Pawn

embedded scripting language

Debugger Interface

January 2016 CompuPhase

Requests for corrections and additions to the manual and the software can be directed to CompuPhase at the above address.

Typeset with TEX in the “DejaVu” typeface family.

Remote debugging interface

When debugging script on the same machine as the debugger, the PAWN abstract machine invokes a “debug hook” function directly inside the debugger. Synchronization is usually not an issue, because the hook function is normally called on the same task or thread as that the abstract machine runs on. The debugger also has direct access to the code and data of the running script, as they run on the same machine and (more importantly) in the same process.

Remote debugging adds two stubs and a protocol to the picture. The abstract machine still calls into a debug hook function, but this function is a stub that sends all events over a communication line. The debugger is not invoked from the abstract machine directly, but from another stub, which receives its signals from the remote host. A communication line, such as an RS232 line, connects the two stubs.

The PAWN debugger already contains a stub for remote debugging over a serial line. To add remote debugging, we therefore need to implement a stub for the remote host that conforms to the protocol that the PAWN debugger uses.

The terminology used in this application note is that the “remote host” is the platform that runs the PAWN script, and the “local host” the the platform (a PC) that runs the debugger. Both sides have the PAWN abstract machine, but only the remote host runs the script.

The debugger hook “stub”

Set-up and handshake

Setting up a debugger hook stub on the remote host takes the same steps as for the case of local debugging, but before going forward with it: you may want to check that 1) the script contains debug information and 2) that a debugger is trying to connect.

To test whether a script contains debug information, call function `amx_Flags` after `amx_Init`. If the script has debug information, proceed with the following step (checking for a debugger at the local host); if not, you will probably want to run the script without debug hook.

The reason for having the remote device check for a debugger (on the local host) is that the abstract machine blocks on the the debug hook (upon encountering a BREAK instruction). The script does not continue until the debug hook returns. The hook function itself sends the event to the local host and waits for the reply of the debugger. If no debugger is present at the local host, though, the script will just hang after dropping on its first BREAK instruction.

To avoid this, the PAWN debugger sends a handshake character over the serial line until it gets a response. The remote host should set up a serial connection, and enter a polling loop with a time-out. If it receives the handshake character within the time-out period, it replies with its own handshake and proceeds installing the debug hook. If it *does not* receive the handshake character, it should probably just run the script *without* setting up a debug hook. The suggested time-out is three seconds.

The handshake character of the current version of the PAWN debugger is the inverted exclamation mark (“¡”, character code $A1_h$ in the ISO 8859-1 “Latin-1” character set). Previous versions of the PAWN debugger used the standard exclamation mark (“!”, ISO 8859-1 21_h), but this is also a special character during normal operation of a debug session. On reception of the “¡”, the remote device installs a debug hook stub function, and starts running the script. The PAWN debugger waits for the first response from this debug hook stub. If, for any reason, the script cannot run, but the remote device still wants to communicate with the PAWN

debugger, it should reply to the "¡" by sending the inverted question mark ("¿", character code BF_h in the ISO 8859-1 "Latin-1" character set) plus a closing square bracket ("]"). After sending this reply, the remote device should wait for the debugger to respond (with a command).

To install a debug hook, call function `amx_SetDebugHook` after `amx_Init` and before `amx_Exec`. For example, if the debug hook function will be called `DebugStub`, you could add the following line after the call to `amx_Init`:

```
err = amx_SetDebugHook(&amx, DebugStub);
```

The remote host now has to contain a function called `DebugStub`, with the following definition:

```
int AMXAPI DebugStub(AMX *amx)
{
  /* forward events and handle requests */
}
```

The interesting parts is, of course, what happens between the braces of the function `DebugStub`.

Overall operation

The debug hook function in the remote host is called from the abstract machine, each time that this abstract machine drops on a BREAK instruction. The tasks of the debug function are to send a signal to the debugger (at the local host) and to then handle any commands that the debugger wants to carry out on the remote host.

The signal to the remote host is a "¿" character followed by the code address and a "]". The code address is in the `cip` field of the `AMX` structure passed to the debug hook. The code address must be in hexadecimal format. For example, the string to send could be created with the C code:

```
char str[128];
sprintf(str, "\xbf%x]", amx->cip);
```

After sending the string, the debug hook function must enter a loop and wait until it receives the character "!" (ASCII 21_h). This character indicates that the script must continue to run, until the next BREAK instruction. Upon reception of the "!" character, the debug hook function returns `AMX_ERR_NONE`.

If the PAWN debugger is in "run" mode, it will immediately reply to any incomping signal with a "!", therefore instructing the remote host to continue without delay.

