A GDB Server for the Win32 API

Martin Ruckert
Munich University of Applied Sciences
Department of Mathematics and Computer Science

Version 1.0β DRAFT

1. Table of Contents

2. Introduction

The gdbserver is a program to interface `gdb` with a running program, possibly on a different host, over a standard interface. Since different hosts provide different low level facilities the program is split into two parts, the host specific part and the `gdb` specific part. Since this `gdbserver` is intended to be run under the Windows operating system, the `gdb` specific part is subdivided again into an operating system independent part and a Windows specific part.

From the outset, the `gdbserver` program is just a big C program and it consists of

⟨ include files 26 ⟩ (1)
⟨ function prototypes 7 ⟩
⟨ global variables 13 ⟩

and

⟨ functions 30 ⟩ (2)

Last not least, we have the *main* program

int *main*(**int** *argc*, **char** ∗*argv*[ ]){ (3)

where we

⟨ define local variables 9 ⟩ (4)
⟨ set an error exit point 143 ⟩
⟨ initialize variables 14 ⟩
⟨ start the target program 64 ⟩
⟨ open a connection to gdb 133 ⟩
⟨ set an error reentry point 142 ⟩
⟨ receive and dispatch messages 8 ⟩

and finally

⟨ close the connection to gdb 135 ⟩; (5)

We conclude the *main* program with a successful

return 0; } (6)

The most interesting part of it all is how we ⟨ receive and dispatch messages 8 ⟩. That is, how commands and answers are exchanged between `gdb` and gdbserver. This, we will investigate in the next section. Then follows a discussion about the Win32 API and how it is used to implement the access functions for the running target program. At the end, we describe how messages are properly packaged, and all the necessary details on how to establish a TCP/IP connection with `gdb`, and send the packaged data. We conclude this paper with a section on messages and error reporting, and provide several indices on content as well as on programming details.

3. The gdb Remote Protocol

Messages are transmitted in packets. Two functions handle the low level details:

⟨ function prototypes 7 ⟩ ≡ (7)

```
int getpkt(char *buffer);
int putpkt(char *buffer);
```

Used in 1.

both functions return the number of characters successfully transmitted or a negative value in case of error.

Using these functions, we can formulate a loop to

⟨ receive and dispatch messages 8 ⟩ ≡ (8)

```
target_wait();
while (getpkt(buffer) > 0) {
  ⟨ dispatch a message 10 ⟩
  putpkt(buffer);
  ⟨ check for end of target process 32 ⟩
}
```

Used in 4.

The buffer that we use for exchanging messages, as well as its size, is a local variable

⟨ define local variables 9 ⟩ ≡ (9)

```
char *buffer;
unsigned int buffer_size;
```

Used in 4.

To allocate the message *buffer*, we should know how big a buffer is needed to cover all cases. This is hardly possible. In any case, we can make it large, lets say 1028 characters, and big enough to hold all registers (see the G and g command below).

After these preparations we can discuss the messages in Detail. Most command messages are identified by their first character and we use a switch accordingly.

⟨ dispatch a message 10 ⟩ ≡ (10)

```
switch (buffer[0]) {
```

Used in 8.

This will pick one of the following cases. Any gdb server is required to support the g, G, m, M, c, and s commands all other commands are optional.

3.1. Read Registers

The command g, requests the transmission of all the registers. The command consists just of a single 'g'. The program that is currently being debugged, logically consists of several processes, and each process may have several threads. The distinction here is, that each thread maintains its own execution state, including a set of registers, but all threads of one process share the same memory. Hence, we have to select the desired thread first, then we answer the registers. This implementation of gdbserver is restricted to debugging a single thread, and therefore we skip this step.

⟨ dispatch a message 10 ⟩ +≡ (11)

```
case 'g': ⟨ answer registers 12 ⟩
  break;
```

The answer consists of a lengthy string of hexadecimal digits, where each byte of register data is coded by two digits. The registers, the size of the registers, and the byte order (big endian or little endian) is determined by the target architecture. gdb itself has two internal macros, REGISTER_RAW_SIZE and REGISTER_NAME that contain the required information.

⟨ answer registers 12 ⟩ ≡ (12)

```
{
  int n;
  char *p = buffer;
  for (n = 0; n < target_registers; n++) {
    hexfrombin(p, target_register_value(n), target_register_size(n));
    p = p + 2 * target_register_size(n);
```

```
    }
    *p = 0;
  }
```
Used in 11.

We used three target dependent entities:

- *target_registers*, the number of registers available,
- *target_register_size*, a function that returns the size of the register in byte, and
- *target_register_value*, a function returning a pointer to a place in memory, from where the register value can be read in target byte order.

It is crucial that our buffer is large enough. For this we have a

⟨global variables 13⟩ ≡ (13)

```
  static unsigned int registerpkg_size;
```
Used in 1.

Which is set when we

⟨initialize variables 14⟩ ≡ (14)

```
  {
    int n;
    registerpkg_size = 0;
    for (n = 0; n < target_registers; n++) registerpkg_size = registerpkg_size + 2 * target_register_size(n);
  }
```
Used in 4.

After that, we can finally allocate the message *buffer*.

⟨initialize variables 14⟩ +≡ (15)

```
  if (registerpkg_size < 1024) buffer_size = 1024;
  else buffer_size = registerpkg_size + 1;                 /* for the trailing zero byte */
  buffer = malloc(buffer_size);
  if (buffer ≡ NULL) fatal_error("Out␣of␣memory");
```

3.2. Write Registers

indexwrite registers The command `G`, requests the setting of all the registers. Again we select the thread, before we extract the registers from the buffer, and conclude with providing an answer to `gdb`.

⟨dispatch a message 10⟩ +≡ (16)

```
case 'G': ⟨obtain registers 17⟩answer_ok(buffer);
  break;
```

The command encodes after the leading letter 'G' all the registers exactly in the same format as in the 'g' command discussed before. Hence, we have:

⟨obtain registers 17⟩ ≡ (17)

```
  {
    int i;
    char *p = buffer;
    for (i = 0; i < target_registers; i++) {
      binfromhex(target_register_address(i), target_register_size(i), p);
      p = p + 2 * target_register_size(i);
    }
    if (strlen(buffer) ≠ registerpkg_size) message("register␣package␣has␣wrong␣length");
  }
```
Used in 16.

The function *target_register_address* is similar to *target_register_value*. Both deliver an address in memory that is associated with the given register. Where as *target_register_value* assumes that your are going to read the memory at that address, the function *target_register_address* assumes you are writing this address.

3.3. Write Single Register

The command `P`, is used by **gdb** to set a single register. The usual answer is 'OK'.

```
⟨dispatch a message 10⟩ +≡                                                (18)
case 'P':
  {
    int n;
    ⟨obtain register number n 19⟩
    ⟨obtain one register value 20⟩
  }
  answer_ok(buffer);
  break;
```

In this command, the 'P' is followed by the hex encoded register number n,

```
⟨obtain register number n 19⟩ ≡                                          (19)
  n = intfromhex(buffer + 1);
  if (n > target_registers) {
    message("wrong␣register␣number");
    answer_error(buffer, 1);
    break;
  }                                                       Used in 18 and 21.
```

After the register number follows an equal sign followed by the hex encoded value of the register (in target byte order).

```
⟨obtain one register value 20⟩ ≡                                          (20)
  binfromhex(target_register_address(n), target_register_size(n), strchr(buffer + 1, '=') + 1);     Used in 18.
```

3.4. Read Single Register

The command `p`, is used by **gdb** to obtain a single Register. The format of the 'p' command is as one should expect from the previous commands: The letter 'p' is followed by the register number, and the answer is the register value hex encoded in target byte order.

```
⟨dispatch a message 10⟩ +≡                                                (21)
case 'p':
  {
    int n;
    ⟨obtain register number n 19⟩
    ⟨answer register n 22⟩
  }
  break;
```

