WATCOM Debugger Trap File Interface

VERSION 17.0

July 3, 1994

Table of Contents

Table of Contents

Introduction

The WATCOM debugger consists of a number of separate pieces of code. The main executable, WD.EXE, 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. This design has two main 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. 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 contraints or other considerations which prevent the debugger proper from running on the remote system (such as Novell Netware 386).

This document describes the interface used by version 4.0 of the WATCOM debugger (shipped with the 10.0 C/C++ and FORTRAN 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 Some Definitions

1.1 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.

1.2 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.

The Request Interface

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:

```
typedef struct {
    void FAR            *ptr;
    unsigned_16         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 The Interface Routines

The trap file interface must provide three routines: `TrapInit`, `TrapRequest`, and `TrapFini`. These routines are invoked with standard WATCOM register calling conventions. How the debugger determines the address of these routines after loading a trap file is system dependent and described later.

2.1 TrapInit

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

```
trap_version FAR TrapInit( char  FAR    *parm,
                           char  FAR    *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 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 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:

```
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 17, the current minor version is 0. 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 TrapRequest

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

```
unsigned 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`.

2.2.1 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, 2, 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.

2.3 TrapFini

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

```
void FAR TrapFini(void);
```

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

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.

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.

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 17, the current minor version is 0. The `remote` field informs the trap file whether a remote server is between the 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.

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
```

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 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
```

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
```

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
unsigned_32     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.

NOTE: 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.

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.

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;
```

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

```
CPU_86  =  0
CPU_186 =  1
CPU_286 =  2
CPU_386 =  3
CPU_486 =  4
CPU_586 =  5
```

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

```
FPU_EMU = -1     - Software emulated FPU
FPU_NO  =  0     - No FPU
FPU_87  =  1
FPU_287 =  2
FPU_387 =  3
```

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
OS_OS2       =  2
OS_PHAR      =  3   - Phar Lap 386 DOS Extender
OS_ECLIPSE   =  4   - Eclipse 386 DOS Extender
OS_NW386     =  5   - NetWare 386
OS_QNX       =  6
OS_RATIONAL  =  7   - DOS/4G
OS_WINDOWS   =  8
OS_PENPOINT  =  9
OS_NT        = 10
OS_AUTOCAD   = 11   - ADS/ADI development
```

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.

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
unsigned_32     handle;
```

The `req` field contains the request. The `in_addr` tells the address to map. The `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
unsigned_32     lo_bound;
unsigned_32     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.

1.9 REQ_ADDR_INFO (8)

Request to check if a given address is using 32-bit addressing (the selector's B-bit is on) by default. The debugger requires this information to properly disassemble instructions.

Request message:

```
access_req      req
addr48_ptr      in_addr
```

The `req` field contains the request and the `in_addr` tells the input address.

Return message:

```
unsigned_8      is_32
```

The field returns one if the address is a USE32 segment, zero otherwise.

1.10 REQ_CHECKSUM_MEM (9)

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`.

1.11 REQ_READ_MEM (10)

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.

1.12 REQ_WRITE_MEM (11)

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.

1.13 REQ_READ_IO (12)

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.

1.14 REQ_WRITE_IO (13)

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.

1.15 REQ_READ_CPU (14)

Request to read the CPU registers.

Request message:

```
access_req      req
```

Return message:

```
bytes           data
```

The `data` field contains the register information requested. It contains the following structure:

```
struct cpu_regs {
    unsigned_32 EAX;
    unsigned_32 EBX;
    unsigned_32 ECX;
    unsigned_32 EDX;
    unsigned_32 ESI;
    unsigned_32 EDI;
    unsigned_32 EBP;
    unsigned_32 ESP;
    unsigned_32 EIP;
    unsigned_32 EFL;
    unsigned_32 CR0;
    unsigned_32 CR2;
    unsigned_32 CR3;
    unsigned_16 DS;
    unsigned_16 ES;
    unsigned_16 SS;
    unsigned_16 CS;
    unsigned_16 FS;
    unsigned_16 GS;
};
```

1.16 REQ_READ_FPU (15)

Request to read the FPU registers.

Request message:

```
access_req      req
```

Return message:

```
bytes           data
```

The `data` field contains the register information requested. Its format is the same as the result of a "fsave" instruction in a 32-bit segment (the instruction pointer and operand pointer fields take up 8 bytes each). Implementations of trap files in 16-bit environments should expand the instruction pointer and operand pointer fields from 4 bytes to 8 (shuffling the data register fields down in memory) before returning the result to the debugger.

1.17 REQ_WRITE_CPU (16)/REQ_WRITE_FPU (17)

Requests to write to the CPU or FPU state.

Request message:

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

Information in `data` field will be transfered to the debuggee's registers. The formats of data can be found in REQ_READ_CPU/REQ_READ_FPU

NOTE: For the REQ_WRITE_FPU case, the data will be in a 32-bit "fsave" instruction format, so 16-bit environments will have to squish the instruction and operand pointer fields back to their 4 byte forms.

Return message:

```
NONE
```

1.18 REQ_PROG_GO (18)/REQ_PROG_STEP (19)

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 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 14-15  :  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.

1.19 REQ_PROG_LOAD (20)

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
unsigned_32     task_id
unsigned_32     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_32             - 32-bit program
Bit 1      :  LD_FLAG_IS_PROT           - Protected mode
Bit 2      :  LD_FLAG_IS_STARTED        - Program already started
Bit 3      :  LD_FLAG_IGNORE_SEGMENTS   - Ignore segments (flat)
Bit 4 - 7  :  not used
```

1.20 REQ_PROG_KILL (21)

Request to kill the program.

Request message:

```
access_req      req
unsigned_32     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.

