Open Watcom Debugger Interface

Originally written by WATCOM International Corp.
Revised by Open Watcom contributors

Table of Contents

WATCOM Debugging Information Format VERSION 4.0 1

1 Debugging Information Format 3

2 Object file structures 5
2.1 Version number and source language identification 5
2.2 Line number information 5
2.3 Location information 5
2.4 Typing information 8
2.4.1 TYPE_NAME (value 0x1?) 8
2.4.2 ARRAY (value 0x2?) 9
2.4.3 SUBRANGE (value 0x3?) 9
2.4.4 POINTER (value 0x4?) 10
2.4.5 ENUMERATED (value 0x5?) 10
2.4.6 STRUCTURE (value 0x6?) 11
2.4.7 PROCEDURE (value 0x7?) 12
2.4.8 CHARACTER_BLOCK (value 0x8?) 12
2.5 Local symbol information 13
2.5.1 VARIABLE (value 0x1?) 13
2.5.2 CODE (value 0x2?) 13
2.5.3 NEW_BASE (value 0x3?) 15

3 Executable file structures 17
3.1 Master debug header 17
3.2 Source language table 19
3.3 Segment address table 19
3.4 Section debug information 19
3.4.1 Section debug header 19
3.4.2 Local symbols class 20
3.4.3 Types class 20
3.4.4 Line numbers class 20
3.4.4.1 Special Line Number Table 21
3.4.5 Module information class 22
3.4.6 Global symbols class 23
3.4.7 Address information class 24

Debugger Trap File Interface VERSION 1.3 25

1 Trap File Interface 27
1.1 Some Definitions 27
1.1.1 Byte Order 27
1.1.2 Pointer Sizes 28
1.1.3 Base Types 28

2 The Request Interface 29
2.1 Request Structure. 29
2.2 The Interface Routines 29
2.2.1 TrapInit 29
2.2.2 TrapRequest 30
2.2.2.1 Request Example 30
2.2.3 TrapFini 31

Table of Contents

3 The Requests 33
3.1 Core Requests 33
3.1.1 REQ_CONNECT (0) 33
3.1.2 REQ_DISCONNECT (1) 34
3.1.3 REQ_SUSPEND (2) 34
3.1.4 REQ_RESUME (3) 34
3.1.5 REQ_GET_SUPPLEMENTARY_SERVICE (4) 34
3.1.6 REQ_PERFORM_SUPPLEMENTARY_SERVICE (5) 35
3.1.7 REQ_GET_SYS_CONFIG (6) 35
3.1.8 REQ_MAP_ADDR (7) 37
3.1.9 REQ_CHECKSUM_MEM (8) 37
3.1.10 REQ_READ_MEM (9) 38
3.1.11 REQ_WRITE_MEM (10) 38
3.1.12 REQ_READ_IO (11) 38
3.1.13 REQ_WRITE_IO (12) 39
3.1.14 REQ_PROG_GO (13)/REQ_PROG_STEP (14) 39
3.1.15 REQ_PROG_LOAD (15) 40
3.1.16 REQ_PROG_KILL (16) 41
3.1.17 REQ_SET_WATCH (17) 41
3.1.18 REQ_CLEAR_WATCH (18) 41
3.1.19 REQ_SET_BREAK (19) 42
3.1.20 REQ_CLEAR_BREAK (20) 42
3.1.21 REQ_GET_NEXT_ALIAS (21) 42
3.1.22 REQ_SET_USER_SCREEN (22) 43
3.1.23 REQ_SET_DEBUG_SCREEN (23) 43
3.1.24 REQ_READ_USER_KEYBOARD (24) 43
3.1.25 REQ_GET_LIB_NAME (25) 43
3.1.26 REQ_GET_ERR_TEXT (26) 44
3.1.27 REQ_GET_MESSAGE_TEXT (27) 44
3.1.28 REQ_REDIRECT_STDIN (28)/REQ_REDIRECT_STDOUT (29) 45
3.1.29 REQ_SPLIT_CMD (30) 45
3.1.30 REQ_READ_REGS (31) 45
3.1.31 REQ_WRITE_REGS (32) 46
3.1.32 REQ_MACHINE_DATA (33) 46
3.2 File I/O requests 46
3.2.1 REQ_FILE_GET_CONFIG (0) 47
3.2.2 REQ_FILE_OPEN (1) 47
3.2.3 REQ_FILE_SEEK (2) 47
3.2.4 REQ_FILE_READ (3) 48
3.2.5 REQ_FILE_WRITE (4) 48
3.2.6 REQ_FILE_WRITE_CONSOLE (5) 49
3.2.7 REQ_FILE_CLOSE (6) 49
3.2.8 REQ_FILE_ERASE (7) 49
3.2.9 REQ_FILE_STRING_TO_FULLPATH (8) 50
3.2.10 REQ_FILE_RUN_CMD (9) 50
3.3 Overlay requests 51
3.3.1 REQ_OVL_STATE_SIZE (0) 51
3.3.2 REQ_OVL_GET_DATA (1) 52
3.3.3 REQ_OVL_READ_STATE (2) 52
3.3.4 REQ_OVL_WRITE_STATE (3) 52
3.3.5 REQ_OVL_TRANS_VECT_ADDR (4) 53
3.3.6 REQ_OVL_TRANS_RET_ADDR (5) 53

Table of Contents

3.3.7 REQ_OVL_GET_REMAP_ENTRY (6) 53
3.4 Thread requests 54
3.4.1 REQ_THREAD_GET_NEXT (0) 54
3.4.2 REQ_THREAD_SET (1) 54
3.4.3 REQ_THREAD_FREEZE (2) 55
3.4.4 REQ_THREAD_THAW (3) 55
3.4.5 REQ_THREAD_GET_EXTRA (4) 55
3.5 RFX requests 56
3.5.1 REQ_RFX_RENAME (0) 56
3.5.2 REQ_RFX_MKDIR (1) 56
3.5.3 REQ_RFX_RMDIR (2) 56
3.5.4 REQ_RFX_SETDRIVE (3) 57
3.5.5 REQ_RFX_GETDRIVE (4) 57
3.5.6 REQ_RFX_SETCWD (5) 57
3.5.7 REQ_RFX_GETCWD (6) 58
3.5.8 REQ_RFX_SETDATETIME (7) 58
3.5.9 REQ_RFX_GETDATETIME (8) 58
3.5.10 REQ_RFX_GETFREESPACE (9) 59
3.5.11 REQ_RFX_SETFILEATTR (10) 59
3.5.12 REQ_RFX_GETFILEATTR (11) 59
3.5.13 REQ_RFX_NAMETOCANONICAL (12) 60
3.5.14 REQ_RFX_FINDFIRST (13) 60
3.5.15 REQ_RFX_FINDNEXT (14) 61
3.5.16 REQ_RFX_FINDCLOSE (15) 61

4 System Dependent Aspects 63
4.1 Trap Files Under DOS 63
4.2 Trap Files Under OS/2 63
4.3 Trap Files Under Windows. 64
4.4 Trap Files Under Windows NT. 64
4.5 Trap Files Under QNX 64
4.6 Trap Files Under Netware 386 or PenPoint 65

Overlay Manager Interface VERSION 3.0 67

1 Overlay manager interface 69
1.1 The Hook Routine 69
1.2 The Handler Routine 70
1.2.1 GET_STATE_SIZE 70
1.2.2 GET_OVERLAY_STATE 70
1.2.3 SET_OVERLAY_STATE 70
1.2.4 TRANSLATE_VECTOR_ADDR 71
1.2.5 TRANSLATE_RETURN_ADDR 71
1.2.6 GET_OVL_TBL_ADDR 71
1.2.7 GET_MOVED_SECTION 72
1.2.8 GET_SECTION_DATA 72
1.3 Overlay Table Structure 73

WATCOM Debugging Information Format VERSION 4.0

1 Debugging Information Format

This document describes the object and executable file structures used by the Open Watcom Debugger to provide symbolic information about a program. This information is subject to change.

Note that version 4.0 of the Open Watcom debugger supports the DWARF and CodeView symbolic debugging information formats in addition to the format described in this document. For the purposes of discussion, this format will be known as the "WATCOM" format. DWARF is now the primary format used by Open Watcom compilers. Support for generating the WATCOM format will probably remain but is only useful for debugging DOS overlays.

Before reading this document you should understand the Intel 8086 Object Module Format (OMF). This format is described in the Intel document *8086 Relocatable Object Module Formats* and also the October 1985 issue of *PC Tech Journal*.

Responsibility for the Intel/Microsoft OMF specification has been taken over by the Tools Interface Standards (TIS) Committee. The TIS standards (including the OMF spec) may be obtained by phoning the Intel literature center at 1-800-548-4725 and asking for order number 241597.

This document is for the Open Watcom Debugger version 4.0 (or above.)

4 Debugging Information Format

2 Object file structures

The compiler is responsible for placing extra information into the object file in order to provide symbolic information for the Open Watcom Debugger. There are three classes of information, each of which may be present or absent from the file individually. These classes are line number, type and local symbol information.

For the Open Watcom C compiler, line number information is provided when the "/d1" switch is used and all three classes are provided when the "/d2" switch is used.

2.1 Version number and source language identification

Since there may be different versions of the type and local symbol information, and there may be multiple front-ends a special OMF COMENT record is placed in the object file. It has the following form:

```
comment_class = 0xfe
'D'
major_version_number (char)
minor_version_number (char)
source_language (string)
```

The **comment_class** of 0xfe indicates a linker directive comment. The character 'D' informs the linker that this record is providing debugging information. The **major_version_number** is changed whenever there is a modification made to the types or local symbol classes that is not upwardly compatible with previous versions. The **minor_version_number** increments by one whenever a change is made to those classes that is upwardly compatible with previous versions. The **source_language** field is a string which determines what language that the file was compiled from.

If the debugging comment record is not present, the local and type segments (described later) are not in WATCOM format and should be omitted from the resulting executable file's debugging information. The current major version is one, and the current minor version is three.

2.2 Line number information

Line number information is provided by standard Intel OMF LINNUM records. A kludge has been added that allows for line numbers to refer to more than one source file. See the section on the "Special Line Number Table" in the executable structures portion of the document for more details.

2.3 Location information

A type or symbol definition may contain a location field. This field is of variable length and identifies the memory (or register) location of the symbol in question. A location field may consist of a single entry, or a list of entries. Each entry describes an operation of a stack machine. The value of the location field is the top entry of the stack after all the operations have been performed. To tell whether a field is a single entry or a list, the first byte is examined. If the value of the byte is greater than 0x80, then the field consists of a

list of entries, and the length in bytes of the list is the value of the first byte minus 0x80. If the first byte is less than 0x80, the byte is the first byte of a single entry field. The top nibble of the first byte in each entry is a general location class while the low nibble specifies the sub-class.

```
BP_OFFSET    (value 0x1?)
    BYTE     (value 0x10) offset_byte
    WORD     (value 0x11) offset_word
    DWORD    (value 0x12) offset_dword

CONST        (value 0x2?)
    ADDR286  (value 0x20) memory_location_32_pointer
    ADDR386  (value 0x21) memory_location_48_pointer
    INT_1    (value 0x22) const_byte
    INT_2    (value 0x23) const_word
    INT_4    (value 0x24) const_dword

MULTI_REG    (value 0x3?)
    Low nibble is number of register bytes that follow - 1.
    The registers are specified low order register first.

REG          (value 0x4?)
    Low nibble is low nibble of the appropriate register value.
    This may only be used for the first 16 registers.

IND_REG      (value 0x5?)
    CALLOC_NEAR (value 0x50) register_byte
    CALLOC_FAR  (value 0x51) register_byte, register_byte
    RALLOC_NEAR (value 0x52) register_byte
    RALLOC_FAR  (value 0x53) register_byte, register_byte

OPERATOR      (value 0x6?)
    IND_2        (value 0x60)
    IND_4        (value 0x61)
    IND_ADDR286  (value 0x62)
    IND_ADDR386  (value 0x63)
    ZEB          (value 0x64)
    ZEW          (value 0x65)
    MK_FP        (value 0x66)
    POP          (value 0x67)
    XCHG         (value 0x68) stack_byte
    ADD          (value 0x69)
    DUP          (value 0x6a)
    NOP          (value 0x6b)


Here is the list of register numbers:
 0-AL,  1-AH,   2-BL,   3-BH,   4-CL,   5-CH,   6-DL,   7-DH
 8-AX,  9-BX,   10-CX,  11-DX,  12-SI,  13-DI,  14-BP,  15-SP
16-CS,  17-SS,  18-DS,  19-ES
20-ST0, 21-ST1, 22-ST2, 23-ST3, 24-ST4, 25-ST5, 26-ST6, 27-ST7
28-EAX, 29-EBX, 30-ECX, 31-EDX, 32-ESI, 33-EDI, 34-EBP, 35-ESP
36-FS,  37-GS
```

CONST pushes a single constant value onto the expression stack. INT_1 and INT_2 constant values are sign-extended to four bytes before being pushed.

The OPERATOR class performs a variety of operations on the expression stack.

IND_2 Pick up two bytes at the location specified by the top entry of the stack, sign-extend to four bytes and replace top of stack with the result.

IND_4 Replace the top of stack with the contents of the four bytes at the location specified by the top of stack.

IND_ADDR286 Replace the top of stack with the contents of the four bytes, treated as a far pointer, at the location specified by the top of stack.

IND_ADDR386 Replace the top of stack with the contents of the six bytes, treated as a far pointer, at the location specified by the top of stack.

ZEB Zero extend the top of stack from a byte to a dword (clear the high three bytes).

ZEW Zero extend the top of stack from a word to a dword.

MK_FP Remove the top two entries from the stack, use the top of stack as an offset and the next element as a segment to form a far pointer and push that back onto the stack.

POP Remove the top entry from the stack.

XCHG Exchange the top of stack with the entry specified by **`stack_byte`**. "XCHG 1" would exchange the top of stack with the next highest entry.

ADD Remove the top two entries from the stack, add them together and push the result.

DUP Duplicate the value at the top of the stack.

NOP Perform no operation.

REG and MULTI_REG push the 'lvalue' of the register. If they are the only entry then the symbol exists in the specified register. To access the value of the register, you must indirect it.

BP_OFFSET locations are for variables on the stack. The values given are offsets from the BP register for 286 programs and from the EBP register for 386 programs. A BP_OFFSET could also be expressed with the following series of operations:

```
MULTI_REG(1) SS
IND_2
MULTI_REG(1) EBP
IND_4
MK_FP
INT_1 offset_byte
ADD
```

The IND_REG location type is used for structured return values. The register or register pair is used to point at the memory location where the structure is returned. CALLOC means that the calling procedure is responsible for allocating the return area and passing a pointer to it as a parameter in the specified registers. RALLOC means that the called routine allocated the area and returns a pointer to it in the given registers.

