ybug - System Control Tool for SpiNNaker

SpiNNaker Group, School of Computer Science, University of Manchester

Steve Temple - 19 Sep 2014 - Version 1.33

Introduction

ybug is a program which runs on a host computer (workstation) and provides an interactive text-based interface to a SpiNNaker system. It allows the system to be bootstrapped and programs and data to be downloaded. There are a number of low-level debugging features such as the ability to inspect and change memory in any SpiNNaker chip in the system. **ybug** communicates with the target system using a protocol based on UDP/IP and so can control systems located anywhere on the Internet.

Installation and dependencies

ybug is written in Perl and uses a number of locally developed Perl libraries. Installation just requires the copying of the main source file and the libraries to a suitable place on your workstation and the setting of some environment variables to reference these. **ybug** needs the library `String::CRC32` which is commonly installed but can be also found at CPAN or as package `libstring-crc32-perl` (Ubuntu, etc) or `perl-String-CRC32` (Fedora, etc).

There is also one optional, but very useful, dependency on the Perl library which interfaces to the GNU ReadLine library. This provides command history and filename completion and requires the installation of the library `Term::ReadLine::Gnu`. This can be downloaded either from CPAN or via a package such as `libterm-readline-gnu-perl` (Ubuntu, etc) or `perl-Term-ReadLine-Gnu` (Fedora, etc).

Background

ybug began as a simple hack to talk to early SpiNNaker systems. Like many hacks, it has not (yet) been superseded. This note documents version 1.33 of **ybug**.

Starting *ybug* and the Command Line Interface (CLI)

ybug is started from the Unix shell by giving the command `ybug` with a single compulsory argument which is the IP address of the SpiNNaker system that you wish to control. The IP address can be numeric or a host name which will be looked up in the usual way. **ybug** accepts options which begin with `-`. The option `-help` will list all options. If you don't have the ReadLine library loaded and **ybug** complains about this, the `-norl` flag may help.

When the program starts you are presented with a prompt which has four components. The first is the hostname or IP address which was given when the program was started. The second, third and fourth are three numbers which indicate which chip and core on the SpiNNaker system certain **ybug** commands will be directed to.

At this point you can type commands which will be executed by the **ybug** CLI. A line beginning with the character `#` will be ignored and can be used for comments. Otherwise the first non-blank field on the line is interpreted as a command and further fields are interpreted as arguments to the command. If the ReadLine library is in use then completion of the command name (the first item on the command line) is attempted as is filename completion on subsequent items on the line.

Some commands not directly related to SpiNNaker are as follows

- `?` - provides a short-form list of all commands which are available.

- `help [<name>]` - provides a long list of all commands, their arguments and purpose. If a valid command name is given as an argument, just that command is listed.

- `<command> ?` - performs the same function as `help <command>`

- `version` - displays the **ybug** version number

- `expert` - enables some commands which are rarely required or may be dangerous if used carelessly

- `echo <string>` - echoes the string to the console. There is no newline printed at the end of the string but the characters `\n` will generate a newline anywhere in the string.

- `pause <string>` - echoes the string to the console (like `echo`) and then waits for the user to press `Enter`.

- `@ <file.F> [quiet]` - reads subsequent commands from the supplied file. Commands are echoed unless the `quiet` flag is provided. This command can be nested up to 10 deep.

- `quit` - exits **ybug**

Arguments

ybug commands take a variety of arguments. These may be file names, numbers in decimal or hex, IP addresses, etc. The format of each argument required by a command is documented in the help text where this is possible. In the help text, argument names are followed by a character which indicates what form the argument should take.

Where a hexadecimal number is expected, a preceding "0x" is not necessary and is ignored if provided. Where a decimal number is expected, a preceding "0x" is allowed and the number converted.

D	Decimal number
X	Hexadecimal number
R	Real number
F	File name
M	MAC address
P	IP address

Selecting a core or chip

Most **ybug** commands cause communication to take place with the SpiNNaker system. Some commands need to communicate with a particular core on a particular chip. Other commands need to communicate with the monitor processor on a particular chip while others need to communicate with the monitor processor on the **root chip** which is usually the one connected to the system's Ethernet interface.

SpiNNaker chips are addressed with a pair of coordinates which are their X and Y position in the grid of chips. The chip at coordinate (0, 0) is normally the root chip. The range of X and Y coordinates will vary according to the size of the SpiNNaker system. SpiNNaker cores are addressed as a number in the range 0 to 17. Core 0 is the monitor processor and cores 1 to 17 are application processors on which application programs are run. The current X, Y and Core settings are shown in **ybug**'s prompt.