The debug hook can receive other commands from the debugger. These start with a question mark ("?"), followed by an upper case letter, parameters, and end with a newline (ASCII $0A_h$). The debug hook may also receive a stray "¡" character (character code $A1_h$), which was still in the queue from the handshake procedure —this character should simply be ignored.

The replies of the remote device to all commands follows the syntax "¿" + *parameters* + "]". If the local host receives data that does not conform with this syntax, it should be assumed normal output of the remote device, that it can print (on the console) or forward. Thus, the normal serial output of a script and the debugger interface may share a single serial line.

A fairly minimal template for the debug hook function is below. In this function, it is assumed that the functions `send_rs232` and `recv_rs232` are available, and that both these functions have as parameters a character buffer and the number of characters to send or receive respectively.

Template debug hook function

```
int AMXAPI DebugStub(AMX *amx)
{
  char str[128];
  int idx;

  sprintf(str, "\xbf%x]", amx->cip);
   send_rs232(str, strlen(str));

  for ( ;; ) {
     recv_rs232(str, 1);
    switch (str[0]) {
    case '!':
      return AMX_ERR_NONE;

    case '?':
      for (idx = 1; str[idx - 1] != '\n'; idx++)
         recv_rs232(str + idx, 1);
      str[idx] = '\0';    /* zero-terminate, for convenience */

       /* now handle the commands */

      break;
    } /* switch */
```

```
    } /* for */
  }
```

Command overview

The remaining part, now is to handle the commands. The table of commands that the debugger sends to the remote host is:

B	value	Set baud rate
G	name	Transfer file to the local host (“get”)
L	directory	List files on the remote host
M	address,size	Transfer memory contents
P	size,name	Transfer file to the remote host (“put”)
R		Transfer registers FRM, STK and HEA
T	time stamp	Synchronizes the time of the remote host
U		Unhook, with optional reset
W	address,data	Write memory

Only three commands are mandatory for the PAWN debugger: “M”, “R” and “U”. I recommend that a remote host responds with “¿0]” to any unknown command, as to avoid that the debugger waits for an answer with a time-out.

All values transferred between the debugger and the debug hook function are in hexadecimal format, but without any prefixes. As stated earlier, every command (from the local host to the remote device) ends with a newline character (ASCII $0A_h$).

Transferring registers

If the PAWN debugger is in stepping mode, it first sends the “R” command (meaning that it sends the string “?R” (plus a newline character). It expects as the return value from the remote host, a string starting with an “¿” and followed with the values of the registers FRM, STK and HEA. These are registers of the PAWN abstract machine, and they can be read from the AMX structure that the debug hook received. For example, the debug hook in the remote device can create this string with the C code:

```
char str[128];
sprintf(str, "\xbf%x,%x,%x]", amx->frm, amx->stk, amx->hea);
```

command	**reply**
?R	¿*frm*,*stk*,*hea*]

Transferring memory

The “M” command has two parameters: the start address to read memory from and the number of cells to return. The debugger hook must return a single string that holds the values (in hexadecimal), separated by commas. The string must start with a “¿” and end with a “]”. So, if three cells were requested, the reply string would have the form “¿1234,9af,17]”. In its current implementation, the PAWN debugger asks for a maximum of 10 values with the “M” command. With 32-bit cells, the required minimal string length is 92 bytes (max. 10 values of max. 8 hexadecimal digits, with 9 commas separating the values, the “¿” prefix, the “]” suffix and, for convenience, a terminating zero byte).

command	**reply**
?M*address*,*size*	¿*value*,*value*,...]

The address on the “M” command is relative to the data section of the abstract machine. To convert this address into a pointer to physical memory, the local host can use a function like the one below:

Translate virtual addresses to physical memory addresses

```
static cell *VirtAddressToPhys(AMX *amx, cell amx_addr)
{
  AMX_HEADER *hdr = (AMX_HEADER *)amx->base;
  unsigned char *data = amx->data ? amx->data : amx->base+(int)hdr-
>dat;

  if (amx_addr >= amx->hea && amx_addr < amx->stk
      || amx_addr < 0
      || amx_addr >= amx->stp)
    return NULL;

  return (cell *)(data + (int)amx_addr);
}
```

The function tests whether the code and data sections of the abstract machine are combined or separate (for example, if the code runs from ROM and the data is in RAM, both sections are separate), and it also checks whether the address is valid: either in the data section, or in the valid portion of the stack or the heap. The return value is a pointer to the cell whose “virtual” address was passed.