2.4 Typing information

The Open Watcom Debugger typing information is contained in a special segment in the object file. The segment name is "$$TYPES" and the segment class is "DEBTYP". To allow greater flexibility in demand loading the typing information and also let it exceed 60K for a single module, each object file may have multiple $$TYPES segments. Each segment is identified by an entry in the demand link table (described in the executable file structures section). No individual segment may exceed 60K and no individual type record may be split across a segment boundry. Also, any type which is described by multiple records (structures, enums, procedures) may not be split across a segment boundry. Since each segment is loaded as a whole by the debugger when demand loading, increasing the segment size requires larger amounts of contiguous memory be present in the system. Decreasing the size of the individual segments reduces memory requirements, but increases debugger lookup time since it has to traverse more internal structures. The current code generator starts a new type segment when the current one exceeds 16K. The segments are considered to be a stream of variable length definitions, with each definition being preceded by a length byte. A number of the definitions contain indices of some form. These indices are standard Intel format, with 0 meaning no index, 1 to 127 is represented in one byte, 128 to 32767 in high byte/low byte form with the top bit on in the high byte. Definitions are given index numbers by the order in which they appear in the module, with the first being index one. Character strings representing names are always placed at the end of a definition so that their length can be calculated by subtracting the name's start point from the length of the record. They are not preceded by a length byte or followed by a zero byte.

The first byte identifies the kind of the type definition that follows. The top nibble of the byte is used to indicate the general class of the type definition (there are eight of these). The low order nibble is used to qualify the general type class and uniquely identify the definition type.

2.4.1 TYPE_NAME (value 0x1?)

This definition is used to give names to types. There are three sub-classes.

```
SCALAR     (value 0x10) scalar_type_byte, name
SCOPE      (value 0x11) name
NAME       (value 0x12) scope_index, type_index, name
CUE_TABLE  (value 0x13) table_offset_dword
EOF        (value 0x14)
```

SCALAR is used to give a name to a basic scalar type. It can also be used to give a type index to a scalar type without a name by specifying the null name. The **`scalar_type_byte`** informs the Open Watcom Debugger what sort of scalar item is being given a name. It has the following form:

```
BIT: 7 6 5 4 3 2 1 0
     | |   | |     |
     | |   | +-----+--- size in bytes - 1
     | +---+----------- class (000 - integer)
     |                        (001 - unsigned)
     |                        (010 - float)
     |                        (011 - void (size=0))
     |                        (100 - complex)
     +----------------- unused
```

To create an unnamed scalar type, for use in other definitions, just use a zero length name.

Notes: BASIC would have been a better name for this, since complex is not a scalar type, but the name was chosen before complex support was added.

SCOPE is used to restrict the scope of other type names. A restricted scope type name must be preceded by its appropriate scope name in order for the Open Watcom Debugger to recognize it as a type name. This is useful for declaring C structure, union, and enum tag names. You declare SCOPE names of "struct", "union", and "enum" and then place the appropriate value in the **scope_index** field of the NAME record when declaring the tag.

NAME gives an arbitrary type a name. The field, **scope_index**, is either zero, which indicates an unrestricted type name, or is the type index of a SCOPE definition, which means that the type name must be preceded by the given scope name in order to be recognized.

The next two records are kludges to allow OMF line numbers to refer to more than one source file. See the section of on the "Special Line Number Table" in the executable structure for more details.

CUE_TABLE is followed by **table_offset_dword** which gives the offset in bytes from the begining of the typing information for a module to the special line number table. If this record is present, it must be in the first $$TYPES segment for the module and preferably as close to the begining of the segment as possible.

EOF marks the end of the typing information for the module and the begining of the special line number table.

2.4.2 ARRAY (value 0x2?)

This definition is used to define an array type. There are 6 sub-classes.

```
BYTE_INDEX     (value 0x20) high_bound_byte, base_type_index
WORD_INDEX     (value 0x21) high_bound_word, base_type_index
LONG_INDEX     (value 0x22) high_bound_dword, base_type_index
TYPE_INDEX     (value 0x23) index_type_index, base_type_index
DESC_INDEX     (value 0x24) scalar_type_byte, scalar_type_byte,
                                bounds_32_pointer, base_type_index
DESC_INDEX_386 (value 0x25) scalar_type_byte, scalar_type_byte,
                                bounds_48_pointer, base_type_index
```

BYTE_INDEX, WORD_INDEX, LONG_INDEX are all used to describe a restricted form of array. If one of these forms is used then the index type is an integer with the low bound of the array being zero and the high bound being whatever is specified.

The DESC_INDEX form is used when the array bounds are not known at compile time. The **bounds_32_pointer** is a far pointer to a structure in memory. The type and size of the first field is given by the first **scalar_type_byte** and indicates the lower bound for the index. The second field's type and size is given by the second **scalar_type_byte** . This field gives the number of elements in the array.

The DESC_INDEX_386 is the same as DESC_INDEX except that a 48-bit far pointer is used to locate the structure in memory.

2.4.3 SUBRANGE (value 0x3?)

This definition is used to define a subrange type. There are 3 sub-classes.

```
BYTE_RANGE (value 0x30) lo_bnd_byte, hi_bnd_byte, base_type_index
WORD_RANGE (value 0x31) lo_bnd_word, hi_bnd_word, base_type_index
LONG_RANGE (value 0x32) lo_bnd_dword, hi_bnd_dword, base_type_index
```

If the base type is unsigned then the low and high bounds should be interpreted as containing unsigned quantities, otherwise they contain integers. However, the decision to use the byte, word, or long form of the definition is always made considering the high and low bounds as signed numbers.

2.4.4 POINTER (value 0x4?)

This definition is used to define a pointer type. There are 10 sub-classes.

```
NEAR            (value 0x40) base_type_index [,base_locator]
FAR             (value 0x41) base_type_index
HUGE            (value 0x42) base_type_index
NEAR_DEREF      (value 0x43) base_type_index [,base_locator]
FAR_DEREF       (value 0x44) base_type_index
HUGE_DEREF      (value 0x45) base_type_index
NEAR386         (value 0x46) base_type_index [,base_locator]
FAR386          (value 0x47) base_type_index
NEAR386_DEFREF  (value 0x48) base_type_index [,base_locator]
FAR386_DEREF    (value 0x49) base_type_index
```

When a symbol is one of the *_DEREF types, the Open Watcom Debugger will automatically dereference the pointer. This "hidden" indirection may be used to define reference parameter types, or other indirectly located symbols. The *_DEREF types have now been superceeded by location expressions. They should no longer be generated. The NEAR* pointer types all have an optional **base_locator** field. The debugger can tell if this field is present by examining the length of the debug type entry at the begining of the record and seeing if there are additional bytes after the **base_type_index** field. If there are more bytes, the **base_locator** is a location expression whose result is an address, the value of which is the base selector and offset value when indirecting through the pointer (based pointers). The contents of the based pointer variable are added to result of the location expression to form the true resulting address after an indirection. The address of the pointer variable being indirected through is pushed on the stack before the location expression is evaluated (needed for self-based pointers). If the **base_locator** field is not present, the debugger will use the default near segment and a zero offset.

2.4.5 ENUMERATED (value 0x5?)

This definition is used to define an enumerated type. There are 4 sub-classes.

```
LIST       (value 0x50) #consts_word, scalar_type_byte
CONST_BYTE (value 0x51) value_byte, name
CONST_WORD (value 0x52) value_word, name
CONST_LONG (value 0x53) value_dword, name
```

LIST is used to inform the Open Watcom Debugger of the number of constants in the enumerated type and the scalar type used to store them in memory. It will be followed immediately by all the constant definitions for the enumerated type. See TYPE_NAME for a description of the **scalar_type_byte**.

CONST_BYTE, CONST_WORD, and CONST_LONG define the individual constant values for an enumerated type. The type of the constant is provided by the preceeding LIST definition. The decision to use the byte, word, or long form of the definition is made always by considering the value as a signed number. The CONST_* definition records are not counted when determining type index values.

The LIST record and its associated CONST_* records must all be contained in the same $$TYPES segment.

2.4.6 STRUCTURE (value 0x6?)

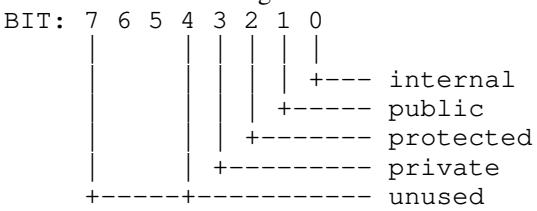
This definition is used to define a structure type. There are 10 sub-classes.

```
LIST        (value 0x60) #fields_word [,size_dword]
FIELD_BYTE  (value 0x61) offset_byte, type_index, name
FIELD_WORD  (value 0x62) offset_word, type_index, name
FIELD_LONG  (value 0x63) offset_dword, type_index, name
BIT_BYTE    (value 0x64) offset_byte, start_bit_byte, bit_size_byte,
                             type_index, name
BIT_WORD    (value 0x65) offset_word, start_bit_byte, bit_size_byte,
                             type_index, name
BIT_LONG    (value 0x66) offset_dword, start_bit_byte, bit_size_byte,
                             type_index, name
FIELD_CLASS (v    0x67) attrib_byte, field_locator, type_index, name
BIT_CLASS   (value 0x68) attrib_byte, field_locator, start_bit_byte,
                             bit_size_byte, type_index, name
INHERIT_CLASS (v  0x69) adjust_locator, ancestor_type_index
```

LIST is used to introduce a structure definition. It is followed immediately by all the field definitions that make up the structure. The optional **size_dword** gives the size of the structure in bytes. If it is not present, the debugger calculates the size of the structure based on field offsets and sizes.

FIELD_BYTE, FIELD_WORD, FIELD_LONG, and FIELD_CLASS define a single field entry in a structure defintion.

BIT_BYTE, BIT_WORD, BIT_LONG, and BIT_CLASS define a bit field in a structure. :The FIELD_CLASS and BIT_CLASS records are used for defining fields in a C++ class. The **attrib_byte** contain a set of bits describing attributes of the field:



An internal field is one that is generated for compiler support. It is not normally displayed to the user. The other bits have their usual C++ meanings.

The **field_locator** is a location expression describing how to calculate the field address. Before begining to evaluate the expression, the debugger will implicitly push the base address of the class instance onto the stack. The following is an example of the location expression used to calculate an ordinary field at offset 10 from the start of the class:

```
INT_1   10
ADD
```

The INHERIT_CLASS record indicates that a particular class should inherit all the fields specified by **ancestor_type_index**. This field must point at either a STRUCTURE LIST record or a TYPE NAME that eventually resolves to a STRUCTURE LIST. The **adjust_locator** is a location expression that tells the debugger how to adjust the field offset expressions in the inherited class to their proper values for a class of this instance.

The FIELD_*, BIT_*, and INHERIT_CLASS records are not counted when determining type index values.

A C union, or Pascal variant record is described by having a number of fields all beginning at the same offset. The Open Watcom Debugger will display the fields in the reverse order that the records define them. This means that ordinarily, the records should be sorted by descending offsets and bit positions.

The LIST record and it's associated field descriptions must all be contained in the same $$TYPES segment.

2.4.7 PROCEDURE (value 0x7?)

This definition is used to define a procedure type. There are 4 sub-classes.

```
NEAR      (value 0x70) ret_type_index, #parms_byte
{,parm_type_index}
FAR       (value 0x71) ret_type_index, #parms_byte
{,parm_type_index}
NEAR386   (value 0x72) ret_type_index, #parms_byte
{,parm_type_index}
FAR386    (value 0x73) ret_type_index, #parms_byte
{,parm_type_index}
EXT_PARMS (value 0x74) {,parm_type_index}
```

The EXT_PARMS sub-class is used when there are too many parameter types to fit into one PROCEDURE record. This condition can be recognized when the #parms_byte indicates there are more parameter types than fit into the record according to the length field at the beginning. In this case the remaining parameter types are continued in the record immediately following, which will always be of type EXT_PARMS. The EXT_PARMS record must be contained in the same $$TYPES segment as the preceeding procedure record.

2.4.8 CHARACTER_BLOCK (value 0x8?)

Items of type CHARACTER_BLOCK are length delimited strings. There are 4 sub-classes.

```
CHAR_BYTE     (value 0x80) length_byte
CHAR_WORD     (value 0x81) length_word
CHAR_LONG     (value 0x82) length_dword
CHAR_IND      (value 0x83) scalar_type_byte, length_32_pointer
CHAR_IND_386 (value 0x84) scalar_type_byte, length_48_pointer
CHAR_IND_LOC (value 0x85) scalar_type_byte, address_locator
```

The CHAR_BYTE, CHAR_WORD, and CHAR_LONG forms are used when the length of the character string is known at compile time. Even though the length given is an unsigned quantity, the decision on which form to use is made by considering the value to be signed. The CHAR_IND form is used when the length of the string is determined at run time. The **`length_32_pointer`** gives the far address of a location containing the length of the string. The size of this location is given by the **`scalar_type_byte`**. The CHAR_IND_386 form is the same as CHAR_IND except that the location of the length is given by a 48-bit far pointer. The CHAR_IND_LOC form is the same as CHAR_IND except that the address of the length is given by a location expression.

2.5 Local symbol information

The Open Watcom Debugger local symbol information is contained in a special segment in the object file. The segment name is "$$SYMBOLS" and the segment class is "DEBSYM". The segment is considered to be a stream of variable length definitions, with each definition being preceded by a length byte. A number of the definitions contain indices of some form. These indices are standard Intel format, with 0 meaning no index, 1 to 127 is represented in one byte, 128 to 32767 in high byte/low byte form with the top bit on in the high byte. Character strings representing names are always placed at the end of a definition so that their length can be calculated by subtracting the name's start point from the length of the record. They are not preceded by a length byte or followed by a zero byte.

The first byte identifies the kind of the symbol definition that follows. The top nibble of the byte is used to indicate the general class of the symbol definition. The low order nibble is used to qualify the general definition class.

Symbol definitions are used to provide the Open Watcom Debugger with the location and scoping of source language local symbols. There are two general classes of symbol definition, one for variables and one for code.

2.5.1 VARIABLE (value 0x1?)

This definition is used to define the location of a data symbol. There are 4 sub-classes.

```
MODULE     (value 0x10) memory_location_32_pointer, type_index, name
LOCAL      (value 0x11) address_locator, type_index, name
MODULE386  (value 0x12) memory_location_48_pointer, type_index, name
MODULE_LOC (value 0x13) address_locator, type_index, name
```

MODULE defines either an exported, domestic, or imported variable in the module. It is not necessary to generate symbol information for an imported variable since the Open Watcom Debugger will look for local symbol information in the module which defines the variable if required.

LOCAL defines a symbol that is local to a code block or procedure. The defining block is the first one previous to this definition. Local symbols only "exist" for the purpose of the Open Watcom Debugger lookups when the program is executing in a block which defines the symbol.