To answer, we just pack the register value into the buffer and terminate with a zero byte.

```
⟨answer register n 22⟩ ≡                                                  (22)
  hexfrombin(buffer, target_register_value(n), target_register_size(n));
  buffer[target_register_size(n) * 2] = 0;                        Used in 21.
```

3.5. Read Memory

The command `m`, is used by **gdb** to inspect memory locations. It provides an address a and the number n of bytes desired. Then we allocate space for a copy of the requested target memory. A target specific function *target_get_memory*, will then actually read the memory and return the number of bytes read. From this byte string, we construct an answer.

```
⟨dispatch a message 10⟩ +≡                                                (23)
case 'm':
  {
    unsigned int a;
```

```
    unsigned int l;
    unsigned char *m;
    ⟨ get address a and length l 24 ⟩
    m = malloc(l);
    if (m ≡ NULL)
      error ("out␣of␣memory") ;
    l = target_get_memory(m, a, l);
    ⟨ answer memory string 25 ⟩
    free(m);
  }
  break;
```

In the 'm' command, the letter 'm' is followed by the address coded in hex, then a comma, and then the number of bytes needed. We trim the number of bytes requested down to a size that our buffer can handle. This is ok, since this command may return fewer bytes than requested anyway.

⟨ get address a and length l 24 ⟩ ≡ (24)

```
  a = intfromhex(buffer + 1);
  l = intfromhex(strchr(buffer + 1, ',') + 1);
  if (l ≥ buffer_size/2) l = (buffer_size/2) − 1;
```

Used in 23 and 27.

The answer is easily obtained the usual way:

⟨ answer memory string 25 ⟩ ≡ (25)

```
  hexfrombin(buffer, m, l);
```

Used in 23.

For the function *strchr* we need

⟨ include files 26 ⟩ ≡ (26)

```
#include <string.h>
```

Used in 1.

3.6. Write Memory

The command `M`, is used by `gdb` to write into memory. After the command character 'M', the address a and the length l are encoded in hex, separated by a comma. Then follows a colon. After the colon, the memory content is coded as usual in hex and target byte order.

⟨ dispatch a message 10 ⟩ +≡ (27)

```
case 'M':
  {
    unsigned int a;
    unsigned int l;
    unsigned char *m;
    ⟨ get address a and length l 24 ⟩
    m = malloc(l);
    if (m ≡ NULL)
      error ("out␣of␣memory") ;
    binfromhex(m, l, strchr(buffer + 1, ':') + 1);
    if (target_put_memory(m, a, l) ≡ l) answer_ok(buffer);
    else answer_error(buffer, 1);
    free(m);
  }
  break;
```

3.7. Continue Execution

The command `c`, is used by `gdb` to continue execution of the current thread. The 'c' might be followed by an optional address, which we read and pass on to a low level function *target_continue*, the we wait until the target stops again an report the cause and the circumstances of this stop to the waiting debugger.

```
⟨ dispatch a message 10 ⟩ +≡                                              (28)
case 'c':
  {
    unsigned int a;
    ⟨ get optional address a 29 ⟩target_continue( );
    target_wait( );
    answer_stopped(buffer);
  }
  break;
```

To test for the presence for an address, we just consider the byte following the 'c', and the read the hex coded address.

```
⟨ get optional address a 29 ⟩ ≡                                           (29)
  if (buffer[1] ≠ 0) a = intfromhex(buffer + 1);
  else a = 0;                                                Used in 28 and 35.
```

The reporting of a stopped thread may take several forms. In the simplest case, we answer an 'S' followed by the signal number (coded in hex); in the more elaborate case, we provide some information on certain registers which we assume the debugger will need anyway, as for instance the program counter, the stack pointer and the frame pointer. In this case, the answer consists of a 'T' followed by the signal number, followed by a sequence of register specifications.

While we are at it, two more answers are possible as a response to a stopped thread: It the thread exited we answer a 'W' followed by the exit code; it the thread was killed by a signal, we answer 'X' followed by the signal number.

```
⟨ functions 30 ⟩ ≡                                                         (30)
  void answer_stopped(char *buffer)
  {
    int status;
    int sig;
    status = target_status( );
    sig = target_signal( );
    if (status ≡ EXITED) {
      buffer[0] = 'W', hexfromint(buffer + 1, sig);
    }
    else if (status ≡ KILLED) {
      buffer[0] = 'X', hexfromint(buffer + 1, sig);
    }
    else {
      int l;
      buffer[0] = 'T';
      l = hexfromint(buffer + 1, sig);
      answer_registers(buffer + 1 + l);
    }
  }                                                               Used in 2.
```

We used the function

```
⟨ function prototypes 7 ⟩ +≡                                               (31)
  extern void answer_registers(char *buffer);
  extern int hexfromint(char *to, unsigned int from);
```

We use the return value of *target_status* also to

⟨ check for end of target process 32 ⟩ ≡ (32)

```
{
  int status;
  status = target_status();
  if (status ≡ EXITED) {
    message("Target␣process␣exited");
    break;
  }
  else if (status ≡ KILLED) {
    message("Target␣process␣was␣killed");
    break;
  }
}
```

Used in 8.

The **break** here will cause the message processing loop to exit.

Each register specification consists of the register number, a colon, the register value, and a semicolon. The registers that gdb needs are target dependent. So we use the function *target_expedite* to give us a pointer to an array of register numbers, that must end in a negative value. All these values are then packed in the return packet.

⟨ functions 30 ⟩ +≡ (33)

```
void answer_registers(char *buffer)
{
  int *n;
  n = target_expedite();
  while (*n ≥ 0) {
    int l;
    l = hexfromint(buffer, *n);
    buffer = buffer + l;
    buffer[0] = ':';
    buffer++;
    l = hexfrombin(buffer, target_register_value(*n), target_register_size(*n));
    buffer = buffer + l;
    buffer[0] = ';';
    buffer++;
    n++;
  }
  buffer[0] = 0;
}
```

We used the function

⟨ function prototypes 7 ⟩ +≡ (34)

```
extern int hexfrombin(char *to, char *from, int fromsize);
```

3.8. Single Step Execution

The command s, is used by gdb to continue execution of the current thread by only a single step. Its format, as well as the reply is analogous to the continue command.

⟨ dispatch a message 10 ⟩ +≡ (35)

```
case 's':
  {
    unsigned int a;
    ⟨ get optional address a 29 ⟩
```

```
    target_step();
    target_wait();
    answer_stopped(buffer);
  }
  break;
```

3.9. Query Last Signal

The command `?`, is used by `gdb` to inquire about the last signal that was received by the process. The reply is again the same as for the step or continue command.

⟨dispatch a message 10⟩ +≡ (36)

```
case '?': answer_stopped(buffer);
  break;
```

3.10. Kill Process

The command `k`, is used by `gdb` to kill the current thread. We kill the `gdbserver` too (not perfect).

⟨dispatch a message 10⟩ +≡ (37)

```
case 'k': target_kill();
  exit(0);
```

3.11. Enable extended Mode

The command `!`, is used by `gdb` to switch the `gdbserver` into extended mode. Which, by now, is not supported.

⟨dispatch a message 10⟩ +≡ (38)

```
case '!': answer_nothing(buffer);
  break;
```

3.12. Detach from Remote System

The command `D`, is send by `gdb` if it detaches from the server. It is not yet supported. There is no answer to this command.