The **ybug** command `sp` is used to select the chip and/or core. It can have up to three arguments as follows

```
        sp root                         ChipX = 0, ChipY = 0, Core = 0
        sp <core>                       Core = core
        sp <X> <Y>                      ChipX = X, ChipY = Y, Core = 0
        sp <X> <Y> <core>               ChipX = X, ChipY = Y, Core = core
```

ybug commands which know that they need to talk to a specific chip or core which is in conflict with the currently selected chip/core will ignore the relevant parts of the selection.

Bootstrapping a SpiNNaker system

After a SpiNNaker system has been powered-on or reset, it needs to be bootstrapped by loading its control software (a program called **SC&MP**). Once the system has been bootstrapped it should not normally be necessary to reboot it unless an application program goes badly wrong and corrupts some critical data on a chip. At this point the system will normally need to be reset and rebooted.

The **ybug** command to reboot a system is `boot` and it needs two arguments. The first is the name of a boot file and the second is the name of a configuration file which is used to configure the bootstrap for the particular system which is being booted. A set of standard configuration files will be supplied with your system. For example, if you have a SpiNN-3 board you should use the configuration file `spin3.conf`. Various operating parameters of the system, such as the clock speed of the processors, can be configured by editing the config file. A typical boot command looks like this.

```
        > boot scamp.boot spin4.conf
```

The boot and config files are searched for using the path in the environment variable `$SPINN_PATH`. This variable must be set (and exported) otherwise the files will not be found (ie there is no default search location).

Booting may take a couple of seconds and you should see a message confirming that the bootstrap was successful. If not, consider resetting the system before trying again.

Commands to control *ybug*

These commands control the way that **ybug** operates and do not directly interact with the SpiNNaker system.

- `debug <num.D>` - sets a debug variable which controls how much debug information is displayed as **ybug** operates. 0 means no debug and 1 through 4 provide increasing amounts of information. The information is mostly related to data packets going to and from SpiNNaker. Without an argument, the current setting is displayed.

- `sleep <time.R>` - causes **ybug** to pause for the specified number of seconds. Can be useful in command files where time needs to be allowed between successive commands. If no time is given, 1 second is used.

- `timeout <time.R>` - sets the timeout (in seconds) for responses from SpiNNaker when commands are sent to it. The default is 0.5 seconds and this is likely to be adequate for most connections. Without an argument, the current setting is displayed.

Commands to load and control Applications

A SpiNNaker application is a program which runs on a single SpiNNaker core. Many cores may be loaded with the same application. While an application is running on SpiNNaker it has an Application ID which is allocated when the application is loaded. The AppID is a number in

the range 16 to 254. AppIDs below 16 and above 254 are reserved for system use. The AppID is used to control the application as a whole while it runs on SpiNNaker and also serves to identify shared resources on the chip which are currently being used by the application.

A SpiNNaker application is stored in an APLX file and files of this type are loaded into SpiNNaker to start an application running.

- `app_load <file.F> <region> <cores> <app_id.D> [wait]` - loads an application onto the specified set of cores on the chips in the specified region. The application is contained in the file which should be in APLX format.

  The region specifies a set of chips ranging from all chips to a single chip. The definition of regions on SpiNNaker is documented elsewhere but some examples are given here.

`all`	All chips
`@X,Y`	Chip (X, Y)
`.`	Current chip (as selected by `sp`)
`0.0.0`	16 chips bounded by (0,0), (0,3), (3,3), (3,0)

  The cores can be specified as a comma separated list or a minus separated range, a combination of these two or the string `all`. Some examples

`1`	core 1
`1-4`	cores 1 to 4
`7,9`	cores 7 and 9
`1-4,7,9`	cores 1 to 4 and 7 and 9
`all`	all cores (1 to 17)

  The application starts to load onto all cores on a chip at the same time but variations in access to the shared memory where the APLX file is loaded may mean that not all cores start to run the application simultaneously. Similarly, each chip will start to load at a different time. Chips further away from the root chip start to load later. If this is an issue, there are various software techniques available to synchronise start-up.

  The application is given the supplied AppID when it starts. You should not load another application with the same AppID into the system while the original one is still running. Applications remain in the system even after they terminate (ie exit `c_main`). This allows debugging and inspection of their state but also means that they must be explicitly removed.