2.5.2 CODE (value 0x2?)

This definition is used to define an object in the code. There are 6 sub-classes.

```
BLOCK         (value 0x20) start_offset_word, size_word,
                             parent_block_offset
NEAR_RTN      (value 0x21) <BLOCK>, pro_size_byte, epi_size_byte,
                             ret_addr_offset_word, type_index,
                             return_val_loc, #parms_byte
                             {,parm_location}, name
FAR_RTN       (value 0x22) <BLOCK>, pro_size_byte, epi_size_byte,
                             ret_addr_offset_word, type_index,
                             return_val_loc, #parms_byte
                             {,parm_location}, name
BLOCK_386     (value 0x23) start_offset_dword, size_dword,
                             parent_block_offset
NEAR_RTN_386  (value 0x24) <BLOCK_386>, pro_size_byte, epi_size_byte,
                             ret_addr_offset_dword, type_index,
                             return_val_loc, #parms_byte
                             {,parm_location}, name
FAR_RTN_386   (value 0x25) <BLOCK_386>, pro_size_byte, epi_size_byte,
                             ret_addr_offset_dword, type_index,
                             return_val_loc, #parms_byte
                             {,parm_location}, name
MEMBER_SCOPE  (value 0x26) parent_block_offset, class_type_index
                             [obj_ptr_type_byte, object_loc]
```

BLOCK is used to indicate a block of code that contains local symbol definitions. The field **parent_block_offset** is used to tell the Open Watcom Debugger the next block to search for a symbol definition if it is not found in this block. The field is set to zero if there is no parent block.

NEAR_RTN and FAR_RTN are used to specify a routine definition. Notice that the first part is identical to a code block definition. The **ret_addr_offset_word** is the offset from BP (or EBP) that the return address is located on the stack. The **#parms_byte** and **parm_location**'s following are only for those parms which are passed in registers. The remainder of the parms are assumed to be passed on the stack.

The MEMBER_SCOPE record is used for C++ member functions. It introduces a scope where the the debugger looks up the fields of the class identified by **class_type_index** as if they were normal symbols. If the **obj_ptr_type_byte** and **object_loc** location expression portions of the record are present, it indicates that the function has a C++ "this" pointer, and all fields of the class structure are accessable. The location expression evaluates to the address of the object that the member function is manipulating. The **obj_ptr_type_byte** contains a value from the low order nibble of a POINTER type record. It indicates the type of 'this' pointer the routine is expecting. I.e.:

Value	*Definition*
0	16-bit near pointer
1	16-bit far pointer
6	32-bit near pointer
7	32-bit far pointer

If the portions following the **class_type_index** are absent from the record, the routine is a static member function and only has access to static data members.

To use this record, the member function's **parent_block_offset** is pointed at the MEMBER_SCOPE record, and the MEMBER_SCOPE's **parent_block_offset** field is pointed at what the member function would normally be pointing at. In effect, a new block scope has been introduced.

The *_386 versions of the records are identical to their 286 counterparts excepts that the **start_offset**, **size**, and **ret_addr_offset** fields have been widened to 32 bits.

Notes: There should be a better mapping of parm number to parm location. There is no provision for Pascal calling conventions (reversed parm order) or other strangeness.

The BLOCK definition contains a **start_offset_word** (or **start_offset_dword** in a BLOCK_386). This is the offset from a given memory location provided by NEW_BASE entries and indicates the address of the start of executable code for the block.

All the code location definitions are assumed to be sorted in order of increasing end offsets (start offset + size). This ensures that the first scope that the debugger encounters in a traversal of the symbolic information is the closest enclosing scope.

2.5.3 NEW_BASE (value 0x3?)

```
ADD_PREV_SEG (value 0x30) seg_increment_word
SET_BASE     (value 0x31) memory_location_32_pointer
SET_BASE386  (value 0x32) memory_location_48_pointer
```

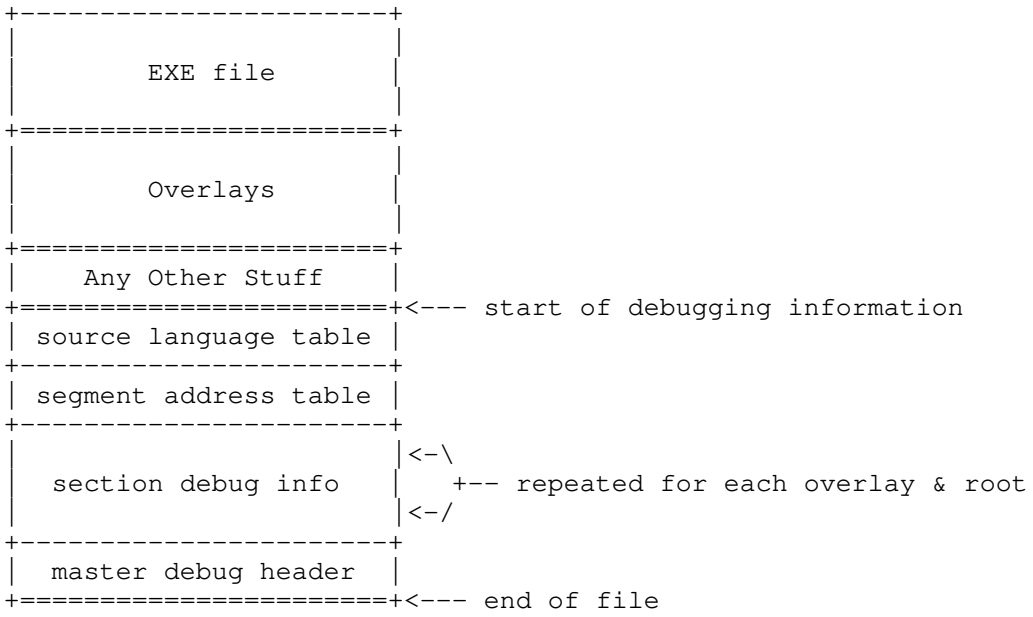
For ADD_PREV_SEG, the specified amount is added to the segment value of the code start address of the module. The code start offset is reset to zero. All BLOCK definitions occuring after this item are relative to the new value. After a SET_BASE or SET_BASE386 all BLOCK definitions are relative to the memory location that is given by the record.

Notes: Avoid the use of the ADD_PREV_SEG record. Its operation is only valid in real mode. It is included for backwards compatiblity only.

3 Executable file structures

The linker is responsible for processing the debugging information contained in the object files and some of its internal structures and appending them to the executable file.

After linking, the executable file looks like this:



The section marked as "EXE file" is the normal executable file. All debugging information is appended to the end of the file, after any overlay sections or other information. The **master debug header** begins at a fixed offset from the end of the file, and provides the location of the remainder of the debug information. The **source language table** contains the source languages used by the program. The **section debug info** is repeated once for the root and each overlay section defined in the executable. It contains all the debugging information for all object modules defined in the root or a particular overlay section. The **section debug info** is further divided into a number of debugging information classes, these will be explained later. All offsets in the debugging information that refer to other information items are relative to the start of the information, the start of a section of information, or the start of a class of the information. In other words, the information is not sensitive to its location in the executable file.

3.1 Master debug header

The master debug header allows the Open Watcom Debugger to verify the fact that there is debugging information, to locate the other sections and to verify that it is capable of handling the version of debugging information. The master header structure is as follows:

```
struct master_dbg_header {
    unsigned_16 signature;
    unsigned_8  exe_major_ver;
    unsigned_8  exe_minor_ver;
    unsigned_8  obj_major_ver;
    unsigned_8  obj_minor_ver;
    unsigned_16 lang_size;
    unsigned_16 segment_size;
    unsigned_32 debug_size;
};
```

The **signature** word contains the value 0x8386. This is the first indication to the Open Watcom Debugger that there is debugging information present. The **exe_major_ver** field contains the major version number of the executable file debugging information structures. The major version number will change whenever there is a modification to these structures that is not upwardly compatible with the previous version. The current major version number is three. The **exe_minor_ver** field contains the minor version number of the executable file debugging information structures. The minor version number increments by one whenever there is a change to the structures which is upwardly compatible with the previous version. The current minor version number is zero. This means that in order for the Open Watcom Debugger to process the debugging information the following must be true:

1. FILE exe debug info major version == debugger exe debug info major version

2. FILE exe debug info minor version <= debugger exe debug info minor version

The **obj_major_ver** field contains the major version number of the object file debugging information structures (internal format of the types and local symbol information). The major version number will change whenever there is a modification to these structures that is not upwardly compatible with the previous version. The current major version number is one. The **obj_minor_ver** field contains the minor version number of the object file debugging information structures. The minor version number increments by one whenever there is a change to the structures which is upwardly compatible compatible with the previous version. The current minor version number is three. This means that in order for the debugger to process the debugging information the following must be true:

1. FILE obj debug info major version == debugger obj debug info major version

2. FILE obj debug info minor version <= debugger obj debug info minor version

These two fields are filled in by the linker by extracting the version information from special debug comment record in the processed object files. If two object files in the link contain different major version numbers, the linker should report an error or warning and not process the type or local symbol information for the 'incorrect' file. The minor version number placed in the master header should be the maximum of all the minor version numbers extracted from the object files.

The **lang_size** field contains the size of the source language table at the beginning of the debug information. The **segment_size** field informs the debugger of the size, in bytes, of the segment address table. The field, **debug_size**, gives the total size of the debugging information, including the size of the master header itself. This allows the debugger to calculate the start of the debugging information by subtracting the value of the **debug_size** field from the location of the end of file. This gives the start of the source language and segment address tables, whose sizes are known from the master header. Once the location of the first section of debugging information is determined, it can be processed. Within the section information is a indicator of its total size, which allows the debugger to find the start of the next section, and process that as well. This continues until all the debug sections have been processed. the debugger knows there are no more debug sections to process when the indicated start of a section is the same as the start of the master header.

18 Master debug header

3.2 Source language table

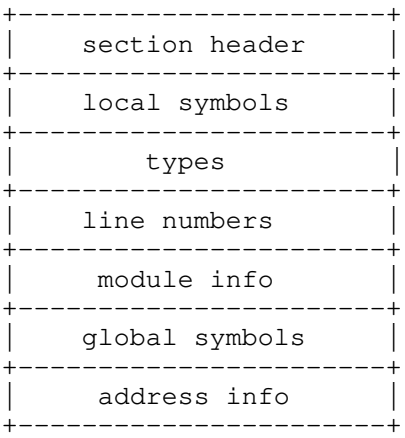
The source language table is merely the collection of unique source languages used in the program. The strings are extracted from the special debug comment records in the object files and placed in this section one after another with zero bytes separating them.

3.3 Segment address table

The segment address table is an array of all the unique segment numbers used by the executable. Essentially, any segment value that would appear in the map file will be represented in the table.

3.4 Section debug information

Each **section debug info** contains the following:



The local symbols, types and line numbers classes are demand loaded by the debugger as it requires pieces of the classes for various modules. The module info, global symbols, and address info classes are permanently loaded by the debugger at the start of a debugging session. The global symbol, module, and address info classes have no size restriction, however there is a limit of 65536 modules per section and there are some restrictions on how the address info class may be laid out. These restrictions are described in the section explaining the address info class.

3.4.1 Section debug header

The section header class allows the debugger to determine the size of the section information and the location of the permanently loaded classes. The header structure is as follows:

```
struct section_dbg_header {
    unsigned_32 mod_offset;
    unsigned_32 gbl_offset;
    unsigned_32 addr_offset;
    unsigned_32 section_size;
    unsigned_16 section_id;
};
```

The **mod_offset**, **gbl_offset**, and **addr_offset** fields are offsets, from the beginning of the section debug header to the module info, global symbol, and address info classes of debugging information.

The **section_size** field is the size of the debugging information for the section, including the section header. The following conditions must hold true for the debugger to recognize the debugging information as valid:

1. mod_offset < gbl_offset
2. gbl_offset < addr_offset
3. addr_offset < section_size

The **section_id** field contains the overlay number for this section. This is zero for the root.

3.4.2 Local symbols class

The local symbols segments are processed normally by the linker, except that the data in the segments is placed in this section, no relocation entries are output for any fixups in the data and fields in the module structure are intialized to point to the beginning and size of each object file's contribution to the section.

3.4.3 Types class

The type segments are processed normally by the linker, except that the data in the segments is placed in this section, no relocation entries are output for any fixups in the data and fields in the module structure are intialized to point to the beginning and size of each object file's contribution to the section.

3.4.4 Line numbers class

The LINNUM records for each object file are collected and placed in this class using an array of arrays. The top level array is the following structure:

```
struct line_segment {
    unsigned_32     segment;
    unsigned_16     num;
    line_info       line[1];
}
```

The **segment** field contains a offset, from the start of the address info class, to an addr_info structure (see the address info class description). This provides the segment value for the array of line_info's following. The next field, **num**, provides the number of line_info's in the array. The **line** is a variable size array containing the following structure:

```
struct line_info {
    unsigned_16  line_number;
    unsigned_32  code_offset;
};
```

The **line_number** contains the source line number whose offset is being defined. If the top bit of the line number is on, this line number refers to an entry in the special line number table. See the "Special Line Number Table" section for more details. The **code_offset** field contains the offset from the begining of the module for the first instruction associated with the line number. To get the true code address for the instruction you must add **code_offset** to the address given by the **segment** field in the line_segment structure. All the instructions up to the next element's **code_offset**, or the end of the object file's code for that segment if there is no next **code_offset** are considered to be part of the **line_number** source line. Within each line_segment structure the line_info array is assumed to be sorted in order of ascending

code_offset's. The module structure for the object file contains fields which indicate the start and size of the line_segment array within the class.

Each line_segment structure may not exceed 60K, however the total amount of line information for a module may exceed 60K with multiple line_segment structures and multiple entries in the demand link table (described in the module information section).

To obtain a line number from an address, the debugger performs the following steps

1. Given an address, the defining module is found from the address information class. This allows the debugger to find and load the line number information for that module, if it is not already loaded.

2. Walk down the array of line_segment structures until one with the appropriate segment is found.

3. Binary search the array of line_info's until the proper one is located.

3.4.4.1 Special Line Number Table

The OMF line number record does not allow for more than one source file to be referenced in an object file. This kludge gets around the restriction. If the top bit is on in **line_number** than that field refers to an entry in the special line number table. The debugger then searches the typing information for the module for a **CUE_TABLE** record. If it finds one, it uses the offset given to find the begining of the table in the typing information. The table looks like this:

```
/* cue entry table */
unsigned_16 cue_count

struct {
    unsigned_16 cue;
    unsigned_16 fno;
    unsigned_16 line;
    unsigned_16 column;
} cue_entry; /* repeated cue_count times, sorted by the 'cue' field
*/

/* file name index table */
unsigned_16 file_count

struct {
    unsigned_16 index;
} file_name_index_entry; /* repeated file_count times */

/* file name table */
A list of zero terminated source file names
```

To find the correct cue entry given the value in a **line_number**, search the **cue_entry** table for the cue which satisfies the following:

```
cue_entry[entry].cue <= (line_number & 0x7fff) <
cue_entry[entry+1].cue
```

Once you have the cue entry, you can extract the true line number by:

```
line = cue_entry[entry].line + (line_number & 0x7fff)
                - cue_entry[entry].cue;
```

The file name is found by:

```
fname_index = file_name_index_table[ cue_entry[entry].fno ]
fname =  file_name_table[ fname_index ]
```

The code offset and segment are found in the **line_info** and **line_segment** structures as usual.