End of a debug session

The “U” command instructs the debug hook to unhook itself, so that the script continues running without debugger. If the “U” is

followed by a “*”, the remote host should be reset. It is up to the remote host to decide to what degree the system should reset. In any case, the “U” command indicates the end of a debugging session. There is no reply to this command.

command	**reply**
?U	
?U*	

Advanced commands

The previous chapter covered the essential commands for a debug hook: when these commands are not present, the PAWN debugger does not function well. Some of the remaining commands are more complex, others fall outside the scope of the PAWN debugger (these commands are used by some utilities, but they are not used by the current version of the debugger.

Writing memory

The "W" command does the inverse of the "M" command. This command contains a list of values that it wants the debug hook to store in the data section of the abstract machine, at the given address. The debugger will only send this command after a user request. Typically, the user has entered a command in the debugger to adjust the value of a variable.

The parameters of the "W" command are the start address, relative to the data section of the abstract machine, followed by a list of values. The values are separated with commas (and in hexadecimal). The list ends with a newline.

The address is relative to the data section of the abstract machine. The remote host needs to convert this address to a pointer to physical memory —for example, using the `VirtAddressToPhys` function presented at page 6.

The debug hook should reply with the string "`¿1]`" for success and "`¿0]`" for failure.

command	**reply**
`?`*address*`,`*value*`,...`	`¿1]`
	`¿0]`

Transferring files

There are two commands for transferring files: one to "get" a file from the remote host and one to "put" a file onto the remote host. Both transfers use a simple protocol, where the file data is sent in blocks and each block is acknowledged with a simple checksum. The block size must be negotiated at the start of the transfer.

The PAWN debugger only uses the "P" command (for "put"), for storing a compiled script onto the remote host —e.g. before starting to debug it. The transfer in the opposite direction, the "G" command, is not used in the current version of the PAWN debugger.

• "Put" a file

The parameters of the "P" command are the file size (the number of bytes that will be transferred) and the filename. The filename is optional; if the debugger does not send a filename, the remote host may use a fixed (default) filename, or a temporary filename.

The reply of the "P" command carries the block size that the debugger should use to send the data (like all numbers in the debugger interface, this value is in hexadecimal). For example, if the remote host replies with `¿100]`, the debugger will transfer the file in blocks of 256 bytes of binary data. The remote host acknowledges each received block by sending a checksum.

Each block starts with a single-byte "start code". This code is either an ACK (ASCII 06_h) or a NAK (ASCII 15_h). If the debugger sends an ACK, the block that follows is the next sequential block of data for the file. If it sends a NAK, the block that follows is a repeated send of the preceding block. The basic walk-through of the transfer is: after sending the "P" command and receiving the block size, the debugger sends an ACK and the first block of data from the file. The debugger hook in the remote host receives the data and replies with the checksum. The debugger compares the received checksum with the one it computed itself. If it matches, it sends an ACK plus the *next* block of data; if it mismatches, it sends a NAK and retransmits the previous block.

The checksum is the "Internet checksum", but as an 8-bit variant. This checksum is often called the "one's complement", because it wraps the carry around on overflow. Below is an example function to calculate the checksum (for blocks smaller then 4 MiB); the first loop adds all bytes in a buffer together and the second loop adds back any overflow.

Checksum calculation

```
unsigned char checksum(const unsigned char *buffer, size_t size)
{
  unsigned long chksum = 1;
  size_t i;

  for (i = 0; i < size; i++)
    chksum += buffer[i];
```

```
    while (chksum > 0xff)
      chksum = (chksum & 0xff) + (chksum >> 8);

    return (unsigned char)chksum;
  }
```

Fast mode
A remote host may activate "fast transfer mode". Fast transfer mode differs from the above description in the following ways:

- ◇ The remote host appends a ",crc" to its acknowledgement for the block size. For example, when the block size is 256 bytes, it would transmit ¿100,crc] instead of ¿100].
- ◇ The blocks do not start with a start code. Each block is just the data. As a result, there are no retransmits, as there is no method to flag is retransmit.
- ◇ The remote host acknowledges only the very last block, but the "checksum" in that acknowledgements is the 32-bit CRC of the received file.
- ◇ Flow control is needed to regulate the transfer speed. On RS232 lines, this should be hardware flow control (RTS/CTS); USB virtual COM ports can use "NAK" delays.

• "Get" a file

The PAWN debugger does not currently use the "G" command. This command (and a few others) are intended for file synchronization utilities (on remote hosts that implement a file system). Notwithstanding this, I will refer to the utility running on the local host as "the debugger" in this section.