  If the `wait` flag is specified, the application is loaded but does not start to execute until it receives a `start` signal. This is useful for loading many different applications and having them all start at the same time.

- `app_stop <app_id.D>[-<app_id.D>]` - causes the application with the given AppID to stop on all chips on which it is running. It will be replaced with a default system application and all shared resources that it claimed during execution will be freed. A range of AppIDs may be given subject to certain restrictions. The number of AppIDs in the range must be a power of 2 and the lower AppID must be a multiple of that power of 2. Examples of valid ranges are `16-23`, `64-95`, `80-81`.

- `app_sig <region> <app_id.D>[-<app_id.D>] <signal> [<state>]` - sends a signal to all cores running AppIDs in the given range in chips in the given region. This feature is still in development but it allows signals similar to Unix signals to be sent to running applications and also allows counting how many cores in the application are in a particular execution state. For example to count cores running AppId 16 which are in state `RUN` and to count cores running AppID 0 which are in state `IDLE` (the default state)

  ```
  > app_sig all 16 count run
  > app_sig all 0 count idle
  ```

Some valid signals are

stop	Terminate application and clean up
start	Start an application when `wait` was given in `app_load`
sync0	Proceed past barrier (eg on API start-up)
count	Count cores in a given state

- `ps <core.D>|x|d` - displays the status of all cores on a chip or a single core. When used without arguments, the `ps` command displays the state of every core on the chip showing the application it is running, its state, how long it has been running and when it was loaded.

  With the argument `d` or `x` it displays the four user variables associated with each core in decimal or hex. This is a useful way of getting debugging and status information out while an application is running.

  With a numeric argument (a core number) the command displays a more detailed set of information about a single core. Where the core has crashed this will include a register dump which may help with diagnosing the problem.

Commands to load, dump, inspect and alter memory

These commands allow the memory of many chips or a particular chip or core to be displayed and altered. Before those commands which relate to a specific chip or core are issued, the appropriate chip and core must be selected (with the `sp` command). Some memory on a chip (eg SDRAM) can be accessed from any core whereas other memory (eg DTCM) can only be accessed if the appropriate core is selected.

Note that memory-mapped peripherals may also be accessed with these commands but great care must be taken to only access them in valid ways (ie using the correct data size and alignment) otherwise a system crash may result.

- `smemw <addr.X>` - displays 256 bytes of memory as a hex dump. The memory is loaded as 64 words starting at the supplied address and the display is organised in word format.

- `smemh <addr.X>` - displays 256 bytes of memory as a hex dump. The memory is loaded as 128 half-words starting at the supplied address and the display is organised in half-word format.

- `smemb <addr.X>` - displays 256 bytes of memory as a hex dump. The memory is loaded as 256 bytes starting at the supplied address and the display is organised in byte format. Where the byte has a printable ASCII representation, it is also shown as a character.

- `sw <addr.X> [<data.X>]` - displays (one argument) or sets (two arguments) a single word in memory at the given address (which should be word aligned).

- `sh <addr.X> [<data.X>]` - displays (one argument) or sets (two arguments) a single half-word in memory at the given address (which should be half-word aligned).

- `sb <addr.X> [<data.X>]` - displays (one argument) or sets (two arguments) a single byte in memory.

- `sload <file.F> <addr.X>` - reads a file and copies its contents into memory beginning at the specified address. The data is written as bytes so that any length of file may be used.

- `sdump <file.F> <addr.X> <len.X>` - reads the specified length of SpiNNaker memory starting at the given address and copies it to a file. The data is read as bytes. Note that the length is specified in hexadecimal.

- `sfill <from.X> <to.X> <word.X>` - fills SpiNNaker memory with the specified word, beginning at the specified `from` address and ending at the word below the `to` address. Both addresses should be word aligned.

- `data_load <file.F> <region> <addr.X>` - writes the content of the specified file to memory at the given address in all chips specified by `region`. This is useful if the same data has to be sent to many chips simultaneously and the data is going to a shared area of memory (ie not ITCM or DTCM). Writing to a single chip is more efficient using `sload`.

The next three commands are only available in `expert` mode.

- `gw <addr.X> <data.X>` - writes the specified data as a word to the given address on every SpiNNaker chip in the system. Note that only a limited set of addresses is supported, to allow access to some important peripherals and parts of memory. Probably not for everyday use but can be useful for debugging.

- `gh <addr.X> <data.X>` - writes the specified data as a half-word to the given address on every SpiNNaker chip in the system. See `gw` for further details.