3.4.5 Module information class

The module information class is built from the linker's list of object files that it processes to build the executable file, which are either specified on the linker command line or extracted from libraries. All the modules are implicitly given an index number by their order in the class. These index numbers start at zero and are used by other classes to identify individual modules. The module structure contains the following fields:

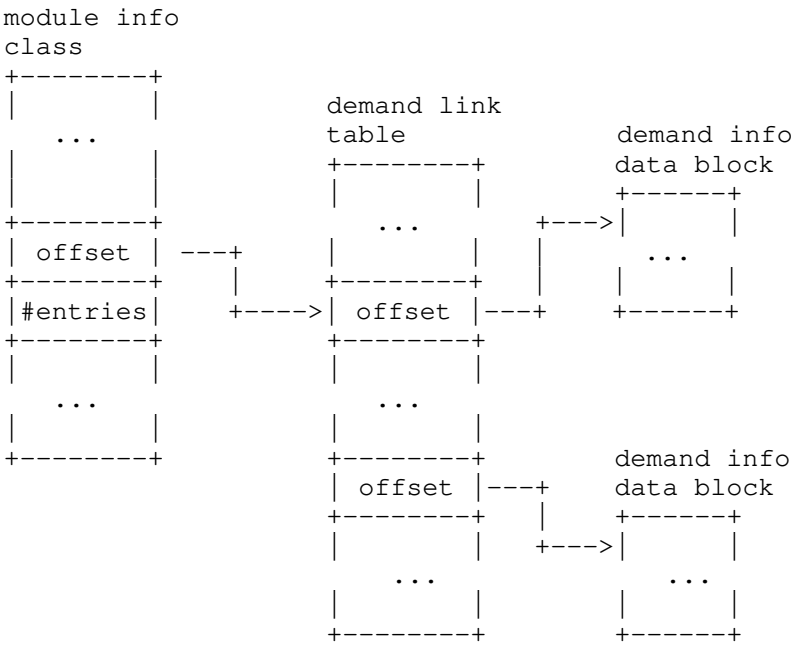
```
struct mod_info {
    unsigned_16 language;
    demand_info locals;
    demand_info types;
    demand_info lines;
    unsigned_8  name[1];
};
```

The **language** field contains an offset, from the start of the source language table to the string of the source language for this module. The **name** field is a variable length array of characters with the first element of the array being the length of the name. The remaining characters identify the source file the compiler used to generate the object file (e.g. "C:\DEV\WV\C\DBGMAIN.C"). The source file name is obtained from the THEADR record of the object file. the debugger uses the file name part of the file specification as its "module name". The remaining fields, **locals**, **types**, and **lines** are a structure type which define the location and size of this module's demand loaded information from those classes. The structure contains these fields:

```
struct demand_info {
    unsigned_32 offset;
    unsigned_16 num_entries;
};
```

The **offset** field contains the offset from the beginning of the debugging information section to first entry in the demand link table containing the information for that particular demand load class. The **num_entries** field gives the number of contiguous entries in the demand link table that are present for the module's demand load information of that particular class.

The demand link table consists of an array of unsigned_32 offsets, which are relative from the debugging information section, to the individual demand info class data blocks. The array is in ascending order of offsets so that the debugger may calculate the size of a particular demand load data block by subtracting the offset of the next data block from the offset of the current data block. This implies that there is an extra entry at the end of the table whose offset points to the end of the final demand load data block so that the debugger always has a 'next' link entry to calculate size of a data block with. The size of each individual block may not exceed 60K. A picture may be useful here to show how all the pieces fit together:



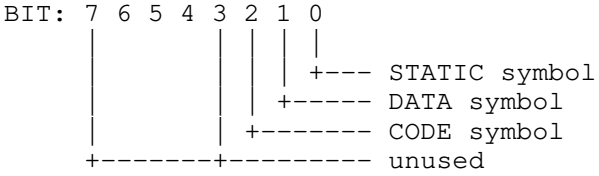
When the debugger wishes to look something up in a demand load class for a module. It uses the offset in the mod_info structure to locate the array entry in the demand link table which has the offset for the first info data block. It then loads the first block and searches it for the information. If the information is not present in that block, it moves to the next entry in the demand link table and repeats the above process. This continues until all the entries for that particular class of the module (identified by the **num_entries** field in the mod_info structure) have been examined, or the information is located.

3.4.6 Global symbols class

All PUBDEF records processed by the linker create entries in this class. The fields in the structure are:

```
struct gbl_info {
    addr48_ptr      addr;
    unsigned_16     mod_index;
    unsigned_8      kind;
    unsigned_8      name[1];
};
```

The **addr** field contains the location in memory associated with this symbol. The value placed in this field is the same that the linker places in the map file (i.e. unrelocated, as if the executable loads at location 0:0). The field contains a 48 bit value (32 bit offset followed by a 16 bit segment). The **mod_index** field is an index which identifies the module which defines the symbol (i.e. contained the [L]PUBDEF record). The **kind** gives rudimentary typing information for the symbol. It consists of the following set of bits:



Bit zero is 1 if the global was defined by a LPUBDEF record and 0 if it was defined by a PUBDEF record. LPUBDEF symbols are generated by the code generator for static symbols, so this allows a debugger to see static symbols even when no compiler debug switches are being used. Bit one is 1 if the producer of the information is able to determine that the symbol is a data symbol. Bit two is one if the producer is able to

determine that the symbol is a code symbol. Both bits may be zero if the producer is unable to determine whether the symbol is a code or data item. The final field, **name** is a variable length array, with the first character indicating the length of the name, and the remaining characters being the actual name of the symbol.

3.4.7 Address information class

The address information class allows the debugger, given a memory address, to determine the module which defines that memory address. The linker builds this class from the SEGDEF and GRPDEF records in the object files that it processes. The class consists of an array of structures with the following fields:

```
struct seg_info {
    addr48_ptr      addr;
    unsigned_16     num;
    addr_info       sects[1];
};
```

The **addr** field identifies the start of a segment in memory. This field contains the unrelocated value of the segment starting address (i.e. as if the executable had been loaded at 0:0). The the low order 15 bits of the next field, **num** tells how many of the **sects** entries there are in the structure. The top bit of the field is a one when the segment belongs to "NonSect". "NonSect" is the overlay section which holds all program data that is not in the root or an overlay section. Typically this consists of DGROUP and FAR_DATA segments. NonSect always is located at the highest address of all sections. It is preloaded by the overlay manager and is never moved. If the segment does not belong to NonSect, the top bit of the **num** field is zero. The **sects** field is a variable size array of structures. This addr_info structure contains the following fields:

```
struct addr_info {
    unsigned_32     size;
    unsigned_16     mod_index;
};
```

The **mod_index** field indicates the module in the module information class which defines this piece of the segment. The **size** field identifies how large a piece of the segment specified by the seg_info structure belongs to the module. The starting address of the segment piece is given by adding all the previous size fields in the **sects** array to the original starting address in the seg_info structure.

The size of a seg_info structure may not exceed 60K. If a single physical segment would have more **sects** than would fit into this restriction (**num** greater than 10238), it should be split into two separate seg_info structures.

To identify the module that defines a location in memory, the debugger does the following:

1. Walk down the array of seg_info structures until one is found with the same segment address as the location that is being identified. If no such seg_info is found, or the starting offset of the segment is greater than the offset of the memory location, then there is no defining module.

2. Walk down the array of addr_info's in the seg_info structure until an entry is found whose starting offset is less than or equal to the memory location offset and whose ending offset is greater than the memory location offset. If there is no such entry, there is no defining module.

3. Otherwise, the **mod_offset** field of the addr_info entry is added to the beginning of the module information class, which gives a pointer to the module structure that defines the memory location.

Debugger Trap File Interface VERSION 1.3

1 Trap File Interface

The Open Watcom debugger consists of a number of separate pieces of code. The main executable, WD.EXE (wd on UNIX systems), provides a debugging 'engine' and user interface. When the engine wishes to perform an operation upon the program being debugged such as reading memory or setting a breakpoint, it creates a request structure and sends it to the 'trap file' (so called because under DOS, it contains the first level trap handlers). The trap file examines the request structure, performs the indicated action and returns a result structure to the debugger. The debugger and trap files also use Machine Architecture Description (MAD) files which abstract the CPU architecture. This design has the following benefits:

1. OS debugging interfaces tend to be wildly varying in how they are accessed. By moving all the OS specific interface code into the trap file and having a defined interface to access it, porting the debugger becomes much easier.

2. By abstracting the machine architecture specifics through MAD files, it becomes possible to use one debugger for several target CPU architectures (such as x86 and Alpha AXP). Unlike most other debuggers, the Open Watcom debugger is not tied to a single host/target combination and if appropriate trap and MAD files are available, the debugger running on any host can remotely debug any target.

3. The trap file does not have to actually perform the operation. Instead it could send the request out to a remote server by a communication link such as a serial line or LAN. The remote server can retrieve the request, perform the operation on the remote machine and send the results back via the link. This enables the debugger to debug applications in cases where there are memory constraints or other considerations which prevent the debugger proper from running on the remote system (such as Novell Netware 386).

This document describes the interface initially used by version 4.0 of the WATCOM debugger (shipped with the 10.0 C/C++ and FORTRAN releases). It has been revised to describe changes incorporated in Watcom 11.0 release, as well as subsequent Open Watcom releases. It is expected to be modified in future releases. Where possible, notification of expected changes are given in the document, but all aspects are subject to revision.

1.1 Some Definitions

Next follow some general trap definitions.

1.1.1 Byte Order

The trap file interface is defined to use little endian byte order. That is, the least significant byte is stored at the lowest address. Little endian byte order was chosen for compatibility with existing trap files and tools. Fixed byte order also eases network communication between debuggers and trap files running on machines with different byte order.

1.1.2 Pointer Sizes

In a 16-bit hosted environment such as DOS, all pointers used by the trap file are "far" 16:16 pointers. In a 32-bit environment such as Windows NT the pointers are "near" 0:32 pointers.

1.1.3 Base Types

A number of basic types are used in the interface. They are defined as follows:

Type	*Definition*
unsigned_8	1 byte unsigned quantity
unsigned_16	2 byte unsigned quantity
unsigned_32	4 byte unsigned quantity
access_req	The first field of every request is of this type. It is a 1 byte field which identifies the request to be performed.

addr48_ptr	This type encapsulates the concept of a 16:32 pointer. All addresses in the debuggee memory are described with these. The debugger always acts as if the debuggee were in a 32-bit large model environment since the 32-bit flat model and all 16-bit memory models are subsets. The structure is defined as follows:
	`typedef struct {`
	`  unsigned_32  offset;`
	`  unsigned_16  segment;`
	`} addr48_ptr;`
	
	The **`segment`** field contains the segment of the address and the **`offset`** field stores the offset of the address.

bytes	The type **`bytes`** is an array of unsigned_8. The length is provided by other means. Typically a field of type **`bytes`** is the last one in a request and the length is calculated from the total length of the request.
string	The type **`string`** is actually an array of characters. The array is terminated by a null ('\0') character. The length is provided by other means. Typically a field of type **`string`** is the last one in a request and the length is calculated from the total length of the request.
trap_error	Some trap file requests return debuggee operating system error codes, notably the requests to perform file I/O on the remote system. These error codes are returned as an unsigned_32. The debugger considers the value zero to indicate no error.
trap_phandle	This is an **`unsigned_32`** which holds process (task) handle. A task handle is used to uniquely identify a debuggee process.
trap_mhandle	This is an **`unsigned_32`** which holds a module handle. Typically the main executable will be one module, and on systems which support DLLs or shared libraries, each library will be identified by a unique module handle.

2 The Request Interface

Next follow detailed description of interface elements.

2.1 Request Structure.

Each request is a composed of two sequences of bytes provided by the debugger called messages. The first set contains the actual request code and whatever parameters that are required by the request. The second sequence is where the result of the operation is to be stored by the trap file.

The two sequences need not be contiguous. The sequences are described to the trap file through two arrays of message entry structures. This allows the debugger to avoid unnecessary packing and unpacking of messages, since **mx_entry**'s can be set to point directly at parameter/result buffers.

Multiple requests are **not** allowed in a single message. The **mx_entry**'s are only used to provide scatter/gather capabilities for one request at a time.

The message entry structure is as follows (defined in **trptypes.h**):

```
typedef struct {
    void            *ptr;
    unsigned        len;
} mx_entry;
```

The **ptr** is pointing to a block of data for that message entry. The **len** field gives the length of that block. One array of **mx_entry**'s describes the request message. The second array describes the return message.

It is not legal to split a message into arbitrary pieces with mx_entries. Each request documents where an **mx_entry** is allowed to start with a line of dashes.

2.2 The Interface Routines

The trap file interface must provide three routines: **TrapInit**, **TrapRequest**, and **TrapFini**. How the debugger determines the address of these routines after loading a trap file, as well as the calling convention used, is system dependent and described later. These functions are prototyped in **trpimp.h**.

2.2.1 TrapInit

This function initializes the environment for proper operation of **TrapRequest**.

```
trap_version TRAPENTRY TrapInit( char       *parm,
                                 char       *error,
                                 unsigned_8 remote
);
```

The **parm** is a string that the user passes to the trap file. Its interpretation is completely up to the trap file. In the case of the Open Watcom debugger, all the characters following the semicolon in the **/TRAP** option are passed as the **parm**. For example:

```
wd /trap=nov;testing program
```

The **parm** would be "testing". Any error message will be returned in **error**. The **remote** field is a zero if the Open Watcom debugger is loading the trap file and a one if a remote server is loading it. This function returns a structure **trap_version** of the following form (defined in **trptypes.h**):

```
typedef struct {
    unsigned_8  major;
    unsigned_8  minor;
    unsigned_8  remote;
} trap_version;
```

The **major** field contains the major version number of the trap file while the **minor** field tells the minor version number of the trap file. **Major** is changed whenever there is a modification made to the trap file that is not upwardly compatable with previous versions. **Minor** increments by one whenever a change is made to the trap file that is upwardly compatible with previous versions. The current major verion is 1, the current minor version is 3. The **remote** field informs the debugger whether the trap file communicates with a remote machine.

TrapInit must be called before using **TrapRequest** to send a request. Failure to do so may result in unpredictable operation of **TrapRequest**.

