Generic Simple OSS Interface

October 26, 2013

1 Introduction

In our discussions with Wlodek, we emphasized several times the need for a simplification of the praxis-side OSS interface understood as the part responsible for interpreting commands and their arguments, mostly (but not exclusively) arriving to the node via UART, as well as sending responses to the OSS. The old interface (or rather set of interfaces) used so far in production praxes is character-oriented, which means that data has to be un-formatted (converted from ASCII to binary) on its way into the praxis and formatted on its way back to the OSS. That was done by the node and required memory and time.

With the introduction of *piter* and its *plugins* (see the *Serial* document), there has been no need to maintain the human-friendliness of OSS data at the praxis's end, even if that data is entered/interpreted directly and manually by a human operator via a character terminal. Thus, we should focus on making the low-level interface most friendly from the node's point of view. Especially for CC430, where flash memory is more limited than on MSP430F1611, getting rid of the costly formatting (un-formatting) of the data looks like a much needed step in the right direction. Note that barring some truly exceptional cases (like when the node has to talk directly to some exotic, non-programmable equipment) there is absolutely no need for the node to deal with text parsing, character representation of numbers, etc.

Moreover, we should be aiming at the UART packet interface to be used at the bottom of OSS communication. That will help us get rid of the legacy *device* concept of PicOS, whose sole present representative is the abominable UART. While, at first sight, it might seem that the packet interface introduces more overhead than the old (direct/line) interface, we should remember that the largest component of the packet interface (i.e., VNETI/TCV) is unavoidably needed for RF communication, so the actual overhead for UART communication is in fact quite modest. One of our old arguments for sticking to the character-oriented line interface has been the occasional need to inter-operate with custom equipment. So far, that has happened only once: EcoNet satnode talking to a satellite box. Situations like that can (and should) be viewed as special cases, because the set of commands is necessarily going to be quite idiosyncratic to the equipment. Such cases can be handled by software adapters, if we want to keep the praxis unaware of those idiosyncrasies, and the special equipment can be interfaced to the network via the OSS. If the node really has to talk directly to the equipment, it will need a special program, but there isn't really a way around that. Note that one option of the packet interface (mode-L) provides for transparent line-like communication on the OSS side (so the principal unification of the "bottom layer" [and elimination of the old device-like interface] won't spoil anything).

1.1 A few assumptions

The general idea behind the new interface is to make it entirely binary and as simple as possible to interpret on the node's side, while providing for potentially unlimited flexibility at the OSS program's end.

1.1.1 The low-level packet scheme

The first question (which has been asked a few times before without a clear answer) is this: do we want to base the mechanism on the simple mode-N packet scheme, or, perhaps, we would be better off with a reliable scheme (mode-P or BOSS)? Judging by the lack of enthusiasm in adopting reliable UART communication schemes so far, I believe that we want to use N-type packets and nothing more. While BOSS has been useful in some circumstances (I used it in the SA and REFLOW praxes, which are not what we would consider our main line), it does not immediately look beneficial to our “standard” praxes. The N-mode UART-as-a-PHY is the simplest packetized scheme for the UART. It comes with built-in CRC checking (obviating the need for application-level parity control) which can be easily switched off (or replaced by something simpler), if desirable.

1.1.2 The new script versus piter

Having settled that low-level interface part, the next question was whether to implement the user-end interface as a *piter* plugin or as a stand-alone script. I have opted for the latter for these reasons:

1. The simplicity of view by the user. The way to start the OSS program should be very simple and natural, possibly devoid of any options. The program should be associated with the praxis, i.e., the praxis-specific components should be kept in the praxis directory, while the generic part should be shared and accessed in the same way by all praxes.
2. The generic component of the new OSS interface would look like a *piter* plugin itself, with its praxis-specific items appearing as a plugin to the plugin. That would be confusing and complicated. One possibility would be to simply extend *piter* by the generic component (which we are always free to consider), but that would necessarily add to *piter's* options (making it more complicated).
3. There wouldn't really be too many advantages from incorporating the tool into *piter*. First, it wouldn't simplify the programming. Second, it wouldn't even prevent replication of functionality (because the essential components are already modularized into Tcl packages, and used by *piter* as well).