- `gb <addr.X> <data.X>` - writes the specified data as a byte to the given address on every SpiNNaker chip in the system. See `gw` for further details.

Commands to control IPTags

An IPTag is a numeric identifier which is used to index a table in a chip with an Ethernet interface. Entries in the table contain an IP address and port number. They are used to direct SDP packets which arrive at the Ethernet-connected chip from within SpiNNaker bearing an IPTag to the appropriate IP address and port using the UDP protocol.

A typical use is to direct debugging output from `io_printf` functions executed on application cores to a host system which then displays the debug text. IPTag 0 is normally used for this purpose. The `iptag` command is used to control the IPTag tables.

- `iptag` - without arguments, the `iptag` command displays the contents of the IPTag table. Only valid entries are displayed. The table has two sections and the size of these sections is shown by this command as well as the timeout which is applied to transient IPTags. Lower entries in the table are permanent tags which are set up explicitly either by a host or by a SpiNNaker application. Higher entries are transient and only last for the lifetime of a SCP (command) transaction between the host and SpiNNaker. The number of packets which have passed through the IPTag since it was created is also displayed.

- `iptag <tag.D> set <ip_addr.P> <port.D>` - this form of the command sets an IPTag. It is usual for tag 0 to be pre-allocated to the controlling host with a port number of 17892 (the Tubotron port). The `ip_addr` can be a numeric address or a hostname in which case it will be looked up using DNS.

  The character `.` can also be supplied for the `ip_addr` in which case the IP address of the host on which **ybug** is running will be used. The IP address `0.0.0.0` will cause the source IP address of the UDP/IP packet which carries the `IPTag` command into SpiNNaker to be used. This is useful where the packet has been through address translation (eg NAT) en-route to SpiNNaker. Some examples.

  ```
  iptag 3 set 192.168.0.4 2222      Set IPTag 3, port=2222. IP=192.168.0.4
  iptag 2 set .  15555              Set IPTag 2, port=15555 IP = host IP address
  ```

- `iptag <tag.D> reverse <port.D> <dest_addr.X> <dest_port.X>` - this creates a reverse tag which forwards incoming UDP packets on the specified port to the SpiNNaker chip specified by `dest_addr` and `dest_port`. The UDP payload (which must be small enough to fit in an SDP packet) is copied into an SDP packet and delivered within SpiNNaker. When a packet arrives, a reverse path is set up so that a response can be sent by using the same IP tag.

      iptag 5 reverse 12345 0304 23   Set reverse IPTag 5, port=1234

  The example above will cause incoming UDP packets on port 12345 to be sent on to chip (3,4), core 3, port 1.

- `iptag <tag.D> clear` - this form of the command clears (removes) an IPTag table entry.

Debugging Commands

A number of commands are provided specifically to help with debugging and others are likely to be added on demand.

- `sver` - this command queries the currently addressed core to find out what operating software it is currently running. This will usually be SC&MP for monitor processors (core 0) and SARK for application processors. The information includes the version number of the software, the system it is running on (usually SpiNNaker) and the date it was built. The physical core number of the core is also shown (in square brackets).

- `led <0123>* on|off|inv|flip` - controls the LEDs attached to a selected SpiNNaker chip. Up to 4 LEDs can be turned on, off or inverted. Some SpiNNaker boards have fewer than 4 LEDs.

- `heap sdram|sysram|system` - displays the allocated and free blocks in one of the three shared heaps on the selected chip. With no argument, all three heaps are displayed. The Tag and AppID associated with each allocated block are shown.

- `iobuf <core.D> [<file.F>]` - dumps the contents of the IO buffer of the specified core. The IO buffer contains the output of `io_printf` functions which have been executed on the core. It is assumed that this is ASCII text and it is printed to the terminal. The IO buffers are allocated in the System Heap and are kept until the application is terminated with `app_stop`. If the `file` argument is given, the text is written to the file rather than the terminal.

Commands to control the Router

These commands control the router on the selected chip. They are mainly concerned with controlling the MC routing tables.

- `rtr_load <file.F> <app_id.D>` - loads the specified file into the router MC table and associates the given AppID with the set of entries that is created. It is assumed that the file is in the appropriate format though some rudimentary checks are performed to verify that this is the case. The number of entries that have to be allocated in the router table is determined from the file and a block of that size is requested. If this is successful then the file is loaded into the appropriate entries in the table.