⟨dispatch a message 10⟩ +≡ (39)

```
case 'D': answer_nothing(buffer);
  break;
```

3.13. Other Commands

Not yet implemented are the commands: d, C, A, i, q, Q, S, T, X, z, and Z. Some commands probably never get implemented: b, B, r, and t.

A command that is not implemented will end up in the **default** section of the switch.

⟨dispatch a message 10⟩ +≡ (40)

```
default: answer_nothing(buffer);
  break; }
```

which concludes the description of how we ⟨dispatch a message 10⟩.

4. The Win32 Target Architecture

The target part of `gdbserver` must be rewritten for each and every target. Therefore it is of special importance to keep this part as short as possible and to give an exact specification of all and everything that must be in this section.

To simplify things, this section produces a header file called "target.h" that will contain declarations of all functions, variables, types, etc. that are provided by the rest of gdbserver to be used by the target section, we call this ⟨target imports 60⟩, and all the stuff that is provided by the target section to be used by the gdbserver, we call this ⟨target exports 44⟩.

⟨`target.h` 41⟩ ≡ (41)

```
  ⟨target imports 60⟩
  ⟨target exports 44⟩
```

This header file is then included into the `gdbserver` file

⟨include files 26⟩ +≡ (42)

```
#include "target.h"
```

Second, this section produces the file "target.c" which contains the implementation of all the ⟨target functions 45⟩. Again, at the very beginning, we include the target header file.

⟨`target.c` 43⟩ ≡ (43)

```
#include <windows.h>
#include "target.h"
  ⟨private target types 53⟩⟨private target variables 56⟩
  ⟨target functions 45⟩
```

The target.c file is compiled separately and linked with the rest to form a complete gdbserver.

As an important simplification, each instance of this `gdbserver` will handle only one type of target. There are no provisions to switch to a different target at run time, something the old `gdbserver` could do in principle, but rarely does. Further, in this section the implementation of the target functions assumes that the target program is running on an Intel processor under the windows operating system, and we use the native win32 API.

After these preliminaries, let us jump to the core function: *target_wait*, which waits for the running target program to stop.

⟨target exports 44⟩ ≡ (44)

```
  extern void target_wait(void);
```

Used in 41.

4.1. Waiting for an Event

To implement this, the Win32 API has the function *WaitForDebugEvent* which needs the address of a `DEBUG_EVENT` structure and a timeout. We choose `INFINITE` for the timeout, which does what it says. After calling this function, we will ⟨process the *event* 47⟩. We enclose the whole thing in an infinite loop. This way, the processing can determine whether to return from the function and notify the `gdbserver` about this event, or ignore, or otherwise handle the event, stay in the loop, and wait for the next event to occur.

⟨target functions 45⟩ ≡ (45)

```
#include <stdio.h>
  void target_wait(void)
  {
    static DEBUG_EVENT event;
    while (1) {
      if (¬WaitForDebugEvent(&event, INFINITE))
        error ("WaitForDebugEvent␣timed␣out") ;
      fprintf(stderr, "Event:␣%d␣\t␣Thread␣Id:␣%x\n", event.dwDebugEventCode, event.dwThreadId);
      ⟨process the event 47⟩
    }
  }
```

Used in 43.

After we have waited for the event, the *event* structure is filled with information about the event. For instance it contains the *dwThreadId* identifying the thread that had the event. Under Windows, when a program is started, it will create all kinds of threads to manage for instance the user interface. Since we are not interested in these events, we start out processing with checking the *dwThreadId* against the targets *id* which belongs to the

⟨ thread information 46 ⟩ ≡ (46)

```
DWORD thread_id;
```

Used in 53.

stored in the structure *t*. If it does not match, we ⟨ ignore the *event* 48 ⟩.

⟨ process the *event* 47 ⟩ ≡ (47)

```
if (event.dwThreadId ≠ t.thread_id ∨ event.dwProcessId ≠ t.process_id) {
  fprintf(stderr, "Event␣has␣wrong␣ids\n");
  ⟨ ignore the event 48 ⟩
}
```

Used in 45.

To ignore an event, we use the *ContinueDebugEvent* function. It needs a Process Id and a Thread Id, both of which, we can take from the *event* structure and a parameter to determine the kind of continuation. Since we are still interested in receiving further events, we use the value `DBG_CONTINUE`. After the thread has gained the permission to continue, the *target_wait* function will **continue** as well in its processing loop.

⟨ ignore the *event* 48 ⟩ ≡ (48)

```
{
  fprintf(stderr, "Continue␣%x␣%x␣%x\n", event.dwProcessId, event.dwThreadId, DBG_CONTINUE);
  ContinueDebugEvent(event.dwProcessId, event.dwThreadId, DBG_CONTINUE);
  continue;
}
```

Used in 47, 50, 51, and 63.

The *event* structure provides a clue of the kind of event that occurred with an *dwDebugEventCode*. Hence, we start a switch and consider all the cases separately.

⟨ process the *event* 47 ⟩ +≡ (49)

```
switch (event.dwDebugEventCode) {
```

Now to the different cases.

Most of them, we simply ignore.

⟨ process the *event* 47 ⟩ +≡ (50)

```
case UNLOAD_DLL_DEBUG_EVENT: case OUTPUT_DEBUG_STRING_EVENT: ⟨ ignore the event 48 ⟩
```

Others need some extra processing, like closing handles, before we finally ignore them.

⟨ process the *event* 47 ⟩ +≡ (51)

```
case CREATE_PROCESS_DEBUG_EVENT: CloseHandle(event.u.CreateProcessInfo.hFile);
  CloseHandle(event.u.CreateProcessInfo.hProcess);
  CloseHandle(event.u.CreateProcessInfo.hThread);
⟨ ignore the event 48 ⟩case LOAD_DLL_DEBUG_EVENT:
  CloseHandle(event.u.LoadDll.hFile); ⟨ ignore the event 48 ⟩
```

If the thread we are debugging is exiting, the debugger needs notification. We return.

⟨ process the *event* 47 ⟩ +≡ (52)

```
case EXIT_THREAD_DEBUG_EVENT: ⟨ invalidate the cache 77 ⟩
  t.status = EXITED;
  t.signal = event.u.ExitThread.dwExitCode; ⟨ set resume_mode 92 ⟩return;
case EXIT_PROCESS_DEBUG_EVENT: ⟨ invalidate the cache 77 ⟩
  t.status = EXITED;
  t.signal = event.u.ExitProcess.dwExitCode; ⟨ set resume_mode 92 ⟩return;
```

But before we do so, we record some information about the process. All information about a thread is stored in a *thread_info* structure.

⟨private target types 53⟩ ≡ (53)

```
typedef struct thread_info {
    ⟨thread information 46⟩
} thread_info;
```

Used in 43.

Two of the information items, which we used above are the *status* and the *signal*

⟨thread information 46⟩ +≡ (54)

```
int status;
int signal;
```

The valid values for the *status* are

⟨target exports 44⟩ +≡ (55)

```
#define RUNNING 0
#define EXITED 1
#define KILLED 2
#define STOPPED 3
```

In *signal* we store exit codes and signals received. *exception* is a flag indicating that an EXCEPTION_DEBUG_EVENT has occurred (see below). Since in this implementation only a single thread is considered, we use only a single **thread_info** variable *t*.

⟨private target variables 56⟩ ≡ (56)

```
static thread_info t;
```

Used in 43.

All access to the target is done by a functional interface. Hence, for variables like *status* and *signal* there are functions to inspect them.

⟨target exports 44⟩ +≡ (57)

```
extern int target_signal(void);
extern int target_status(void);
```

These functions just return the appropriate value.

⟨target functions 45⟩ +≡ (58)

```
int target_signal(void)
{
    return t.signal;
}
int target_status(void)
{
    return t.status;
}
```

Very similar is the processing, when the tread is created.