Consequently, I decided to write a separate script. Its present GUI does look like *piter's* terminal, but it can be changed (upgraded, extended, parametrized) once we decide that we are comfortable with the general idea. *Piter* will be retained as our own general-purpose terminal emulator (incorporating all the communication options described in the *Serial* document), while the new script will be used to implement convenient, specific OSS interfaces for specific praxes.

1.1.3 The language of specification

The third design consideration was the description language for the praxis-specific components of the interface (as well as the demarcation line between the generic and praxis-specific parts). The obvious suggestion was to use XML, and I spent a considerable amount of time trying to come up with a set of XML constructs that would allow us to describe flexible collections of commands and arguments that we might find useful for the various praxes. All those attempts have led me to the same dead end: insufficient dynamics of the description.

For illustration, consider one of the hypothetical commands of my new TEST praxis which I would expect to replace the present mess of several single-letter commands referring to the same *component*, i.e., the RF function of the test. Here is a sample list of parameters that the command may want to affect:

1. Radio status: off, TX on, RX on, TX and RX on, WOR on, WOR and TX on
2. Channel number
3. Interval between transmissions: fixed or randomized, one number or two numbers: min and max
4. Transmission power
5. Packet length: again fixed or variable (randomized)
6. Optional content of the packet (some bytes to send)

Note that, even with moderate postulates regarding the flexibility and user convenience, the types of the parameters and their constraints (which a decent interface should verify and enforce) are difficult to describe in static XML constructs. Here are two examples how those commands might be entered by the user:

```
radio -state tx+wor -channel 3 -interval 256-3072 -length 16
radio -state tx -channel 0 -interval 4096 -length 12-48 "Baca is OK"
```


Note that the *-state* argument is most naturally specified symbolically with a number of rather nontrivial options, e.g., *rx+tx* makes sense while *rx+wor* or *tx+off* does not. Listing these options may still be fine in an XML element, e.g.,

```
<argument type="byte" ...>
       tx=1,rx=2,wor=4,tx+rx,tx+wor,on=3,off=0, …
</argument>
```

but it doesn't look general. Besides, we may not want to confine the command (and argument) format to some preselected style (as a predefined XML element would force us to). Also note that generally code is needed to assess the correctness of an argument list (static XML description can only offer a selection from a predefined set of options). For example, the second argument of *-interval* (or *-length*) must not be less than the first argument. Also, if the *-state* argument is *off* (or doesn't include *tx*) then, perhaps, it should be illegal to specify transmission-related attributes within the same command. There is no way to build such descriptions with static tags without effectively creating some kind of a programming language. So why bother, if we already have a decent programming language handy, namely *Tcl* in which the generic part of the script is written?

Consequently, the praxis-specific part of the interface is described in *Tcl*. Some elements of that description look like declarations (command and message layouts), some other are pieces of code describing the ways to parse textual commands or the dynamic relationships between the various components of commands and/or messages.

1.2 How it works (in a nutshell)

There is one generic script, named *oss*, which we may want to put into the user's *bin* directory, akin to *pip*, *piter*, *udaemon*, etc. When called, that script expects a praxis-specific description of the interface. By default, that description is looked up in a file named *ossi.tcl* (in the current directory) but the path can be specified explicitly as a direct argument to the script (it doesn't expect any other arguments at present). The script reads the description, verifies it, and opens a terminal window with a few extra buttons.

One function (which at present is available from the terminal window, but may be turned into a separate script) is to retrieve from the description a header file to be included by the praxis program. That header file contains declarations of the data structures (as well as some constants) that the praxis program can use to interpret the contents of command packets arriving from the OSS program and/or prepare message packets going in the opposite direction. This way the single description is shared by the OSS script as well as the node program.

The global script implements *autoconnection*, i.e., the user need not try to guess the identity of the UART (COM port) over which the praxis node is available. At present, there isn't even an option to specify a particular port (we shall see if such an option is needed). The only option regarding connection is real UART versus VUEE. Even though in principle connection to a VUEE model could be made automatic (and incorporated into the general scheme), the specific node to connect to cannot always be guessed by the script. In production editions of the script (when merging it with a specific description) we may remove the VUEE selection dialog thus rendering the connection procedure completely transparent.