- `rtr_dump` - displays all router MC entries which are currently allocated. The key, mask and route fields are displayed as well as the AppID and core associated with each entry. Note that entries allocated with `rtr_load` will show an AppID of 0. Their real AppID is stored in the router MC heap data structures (see below). Entries which were initialised by application code running on SpiNNaker will show the true AppID.

- `rtr_heap` - displays the router MC heap which lists allocated and free blocks and the AppID associated with allocated blocks. Note that the output will only be meaningful if applications have chosen to use the router MC heap. If they have allocated MC entries manually, the display may be misleading.

- `rtr_diag [clr]` - displays the router diagnostic count registers which were set up when the system booted. This shows the number of packets of various types which have passed through the router. If the `clr` flag is given then the counts are zeroed after the counts have been displayed. Thus, when the command is given again, the counts will refer to the period since the clearing command was given. If an application has modified the configuration of the count registers then the display is likely to be misleading.

Commands to control a Serial ROM

Some SpiNNaker chips have an SPI-based serial EEPROM (Serial ROM or SROM) attached to their GPIO port. This can be used to bootstrap the chip or to hold data which needs to be non-volatile. Chips which have an Ethernet interface use a Serial ROM to hold MAC and IP addresses. and **ybug** allows these parameters to be changed. This is applicable to Spin2, Spin3 and Spin4 boards. These commands are only accessible in `expert` mode.

- `srom_ip [<ip_addr.P> [<gw_addr.P> [<net_mask.P>]]]` - displays and/or sets IP addresses in the SROM. Without an argument this command displays the current IP and MAC addresses. With one argument it sets the IP address of the system. With a second argument it also sets the gateway IP address and with a third argument it also sets the net mask.

  Take great care to set these parameters carefully or you may not be able to communicate with the system after the changes have taken effect. The SROM is read when the system bootstraps so changes made by this command will not take effect until the system is reset.

- `srom_init` - completely initialises an SROM which contains IP addresses. Use `srom_init ?` to get a list of the arguments. This command is probably not required by most users.

- `srom_read [<addr.X>]` - reads and displays as a hex dump the contents of the SROM at the specified address. Useful for debugging SROM problems but probably not required by most users.

- `srom_write <file.f> <addr.X>` - writes the contents of the given file into the SROM at the specified address. Again, one for experts only and likely to be useful if you want to create your own custom bootstrap SROM.

- `srom_erase` - completely erases the SROM (all bytes are set to `0xff`). Only do this is you are sure you know what you are doing!

SpiNNaker Board Control Commands

The 48-node SpiNNaker boards (Spin4 and Spin5) have a Board Management Processor (BMP) which has its own network interface. The BMP controls the overall operation of the board and provides facilities to reset the SpiNNaker chips. A single BMP can control all of the SpiNNaker boards in a sub-rack.

ybug provides an interface to the BMP for this purpose. The IP address of the BMP must be specified on the command line with the `-bmp` flag. The IP address (or host name) must be followed by a "/" character and then a range of slot numbers which represent the various SpiNNaker boards which are being used in the current **ybug** session. Typically this will just be a single board. For example

```
ybug 192.168.240.25 -bmp 192.168.240.0/3-5
ybug spinn-9 -bmp spinn-9c/0
```

The first command indicates that the controlling BMP is at IP address `192.168.240.0` and the SpiNNaker systems that should be controlled in the **ybug** session are in slots 3 through 5 of the sub-rack. The second command indicates that the controlling BMP has hostname `spinn-9c` and that the SpiNNaker system that is being controlled is in slot 0 (ie on the same board as the BMP). This is the common case of a single Spin4 or Spin5 board.

In the situation where many SpiNNaker boards in a sub-rack are being used as individual systems, it is important to correctly specify which boards are being controlled by the BMP. Otherwise, the wrong systems can end up being reset!

- `reset` - resets all of the specified SpiNNaker systems. This applies a hard reset to all SpiN-Naker chips on each controlled system. It will be necessary to boot those systems after this command is given.

- `power on|off` - turns the power to the SpiNNaker systems on or off. This can be used to put the system into a low power mode when it is not being used. Turning the power on also applies a reset to the SpiNNaker chips. Where the system has SpinLink FPGAs (Spin4 and Spin5 boards), these are also put into power-down mode when the power is turned off. Their configuration is reloaded when they are powered on again.

Change log:

- *1.20 - 18aug13 - ST* - initial release - comments to *temples@cs.man.ac.uk*

- *1.30 - 02apr14- ST* - update for 1.30

- *1.33 - 19sep14- ST* - update for 1.33 - IPTag changes, String::CRC32 dependency.