⟨process the *event* 47⟩ +≡ (59)

```
case CREATE_THREAD_DEBUG_EVENT: ⟨invalidate the cache 77⟩
    t.status = STOPPED;
    t.signal = TARGET_SIGNAL_TRAP; ⟨set resume_mode 92⟩return;
```

The value TARGET_SIGNAL_TRAP comes from the file signals.h, which is part of gdb and is included as part of the

⟨target imports 60⟩ ≡ (60)

```
#include "signals.h"
```

Used in 41.

The most common event is the EXCEPTION_DEBUG_EVENT.

⟨process the *event* 47⟩ +≡ (61)

```
case EXCEPTION_DEBUG_EVENT: ⟨invalidate the cache 77⟩
    ⟨convert win32 signal to gdb signal 62⟩
    t.status = STOPPED; ⟨set resume_mode 92⟩return;
```

The signal that caused the thread to stop is part of the *event* structure. `gdb`, however, has its own Unix oriented notion of signals and we have to find for each win32 signal the a corresponding `gdb` signal.

⟨convert win32 signal to `gdb` signal 62⟩ ≡ (62)

```
switch (event.u.Exception.ExceptionRecord.ExceptionCode) {
case EXCEPTION_ACCESS_VIOLATION: t.signal = TARGET_SIGNAL_SEGV;
  break;
case STATUS_STACK_OVERFLOW: t.signal = TARGET_SIGNAL_SEGV;
  break;
case STATUS_FLOAT_DENORMAL_OPERAND: t.signal = TARGET_SIGNAL_FPE;
  break;
case EXCEPTION_ARRAY_BOUNDS_EXCEEDED: t.signal = TARGET_SIGNAL_FPE;
  break;
case STATUS_FLOAT_INEXACT_RESULT: t.signal = TARGET_SIGNAL_FPE;
  break;
case STATUS_FLOAT_INVALID_OPERATION: t.signal = TARGET_SIGNAL_FPE;
  break;
case STATUS_FLOAT_OVERFLOW: t.signal = TARGET_SIGNAL_FPE;
  break;
case STATUS_FLOAT_STACK_CHECK: t.signal = TARGET_SIGNAL_FPE;
  break;
case STATUS_FLOAT_UNDERFLOW: t.signal = TARGET_SIGNAL_FPE;
  break;
case STATUS_FLOAT_DIVIDE_BY_ZERO: t.signal = TARGET_SIGNAL_FPE;
  break;
case STATUS_INTEGER_DIVIDE_BY_ZERO: t.signal = TARGET_SIGNAL_FPE;
  break;
case STATUS_INTEGER_OVERFLOW: t.signal = TARGET_SIGNAL_FPE;
  break;
case EXCEPTION_BREAKPOINT: t.signal = TARGET_SIGNAL_TRAP;
  break;
case DBG_CONTROL_C: t.signal = TARGET_SIGNAL_INT;
  break;
case DBG_CONTROL_BREAK: t.signal = TARGET_SIGNAL_INT;
  break;
case EXCEPTION_SINGLE_STEP: t.signal = TARGET_SIGNAL_TRAP;
  break;
case EXCEPTION_ILLEGAL_INSTRUCTION: t.signal = TARGET_SIGNAL_ILL;
  break;
case EXCEPTION_PRIV_INSTRUCTION: t.signal = TARGET_SIGNAL_ILL;
  break;
case EXCEPTION_NONCONTINUABLE_EXCEPTION: t.signal = TARGET_SIGNAL_ILL;
  break;
default: t.signal = TARGET_SIGNAL_UNKNOWN;
  break;
}
```

Used in 61.

Finally, we conclude the event processing with the **default** case.

⟨process the *event* 47⟩ +≡ (63)

```
default: ⟨ignore the event 48⟩}
```

4.2. Starting the Target Thread

The function *target_start* which we use to

⟨start the target program 64⟩ ≡ (64)

```
if (argc < 3) fatal_error("Use:␣\gdbserver\␣host:port␣target␣[target␣arguments]");
target_start(argc − 2, argv + 2);
```

Used in 4.

is part of the

⟨target exports 44⟩ +≡ (65)

```
extern int target_start(int argc, char ∗argv[]);
```

The function is implemented using the *CreateProcess* windows system call.

⟨target functions 45⟩ +≡ (66)

```
int target_start(int argc, char ∗argv[])
{
  BOOL ret;
  DWORD flags;
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  static char commandline[1024];
  flags = DEBUG_ONLY_THIS_PROCESS;
  flags |= DEBUG_PROCESS;
  memset(&si, 0, sizeof (si));
  si.cb = sizeof (si);
  memset(&pi, 0, sizeof (pi));
  ⟨convert argv to commandline 68⟩
  ret = CreateProcess(NULL,          /∗ Application Name ∗/
  commandline,                       /∗ command line ∗/
  NULL,                              /∗ Security ∗/
  NULL,                              /∗ thread ∗/
  TRUE,                              /∗ inherit handles ∗/
  flags,                             /∗ start flags ∗/
  NULL,                              /∗ the environment ∗/
  NULL,                              /∗ current directory ∗/
  &si, &pi);
  if (¬ret)
    error ("Error␣creating␣process") ;
  t.thread_handle = pi.hThread;
  t.process_handle = pi.hProcess;
  t.thread_id = pi.dwThreadId;
  t.process_id = pi.dwProcessId;
  ⟨invalidate the cache 77⟩return pi.dwProcessId;
}
```

Notice that we immediately after creating the process, we wait for it to stop again, a behavior, which is initiated by setting the creation flags to DEBUG_PROCESS.

We have seen already the *thread_id*, when we considered the processing of events. Here we see how the *id* gets initialized. We keep other important information about the process, the handles for process and thread as part of the

⟨thread information 46⟩ +≡ (67)

```
HANDLE thread_handle;
HANDLE process_handle;
DWORD process_id;
```

The return value is the win32 process id, that could be used by the calling gdbserver.

It remains to see how to convert the Unix style *argv* to the win32 style commandline.

⟨convert *argv* to *commandline* 68⟩ ≡ (68)

```
{
  int i, j, k;
  i = 0;
  j = 0;
  commandline[i] = 0;
  while (argv[j] ≠ NULL) {
    k = strlen(argv[j]);
    if (i + k + 1 < 1024) {
      if (i ≠ 0) commandline[i++] = '␣';
      strcpy(commandline + i, argv[j]);
      i = i + k;
      j++;
    }
    else error ("Commandline␣too␣long.") ;
  }
}
```

Used in 66.

4.3. Terminating the Target Thread

The inverse to starting the target program is terminating it. This is done with the function *target_kill*.

⟨target exports 44⟩ +≡ (69)

```
extern void target_kill(void);
```

Because handles are valuable resource under win32, we have to close all handles here.

⟨target functions 45⟩ +≡ (70)

```
void target_kill(void)
{
  TerminateProcess(t.process_handle, 1);
  CloseHandle(t.process_handle);
  CloseHandle(t.thread_handle);
}
```

Why we keep the handles in the first place? Because we need them to read or write memory (process handle) and to read or write registers (thread handle), as we will see in the next sections.

4.4. Reading Target Memory

The function *target_get_memory* is one of the

⟨target exports 44⟩ +≡ (71)

```
extern int target_get_memory(unsigned char *m, unsigned int a, unsigned int l);
```

The call to *target_get_memory*(m, a, l) will copy l bytes from address a to the buffer m, and return the number of bytes read. The implementation uses the Win32 *ReadProcessMemory* system call.