2 The specification language

The praxis-specific description is provided in a single specification file named *ossi.tcl* by default and sought in the current directory, i.e., in the directory where the global *oss* script is invoked. If an argument is specified to *oss*, then it is interpreted as the file path overriding the default location of the specification file. No other arguments are interpreted by the script at present.

It is possible to merge the praxis-specific file with the global script and, e.g., turn the resulting combo into a stand-alone executable. It will be easy to provide a script for creating such executables from specifications.

The specification itself is also just a *Tcl* script which may call some functions provided by the global and define its own functions amounting to a command parser. Basically, a specification consists of three parts: *signature*, *data*, and *code*.

2.1 The signature

This is a single statement looking like this:

```
oss_interface -id pxid -speed urate -length bl -parser pfuncs
```

where the arguments need not occur in any particular order. Their interpretation is as follows:

pxid	This is the so-called praxis ID, which is a 32-bit integer used to identify the praxis for autoconnection.
urate	The UART rate used by the node. Note that the rate is not auto-detected, i.e., from the viewpoint of the autoconnection procedure it can be viewed as an extension of the praxis ID (two praxes with the same ID but different rates won't be confused).
length	This is the maximum length of the OSS packet which should be equal to the second argument of *phys_uart* on the node's side. It covers the complete size of a command or message. Note that the Network ID field of the VNETI packet is treated as regular payload.
pfuncs	This is a pair of function names (*Tcl proc*) pointing to the user processor for commands (the first function) and messages (the second function). These functions must be defined somewhere in the specification file.

It is mildly recommended to make the signature statement the first statement of the specification (it doesn't really matter at present, but we may add something later to the signature that will have to be known to subsequent declarations). Not all arguments must be specified. In particular:

- The default praxis ID is *0xFFFFFFFF* (i.e., –1). Of course, it doesn't make a lot of sense to use the default ID for all praxes.
- The default UART rate is 9600 bps.
- The default maximum length of an OSS packet is 82 bytes.

It is also legal not to specify the parser functions, which basically means that the entire signature statement is optional (at present). Simple default functions are provided by the global script (see Section 3).

2.2 Commands and messages

A command is a packet directed from the OSS program to the node, and a message is a packet going in the opposite direction. Every command and message starts with a header consisting of two bytes. Here is the data structure describing that header:

```
typedef struct {
      byte code, ref;
} oss_hdr_t;
```

The *code* attribute identifies the command/message. There is no implied relationship between the codes of commands and messages. The *ref* attribute is an additional (and generally optional) feature of the command/message. For every command, the global script sets the *ref* attribute automatically to the previous value + 1 modulo 256 skipping

zero, i.e., for subsequent commands *ref* changes as 1, 2, 3, …, 254, 255, 1, 2, … and so on. This way *ref* can be viewed as a command version number (modulo 256 or so). If desirable, the node program can set *ref* in its messages to match those in commands as a way of acknowledging or simply relating the messages to specific commands. The user-provided message parser can then take appropriate actions. No standard interpretation of message *ref* is built into the global script.

For a command, code 0 is reserved for the built-in *autoconnect* probe, i.e., the query (automatically) sent by the script to check if the right program is listening on the other end of the UART. The complete probe command consists of 6 bytes, i.e., the two header bytes + the four bytes of the praxis ID.

For a message, code 0 is reserved for a "pure" acknowledgment message or for a response to the autoconnect probe. A pure acknowledgment message consists of four bytes: the header + one status word. It makes sense to assume that status = 0 means OK, while any other value indicates a problem. The *ref* attribute of a pure ACK message should match the *ref* value from the respective command. Note that it can never be zero.

The global script does not assume any specific acknowledging strategy. For example, if a command naturally triggers a reverse message (say, the command is a query), it may make sense to set the *ref* attribute of the message to that of the command. In any case, the global script does not presume anything like that, except for the expected response to an autoconnect probe.

