index to this table, and is actually an offset in this block of storage; it gives the byte offset of the starting character of a literal string. All strings are null-terminated, and the first character of the block is always nul. Thus, a zero STR_IDX represents a null string. Wide characters or unicode for names are not yet supported.

2.14 TCON_STR_TAB

This table holds all character strings defined in the user program. It is very similar to STR_TAB, with the exception that the strings need not be null-terminated, and nul characters are allowed anywhere within the string. The exact length of each string is explicitly specified.

2.15 LABEL_TAB

Each entry of this table is a LABEL, which gives the information associated with a WHIRL label. The index to this table is the WHIRL label number.

The LABEL entry has the following structure:

Table 33 Layout of LABEL

Offset	Field	Description	Field size
byte 0	name_idx	STR_IDX to the name string, must be zero if no name	1 word
byte 4	flags	LABEL flags	3 bytes
byte 7	kind	kind of label	1 byte

Table 34 LABEL Kind

Name	Value	Description
LKIND_DEFAULT	0	ordinary label
LKIND_ASSIGNED	1	
LKIND_BEGIN_EH_RANGE	2	
LKIND_END_EH_RANGE	3	
LKIND_BEGIN_HANDLER	4	
LKIND_END_HANDLER	5	

Table 35 Miscellaneous Attributes of an LABEL Entry

Flag/Value	Description
LABEL_TARGET_OF_GOTO_OUTER_BLOCK 0x000001	control might be passed from outside of the current block to this label.
LABEL_ADDR_SAVED 0x000002	address of this label is saved to a variable
LABEL_ADDR_PASSED 0x000040	address of this label is passed to another PU as actual parameter

2.16 PREG_TAB

Each entry of this table is a PREG, which gives the information associated with a pseudo-register in WHIRL. Pseudo-register numbers 0 — 71 are reserved for dedicated hardware pseudo-registers. All compiler-generated pseudo-registers start with number 72. As a result, the index to this table is the pseudo-register number, minus 71 (Note: by definition, index 0 to the PREG_TAB is reserved for undefined value).

The PREG entry has the following structure:

Table 36 Layout of PREG

Offset	Field	Description	Field size
byte 0	name_idx	STR_IDX to the name string, must be zero if no name	1 word

2.17 ST_ATTR_TAB

Each entry of this table associates certain attribute with an ST entry. Symbol attributes specified here usually cannot be represented by a single bit, and are possessed by a very small subset of the ST entries, and thus are too expensive to be included as part of the ST entry proper. For most PU, this table is expected to be empty.

The ST_ATTR entry has the following structure, size 12 bytes:

Table 37 Layout of ST_ATTR

Offset	Field	Description	Field size
byte 0	st_idx	ST_IDX of the corresponding symbol	1 word
byte 4	kind	kind of the ST_ATTR, see Table 38	1 word
byte 8	reg_id	dedicated (physical) register associated with this symbol • symbol must have ST_ASSIGNED_TO_DEDICATED_PREG bit set	1 word
byte 8	section_name	STR_IDX of the name of the Elf section where this symbol is defined • symbol must be in global scope	1 word

Table 38 Kinds of ST_ATTR

Name	Value	Description
ST_ATTR_DEDICATED_REGISTER	0	dedicated register
ST_ATTR_SECTION_NAME	1	section name

2.18 FILE_INFO

This structure is not really part of the symbol table, it holds miscellaneous information that is derived from the symbol table but does not fit well in any global symbol table. Typically, this information is needed by the compiler backend to set up proper mode of operation before any PU is processed.

A FILE_INFO has the following structure, size 8 bytes:

Table 39 Layout of FILE_INFO

Offset	Field	Description	Field size
byte 0	flags	misc. attributes, see Table 40	1 word
byte 4	gp_group	gp-group id of this file, 0 for single-GOT file	1 byte
byte 5	unused	unused, must be zero	3 bytes

Table 40 Miscellaneous Attributes of FILE_INFO

Flag/Value	Description
FI_IPA 0x00000001	IPA generated file
FI_NEEDS_LNO 0x00000002	some PUs in this file has the flag PU_MP_NEEDS_LNO set
FI_HAS_INLINES 0x00000004	some PUs in this file has the flag PU_HAS_INLINES set
FI_HAS_MP 0x00000008	some PUs in this file has the flag PU_HAS_MP set

2.19 Backend-Specific Tables

This section describes addition symbol tables that are created and used solely by the compiler backend. Each entry in these tables holds addition information associated with the corresponding regular symbol table entries. They are discarded at the end of the backend's processing and are never written out to a file.

Note that these tables are not part of the WHIRL symbol table specification and are implementation specific. The following descriptions apply only to the current implementation of the SGI Pro64 compilers.

2.19.1 BE_ST_TAB

This table is parallel to the ST_TAB. Each entry of this table is a BE_ST, which corresponds to an ST entry. The same ST_IDX is used to index an BE_ST entry in a BE_ST_TAB and the corresponding ST entry in the ST_TAB.

The BE_ST entry has the following structure, size 8 bytes:

Table 41 Layout of BE_ST

Offset	Field	Description	Field size
byte 0	flags	BE_ST flags	1 word
byte 4	io_auxst	pointer to an internal data structure used by the Fortran I/O routines.	1 word

Table 42 Miscellaneous Attributes of an BE_ST entry

Flag/Value	Description
BE_ST_ADDR_USED_LOCALLY 0x00000001	address of this symbol is taken somewhere within the current PU • this flag is computed based on the backend's analysis
BE_ST_ADDR_PASSED 0x00000002	address if this symbol is passed by reference • this flag is computed based on the backend's analysis • this flag is different from ST_ADDR_PASSED, which is set by the frontend based on the source language's semantics

Table 42 Miscellaneous Attributes of an BE_ST entry

Flag/Value	Description
BE_ST_W2FC_REFERENCED 0x00000004	whirl2c or whirl2f sees a reference to this symbol
BE_ST_UNKNOWN_CONST 0x00000008	symbol is a constant but with unknown value • generated by LNO
BE_ST_PU_HAS_VALID_ADDR_FLAGS 0x00000010	indicate that the BE_ST_ADDR_USED_LOCALLY and BE_ST_ADDR_PASSED bits are valid for the PU specified by corresponding ST entry. • valid only for CLASS_FUNC • depending on the optimization level, the above two BE_ST_ADDR flags might not be valid • tail-call optimization can be performed only when BE_ST_PU_HAS_VALID_ADDR_FLAGS is set
BE_ST_PU_NEEDS_ADDR_FLAG_ADJUST 0x00000020	indicate that the ST_ADDR_SAVED and ST_ADDR_PASSED bits are no longer valid • typically set by the MP-lowerer • needs to recompute the above two bits before moving on the next phase in the backend

2.20 Symbol Table Interfaces

The symbol table interfaces are described in a separate document. An online version can be found in http://sahara.mti.sgi.com/Projects/Symtab/porting.html/.