⟨target functions 45⟩ +≡ (72)

```
int target_get_memory(unsigned char *m, unsigned int a, unsigned int l)
{
  DWORD count;
  if (¬ReadProcessMemory(t.process_handle,    /* handle to the process whose memory is read */
  (LPCVOID)a,    /* address to start reading */
  m,    /* address of buffer to place read data */
  l,    /* number of bytes to read */
  &count    /* address of number of bytes read */
  ))
```

```
    error ("Unable␣to␣read␣process␣Memory") ;
  if (count ≠ l) message("Partly␣unsuccessful␣read␣process␣Memory");
  return count;
}
```

4.5. Writing Target Memory

The function *target_put_memory* is again one of the

⟨ target exports 44 ⟩ +≡ (73)

```
extern unsigned int target_put_memory(unsigned char *m, unsigned int a, unsigned int l);
```

The function will copy l bytes from buffer m to the address a and returns the number of bytes written. We use the *WriteProcessMemory* system call and must not forget to flush the instruction cache. `gdb` might be forced to set breakpoints by writing into the code segment of the running process. If however the memory location in question is already in the instruction cache. The process will not stop unless ...

⟨ target functions 45 ⟩ +≡ (74)

```
unsigned int target_put_memory(unsigned char *m, unsigned int a, unsigned int l)
{
  DWORD count;
  if (¬WriteProcessMemory(t.process_handle,    /* handle to the process whose memory is read */
  (LPVOID)a,    /* address to start writing */
  (LPVOID)m,    /* address of buffer for write data */
  l,    /* number of bytes to write */
  &count    /* address of number of bytes written */
  )) message("Unable␣to␣write␣process␣Memory");
  if (count ≠ l) message("Partly␣unsuccessful␣write␣process␣Memory");
  FlushInstructionCache(t.process_handle, (LPCVOID)a, l);
  return count;
}
```

4.6. Reading and Writing Target Registers

For reading registers, there is a the *GetThreadContext* call, which will read all registers together into a `CONTEXT` structure. Since the `gdbserver` reads registers individually using the *target_register_value* function, it is not a good idea to use *GetThreadContext* repeatedly. Instead, we maintain the `CONTEXT` structure as part of the

⟨ thread information 46 ⟩ +≡ (75)

```
CONTEXT context;
```

together with a flag to indicate that the cached information is valid.

⟨ thread information 46 ⟩ +≡ (76)

```
int context_valid;
```

We can then easily

⟨ invalidate the cache 77 ⟩ ≡ (77)

```
t.context_valid = 0;
```

Used in 52, 59, 61, and 66.

We use *GetThreadContext* to write the function *validate_cache*

⟨ target functions 45 ⟩ +≡ (78)

```
static void validate_cache(void)
{
  if (t.context_valid) return;
  memset(&(t.context), 0, sizeof (CONTEXT));
  t.context.ContextFlags = CONTEXT_FULL;
  if (¬GetThreadContext(t.thread_handle,    /* handle to thread with context */
  &(t.context)    /* address of context structure */
```

```
    ))
      error ("Unable␣to␣fetch␣registers") ;
      else {
        t.context_valid = 1;
        t.context_changed = 0;
      }
  }
```

In the last line, we see a new piece of

⟨thread information 46⟩ +≡ (79)

```
  int context_changed;
```

This tells us, whether the context was possibly changed, which makes it necessary to write the context back to the thread before the thread can continue. This is how we

⟨set a valid cache 80⟩ ≡ (80)

```
  t.context.ContextFlags = CONTEXT_FULL;
  if (¬SetThreadContext(t.thread_handle,      /* handle to thread with context */
  &(t.context)     /* address of context structure */
  ))
    error ("Unable␣to␣store␣registers") ;
    else {
      t.context_valid = 1;
      t.context_changed = 0;
    }
```

Used in 91.

4.7. Register access functions

After these preparations, we can now define the

⟨target exports 44⟩ +≡ (81)

```
  extern int target_registers;
  extern unsigned char *target_register_value(int i);
  extern unsigned char *target_register_address(int i);
  extern int target_register_size(int i);
  extern int *target_expedite(void);
```

Again we face the problem of mapping gdb's idea of a register and its register number into windows idea of registers as defined by the CONTEXT structure. To do this, we define an array, called *context_mapping*, that is ordered according to gdb's register numbers. That is, we use gdb's register numbers as index into this array. The array then gives us two things, the byte offset of the register inside theCONTEXT structure and the size of it in bytes.

⟨private target types 53⟩ +≡ (82)

```
  typedef struct {
    int offset;
    int size;
  } mapping;
```

To simplify the static initialization of the *context_mapping* variable we use a simple macro that maps the names of fields of the CONTEXT structure to the offset and the size of the field.

⟨private target types 53⟩ +≡ (83)

```
#define map(field){ ( (int) & ( ( ( CONTEXT * ) NULL ) → field ) ) , sizeof ( ( ( ( CONTEXT * ) NULL ) →
          field ) ) }
```

⟨private target variables 56⟩ +≡ (84)

```
  static mapping context_mapping[] = {map(Eax),                    /* eax */
  map(Ecx),                                                        /* ecx */
  map(Edx),                                                        /* edx */
```

```
  map(Ebx),                                                /* ebx */
  map(Esp),                                                /* esp */
  map(Ebp),                                                /* ebp */
  map(Esi),                                                /* esi */
  map(Edi),                                                /* edi */
  map(Eip),                                                /* eip */
  map(EFlags),                                             /* eflags */
  map(SegCs),                                              /* cs */
  map(SegSs),                                              /* ss */
  map(SegDs),                                              /* ds */
  map(SegEs),                                              /* es */
  map(SegFs),                                              /* fs */
  map(SegGs),                                              /* gs */
  map(FloatSave.RegisterArea[0 * 10]),                     /* st0 */
  map(FloatSave.RegisterArea[1 * 10]),                     /* st1 */
  map(FloatSave.RegisterArea[2 * 10]),                     /* st2 */
  map(FloatSave.RegisterArea[3 * 10]),                     /* st3 */
  map(FloatSave.RegisterArea[4 * 10]),                     /* st4 */
  map(FloatSave.RegisterArea[5 * 10]),                     /* st5 */
  map(FloatSave.RegisterArea[6 * 10]),                     /* st6 */
  map(FloatSave.RegisterArea[7 * 10]),                     /* st7 */
  map(FloatSave.ControlWord),                              /* fctrl */
  map(FloatSave.StatusWord),                               /* fstat */
  map(FloatSave.TagWord),                                  /* ftag */
  map(FloatSave.ErrorSelector),                            /* fiseg */
  map(FloatSave.ErrorOffset),                              /* fioff */
  map(FloatSave.DataSelector),                             /* foseg */
  map(FloatSave.DataOffset),                               /* fooff */
  map(FloatSave.ErrorSelector),        /* fop */           /* XMM0-7 */
  map(ExtendedRegisters[10 * 16]),                         /* xmm0*16] */
  map(ExtendedRegisters[11 * 16]),                         /* xmm1*16] */
  map(ExtendedRegisters[12 * 16]),                         /* xmm2*16] */
  map(ExtendedRegisters[13 * 16]),                         /* xmm3*16] */
  map(ExtendedRegisters[14 * 16]),                         /* xmm4*16] */
  map(ExtendedRegisters[15 * 16]),                         /* xmm5*16] */
  map(ExtendedRegisters[16 * 16]),                         /* xmm6*16] */
  map(ExtendedRegisters[17 * 16]),     /* xmm7*16] */      /* MXCSR */
  map(ExtendedRegisters[24])                               /* mxcsr */
  };
```

Given this array the following is pretty easy. The number of target registers can be computed from the size of the previous array:

⟨target functions 45⟩ +≡ (85)

```
  int target_registers = sizeof (context_mapping)/sizeof(mapping);
```

A pointer to the location where the value of register i can be found as required by the function *target_register_value* can be obtained by:

⟨target functions 45⟩ +≡ (86)

```
  unsigned char *target_register_value(int i)
  {
    validate_cache();
    return ((unsigned char *) &(t.context)) + context_mapping[i].offset;
  }
```

A second function is provided, called *target_register_address*, which returns the same result as *target_register_value* but assumes that the pointer value returned is used for writing into the `CONTEXT` structure. It sets the *context_changed* flag accordingly.