Such a response should have both header bytes equal zero (*ref* = 0 makes it different from a pure ACK) with the header followed by a *word* being the XOR of the two *words* of the praxis ID. The node should only respond to an autoconnect probe, if its praxis ID matches the value that has arrived in the probe.

Even after a successful connection, the global script will be sending autoconnect probes to the node periodically (at 2s intervals) to make sure that the (same) praxis is still present at the other end alive and well. This is heartbeat detection.

The data part of the praxis-specific file declares the layout of commands and messages in a manner resembling *struct* declarations in C. Here is a sample declaration of a command layout:

```
oss_command radio 0x01 {
        byte status;
        byte power;
        word channel;
        word interval [2];
        word length [2];
        blob data;
}
```

The declaration is effected by invoking *oss_command* (a *Tcl proc*) provided by the global script). The first argument is the command name interpreted as a symbolic identifier for the layout (it need not associate with the command's keyword as entered by the user). The second argument is the command code, i.e., the content of the *code* byte in the command header. Structure names and codes must be unique, but the codes and names of commands may overlap with those of messages (see below).

The contents of the last argument of *oss_command* resemble a *struct* declaration from C. Only these types are legal: *word*, *sint*, *lword*, *lint*, *byte*, *char*, and *blob*. The difference between *word* and *sint* (also *lword* and *lint*, as well as *byte* and *char*) is that the former is unsigned while the latter is signed. Simple (single-dimensional) arrays are OK (the dimension must boil down to a small constant expression that can be evaluated by *Tcl*).

Type *blob* is declared as follows:

```
typedef struct {
        word size;
```


```
        byte content [];
    } blob;
```

Its role is to describe strings of bytes of varying length. At most one *blob* can appear in a command structure and only as the last attribute. The idea is that the offsets of all attributes in a command packet must be known statically.

Message layouts are specified in exactly the same way (with exactly the same rules) as command layouts. The only difference is in the name of the operation. Here is a sample message layout:

```
oss_message status 0x01 {
        byte rstatus;
        byte rpower;
        byte rchannel;
        word rinterval [2];
        word rlength [2];
        word smemstat [3];
        byte spower;
}
```

Note that names and codes must be unique among messages (and among commands), but a message and a command may be assigned the same name and/or code.

The size of a command (or message) layout (viewed as a C structure) determines the size of the respective command (message) packet. Two header bytes will be inserted at the beginning of the block and all those bytes will be sent in a packet as payload. If there is no blob in the layout, then the size of the command/message is fixed and equal to the structure size + 2. The maximum packet length specified in the signature (Section 2.1) must be no less than the maximum of those numbers over all layouts. If there is a blob, then the minimum size of the command/message block assumes that the blob is empty, i.e., it takes just one word (for the size attribute). Needless to say, the maximum packet size should then account for the maximum length blob in the largest layout.

For calculating layout sizes and the offsets of their attributes in the blocks, you should remember that those attributes are aligned based on the size of their types. For illustration, consider this layout:

```
oss_command run 0x10 {
        byte how;
        word time;
        byte when;
        lint tag;
}
```

Here are the sizes and offsets (in bytes) of the four values assuming that the first location after the command header (the offset of the first attribute) is 0:

how	size 1, offset 0
time	size 2, offset 2
when	size 1, offset 4
tag	size 4, offset 8

So the total length of the attribute block is 12 bytes. Note that it is also the length of the corresponding structure seen by the node program. This way, when the block is cast to the structure, the arguments will be properly matched to the structure's attributes.

2.3 The node's view

One function of the global script is to generate from the specification a C header to be used by the node program to view the command and message packets according to the prescribed layouts. At present, there is a special button in the script's terminal window, labeled *Hdr* triggering this action. This is how such a header looks:

```
#include "sysio.h"
```


```
#define OSS_PRAXIS_ID      65537
#define OSS_UART_RATE      9600
#define OSS_PACKET_LENGTH  82

typedef struct {
       word size;
       byte content [];
} blob;

typedef struct {
       byte code, ref;
} oss_hdr_t;

// ==================
// Command structures
// ==================

#define command_radio_code   0x01
typedef struct {
       byte   status;
       byte   power;
       word   channel;
       word   interval [2];
       word   length [2];
       blob   data;
} command_radio_t;
...
// ==================
// Message structures
// ==================

#define message_status_code  0x01
typedef struct {
       byte   rstatus;
       byte   rpower;
       byte   rchannel;
       word   rinterval [2];
       word   rlength [2];
       word   smemstat [3];
       byte   spower;
} message_status_t;
...
```