2.2.2 TrapRequest

All requests between the server and the remote trap file are handled by TrapRequest.

```
unsigned TRAPENTRY TrapRequest( unsigned num_in_mx,
                                mx_entry *mx_in,
                                unsigned num_out_mx,
                                mx_entry *mx_out
);
```

The **mx_in** points to an array of request mx_entry's. The **num_in_mx** field contains the number of elements of the array. Similarly, the **mx_out** will point to an array of return mx_entry's. The number of elements will be given by the **num_out_mx** field. The total number of bytes actually filled in to the return message by the trap file is returned by the function (this may be less than the total number of bytes described by the **mx_out** array).

Since every request must start with an **access_req** field, the minimum size of a request message is one byte.

Some requests do not require a return message. In this case, the program invoking TrapRequest **must** pass zero for **num_out_mx** and NULL for **mx_out**.

2.2.2.1 Request Example

The request REQ_READ_MEM needs the memory address and length of memory to read as input and will return the memory block in the output message. To read 30 bytes of memory from address 0x0010:0x8000 into a buffer, we can write:

```
mx_entry        in[1];
mx_entry        out[1];
unsigned char   buffer[30];
struct in_msg_def {
    access_req          req;
    addr48_ptr          addr;
    unsigned_16         len;
} in_msg = { REQ_READ_MEM, { 0x8000, 0x0010 }, sizeof( buffer ) };

unsigned_16 mem_blk_len;

in[0].ptr = &in_msg;
in[0].len = sizeof( in_msg );
out[0].ptr = &buffer;
out[0].len = sizeof( buffer );

mem_blk_len = TrapRequest( 1, in, 1, out );

if( mem_blk_length != sizeof( buffer ) ) {
    printf( "Error in reading memory\n" );
} else {
    printf( "OK\n" );
}
```

The program will print "OK" if it has transferred 30 bytes of data from the debuggee's address space to the **`buffer`** variable. If less than 30 bytes is transfered, an error message is printed out.

2.2.3 TrapFini

The function terminates the link between the debugger and the trap file. It should be called after finishing all access requests.

```
void TRAPENTRY TrapFini( void );
```

After calling **`TrapFini`**, it is illegal to call **`TrapRequest`** without calling **`TrapInit`** again.

3 The Requests

This section descibes the individual requests, their parameters, and their return values. A line of dashes indicates where an **mx_entry** is allowed (but not required) to start. The debugger allows (via REQ_GET_SUPPLEMENTARY_SERVICE/REQ_PERFORM_SUPPLEMENTARY_SERVICE) optional components to be implemented only on specific systems.

The numeric value of the request which is placed in the **req** field follows the symbolic name in parentheses.

3.1 Core Requests

These requests need to be implemented in all versions of the trap file, although some of them may only be stub implementations in some environments. Note that structures suitable for individual requests are declared in **trpcore.h**.

3.1.1 REQ_CONNECT (0)

Request to connect to the remote machine. This must be the first request made.

Request message:

```
access_req      req
unsigned_8      major;    <-+- struct trap_version
unsigned_8      minor;      |
unsigned_8      remote;   <-+
```

The **req** field contains the request. The **trap_version** structure tells the version of the program making the request. The **major** field contains the major version number of the trap file while the **minor** field tells the minor version number of the trap file. The **major** is changed whenever there is a modification made to the trap file that is not upwardly compatable with previous versions. The **minor** increments by one whenever a change is made to the trap file that is upwardly compatable with previous versions. The current major version is 1, the current minor version is 3. The **remote** field informs the trap file whether a remote server is between the Open Watcom debugger and the trap file.

Return message:

```
unsigned_16 max_msg_size
---------------------------
string      err_msg
```

If error has occurred, the **err_msg** field will returns the error message string. If there is no error, **error_msg** returns a null character and the field **max_msg_size** will contain the allowed maximum size of a message in bytes. Any message (typically reading/writing memory or files) which would require more than the maximum number of bytes to transmit or receive must be broken up into multiple requests. The minimum acceptable value for this field is 256.

3.1.2 REQ_DISCONNECT (1)

Request to terminate the link between the local and remote machine. After this request, a REQ_CONNECT must be the next one made.

Request message:

```
access_req      req
```

The **req** field contains the request.

Return message:

```
NONE
```

3.1.3 REQ_SUSPEND (2)

Request to suspend the link between the server and the remote trap file. The debugger issues this message just before it spawns a sub-shell (the "system" command). This allows a remote server to enter a state where it allows other trap files to connect to it (normally, once a remote server has connected to a trap file, the remote link will fail any other attempts to connect to it). This allows the user for instance to start up an RFX process and transfer any missing files to the remote machine before continuing the debugging process.

Request message:

```
access_req      req
```

The **req** field contains the request.

Return message:

```
NONE
```

3.1.4 REQ_RESUME (3)

Request to resume the link between the server and the remote trap file. The debugger issues this request when the spawned sub-shell exits.

Request message:

```
access_req      req
```

The **req** field contains the request.

Return message:

```
NONE
```

3.1.5 REQ_GET_SUPPLEMENTARY_SERVICE (4)

Request to obtain a supplementary service id.

Request message:

```
access_req  req
------------------------
string      service_name
```

The **req** field contains the request. The **service_name** field contains a string identifying the supplementary service. This string is case insensitive.

Return message:

```
trap_error      err;
trap_shandle    id;
```

The **err** field is non-zero if something went wrong in obtaining or initializing the service. **Id** is the identifier for a particular supplementary service. It need not be the same from one invocation of the trap file to another. If both it and the **err** field are zero, it means that the service is not available from this trap file.

Notes: In the future, we might allow for user developed add-ons to be integrated with the debugger. There would be two components, one to be added to the debugger and one to be added to the trap file. The two pieces could communicate with each other via the supplementary services mechanism.

3.1.6 REQ_PERFORM_SUPPLEMENTARY_SERVICE (5)

Request to perform a supplementary service.

Request message:

```
access_req  req
unsigned_32 service_id
------------------------
unspecified
```

The **req** field contains the request. The **service_id** field indicates which service is being requested. The remainder of the request is specified by the individual supplementary service provider.

Return message:

```
unspecified
```

The return message is specified by the individual supplementary service provider.

3.1.7 REQ_GET_SYS_CONFIG (6)

Request to get system information from the remote machine.

Request message:

```
access_req      req
```

The **req** field contains the request.

Return message:

```
unsigned_8  cpu;
unsigned_8  fpu;
unsigned_8  osmajor;
unsigned_8  osminor;
unsigned_8  os;
unsigned_8  huge_shift;
mad_handle  mad;
```

The **mad** field specifies the MAD (Machine Architecture Description) in use and determines how the other fields will be interpreted. Currently the following MADs are used:

```
MAD_X86  - Intel Architecture IA-32 compatible
MAD_X64  - Intel Architecture X64 compatible
MAD_AXP  - Alpha Architecture
MAD_PPC  - PowerPC Architecture
MAD_MIPS - MIPS Architecture
MAD_MSJ  - Java Virtual Machine (Microsoft)
MAD_JVM  - Java Virtual Machine (Sun)
```

The **cpu** fields returns the type of the remote CPU. The size of that field is unsigned_8. Possible CPU types for MAD_X86 are:

```
bits 0-3
   X86_86  = 0   - 8086
   X86_186 = 1   - 80186
   X86_286 = 2   - 80286
   X86_386 = 3   - 80386
   X86_486 = 4   - 80486
   X86_586 = 5   - Pentium
   X86_686 = 6   - Pentium Pro/II/III
   X86_P4  = 15  - Pentium 4
bit  4           - MMX registers
bit  5           - XMM registers
bits 6,7         - unused
```

The **fpu** fields tells the type of FPU. The size of the field is unsigned_8. FPU types for MAD_X86 include:

```
X86_EMU = -1     - Software emulated FPU
X86_NO  =  0     - No FPU
X86_87  =  1     - 8087
X86_287 =  2     - 80287
X86_387 =  3     - 80387
X86_487 =  4     - 486 integrated FPU
X86_587 =  5     - Pentium integrated FPU
X86_587 =  6     - Pentium Pro/II/III integrated FPU
X86_P47 =  15    - Pentium 4 integrated FPU
```

The **osmajor** and **osminor** contains the major and minor version number for the operating system of the remote machine. The type of operating system can be found in **os** field. The size of this field is unsigned_8. The OS can be :

```
OS_IDUNNO    =  0   - Unknown operating system
OS_DOS       =  1   - DOS
OS_OS2       =  2   - OS/2
OS_PHAR      =  3   - Phar Lap 386 DOS Extender
OS_ECLIPSE   =  4   - Eclipse 386 DOS Extender (obsolete)
OS_NW386     =  5   - NetWare 386
OS_QNX       =  6   - QNX 4.x
OS_RATIONAL  =  7   - DOS/4G or compatible
OS_WINDOWS   =  8   - Windows 3.x
OS_PENPOINT  =  9   - PenPoint (obsolete)
OS_NT        = 10   - Win32
OS_AUTOCAD   = 11   - ADS/ADI development (obsolete)
OS_NEUTRINO  = 12   - QNX 6.x
OS_LINUX     = 13   - Linux
OS_FREEBSD   = 14   - Free BSD
```

The **huge_shift** field is used to determine the shift needed for huge arithmetic in that system. It stores the number of left shifts required in order to calculate the next segment correctly. It is 12 for real mode

programs. The value in a protect mode environment must be obtained from the OS of the debuggee machine. This field is only relevant for 16-bit segmented architectures.

3.1.8 REQ_MAP_ADDR (7)

Request to map the input address to the actual address of the remote machine. The addresses in the symbolic information provided by the linker do not reflect any relocation performed on the executable by the system loader. This request obtains that relocation information so that the debugger can update its addresses.

Request message:

```
access_req      req;
addr48_ptr      in_addr;
trap_mhandle    mod_handle;
```

The **req** field contains the request. The **in_addr** tells the address to map. The **mod_handle** field identifies the module which the address is from. The value from this field is obtained by REQ_PROG_LOAD or REQ_GET_LIB_NAME. There are two magical values for the **in_addr.segment** field.

```
MAP_FLAT_CODE_SELECTOR  = -1
MAP_FLAT_DATA_SELECTOR  = -2
```

When the **in_addr.segment** equals one of these values, the debugger does not have a map segment value and is requesting that the trap file performs the mapping as if the given offset was in the flat address space.

Return message:

```
addr48_ptr      out_addr
addr48_off      lo_bound;
addr48_off      hi_bound;
```

The mapped address is returned in **out_addr**. Note that in addition to the segment portion being modified, the offset of the portion of the address may be adjusted as well if the loader performs offset relocations (like OS/2 2.x or Windows NT). The **lo_bound** and **hi_bound** fields identify the lowest and highest input offsets for which this mapping is valid. If the debugger needs to map another address whose input segment value is the same as a previous request, and the input offset falls within the valid range identified by the return of that previous request, it can perform the mapping itself and not bother sending the request to the trap file.

3.1.9 REQ_CHECKSUM_MEM (8)

Request to calculate the checksum for a block of memory in the debuggee's address space. This is used by the debugger to determine if the contents of the memory block have changed since the last time it was read. Since only a four byte checksum has to be transmitted back, it is more efficient than actually reading the memory again. The debugger does not care how the checksum is calculated.

Request message:

```
access_req      req;
addr48_ptr      in_addr;
unsigned_16     len;
```

The **req** field stores the request. The **in_addr** contains the starting address and the **len** field tells how large the block of memory is.

Return message:

```
unsigned_32     result
```

The checksum will be returned in **result**.

3.1.10 REQ_READ_MEM (9)

Request to read a block of memory.

Request message:

```
access_req      req;
addr48_ptr      mem_addr;
unsigned_16     len;
```

The **mem_addr** contains the address of the memory block to read from the remote machine. The length of the block is determined by **len**. The memory data will be copied to output message.

Return message:

```
bytes       data
```

The **data** field stores the memory block read in. The length of this memory block is given by the return value from TrapRequest. If error has occurred in reading memory, the length of the data returns will not be equal to the number of bytes requested.

3.1.11 REQ_WRITE_MEM (10)

Request to write a block of memory.

Request message:

```
access_req      req
addr48_ptr      mem_addr
------------------------
bytes           data
```

The **data** field stores the memory data to be transferred. The data will be stored in the debuggee's address space starting at the address in the **mem_addr** field.

Return message:

```
unsigned_16 len
```

The **len** field tells the length of memory block actually written to the debuggee machine. If error has occurred in writing the memory, the length returned will not be equal to the number of bytes requested.

3.1.12 REQ_READ_IO (11)

Request to read data from I/O address space of the debuggee.

Request message:

```
access_req      req
unsigned_32     IO_offset
unsigned_8      len
```

The **IO_offset** contains the I/O address of the debuggee machine. The length of the block is determined by **len**. It must be 1, 2 or 4 bytes. The data will be copied from **IO_offset** to the return message.

Return message:

```
bytes       data
```

The **data** field stores the memory block read in. The length of this memory block is given by the return value from TrapRequest. If an error has occurred in reading, the length returned will not be equal to the number of bytes requested.

3.1.13 REQ_WRITE_IO (12)

Request to write data to the I/O address space of the debuggee.

Request message:

```
access_req      req
unsigned_32     IO_offset
------------------------
bytes           data
```

The **IO_offset** contains the I/O address of the debuggee machine. The data stored in **data** field will be copied to **IO_offset** on the debuggee machine.

Return message:

```
unsigned_8  len
```

The **len** field tells the number of bytes actually written out. If an error has occurred in writing, the length returned will not be equal to the number of bytes requested.

3.1.14 REQ_PROG_GO (13)/REQ_PROG_STEP (14)

Requests to execute the debuggee. REQ_PROG_GO causes the debuggee to resume execution, while REQ_PROG_STEP requests only a single machine instruction to be executed before returning. In either case, this request will return when a breakpoint, watchpoint, machine exception or other significant event has been encountered. While executing, a trap file is allowed to return spurious COND_WATCH indications. The debugger always checks its own watchpoint table for changes before reporting to the user. This means that a legal implementation of a trap file (but **very** inefficient) can just single step the program and return COND_WATCH for every instruction when there are active watchpoints present.

Request message:

```
access_req      req
```

The request is in **req** field.

Return message:

```
addr48_ptr      stack_pointer
addr48_ptr      program_counter
unsigned_16     conditions
```

The **stack_pointer** and **program_counter** fields store the latest values of SS:ESP and CS:EIP (or their non-x86 equivalents) respectively. The **conditions** informs the debugger what conditions have changed since execution began. It contains the following flags:

```
Bit 0   : COND_CONFIG           - Configurations change
Bit 1   : COND_SECTIONS         - Program overlays change
Bit 2   : COND_LIBRARIES        - Libraries (DLL) change
Bit 3   : COND_ALIASING         - Alias change
Bit 4   : COND_THREAD           - Thread change
Bit 5   : COND_THREAD_EXTRA     - Thread extra change
Bit 6   : COND_TRACE            - Trace point occurred
Bit 7   : COND_BREAK            - Break point occurred
Bit 8   : COND_WATCH            - Watch point occurred
Bit 9   : COND_USER             - User interrupt
Bit 10  : COND_TERMINATE        - Program terminated
Bit 11  : COND_EXCEPTION        - Machine exception
Bit 12  : COND_MESSAGE          - Message to be displayed
Bit 13  : COND_STOP             - Debuggee wants to stop
Bit 14  : COND_RUNNING          - Debuggee is running
Bit 15  :  not used
```

When a bit is off, the debugger avoids having to make additional requests to determine the new state of the debuggee. If the trap file is not sure that a particular item has changed, or if it is expensive to find out, it should just turn the bit on.