⟨target functions 45⟩ +≡ (87)

```
unsigned char *target_register_address(int i)
{
  unsigned char *p;
  p = target_register_value(i);
  t.context_changed = 1;
  return p;
}
```

Next, one of the most simple

⟨target functions 45⟩ +≡ (88)

```
int target_register_size(int i)
{
  return context_mapping[i].size;
}
```

To conclude this section, we consider a mechanism to send some register values to gdb even before it asks for it. It is called to expedite registers. These are registers that `gdb` will need in any case after a program stops. The function *target_expedite* will return a pointer to an array of register numbers, that must end in a negative value. All these values are then packed in the return packet. For the Intel x86 processor the registers are stack pointer (Esp), base pointer (Ebp), and instruction pointer (Eip), with numbers 4, 5, and 8.

⟨target functions 45⟩ +≡ (89)

```
int *target_expedite(void)
{
  static int expedite[] = {4, 5, 8, −1};
  return expedite;
}
```

4.8. Resuming a Thread

This is the last problem we consider: How to continue a thread after the debugger has inspected it. The two functions are

⟨target exports 44⟩ +≡ (90)

```
extern void target_step(void);
extern void target_continue(void);
```

Let's look at *target_continue* first.

⟨target functions 45⟩ +≡ (91)

```
void target_continue(void)
{
  if (t.context_valid ∧ t.context_changed) {⟨set a valid cache 80⟩}
  fprintf(stderr, "Target␣Continue␣%x␣%x␣%x\n", t.process_id, t.thread_id, t.resume_mode);
  if (¬ContinueDebugEvent(t.process_id, t.thread_id, t.resume_mode))
    error ("Unable␣to␣continue␣Thread") ;
}
```

The *resume_mode* is normally `DBG_CONTINUE`

⟨set *resume_mode* 92⟩ ≡ (92)

```
t.resume_mode = DBG_CONTINUE;
```

Used in 52, 59, and 61.

and is stored in the

⟨ thread information 46 ⟩ +≡ (93)

```
DWORD resume_mode;
```

If, however, the thread was stopped by a signal through an `EXCEPTION_DEBUG_EVENT` and the thread is forced by a 'S' or 'C' to receive a certain signal, it can be set to `DBG_EXCEPTION_NOT_HANDLED`.

Now the function *target_step*. To achieve the stepping, we have to set a special bit in the *Eflags* register.

⟨ set stepping flag 94 ⟩ ≡ (94)

```
t.context.EFlags |= #100;
```

Used in 95.

But before we do so, we should make sure, that the data in the *context* cache is valid, and afterwards, we should remember that the cache has changed.

⟨ target functions 45 ⟩ +≡ (95)

```
void target_step(void)
{
  validate_cache();
  ⟨ set stepping flag 94 ⟩
  t.context_changed = 1;
  target_continue();
}
```

Another way to modify the behavior of the target is the setting of the continuation point. This can be done by modifying the instruction pointer. A function to do just that is

⟨ target exports 44 ⟩ +≡ (96)

```
extern void target_set_ip(unsigned int a);
```

The implementation should come as no surprise:

⟨ target functions 45 ⟩ +≡ (97)

```
void target_set_ip(unsigned int a)
{
  validate_cache();
  t.context.Eip = a;
  t.context_changed = 1;
}
```

5. Sending and Receiving Packets

The commands and answers are nicely wrapped into a packet. Each packet starts with a `$` sign and ends with a `#` sign followed by a two digit hexadecimal checksum.

Versions of `gdb` prior to 5.0 did add a sequence id and a colon just before the packet data, which is not supported by this application.

⟨ functions 30 ⟩ +≡ (98)

```
int getpkt(char *buffer)
{
  int length = 0;
  unsigned char my_checksum = 0, checksum = 0;
  do {
    ⟨ skip to the dollar sign 99 ⟩
    ⟨ get packet 100 ⟩
    ⟨ get checksum 101 ⟩
    ⟨ confirm receipt 102 ⟩
  } while (my_checksum ≠ checksum);
  buffer[length] = 0;
  return length;
}
```

A loop will

⟨ skip to the dollar sign 99 ⟩ ≡ (99)

```
{
  int ch;
  while ((ch = readchar()) ≠ '$')
    if (ch < 0) return ch;
}
```

Used in 98.

and terminates the function in case of read errors. Next, we start reading the packet data, adding all the bytes received into our own version of the checksum.

⟨ get packet 100 ⟩ ≡ (100)

```
{
  int ch;
  while ((ch = readchar()) ≠ '#') {
    if (ch < 0) return ch;
    my_checksum = my_checksum + ch;
    buffer[length++] = ch;
  }
}
```

Used in 98.

To get the real checksum, we read two hex digits.

⟨ get checksum 101 ⟩ ≡ (101)

```
{
  int ch;
  ch = readchar();
  if (ch < 0) return ch;
  checksum = fromhexdigit((char) ch);
  ch = readchar();
  if (ch < 0) return ch;
  checksum = (checksum ≪ 4) + fromhexdigit((char) ch);
}
```

Used in 98.

Any packet received gets confirmed with a + or, if corrupted, with a − sign.

⟨ confirm receipt 102 ⟩ ≡ (102)

```
if (checksum ≡ my_checksum) writechar('+');
else writechar('-');
flush();
```

Used in 98.

Now we turn to writing a packet:

⟨ functions 30 ⟩ +≡ (103)

```
int putpkt(char *buffer)
{
  unsigned char my_checksum = 0;
  int receipt;
  do {
    writechar('$');
    ⟨ write buffer 104 ⟩ writechar('#');
    ⟨ write my_checksum 105 ⟩ flush();
    ⟨ get receipt 106 ⟩
  } while (receipt ≡ '-');
  return 1;
}
```

A receipt character of − is a request for retransmission.

⟨ write buffer 104 ⟩ ≡ (104)

```
{
  int i;
  for (i = 0; buffer[i] ≠ 0; i++) {
    writechar(buffer[i]);
    my_checksum = my_checksum + buffer[i];
  }
}
```

Used in 103.

Writing the buffer could use run length encoding to save space. If a character is followed by ∗, the next character minus 29 is taken as a repetition count, which is applied to the character preceding the ∗. We do not use this feature here.

The checksum is written as two hex digits

⟨ write *my_checksum* 105 ⟩ ≡ (105)

```
writechar(tohexdigit(my_checksum ≫ 4));
writechar(tohexdigit(my_checksum & #0F));
```

Used in 103.

The receipt should be a + sign.

⟨ get receipt 106 ⟩ ≡ (106)

```
receipt = readchar();
if (receipt < 0) return receipt;
```

Used in 103.

5.1. Predefined Answers

Some answers are very common: an OK, an empty response to indicate that the command was not understood or is not implemented, or an error response. We provide functions to write these responses into the output buffer.