The three symbolic constants at the top reflect the signature parameters of the specification (see Section 2.1). The two structures defined next are fixed elements of the header file (they look the same in all headers generated by the script). They are followed by the list of structures directly corresponding to the specification layouts for commands and messages (Section 2.2). Note that the name of a command/message structure is derived from the name in the specification by inserting *command_* (or *message_*) in front and adding *_t* at the end.

There are no other inserts or libraries intended for the node program. I have considered creating some, but found each attempt unnecessarily restrictive and pointless. For example, as pointed out by Wlodek, we should not assume that all commands will come in packets with the same kind of low-level headers and, generally, purpose. Some commands are going to be forwarded over RF via proxy nodes whichmay place them at different offsets from the packet's beginning. In most cases, the command will arrive in some kind of buffer coinciding with a packet fragment, but, in some conceivable scenarios, it does not have to. For example, we may have initialization (reset) commands stored in flash (or EEPROM). In my first attempt at offering the node program some advance assistance, I provides a few library functions wrapping VNETI packet

operations in the code (i.e., the command handling function would assume that its argument is a packet buffer). In the second attempt I removed those operations, but still left the (rather trivial) action of extracting the command code and using it as an index into a table of functions. Then, when trying to use those simple aids in a program, I found them too presumptuous and, in a nutshell, unnecessary.

From the viewpoint of the node program the issue is quite simple: a command arrives in some buffer and needs to be interpreted. By casting the proper fragment of the buffer (be it a VNETI packet or a chunk of flash memory) to the right structure, the program gets an immediate view of all the arguments of the command. It is now the matter of inspecting those values and performing the proper action. Not much generic help can be offered to simplify these rather trivial tasks.

The same applies to sending out a message. The node must allocate the buffer, cast it to the message structure, fill the content and dispose of the buffer. The first and last actions can only be accomplished by the node program; the middle one is trivial and needs no library assistance.

2.4 The code

Being a script that is just read and evaluated by the *Tcl* interpreter within some context, the specification file may include functions and variables. If the parser functions are declared in the signature (Section 2.1), their definitions (as *proc*) must be present somewhere in the specification.

Note that in addition to commands directed to the node, you can also enter from the script's window *Tcl* statements which will be executed in the same *namespace* as the functions provided with the specification. That way you can augment your parser with settable variables, affect the behavior of some parser functions, and accomplish anything that can be expected from a powerful shell.

3 Default parsing

By parsing we understand transforming commands entered by the user into command packets sent by the script to the node. The second action carried out by the script consists in reverse parsing, i.e., transforming message packets arriving from the node into presentable forms shown to the user. For brevity, we shall use the same word, “parsing”, for both actions, like in “parser functions” (see above), understanding that, strictly speaking, only one of those functions does parsing, while the other takes care of the reverse action.

A minimalistic (sensible) specification consists of a signature, at least one command, and at least one message. For example, it may look like this:

```
oss_interface -id 0x00010001

oss_command system 0x02 {
       word spin;
       word leds;
       byte request;
       byte blinkrate;
       byte power;
       blob diag;
}

oss_message status 0x01 {
       byte rstatus;
       byte rpower;
       byte rchannel;
       word rinterval [2];
       word rlength [2];
       word smemstat [3];
       byte spower;
```


```
}
```

Note that the script will assume that the UART rate is 9600 bps and the maximum OSS packet length is 82 bytes. No (user) parser functions are provided, so the script will use default ones.