1.21 REQ_SET_WATCH (22)

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.

1.22 REQ_CLEAR_WATCH (23)

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
```

1.23 REQ_SET_BREAK (24)

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`.

1.24 REQ_CLEAR_BREAK (25)

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
```

1.25 REQ_GET_NEXT_ALIAS (26)

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.

1.26 REQ_SET_USER_SCREEN (27)

Request to make the debuggee’s screen visible.

Request message:

```
access_req      req
```

Return message:

```
NONE
```

1.27 REQ_SET_DEBUG_SCREEN (28)

Request to make the debugger’s screen visible.

Request message:

```
access_req      req
```

Return message:

```
NONE
```

1.28 REQ_READ_USER_KEYBOARD (29)

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.

1.29 REQ_GET_LIB_NAME (30)

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

Request message:

```
access_req      req
unsigned_32     handle
```

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

Return message:

```
unsigned_32     handle
--------------------------
string          name
```

The `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.

1.30 REQ_GET_ERR_TEXT (31)

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.

1.31 REQ_GET_MESSAGE_TEXT (32)

Request to get generic message text. After a REQ_PROG_LOAD, or REQ_PROG_GO/REQ_PROG_STEP has returned with COND_MESSAGE/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.

1.32 REQ_REDIRECT_STDIN (33)/REQ_REDIRECT_STDOUT (34)

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.

1.33 REQ_SPLIT_CMD (35)

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.

2 File I/O requests

This section describes requests that deal with file input/output on the 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 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 an new basic type in addition to the ones already described:

Type	*Definition*
trap_fhandle	This is an unsigned_32 which holds a debuggee file handle.

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.

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      :  IO_READ
Bit 1      :  IO_WRITE
Bit 2      :  IO_CREATE
Bit 3 - 7  :  not used
```

For read/write mode, turn both `IO_READ` and `IO_WRITE` bits on.

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.

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:

```
DBG_SEEK_ORG = 0  - Relative to the start of file
DBG_SEEK_CUR = 1  - Relative to the current file position
DBG_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.

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.

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.

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.

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.

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.

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:

```
FILE_EXE  =  0
FILE_DBG  =  1
FILE_PRS  =  2
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.

2.10 REQ_FILE_RUN_CMD (9)

Request to run a command on the 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 Overlay requests

This section describes requests that deal with overlays (currently 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 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 an 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.

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.

3.2 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 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.

3.4 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
```

3.5 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 `ovl_addr` field contains the overlay address. The `ovl_addr` field is defined as follows:

```
typedef struct {
    addr32_ptr  mach;
    unsigned_16 sect_id;
}
```

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 `section_id` field will be zero.

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 `section_id` field in the structure `ovl_addr` will be zero.

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.

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 do not show that "prefix" to the request messages.

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

4.1 REQ_THREAD_GET_NEXT (0)

Request to get next thread.

Request message:

```
access_req      req
unsigned_32     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:

```
unsigned_32     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
```

4.2 REQ_THREAD_SET (1)

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

Request message:

```
access_req      req
unsigned_32     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
unsigned_32     old_thread
```

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

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
unsigned_32     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.

4.4 REQ_THREAD_THAW (3)

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

Request message:

```
access_req      req
unsigned_32     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.

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.

5 RFX requests

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

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

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.

5.2 REQ_RFX_MKDIR (1)

Request to create a directory on the debuggee's 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.

5.3 REQ_RFX_RMDIR (2)

Request to remove a directory on the debuggee's 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.

5.4 REQ_RFX_SETDRIVE (3)

Request to set the current drive on the debuggee's system.

Request message:

```
access_req      req
unsigned_8      drive
```

The `drive` field contains the drive number to be set on the debuggee's system.

Return message:

```
trap_error      err
```

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

5.5 REQ_RFX_GETDRIVE (4)

Request to get the current drive on the debuggee's 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 debuggee's system.

5.6 REQ_RFX_SETCWD (5)

Request to set a directory on the debuggee's 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.

5.7 REQ_RFX_GETCWD (6)

Request to get the current directory name on the debuggee's system.

Request message:

```
access_req      req
unsigned_8      drive
```

The `drive` field contains the target drive number.

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.

5.8 REQ_RFX_SETDATETIME (7)

Request to set the date and time information on the debuggee's 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
```

5.9 REQ_RFX_GETDATETIME (8)

Request to get the date and time information on the debuggee's 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).

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.

Return message:

```
unsigned_32     size
```

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

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.

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.

5.13 REQ_RFX_NAMETOCANNONICAL (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.

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
--------------------------
dta             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 `dta` is as follows:

```
typedef struct dta {
    struct {
        char                i_dunno[13];
        unsigned int        dir_entry_num;
        unsigned int        cluster;
        char                i_still_dunno[4];
    } dos;
    char                attr;
    unsigned int        time;
    unsigned int        date;
    long                size;
    char                name[14];
} dta;
```

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
-------------------
dta             info
```

The `req` field contains the request. The `info` field contains the dta returned from the previous REQ_FIND_NEXT or REQ_FIND_FIRST.

Return message:

```
trap_error      err
--------------------------
dta             info
```

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

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.

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 WATCOM debugger performs these operations for the various systems.

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.

2 Trap Files Under OS/2

A trap file is a normal OS/2 1.x DLL. The system automatically searches the directories specified by the LIBPATH command in the CONFIG.SYS file. Once loaded, the 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" );
}
```

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 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 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 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" );
}
```

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.

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.