⟨ function prototypes 7 ⟩ +≡ (107)

```
void answer_ok(char *buffer); void answer_error (char *buffer, int error ) ;
void answer_nothing(char *buffer);
```

Here are simple implementations:

⟨ functions 30 ⟩ +≡ (108)

```
void answer_ok(char *buffer)
{
  buffer[0] = 'O';
  buffer[1] = 'K';
  buffer[2] = 0;
}
void answer_error (char *buffer, int error ) { buffer[0] = 'E'; buffer[1] = tohexdigit ( ( error ≫ 4 )
    &#0F ) ; buffer[2] = tohexdigit ( error &#0F ) ;
buffer[3] = 0; } void answer_nothing(char *buffer)
{
  buffer[0] = 0;
}
```

5.2. Hexadecimal Numbers

We need functions to convert binary numbers to hexadecimal digits.

We start with two functions,

⟨ function prototypes 7 ⟩ +≡ (109)

```
extern char tohexdigit(int n);
extern int fromhexdigit(char c);
```

that convert a number n in the range 0 to 15 into a hexadecimal digit and vice versa.

⟨ functions 30 ⟩ +≡ (110)

```
char tohexdigit(int n)
{
  n = n & #0F;
  if (n < 10) return '0' + n;
  else return 'A' + n − 10;
}
int fromhexdigit(char c)
{
  if ('0' ≤ c ∧ c ≤ '9') return c − '0';
  else if ('A' ≤ c ∧ c ≤ 'F') return c − 'A' + 10;
  else if ('a' ≤ c ∧ c ≤ 'f') return c − 'a' + 10;
  else error ("Illegal␣hex␣digit") ;
  return 0;
}
```

Occasionally, it is good to have a function to convert whole hexadecimal numbers to binary numbers and back. The two functions *hexfrombin* and *binfromhex* will not alter the byte order and are useful if the hex numbers refer to the target byte order. They both return the number of bytes written (either binary or hex).

The third and fourth function, *intfromhex* and *hexfromint*, rely on the host byte order and use regular **unsigned int**'s. The return value is the the unsigned integer value in the case of *intfromhex*, and the number of hex digits written in the case of *hexfromint*.

⟨ functions 30 ⟩ +≡ (111)

```
int hexfrombin(char *to, char *from, int fromsize)
{
  int i = 0;
  char ch;
  while (0 < fromsize−−) {
    ch = *from++;
    to[i++] = tohexdigit(((ch & #f0) ≫ 4) & #0f);
    to[i++] = tohexdigit(ch & #0f);
  }
  to[i] = 0;
  return i;
}
int binfromhex(char *to, int tosize, char *from)
{
  int i = 0;
  while (0 < tosize−−) {
    to[i] = (fromhexdigit(*from++) ≪ 4) & #f0;
    to[i] = to[i] | (fromhexdigit(*from++) & #0f);
    i++;
  }
  return i;
```

```
}
unsigned int intfromhex(char *from)
{
  unsigned int result = 0;
  while (isxdigit(*from)) {
    result = (result << 4) + fromhexdigit(*from);
    from++;
  }
  return result;
}
int hexfromint(char *to, unsigned int from)
{
  int i = 0;
  unsigned char ch;
  do {
    ch = from & #FF;
    from = from >> 8;
    to[i++] = tohexdigit(((ch & #f0) >> 4) & #0f);
    to[i++] = tohexdigit(ch & #0f);
  } while (from > 0);
  to[i] = 0;
  return i;
}
```

6. Setting up a TCP/IP Connection

A TCP/IP connection is quite something complicated.

In the end, however, we have a socket

⟨global variables 13⟩ +≡ (112)

```
static int remote_socket;
```

that we can use to send and receive data using the send and recv system calls. We encapsulate the system calls in two low level functions to read and write arbitrary data.

⟨functions 30⟩ +≡ (113)

```
static int sockread(int s, void *str, size_t n)
{
  int i;
  i = recv(s, str, n, 0);
  if (i > 0) return i;
  else error ("socket␣read") ;
  return i;
}
static void sockwrite(int s, char *str, size_t n)
{
  int i;
  while (n > 0) {
    i = send(s, str, n, 0);
    if (i > 0) {
      str = str + i;
      n = n - i;
```

```
      }
      else {
        error ("socket␣write") ;
        return;
      }
    }
  }
```

Next, we have three higher level functions, that provide buffered single character input and output using the lower level functions.

⟨ function prototypes 7 ⟩ +≡ (114)

```
static int readchar(void);
static void writechar(char c);
static void flush(void);
```

⟨ functions 30 ⟩ +≡ (115)

```
static int readchar(void)
{
  static unsigned char buffer[BUFSIZ];
  static int index = 0;
  static int size = 0;
  if (index ≥ size) {
    size = sockread(remote_socket, buffer, sizeof (buffer));
    if (size ≤ 0) return −1;
    index = 0;
  }
  return buffer[index ++];
}
```

⟨ global variables 13 ⟩ +≡ (116)

```
static unsigned char out_buffer[BUFSIZ];
static int out_size = 0;
```

⟨ functions 30 ⟩ +≡ (117)

```
static void writechar(char c)
{
  if (out_size ≥ BUFSIZ) flush( );
  out_buffer[out_size ++] = c;
}
```

⟨ functions 30 ⟩ +≡ (118)

```
static void flush(void)
{
  sockwrite(remote_socket, out_buffer, out_size);
  out_size = 0;
}
```

All the rest of handling a TCP/IP connection under windows is contained in the following two functions:

⟨ function prototypes 7 ⟩ +≡ (119)

```
extern void remote_open(char *name);
extern void remote_close(void);
```

Let us first investigate how to set up a TCP/IP connection under windows. The user interface of `gdb` allows to specify the remote connection as “hostname:port” Actually, the hostname gets ignored, and we just

⟨extract the port number 120⟩ ≡ (120)

```
{
  char *s;
  s = strchr(name, ':');
  if (s ≡ NULL) fatal_error("IP␣port␣missing");
  port = atoi(s + 1);
}
```

Used in 132.

For functions like *atoi*, we need

⟨include files 26⟩ +≡ (121)

```
#include <stdlib.h>
```

Under the Windows Operating system, TCP/IP is handled by the so called WinSock Dll, the windows socket dynamic link library. This library needs to be loaded and initialized.

⟨initialize Windows TCP 122⟩ ≡ (122)

```
{
  WSADATA wsaData;
  if (WSAStartup(MAKEWORD(1, 1), &wsaData) ≠ 0) fatal_error("Unable␣to␣initialize␣TCP/ip");
}
```

Used in 132.

For the prototypes, we need

⟨include files 26⟩ +≡ (123)

```
#include <winsock.h>
```

The actual data is exchanged through a mechanism well known from the Unix operating system: sockets. We first need a socket to be able to listen to the given port.

⟨obtain a socket 124⟩ ≡ (124)

```
{ int listen_socket;
listen_socket = socket(PF_INET, SOCK_STREAM, 0);
if (listen < 0) fatal_error("Can't␣open␣socket");
```

Used in 132.

We change the settings for this socket to allow rapid reuse

⟨obtain a socket 124⟩ +≡ (125)

```
{
  int tmp = 1;
  setsockopt(listen_socket, SOL_SOCKET, SO_REUSEADDR, (char *) &tmp, sizeof (tmp));
}
```

We set up an information structure specifying the right port.

⟨obtain a socket 124⟩ +≡ (126)

```
{ struct sockaddr_in sin;
memset(&sin, 0, sizeof (sin));
sin.sin_family = PF_INET;
sin.sin_port = htons(port);
sin.sin_addr.s_addr = INADDR_ANY;
```

And bind the socket to the port.

⟨obtain a socket 124⟩ +≡ (127)

```
if (bind(listen_socket, (struct sockaddr *) &sin, sizeof (sin)) ∨ listen(listen_socket, 1))
  fatal_error("can␣not␣bind␣address");
```

Using this socket we now listen at out port and wait for a connection.