3.1.15 REQ_PROG_LOAD (15)

Request to load a program.

Request message:

```
access_req      req
unsigned_8      true_argv
------------------------
bytes           argv
```

The **true_argv** field indicates whether the argument consists of a single string, or a true C-style argument vector. This field is set to be one for a true argument vector and zero otherwise. The **argv** is a set of zero-terminated strings, one following each other. The first string gives the name of the program to be loaded. The remainder of the **argv** field contains the program's arguments. The arguments can be a single string or an array of strings.

Return message:

```
trap_error      err
trap_phandle    task_id
trap_mhandle    mod_handle
unsigned_8      flags
```

The **err** field returns the error code while loading the program. The **task_id** shows the task (process) ID for the program loaded. The **mod_handle** is the system module identification for the executable image. It is used as input to the REQ_MAP_ADDR request. The **flags** field contains the following information:

```
Bit 0   : LD_FLAG_IS_BIG            - 32-bit program (obsolete)
Bit 1   : LD_FLAG_IS_PROT           - Protected mode (obsolete)
Bit 2   : LD_FLAG_IS_STARTED        - Program already started
Bit 3   : LD_FLAG_IGNORE_SEGMENTS   - Ignore segments (flat)
Bit 4   : LD_FLAG_HAVE_RUNTIME_DLLS - DLL load breaks supported
Bit 5   : LD_FLAG_DISPLAY_DAMAGED   - Debugger must repaint screen
Bit 6,7 : not used
```

3.1.16 REQ_PROG_KILL (16)

Request to kill the program.

Request message:

```
access_req      req
trap_phandle    task_id
```

The **req** field contains the request. The **task_id** field (obtained from REQ_PROG_LOAD) identifies the program to be killed.

Return message:

```
trap_error      err
```

The **err** field returns the error code of the OS kill program operation.

3.1.17 REQ_SET_WATCH (17)

Request to set a watchpoint at the address given.

Request message:

```
access_req      req
addr48_ptr      watch_addr
unsigned_8      size
```

The address of the watchpoint is given by the **watch_addr** field. The **size** field gives the number of bytes to be watched.

Return message:

```
trap_error      err
unsigned_32     multiplier
```

The **err** field returns the error code if the setting failed. If the setting of the watchpoint worked, the 31 low order bits of **multiplier** indicate the expected slow down of the program when it's placed into execution. The top bit of the field is set to one if a debug register is being used for the watchpoint, and zero if the watchpoint is being done by software.

3.1.18 REQ_CLEAR_WATCH (18)

Request to clear a watchpoint at the address given. The trap file may assume all watch points are cleared at once.

Request message:

```
access_req      req
addr48_ptr      watch_addr
unsigned_8      size
```

The address of the watch point is given by the **watch_addr** field. The **size** field gives the size of the watch point.

Return message:

```
NONE
```

3.1.19 REQ_SET_BREAK (19)

Request to set a breakpoint at the address given.

Request message:

```
access_req      req
addr48_ptr      break_addr
```

The address of the break point is given by the **break_addr** field.

Return message:

```
unsigned_32     old
```

The **old** field returns the original byte(s) at the address **break_addr**.

3.1.20 REQ_CLEAR_BREAK (20)

Request to clear a breakpoint at the address given. The trap file may assume all breakpoints are cleared at once.

Request message:

```
access_req      req
addr48_ptr      break_addr
unsigned_32     old
```

The address of the break point is given by the **break_addr** field. The **old** field holds the old instruction returned from the REQ_SET_BREAK request.

Return message:

```
NONE
```

3.1.21 REQ_GET_NEXT_ALIAS (21)

Request to get alias information for a segment. In some protect mode environments (typically 32-bit flat) two different selectors may refer to the same physical memory. Which selectors do this is important to the debugger in certain cases (so that symbolic information is properly displayed).

Request message:

```
access_req      req
unsigned_16     seg
```

The **seg** field contains the segment. To get the first alias, put zero in this field.

Return message:

```
unsigned_16     seg
unsigned_16     alias
```

The **seg** field contains the next segment where an alias appears. If this field returns zero, it implies no more aliases can be found. The **alias** field returns the alias of the input segment. Zero indicates a previously set alias should be deleted.

3.1.22 REQ_SET_USER_SCREEN (22)

Request to make the debuggee's screen visible.

Request message:

```
access_req      req
```

Return message:

```
NONE
```

3.1.23 REQ_SET_DEBUG_SCREEN (23)

Request to make the debugger's screen visible.

Request message:

```
access_req      req
```

Return message:

```
NONE
```

3.1.24 REQ_READ_USER_KEYBOARD (24)

Request to read the remote keyboard input.

Request message:

```
access_req      req
unsigned_16     wait
```

The request will be time out if it waits longer than the period specifies in the **wait** field. The waiting period is measured in seconds. A value of zero means to wait forever.

Return message:

```
unsigned_8      key
```

The **key** field returns the input character from remote machine.

3.1.25 REQ_GET_LIB_NAME (25)

Request to get the name of a newly loaded library (DLL).

Request message:

```
access_req      req
trap_mhandle    mod_handle
```

The **mod_handle** field contains the library handle. It should be zero to get the name of the first DLL or the value from the **mod_handle** of a previous request.

Return message:

```
trap_mhandle    mod_handle
--------------------------
string          name
```

The **mod_handle** field contains the library handle. It contains zero if there are no more DLL names to be returned. The name of the library will be returned in **name** field. If the **name** field is an empty string (consists just of the '\0' character), then this is a indication that the DLL indicated by the given handle has been unloaded, and the debugger should remove any symbolic information for the image. It is an error to attempt to remove a handle that has not been loaded in a previous REQ_GET_LIB_NAME request.

3.1.26 REQ_GET_ERR_TEXT (26)

Request to get the error message text for an error code.

Request message:

```
access_req      req
trap_error      err
```

The **err** field contains the error code number of the error text requested.

Return message:

```
string          error_msg
```

The error message text will be returned in **error_msg** field.

3.1.27 REQ_GET_MESSAGE_TEXT (27)

Request to retrieve generic message text. After a REQ_PROG_LOAD, REQ_PROG_GO or REQ_PROG_STEP has returned with COND_MESSAGE or COND_EXCEPTION, the debugger will make this request to obtain the message text. In the case of a COND_EXCEPTION return text describing the machine exception that caused the return to the debugger. Otherwise return whatever generic message text that the trap file wants to display to the user.

Request message:

```
access_req      req
```

Return message:

```
unsigned_8      flags
--------------------
string          msg
```

The message text will be returned in the **msg** field. The **flags** contains a number of bits which control the next action of the debugger. They are:

```
Bit 0       : MSG_NEWLINE
Bit 1       : MSG_MORE
Bit 2       : MSG_WARNING
Bit 3       : MSG_ERROR
Bit 4 - 7   : not used
```

The MSG_NEWLINE bit indicates that the debugger should scroll its display to a new line after displaying the message. The MSG_MORE bit indicates that there is another line of output to come and the debugger should make another REQ_GET_MESSAGE_TEXT. MSG_WARNING indicates that the message is a warning level message while MSG_ERROR is an error level message. If neither of these bits are on, the message is merely informational.

3.1.28 REQ_REDIRECT_STDIN (28)/REQ_REDIRECT_STDOUT (29)

Request to redirect the standard input (REQ_REDIRECT_STDIN) or standard output (REQ_REDIRECT_STDOUT) of the debuggee.

Request message:

```
access_req      req
------------------------
string          name
```

The file name to be redirected to/from is given by the **name** field.

Return message:

```
trap_error      err
```

When an error has occurred, the **err** field contains an error code indicating the type of error that has been detected.

3.1.29 REQ_SPLIT_CMD (30)

Request to split the command line into the command name and parameters.

Request message:

```
access_req      req
------------------------
string          cmd
```

The **cmd** field contains the command. Command can be a single command line or an array of command strings.

Return message:

```
unsigned_16     cmd_end
unsigned_16     parm_start
```

The **cmd_end** field tells the position in command line where the command name ends. The **parm_start** field stores the position where the program arguments begin.

3.1.30 REQ_READ_REGS (31)

Request to read CPU register contents. The data returned depends on the target architecture and is defined by the MAD file.

Request message:

```
access_req      req
```

Return message:

```
unspecified
```

The return message content is specific to the MAD in use and will contain a **mad_registers** union (defined in **madtypes.h**).

3.1.31 REQ_WRITE_REGS (32)

Request to write CPU register contents. The data is target architecture specific.

Request message:

```
access_req      req
------------------
unspecified
```

The message content is specific to the MAD in use and will contain a **mad_registers** union.

Return message:

```
NONE
```

3.1.32 REQ_MACHINE_DATA (33)

Request to retrieve machine specific data.

Request message:

```
access_req      req;
unsigned_8      info_type;
addr48_ptr      addr;
----------------------------
unspecified
```

The **info_type** field specifies what kind of information should be returned and **addr** determines the address for which the information is requested. The remainder of the message is MAD specific.

Return message:

```
addr48_off      cache_start;
addr48_off      cache_end;
----------------------------
unspecified
```

The return message content is specific to the MAD in use.

3.2 File I/O requests

This section describes requests that deal with file input/output on the target (debuggee) machine. These requests are actually performed by the core request REQ_PERFORM_SUPPLEMENTARY_SERVICE and appropriate service ID. The following descriptions do not show that "prefix" to the request messages.

The service name to be used in the REQ_GET_SUPPLEMENTARY_SERVICE is "Files".

The file requests use a new basic type in addition to the ones already described:

Type	*Definition*
trap_fhandle	This is an **unsigned_64** which holds a debuggee file handle.

3.2.1 REQ_FILE_GET_CONFIG (0)

Request to retreive characteristics of the remote file system.

Request message:

```
access_req      req
```

Return message:

```
char            ext_separator;
char            path_separator[3];
char            newline[2];
```

The **ext_separator** contains the separator for file name extensions. The possible path separators can be found in array **path_separator**. The first one is the "preferred" path separator for that operating system. This is the path separator that the debugger will use if it needs to construct a file name for the remote system. The new line control characters are stored in array **newline**. If the operating system uses only a single character for newline, put a zero in the second element.

3.2.2 REQ_FILE_OPEN (1)

Request to create/open a file.

Request message:

```
access_req          req
unsigned_8          mode
-----------------------
string              name
```

The name of the file to be opened is given by **name**. The **mode** field stores the access mode of the file. The following bits are defined:

```
Bit 0       :  TF_READ
Bit 1       :  TF_WRITE
Bit 2       :  TF_CREATE
Bit 3       :  TF_EXEC
Bit 4 - 7   :  not used
```

For read/write mode, turn both **TF_READ** and **TF_WRITE** bits on. The **TF_EXEC** bit should only be used together with **TF_CREATE** and indicates that the created file needs executable permission (if relevant on the target platform).

Return message:

```
trap_error      err
trap_fhandle    handle
```

If successful, the **handle** returns a handle for the file. When an error has occurred, the **err** field contains a value indicating the type of error that has been detected.

3.2.3 REQ_FILE_SEEK (2)

Request to seek to a particular file position.

Request message:

```
access_req      req
trap_fhandle    handle
unsigned_8      mode
unsigned_32     pos
```

The handle of the file is given by the **handle** field. The **mode** field stores the seek mode. There are three seek modes:

```
TF_SEEK_ORG = 0  - Relative to the start of file
TF_SEEK_CUR = 1  - Relative to the current file position
TF_SEEK_END = 2  - Rrelative to the end of file
```

The position to seek to is in the **pos** field.

Return message:

```
trap_error      err
unsigned_32     pos
```

If an error has occurred, the **err** field contains a value indicating the type of error that has been detected. The **pos** field returns the current position of the file.

3.2.4 REQ_FILE_READ (3)

Request to read a block of data from a file.

Request message:

```
access_req          req
trap_fhandle        handle
unsigned_16         len
```

The handle of the file is given by the **handle** field. The **len** field stores the number of bytes to be transmitted.

Return message:

```
trap_error          err
-------------------------
bytes               data
```

If successful, the **data** returns the block of data. The length of returned data is given by the return value of TrapRequest minus 4 (to account for the size of **err**). The length will normally be equal to the **len** field. If the end of file is encountered before the read completes, the return value will be less than the number of bytes requested. When an error has occurred, the **err** field contains a value indicating the type of error that has been detected.

3.2.5 REQ_FILE_WRITE (4)

Request to write a block of data to a file.

Request message:

```
access_req      req
trap_fhandle    handle
-----------------------
bytes           data
```

The handle of the file is given by the **handle** field. The data is given in **data** field.

Return message:

```
trap_error      err
unsigned_16     len
```

If there is no error, **len** will equal to that in the **data_len** field. When an error has occurred, the **err** field contains a value indicating the type of error that has been detected.

3.2.6 REQ_FILE_WRITE_CONSOLE (5)

Request to write a block of data to the debuggee's screen.

Request message:

```
access_req      req
------------------------
bytes           data
```

The data is given in **data** field.

Return message:

```
trap_error      err
unsigned_16     len
```

If there is no error, **len** will equal to the **data_len** field. When an error has occurred, the **err** field contains a value indicating the type of error that has been detected.

3.2.7 REQ_FILE_CLOSE (6)

Request to close a file.

Request message:

```
access_req      req
trap_fhandle    handle
```

The handle of the file is given by the **handle** field.

Return message:

```
trap_error      err
```

When an error has occurred, the **err** field contains a value indicating the type of error that has been detected.

3.2.8 REQ_FILE_ERASE (7)

Request to erase a file.

Request message:

```
access_req      req
------------------------
string          file_name
```

The **file_name** field contains the file name to be deleted.

Return message:

```
trap_error  err
```

If error has occurred when erasing the file, the **err** field will return the error code number.

3.2.9 REQ_FILE_STRING_TO_FULLPATH (8)

Request to convert a file name to its full path name.

Request message:

```
access_req      req
unsigned_8      file_type
------------------------
string          file_name
```

The **file_type** field indicates the type of the input file. File types can be:

```
TF_FILE_EXE  =  0
TF_FILE_DBG  =  1
TF_FILE_PRS  =  2
TF_FILE_HLP  =  3
```

This is so the trap file can search different paths for the different types of files. For example, under QNX, the PATH environment variable is searched for the FILE_EXE type, and the WD_PATH environment variable is searched for the others. The **file_name** field contains the file name to be converted.