The debugger sends the "G" command, followed by the filename. The remote host replies with the file size and the block size.

command	**reply**
?G*filename*	¿*filesize*,*blocksize*]

After receiving the file size and the block size, the debugger acknowledges it by sending an ACK (ASCII 06_h) plus a dummy checksum with the value 1. After this, the remote host sends a start code and the first block of data (for the first block, the start code is always an ACK). After receiving each block, the debugger responds with an ACK plus the checksum. The remote host verifies the checksum and either sends an ACK followed by the next block of data, or sends a NAK (ASCII 15_h) followed by a retransmission of the previous block of data.

Fast mode
The remote host may active "fast transfer mode", which differs from the the above description in the following ways:

- In the reply to the “G” command, the remote host returns the file size, the block size *and* the 32-bit file CRC.
- The “checksum” value in the acknowledgements is not present.
- The data blocks do not start with an ACK (start code).

Deleting and listing files

File deletion is a special case of the “G” command (see the section “Transferring files”) and transferring the file list (directory) is done with the “L” command. The “G” and “L” commands are not used in the debugger; these commands are intended for file synchronization utilities.

To delete a file, the debugger sends the “G” command, followed by an asterisk and the filename. The remote host then removes the file and replies with “?`1]” (for success) or “?`0]” for failure. No further handshaking occurs and no data is transferred.

command	**reply**
?G**filename*	?`1]
	?`0]

The “L” command optionally specifies a directory name or path, and/or wildcard characters. Whether directories and wildcards are supported depends on the remote host. The remote host replies with zero or more file records and ends with a string that contains only “?`]”.

command	**reply**
?L*path*	?`*size time filename*]
	?`]

The first field on each record is the file size, which is set to -1 for a directory. The second field is the time stamp for the file, as seconds since midnight January 1, 1970 (the UNIX time standard). The last field is the name of the file/directory.

Note that the fields are separated with a space (instead of a comma) and also note that the filename may contain spaces.

The remote host simply sends record after record; the debugger does not acknowledge reception of records.

Changing the baud rate

The parameter of the “B” command is the new baud rate value (in hexadecimal). The remote host does not reply to this command. However, after the local host and the remote host have both changed their baud rates, the handshake procedure (see page 2) must be restarted.

Note that if a session ends the the “`?U`” command (no reset), the remote host will typically keep the baud rate. If the session ends with “`?U*`”, however, the baud rate should reset to its default value. See page 6 for more on the “U” command.

Setting up remote debugging

On the remote host, the compiled script must be present. The script must obviously be compiled with full debugging information. On the local machine, this *same* compiled script must also be present, plus the source code of the script. The reason that the compiled script must be present on both the local and the remote hosts is that the debugger parses the local file for the symbolic information.

To start the PAWN debugger for remote debugging, you need to add the option "`-rs232`" to the command line. This option has two parameters that you may add: the port number and the Baud rate.

Port numbers start at 1 for Microsoft Windows, where "1" stands for COM1. Under Linux, port numbers start at 0, where "0" stands for /dev/ttyS0. The default port is 1 for Microsoft Windows and 0 for Linux. To select another port, use the syntax "`-rs232=2`".

The default Baud rate for the debugger is 57600 bps. If the remote host uses a different bad rate for communication, that rate must be specified on the command line, as the second parameter of the "`-rs232`" option. To specify COM1 (on Microsoft Windows) at 9600 bps, add the option "`-rs232=1,9600`" on the command line.

GUI shell interface

The PAWN debugger is a console application. Although it has VT100 terminal support (including pseudo-terminals like the Microsoft Windows console), its interface is rudimentary for today's standards. It is common to lay a graphical shell on top of the debugger, for full-screen debugging and mouse support.

The terminal support must be switched off, so that the shell has a minimum of effort to parse the debugger's output. The output of the debugger must furthermore the recognizable so that the shell can distinguish it from the output of the compiled script itself. The PAWN debugger uses the "`-term`" option for both these settings. By setting "`-term=off,<&}`", the debugger uses only "streaming console" output and prefixes any line of output with "`<&}`".

With the terminal turned off, each line of output (from the debugger) starts with the prefix and runs to the end of the line. When listing all variables, there will be one variable per line and each line starts with the prefix.

command	**parameters**
digits	the line number (an asterisk suffix if active) plus the text
`file`	the name of the source file
`loc`	the name and value of the local variable
`glb`	the name and value of the global variable
`watch`	the index, name and value of the watched variable
`info`	a general message from the debugger
`dbg>`	the debugger prompt

Resources

The PAWN toolkit can be obtained from **www.compuphase.com** in various formats (binaries and source code archives). The manuals for usage of the language and implementation guides are also available on the site in Adobe Acrobat format (PDF files).

Index

- Names of persons (not products) are in *italics*.
- Function names, constants and compiler reserved words are in `typewriter font`.