⟨obtain a socket 124⟩ +≡ (128)

```
{
  int tmp;
```

```
    tmp = sizeof (sin);
    remote_socket = accept(listen_socket, (struct sockaddr *) &sin, &tmp);
    if (remote_socket < 0) fatal_error("accept");
  }
```

Once this call returns successfully set options on the new socket to enable TCP to keep alive process and to tell TCP not to delay small packets (This can speed up interactive connections dramatically).

⟨obtain a socket 124⟩ +≡ (129)

```
  {
    int tmp;
    tmp = 1;
    setsockopt(remote_socket, SOL_SOCKET, SO_KEEPALIVE, (char *) &tmp, sizeof (tmp));
    tmp = 1;
    setsockopt(remote_socket, IPPROTO_TCP, TCP_NODELAY, (char *) &tmp, sizeof (tmp));
  }
```

After that, we do not need any more the *listen_socket* and all the data structures associated with it.

⟨obtain a socket 124⟩ +≡ (130)

```
  closesocket(listen_socket);        /* No longer need this */
  } }
```

We announce success and are done for mow.

⟨obtain a socket 124⟩ +≡ (131)

```
  message("Connected␣to␣gdb␣...\n");
```

This sequence of actions is packed into a nice function:

⟨functions 30⟩ +≡ (132)

```
  void remote_open(char *name)
  {
    unsigned short int port;
    ⟨extract the port number 120⟩⟨initialize Windows TCP 122⟩⟨obtain a socket 124⟩
  }
```

This function is used to

⟨open a connection to gdb 133⟩ ≡ (133)

```
  remote_open(argv[1]);
```

Used in 4.

Once the TCP/IP connection is no longer needed, we close the socket and tell the Winsock Dll to clean up.

⟨functions 30⟩ +≡ (134)

```
  void remote_close(void)
  {
    closesocket(remote_socket);
    WSACleanup();
  }
```

We use this to

⟨close the connection to gdb 135⟩ ≡ (135)

```
  remote_close();
  message("Remote␣host␣terminated␣connection.");
```

Used in 5.

7. Error Reporting and Messages

Here we define three functions:

⟨target imports 60⟩ +≡ (136)

```
extern void message(char *msg); extern void
error (char *msg) ;
extern void fatal_error(char *msg);
```

We have three stages: normal messages are just printed to stderr

⟨functions 30⟩ +≡ (137)

```
void message(char *msg)
{
  fputs(msg, stderr);
}
```

For *stderr*, we need

⟨include files 26⟩ +≡ (138)

```
#include <stdio.h>
```

On the next level, we have errors, they are printed and tagged with the Word Error.

⟨functions 30⟩ +≡ (139)

```
void error (char *msg)
{
  message("Error:␣");
  message(msg);
  message("\n");
  longjmp(toplevel, 1);
}
```

Further, we use a *longjmp* to return to a predefined location stored in a

⟨global variables 13⟩ +≡ (140)

```
static jmp_buf toplevel;
```

We need

⟨include files 26⟩ +≡ (141)

```
#include <setjmp.h>
```

It gets initialized two ways, we

⟨set an error reentry point 142⟩ ≡ (142)

```
if (setjmp(toplevel)) putpkt("");
```

Used in 4.

and we

⟨set an error exit point 143⟩ ≡ (143)

```
if (setjmp(toplevel)) fatal_error("Unable␣to␣continue");
```

Used in 4.

At the last level, there are fatal errors. The program will not continue past a call to this function.

⟨functions 30⟩ +≡ (144)

```
void fatal_error(char *msg)
{
  message("Fatal␣Error:␣");
  message(msg);
  message("\n");
  exit(1);
}
```

8. Index

9. Crossreference of Identifiers

10. Crossreference of Code

⟨ answer memory string 25 ⟩ Used in section 23.

⟨ answer register n 22 ⟩ Used in section 21.

⟨ answer registers 12 ⟩ Used in section 11.

⟨ check for end of target process 32 ⟩ Used in section 8.

⟨ close the connection to gdb 135 ⟩ Used in section 5.

⟨ confirm receipt 102 ⟩ Used in section 98.

⟨ convert win32 signal to `gdb` signal 62 ⟩ Used in section 61.

⟨ convert *argv* to *commandline* 68 ⟩ Used in section 66.

⟨ define local variables 9 ⟩ Used in section 4.

⟨ dispatch a message 10, 11, 16, 18, 21, 23, 27, 28, 35, 36, 37, 38, 39, 40 ⟩ Cited in section 41. Used in section 8.

⟨ extract the port number 120 ⟩ Used in section 132.

⟨ function prototypes 7, 31, 34, 107, 109, 114, 119 ⟩ Used in section 1.

⟨ functions 30, 33, 98, 103, 108, 110, 111, 113, 115, 117, 118, 132, 134, 137, 139, 144 ⟩ Used in section 2.

⟨ get address a and length l 24 ⟩ Used in sections 23 and 27.

⟨ get checksum 101 ⟩ Used in section 98.

⟨ get optional address a 29 ⟩ Used in sections 28 and 35.

⟨ get packet 100 ⟩ Used in section 98.

⟨ get receipt 106 ⟩ Used in section 103.

⟨ global variables 13, 112, 116, 140 ⟩ Used in section 1.

⟨ ignore the *event* 48 ⟩ Cited in section 47. Used in sections 47, 50, 51, and 63.

⟨ include files 26, 42, 121, 123, 138, 141 ⟩ Used in section 1.

⟨ initialize Windows TCP 122 ⟩ Used in section 132.

⟨ initialize variables 14, 15 ⟩ Used in section 4.

⟨ invalidate the cache 77 ⟩ Used in sections 52, 59, 61, and 66.

⟨ obtain a socket 124, 125, 126, 127, 128, 129, 130, 131 ⟩ Used in section 132.

⟨ obtain one register value 20 ⟩ Used in section 18.

⟨ obtain register number n 19 ⟩ Used in sections 18 and 21.

⟨ obtain registers 17 ⟩ Used in section 16.

⟨ open a connection to gdb 133 ⟩ Used in section 4.

⟨ private target types 53, 82, 83 ⟩ Used in section 43.

⟨ private target variables 56, 84 ⟩ Used in section 43.

⟨ process the *event* 47, 49, 50, 51, 52, 59, 61, 63 ⟩ Cited in section 45. Used in section 45.

⟨ receive and dispatch messages 8 ⟩ Cited in section 7. Used in section 4.

⟨ set a valid cache 80 ⟩ Used in section 91.

⟨ set an error exit point 143 ⟩ Used in section 4.

⟨ set an error reentry point 142 ⟩ Used in section 4.

⟨ set stepping flag 94 ⟩ Used in section 95.

⟨ set *resume_mode* 92 ⟩ Used in sections 52, 59, and 61.

⟨ skip to the dollar sign 99 ⟩ Used in section 98.

⟨ start the target program 64 ⟩ Used in section 4.

⟨ target exports 44, 55, 57, 65, 69, 71, 73, 81, 90, 96 ⟩ Cited in section 41. Used in section 41.

⟨ target functions 45, 58, 66, 70, 72, 74, 78, 85, 86, 87, 88, 89, 91, 95, 97 ⟩ Cited in section 43. Used in section 43.

⟨ target imports 60, 136 ⟩ Cited in section 41. Used in section 41.

⟨ `target.c` 43 ⟩

⟨ `target.h` 41 ⟩

⟨ thread information 46, 54, 67, 75, 76, 79, 93 ⟩ Used in section 53.

⟨ write buffer 104 ⟩ Used in section 103.

⟨ write *my_checksum* 105 ⟩ Used in section 103.