Return message:

```
trap_error      err
-------------------------
string          path_name
```

If no error occurs the **err** field returns a zero and the full path name will be stored in the **path_name** field. When an error has occurred, the **err** field contains an error code indicating the type of error that has been detected.

3.2.10 REQ_FILE_RUN_CMD (9)

Request to run a command on the target (debuggee's) system.

Request message:

```
access_req      req
unsigned_16     chk_size
------------------------
string          cmd
```

The **chk_size** field gives the check size in kilobytes. This field is only useful in the DOS implementation. It contains the value of the /CHECKSIZE debugger command line option and represents the amount of memory the user wishes to have free for the spawned sub-shell. The **cmd** field stores the command to be executed.

Return message:

```
trap_error      err
```

If error has occurred when executing the command, the **err** field will return the error code number.

3.3 Overlay requests

This section describes requests that deal with overlays (supported only under 16-bit DOS). These requests are actually performed by the core request REQ_PERFORM_SUPPLEMENTARY_SERVICE and appropriate service ID. The following descriptions do not show that "prefix" to the request messages.

The service name to be used in the REQ_GET_SUPPLEMENTARY_SERVICE is "Overlays".

The overlay requests use a new basic type in addition to the ones already described:

Type	*Definition*

addr32_ptr	This type encapsulates the concept of a 16:16 pointer into the debuggee's address space. Since overlays are only useful for 16-bit environments, using the addr48_ptr type would be inefficient. The structure is defined as follows:
	`typedef struct {`
	`  unsigned_16  offset;`
	`  unsigned_16  segment;`
	`} addr32_ptr;`
	
	The **`segment`** field contains the segment of the address and the **`offset`** field stores the offset of the address.

ovl_address	This type contains the overlay address and the number of entries down in the overlay stack. The structure is defined as follows:
	`typedef struct {`
	`  addr32_ptr  mach;`
	`  unsigned_16 sect_id;`
	`} ovl_address;`
	
	The **`mach`** field is the machine address. The **`sect_id`** field stores the address section number.

3.3.1 REQ_OVL_STATE_SIZE (0)

Request to return the size of the overlay state information in bytes of the task program. This request maps onto the overlay manager's GET_STATE_SIZE request. See the Overlay Manager Interface document for more information on the contents of the return message.

Request message:

```
access_req      req
```

The **`req`** field contains the request.

Return message:

```
unsigned_16     size
```

The **`size`** field returns the size in bytes. A value of zero indicates no overlays are present in the debuggee and none of the other requests dealing with overlays will ever be called.

3.3.2 REQ_OVL_GET_DATA (1)

Request to get the address and size of an overlay section. This request maps onto the overlay manager's GET_SECTION_DATA request. See the Overlay Manager Interface document for more information on the contents of the return message.

Request message:

```
access_req      req
unsigned_16     sect_id
```

The **sect_id** field indicates the overlay section the information is being requested of.

Return message:

```
unsigned_16     segment
unsigned_32     size
```

The **segment** field contains the segment value where the overlay section is loaded (or would be loaded if it was brought into memory). The **size** field gives the size, in bytes, of the overlay section. If there is no section for the given id, the **segment** field will be zero.

3.3.3 REQ_OVL_READ_STATE (2)

Request to read the overlay table state. This request maps onto the overlay manager's GET_OVERLAY_STATE request. See the Overlay Manager Interface document for more information on the contents of the return message. The size of the returned data is provided by the REQ_OVL_STATE_SIZE trap file request.

Request message:

```
access_req      req
```

Return message:

```
bytes           data
```

The **data** field contains the overlay state information requested.

3.3.4 REQ_OVL_WRITE_STATE (3)

Request to write the overlay table state. This request maps onto the overlay manager's SET_OVERLAY_STATE request. See the Overlay Manager Interface document for more information on the contents of the return message.

Request message:

```
access_req      req
----------------------
bytes           data
```

The **data** field contains the overlay state information to be restored.

Return message:

```
NONE
```

3.3.5 REQ_OVL_TRANS_VECT_ADDR (4)

Request to check if the input overlay address is actually an overlay vector. This request maps onto the overlay manager's TRANSLATE_VECTOR_ADDR request. See the Overlay Manager Interface document for more information on the contents of the messages.

Request message:

```
access_req      req
ovl_address     ovl_addr
```

The **mach** field is the machine address. The **sect_id** field stores the number of entries down in the overlay stack.

Return message:

```
ovl_address     ovl_addr
```

The translated address will be returned in the **ovl_addr** field. If the address is not an overlay vector, then the input address will be returned and the **sect_id** field will be zero.

3.3.6 REQ_OVL_TRANS_RET_ADDR (5)

Request to check if the address is the overlay manager parallel return code. This request maps onto the overlay manager's TRANSLATE_RETURN_ADDR request. See the Overlay Manager Interface document for more information on the contents of the messages.

Request message:

```
access_req      req
ovl_address     ovl_addr
```

Return message:

```
ovl_address     ovl_addr
```

The translated address will be returned in the **ovl_addr** field. If the address is not an parallel return code, then the input address will be returned and the **sect_id** field in the structure **ovl_addr** will be zero.

3.3.7 REQ_OVL_GET_REMAP_ENTRY (6)

Request to check if the overlay address needs to be remapped. This request maps onto the overlay manager's GET_MOVED_SECTION request. See the Overlay Manager Interface document for more information on the contents of the messages.

Request message:

```
access_req      req
ovl_address     ovl_addr
```

The **ovl_addr** field contains the overlay address.

Return message:

```
unsigned_8      remapped
ovl_address     ovl_addr
```

If the address gets remapped the **remapped** field will return one. The remapped address will be returned in the **ovl_addr** field. The input address will be unchanged if the address has not been remapped.

3.4 Thread requests

This section descibes requests that deal with threads. These requests are actually performed by the core request REQ_PERFORM_SUPPLEMENTARY_SERVICE and appropriate service ID. The following descriptions do not show that "prefix" to the request messages.

The service name to be used in the REQ_GET_SUPPLEMENTARY_SERVICE is "Threads".

The thread requests use a new basic type in addition to the ones already described:

Type	*Definition*
trap_thandle	This is an **unsigned_32** which holds a thread handle.

3.4.1 REQ_THREAD_GET_NEXT (0)

Request to get next thread.

Request message:

```
access_req      req
trap_thandle    thread
```

The **thread** contains the either a zero to get information on the first thread, or the value of the **thread** field in the return message of a previous request.

Return message:

```
trap_thandle    thread
unsigned_8      state
```

The **thread** field returns the thread ID. There are no more threads in the list, it will contain zero. The **state** field can have two values:

```
THREAD_THAWED = 0
THREAD_FROZEN = 1
```

3.4.2 REQ_THREAD_SET (1)

Request to set a given thread ID to be the current thread.

Request message:

```
access_req      req
trap_thandle    thread
```

The **thread** contains the thread number to set. If it's zero, do not attempt to set the thread, just return the current thread ID.

Return message:

```
trap_error      error
trap_thandle    old_thread
```

The **old_thread** field returns the previous thread ID. If the set fails, the **err** field will be non-zero.

3.4.3 REQ_THREAD_FREEZE (2)

Request to freeze a thread so that it will not be run next time when executing the task program.

Request message:

```
access_req      req
trap_thandle    thread
```

The **thread** contains the thread number to freeze.

Return message:

```
trap_error      err
```

If the thread cannot be frozen, the **err** field returns non-zero value.

3.4.4 REQ_THREAD_THAW (3)

Request to allow a thread to run next time when executing the program.

Request message:

```
access_req      req
trap_thandle    thread
```

The **thread** contains the thread number to thaw.

Return message:

```
trap_error      err
```

If the thread cannot be thawed, the **err** field returns non zero value.

3.4.5 REQ_THREAD_GET_EXTRA (4)

Request to get extra information about a thread. This is arbitrary textual data which the debugger merely displays in its thread window. The trap file can place any information in the return message which it feels would be useful for the user to know.

Request message:

```
access_req      req
unsigned_32     thread
```

The **thread** field contains the thread ID. A zero value means to get the title string for the thread extra information. This is displayed at the top of the thread window.

Return message:

```
string          extra
```

The extra information of the thread will be returned in **extra** field.

3.5 RFX requests

This section deals with requests that are only used by the RFX (Remote File Xfer) program. These requests are actually performed by the core request REQ_PERFORM_SUPPLEMENTARY_SERVICE and appropriate service ID. The following descriptions do not show that "prefix" to the request messages.

The service name to be used in the REQ_GET_SUPPLEMENTARY_SERVICE is "RFX".

3.5.1 REQ_RFX_RENAME (0)

Request to rename a file on the debuggee's system.

Request message:

```
access_req      req
------------------------
string          from_name
------------------------
string          to_name
```

The file whose name is indicated by the field **`from_name`** will be renamed to the name given by the field **`to_name`**

Return message:

```
trap_error      err
```

If error has occurred, the **`err`** field will return the error code number.

3.5.2 REQ_RFX_MKDIR (1)

Request to create a directory on the target (debuggee) system.

Request message:

```
access_req      req
-----------------------
string          dir_name
```

The **`dir_name`** field contains the name of the directory to be created.

Return message:

```
trap_error      err
```

If error has occurred when creating the directory, the **`err`** field will return the error code number.

3.5.3 REQ_RFX_RMDIR (2)

Request to remove a directory on the target system.

Request message:

```
access_req      req
-----------------------
string          dir_name
```

The **dir_name** field contains the name of the directory to be removed.

Return message:

```
trap_error      err
```

If error has occurred, the **err** field will return the error code number.

3.5.4 REQ_RFX_SETDRIVE (3)

Request to set the current drive on the target system.

Request message:

```
access_req      req
unsigned_8      drive
```

The **drive** field contains the drive number to be set on the target system (0=A,1=B,...).

Return message:

```
trap_error      err
```

If error has occurred, the **err** field will return the error code number.

3.5.5 REQ_RFX_GETDRIVE (4)

Request to get the current drive on the target system.

Request message:

```
access_req      req
```

The **req** field contains the request.

Return message:

```
unsigned_8      drive
```

The **drive** field returns the current drive number on the target system (0=A,1=B,...).

3.5.6 REQ_RFX_SETCWD (5)

Request to set a directory on the target system.

Request message:

```
access_req      req
-----------------------
string          dir_name
```

The **dir_name** field contains the name of the directory to be set.

Return message:

```
trap_error      err
```

If error has occurred, the **err** field will return the error code number.

3.5.7 REQ_RFX_GETCWD (6)

Request to get the current directory name on the target system.

Request message:

```
access_req      req
unsigned_8      drive
```

The **drive** field contains the target drive number (0=current drive,1=A,2=B,...).

Return message:

```
trap_error      err
-------------------------
string          dir_name
```

The **dir_name** field contains the name of the directory to be set. If error has occurred, the **err** field will return the error code number.

3.5.8 REQ_RFX_SETDATETIME (7)

Request to set a file's date and time information on the target system.

Request message:

```
access_req      req
trap_fhandle    handle
time_t          time
```

The **handle** contains the file handle. The **time** field follows the UNIX time format. The **time** represents the time since January 1, 1970 (UTC).

Return message:

```
NONE
```

3.5.9 REQ_RFX_GETDATETIME (8)

Request to get the date and time information for a file on the target system.

Request message:

```
access_req      req
trap_fhandle    handle
```

The **handle** contains the file handle.

Return message:

```
time_t          time
```

The **time** field follows the UNIX time format. The **time** represents the time since January 1, 1970 (UTC).

3.5.10 REQ_RFX_GETFREESPACE (9)

Request to get the amount of free space left on the drive.

Request message:

```
access_req      req
unsigned_8      drive
```

The **drive** field contains the target drive number (0=current drive,1=A,2=B,...).

Return message:

```
unsigned_32     size
```

The **size** field returns the number of bytes left on the drive.

3.5.11 REQ_RFX_SETFILEATTR (10)

Request to set the file attribute of a file.

Request message:

```
access_req      req
unsigned_32     attribute
------------------------
string          name
```

The **name** field contains the name whose attributes are to be set. The **attribute** field contains the new attributes of the file.

Return message:

```
trap_error      err
```

If error has occurred, the **err** field will return the error code number.

3.5.12 REQ_RFX_GETFILEATTR (11)

Request to get the file attribute of a file.

Request message:

```
access_req      req
-------------------
string          name
```

The **name** field contains the name to be checked.

Return message:

```
unsigned_32     attribute
```

The **attribute** field returns the attribute of the file.

3.5.13 REQ_RFX_NAMETOCANONICAL (12)

Request to convert a file name to its canonical form.

Request message:

```
access_req      req
------------------------
string          file_name
```

The **file_name** field contains the file name to be converted.

Return message:

```
trap_error      err
-------------------------
string          path_name
```

If there is no error, the **err** field returns a zero and the full path name will be stored in the **path_name** field. When an error has occurred, the **err** field contains an error code indicating the type of error that has been detected.

3.5.14 REQ_RFX_FINDFIRST (13)

Request to find the first file in a directory.

Request message:

```
access_req      req
unsigned_8      attrib
----------------------
string          name
```

The **name** field contains the name of the directory and the **attrib** field contains the attribute of the files to list in the directory.

Return message:

```
trap_error      err
-----------------------
rfx_find        info
```

If found, the **err** field will be zero. The location and information of about the first file will be in the structure **info.** Definition of the structure **rfx_find** is as follows:

```
typedef struct rfx_find {
    unsigned_8          reserved[21];
    unsigned_8          attr;
    unsigned_16         time;
    unsigned_16         date;
    unsigned_32         size;
    unsigned_8          name[260];
} rfx_find;
```

3.5.15 REQ_RFX_FINDNEXT (14)

Request to find the next file in the directory. This request should be used only after REQ_RFX_FINDFIRST.

Request message:

```
access_req      req
--------------------
rfx_find        info
```

The **req** field contains the request. The **info** field contains the rfx_find structure returned from the previous REQ_FIND_NEXT or REQ_FIND_FIRST.

Return message:

```
trap_error      err
-----------------------
rfx_find        info
```

The **info** field is the same as in REQ_FIND_FIRST.

3.5.16 REQ_RFX_FINDCLOSE (15)

Request to end the directory search operation.

Request message:

```
access_req      req
```

The **req** field contains the request.

Return message:

```
trap_error      err
```

If successful, the **err** field will be zero, otherwise the system error code will be returned.

4 System Dependent Aspects

Every environment has a different method of loading the code for the trap file and locating the TrapInit, TrapRequest, and TrapFini routines. This section descibes how the Open Watcom debugger performs these operations for the various systems.

4.1 Trap Files Under DOS

A trap file is an "EXE" format file with the extension ".TRP". The debugger searches the directories specified by the PATH environment variable. Once found, it is loaded into memory and has the normal EXE style relocations applied to the image. Then the lowest address in the load image (NOTE: not the starting address from EXE header information) is examined for the following structure:

```
typedef struct {
    unsigned_16     signature; /* == 0xDEAF */
    unsigned_16     init_off;
    unsigned_16     acc_off;
    unsigned_16     fini_off;
} trap_header;
```

If the first 2 bytes contain the value 0xDEAF, the file is considered to be a valid trap file and the **`init_off`**, **`acc_off`**, and **`fini_off`** fields are used to obtain the offsets of the TrapInit, TrapRequest, and TrapFini routines repectively.

The starting address field of the EXE header should be set to point at some code which prints out a message about not being able to be run from the command line and then terminates.

4.2 Trap Files Under OS/2

A trap file is a normal OS/2 DLL. The system automatically searches the directories specified by the LIBPATH command in the CONFIG.SYS file. Once loaded, the Open Watcom debugger uses export ordinal 1 from the DLL for TrapInit, export ordinal 2 for TrapFini and export ordinal 3 for TrapRequest. Some example code follows:

```
rc = DosLoadModule( NULL, 0, trap_file_name, &dll_module );
if( rc != 0 ) {
    return( "unable to load trap file" );
}
if( DosGetProcAddr( dll_module, "#1", &TrapInit ) != 0
  || DosGetProcAddr( dll_module, "#2", &TrapFini ) != 0
  || DosGetProcAddr( dll_module, "#3", &TrapRequest ) != 0 ) {
    return( "incorrect version of trap file" );
}
```

4.3 Trap Files Under Windows.

A trap file is a normal Windows DLL. The system automatically searches the directories specified by the PATH environment variable. Once loaded, the Open Watcom debugger uses export ordinal 2 from the DLL for TrapInit, export ordinal 3 for TrapFini and export ordinal 4 for TrapRequest. Some example code follows:

```
dll = LoadLibrary( trap_file_name );
if( dll < 32 ) {
    return( "unable to load trap file" );
}
TrapInit   = (LPVOID) GetProcAddress( dll, (LPSTR)2 );
TrapFini   = (LPVOID) GetProcAddress( dll, (LPSTR)3 );
TrapRequest = (LPVOID) GetProcAddress( dll, (LPSTR)4 );
if( TrapInit == NULL || TrapFini == NULL || TrapRequest == NULL ) {
    return( "incorrect version of trap file" );
}
```

4.4 Trap Files Under Windows NT.

A trap file is a normal Windows NT DLL. The system automatically searches the directories specified by the PATH environment variable. Once loaded, the Open Watcom debugger uses export ordinal 1 from the DLL for TrapInit, export ordinal 2 for TrapFini and export ordinal 3 for TrapRequest. Some example code follows:

```
dll = LoadLibrary( trap_file_name );
if( dll < 32 ) {
    return( "unable to load trap file" );
}
TrapInit   = (LPVOID) GetProcAddress( dll, (LPSTR)1 );
TrapFini   = (LPVOID) GetProcAddress( dll, (LPSTR)2 );
TrapRequest = (LPVOID) GetProcAddress( dll, (LPSTR)3 );
if( TrapInit == NULL || TrapFini == NULL || TrapRequest == NULL ) {
    return( "incorrect version of trap file" );
}
```

4.5 Trap Files Under QNX

A trap file is a QNX load module format file with the extension ".trp" and whose file permissions are not marked as executable. The debugger searches the directories specified by the WD_PATH environment variable and then the "/usr/watcom/wd" directory. Once found, it is loaded into memory and has the normal loader relocations applied to the image. Then the lowest address in the load image (NOTE: not the starting address from load module header information) is examined for the following structure:

```
typedef struct {
    unsigned_16     signature; /* == 0xDEAF */
    unsigned_16     init_off;
    unsigned_16     acc_off;
    unsigned_16     fini_off;
} trap_header;
```

If the first 2 bytes contain the value 0xDEAF, the file is considered to be a valid trap file and the **`init_off`**, **`acc_off`**, and **`fini_off`** fields are used to obtain the offsets of the TrapInit, TrapRequest, and TrapFini routines repectively.

The starting address field of the load image header should be set to point at some code which prints out a message about not being able to be run from the command line and then terminates.

4.6 Trap Files Under Netware 386 or PenPoint

The trap file routines are linked directly into the remote server code and TrapInit, TrapRequest, TrapFini are directly called.

Overlay Manager Interface VERSION 3.0

1 Overlay manager interface

For Open Watcom Debugger to be able to debug overlays, it must be able to make requests of the overlay manager for certain operations. The overlay manager must also be able to inform Open Watcom Debugger when a new overlay section is loaded.

When Open Watcom Debugger loads a DOS program, it looks at the initial CS:IP value for the following structure:

```
struct ovl_header {
    unsigned_8  short_jmp_opcode;       /* == 0xeb */
    signed_8    short_jmp_displacment;
    unsigned_16 signature;              /* == 0x2112 */
    void        (far *hook)();
    unsigned_16 handler_offset;
};
```

Open Watcom Debugger checks to make sure that the first instruction is a short jump (opcode 0xeb) and that the word following that instruction contains the value 0x2112. If this occurs, Open Watcom Debugger assumes that it is debugging an overlaid application.

Open Watcom Debugger then fills in the **hook** field with the far address of a routine that is invoked with a far call whenever a change in the overlay state occurs. The initial CS value and the contents of the **handler_offset** field gives the far address of the overlay manager routine responsible for handling debugger requests.

1.1 The Hook Routine

After the routine addresses have been exchanged, Open Watcom Debugger starts the program executing, to allow the overlay manager to initialize. After the manager has finished its initialization, it performs a far call to the debugger hook routine, with the return address on the stack being the "real" starting address of the program being debugged. All register contents (including flags) should be preserved by the hook routine.

After initialization, the debugger hook routine is invoked with a far call every time a new overlay section is loaded into memory. In this case the AX register contains the section number that was just loaded. The DL register contains a zero or non-zero value if the overlay load was caused by a call or return, respectively. The CX:BX registers form a far pointer to the last byte of the call instruction that caused the overlay load (in the case of a overlay load being caused by a return instruction (DL is non-zero) the far pointer is to the last byte of the call instruction that the return is returning from.)

Notes: More sections than just the one identified by the section number in AX may be loaded by the overlay manager before the hook routine is called. The current overlay manager also loads all of the ancestors of a section (See the WLINK documentation in the Users' Guide for a description of what an ancestor is). To find out what sections are really in memory the debugger should invoke the handler routine with a GET_OVERLAY_STATE request.

1.2 The Handler Routine

The handler routine is responsible for processing requests from the debugger pertaining to overlays. It is invoked by the debugger by performing a far call with a request number in the AX register. The AX register is used to return the result or return status of the request. The CX and BX registers are used for some requests to pass a far pointer to memory.

There are two structures that the handler routines deals with. The first is called an overlay state. An overlay state consists of a block of memory containing all the information necessary for the overlay manager to restore the overlays to their current condition at some later point in time. The first portion of this block is a bit vector, with each bit representing an overlay section. If the bit is a one, then the overlay section is currently in memory. If the bit is a zero then the overlay section is not in memory. To convert from a section number to a bit position use the following formulas:

```
byte_offset = (section_number - 1) / 8;
bit_number  = (section_number - 1) % 8;
```

Following the bit vector is information that the manager uses to restore the overlay stack.

The second structure used is an overlay address. This consists of a far pointer followed by a 16-bit section number.

The following requests are recognized by the debug handler routine.

1.2.1 GET_STATE_SIZE

```
Inputs:                          Outputs:
AX = request number (0)          AX = size of overlay state
```

This request returns the number of bytes required for an overlay state.

1.2.2 GET_OVERLAY_STATE

```
Inputs:                          Outputs:
AX = request number (1)          AX = 1
CX:BX = far pointer to memory
        to store overlay state
```

This request copies the overlay state into the memory pointed at by the CX:BX registers. A one is always returned in AX.

1.2.3 SET_OVERLAY_STATE

```
Inputs:                          Outputs:
AX = request number (2)          AX = 1
CX:BX = far pointer to memory
        to load overlay state
```

This request takes a previously obtained overlay state and causes the overlay manager to return itself to that overlay configuration. A one is always returned in AX. The overlay manager will not explicitly unload a section that is not in memory according to the given overlay state, so a GET_OVERLAY_STATE request following a SET_OVERLAY_STATE may not return the same bit vector portion. This request may also

be used by the debugger to explicitly load a section, so the assembly code may be examined, perhaps. To do this, zero out a block of memory the size of an overlay state, and then turn on the appropriate section number in the bit vector, then make a SET_OVERLAY_STATE request. Remember that not only that section will be loaded, but all of its ancestor sections as well.

1.2.4 TRANSLATE_VECTOR_ADDR

```
Inputs:                          Outputs:
AX = request number (3)          AX = 1 if addr was translated,
CX:BX = far pointer to                0 otherwise
        overlay address
```

This request checks to see if the far pointer portion of the overlay address pointed at by CX:BX is actually an overlay vector. If the address is a vector then the vector address is replaced by the true address of the routine that the vector is for, and the section number portion is filled in with the section number the of routine. A one is returned in AX in this case. If the address is not an overlay vector, then the overlay address is untouched and an zero is returned in AX.

1.2.5 TRANSLATE_RETURN_ADDR

```
Inputs:                          Outputs:
AX = request number (4)          AX = 1 if addr was translated,
CX:BX = far pointer to                0 otherwise
        overlay address
```

In order to handle parallel overlay calls, the overlay manager replaces the true return address on the stack with that of some special code (the parallel return code). It then takes the original return address and section number an places them on the overlay stack. When a routine returns to the overlay manager, it pops the top entry of the overlay stack, makes sure that the original overlay section is loaded, and returns to the original return address.

This function performs much the same function as TRANSLATE_VECTOR_ADDR, except that rather than checking for a vector address, it checks to see if the address is that of the overlay manager parallel return code. If it is then the section number in the overlay address is used as the number of entries down in the overlay stack that the real return address and section number is to be found (zero is the top entry of the overlay stack). The true return address and section number then replaces the contents of the overlay address and a one is returned in AX. If the address is not the parallel return code, then the overlay address is left untouched and a zero is returned in AX.

1.2.6 GET_OVL_TBL_ADDR

```
Inputs:                          Outputs:
AX = request number (5)          AX = 0
CX:BX = far pointer to variable
        of type far pointer to
        be filled in with
        overlay table address
```

This request fills in the far pointer pointed at by CX:BX with the address of the overlay table so that a profiler can find out where sections are located in the executable, or overlay files. The sampler program, when it detects that it is sampling a overlaid application, can perform this function and write the result into the sample file. Since the overlay table is always in the root, the profiler can then find the overlay table and

from that, find the other sections. It should be noted that the format of the overlay table may change, so this call should be avoided if at all possible.

1.2.7 GET_MOVED_SECTION

```
Inputs:                          Outputs:
AX = request number (6)          AX = 1 if the section exists
CX:BX = far pointer to                0 otherwise
        overlay address
```

With the dynamic overlay manager, sections may be loaded, or moved, to positions other than where the linker originally placed them. The debugger must be informed of the new positions so that it can update the locations of its symbolic information. The GET_MOVED_SECTION request is responsible for informing the debugger what sections have moved and their new locations. The debugger will call this request after the hook routine has been called, or the debugger has invoked the SET_OVERLAY_STATE request. The request returns the first section whose id larger than the section number that is in the overlay address being passed in. The overlay manager will fill in the overlay address with the section number that has moved and its new segment address. The offset portion of the overlay address is unused. The request will return a one in AX. If there are no sections numbers larger than the one being passed in that have moved, a zero is returned.

Here is some example debugger code:

```
void CheckMovedSections()
{
    overlay_address     addr;

    addr.sect_id = 0;
    while( OvlHandler( GET_MOVED_SECTION, &addr ) ) {
        HandleMovedSection( addr.sect_id, addr.segment );
    }
}
```

1.2.8 GET_SECTION_DATA

```
Inputs:                          Outputs:
AX = request number (7)          AX = 1 if the section exists
CX:BX = far pointer to                0 otherwise
        overlay address
```

This request returns information on the current location of a section while it is in memory (or where it would be if it was loaded). The section number portion of the overlay address is filled in with the section id that information is being requested about before the request is made. The overlay manager returns zero in AX if the section does not exist. Otherwise it returns one and fills in the overlay address with the location that the section is in memory, or where it would currently go if it was loaded at that time. It also fills in the section number portion of the address with the size of the section in paragraphs.

1.3 Overlay Table Structure

The pointer returned by the GET_OVL_TBL_ADDR request has the following format:

```
typedef struct ovl_table {
    unsigned_8      major;
    unsigned_8      minor;
    void            far *start;
    unsigned_16     delta;
    unsigned_16     ovl_size;
    ovltab_entry    entries[ 1 ];
} ovl_table;
```

The fields **major** and **minor** field contain version numbers for the overlay table structure. If an upwardly compatible change in the structures is made, the minor number will be incremented. If a non-upwardly compatible change to the structures is made, the major field will be incremented. The current major version is 3, the current minor version is 0. The **start** field contains a 32-bit far pointer to the "actual" starting address of the program. The overlay manager jumps to this address after it has finished initializing. (If a debugger/sampler is present then the overlay manager calls into the hook routine with this address on the return stack.) The **delta** field contains the value to be added to each of the segment relocations when a section is loaded into memory (it contains the segment value for the first segment in the program.) The **ovl_size** field contains the size of the overlay area. This is only used in the dynamic overlay manager. The final field, **entries**, is a variable sized array containing one entry for each overlay section in the program (e.g. the tenth element in the array describes overlay section 10.) Each entry has the following form:

```
typedef struct ovltab_entry {
    unsigned_16         flags_anc;
    unsigned_16         relocs;
    unsigned_16         start_para;
    unsigned_16         code_handle;
    unsigned_16         num_paras;
    unsigned_16         fname;
    unsigned_32         disk_addr;
} ovltab_entry;
```

The top bit of the **flag_anc** field contains an indicator, while the program is running, of whether the overlay section is in memory (value one) or must be loaded from disk (value zero). The next highest bit is filled in by the linker and informs the overlay manager that the section must be loaded during the overlay manager initialization. The remaining bits contain the overlay number for the ancestor of this section (zero if there is none). The **relocs** field say how many segment relocation items there are for this section, while the **start_para** field gives the location in memory (relative to the start of the program) that the section should be placed when loaded. The **num_paras** field contains the size of the section in paragraphs, and the **code_handle** field is used for various purposes inside the dynamic overlay loader. The **fname** field has the offset of the address of a zero terminated string for the name of the file containing the overlay section data and relocations (The segment value is the same as the overlay table). If the top bit of the offset is on, then the file is the original EXE file rather than a separate overlay file, and the overlay manager should use the program file name obtained from DOS (if the version is 3.0. or greater). The **disk_addr** field gives the starting offset the overlay data in the overlay file. The segment relocation items immediately follow the data.

The end of the **entries** array is indicated when an element's **flags_anc** field contains the value 0xffff. The remaining fields in that element contain garbage values.