3.1 Command parsing

With the rudimentary, default command parsing, a command is recognized by the name of its structure (layout), interpreted as a keyword to be entered by the user and followed by the parameters prescribed by the structure (in the structure's order). If there is a blob in the structure (which, if present at all, must appear as the last attribute), then it is optional. All other arguments are mandatory. For example, the *system* command (from the minimalistic specification in the previous section) may be entered like this:

```
system 0 0xFFFF 1 + 1 2,1 "baca is sick"
```

This is how the arguments are evaluated: *spin* = 0, *leds* = 0xFFFF (i.e., 65535), *request* = 2 (1+1), *blinkrate* = 2, *power* = 1, *diag* = 98 97 99 97 32 105 115 32 115 105 99 107 0 (these are the ASCII codes of the characters in the string; the size attribute of the blob is 13 and the sequence ends with a zero byte).

At first sight it may seem intriguing how the parser is able to separate the third and fourth arguments, i.e., 1 + 1 from 2 (note that the spaces around + are fine). The standard procedure of parsing a number assumes that the number can be represented by any expression evaluable by *Tcl*. Thus, when expecting a number in the input line, the parser will grab the longest substring (beginning at the current position) that can be evaluated as an expression. This way, 1 + 1 is OK, but the subsequent space and a digit cannot be accommodated into the expression.

If in doubt, the values can be separated by commas. For example: 2 +1 will be parsed as a single value, while 2, +1 will result in two.

The default way to parse a *blob* depends on whether its first non-blank character is " or not. In the former case, a matching " is expected and everything in between will be parsed as a character string obeying UNIX escaping conventions. The ASCII codes of the string characters will be put into consecutive bytes of the blob's *content*. A zero sentinel byte will be added at the end, unless the string already has one (its last character is \0).

If the blob argument does not begin with ", then the parser expects a list of numerical values (as for regular arguments they can be expressions). Each value is truncated to one byte and stored at the respective blob position. The total number of values found in the remainder of the input line determines the blob's size. Note that no arguments can follow a blob.

Once all the arguments have been successfully processed, the message is assembled according to its layout. The *ref* value in the header is set by the script automatically to the previous value + 1 with 256 transformed into 1.

Note that a blob argument can be empty (i.e., it is optional). Such a blob has the size of 0 and no content (it still takes two bytes in the block).

3.2 Messages

The default processing of a message arriving from the node to the script begins by extracting the message *code*. If the *code* is zero and the *ref* attribute of the message header is also zero, the message is assumed to be a response to the autoconnect probe (also known as a heartbeat). Even while connected, the script will periodically issue autoconnect probes to make sure that the node is still there.

Heartbeats are always processed internally (also when the specification file provides non-default parser functions), so the user never sees them.

A zero-*code* message with nonzero *ref* is deemed a “pure” ACK. Its default processing consists in presenting this text:

```
ACK rr ssss
```

where the two hexadecimal numbers that follow *ACK* are the *ref* and *status* values. Recall that a pure ACK message carries a 16-bit status value Section 2.2).

For a non-zero *code*, the script looks up the message's specification layout. If none is found, i.e., the message's structure is not known to the script, the message is dumped as a sequence of bytes, e.g.,

```
Message [03 12], no layout found
Content: 0x00 0x07 0x00 0x2b 0x00 0x04 0x00 0x04 0x20 0x00
         0x20 0x00 0x8a 0x03 0x8a 0x03 0x00 0x01 0x01 0x00
```

The numbers in square brackets are the message's *code* and *ref*. *Content* is a byte dump of the remaining portion of the message packet.

If a layout is present, then the message block is interpreted according to the layout in the most natural way. Here is the view of a *status* message from Section 2.2:

```
status <13>: rstatus=0 rpower=7 rchannel=0 rinterval=[1024 1024]
             rlength=[32 32] smemstat=[906 906 256] spower=1
```

The value following the message name (*status*) is the *ref* attribute of the message.

4 Programming parser functions

If the signature of the specification file (Section 2.1) defines parser functions, they take over from the default parser. Each of the two functions can be defined independently, e.g.,

```
oss_interface -id 0xFE110001 -parser { my_cmd_prsr }
```

only defines the command parser, while:

```
oss_interface -id 0xFE110001 -parser { "" my_msg_pcsr }
```

only defines the message processor.

The signatures of the two functions are:

```
proc my_cmd_prsr { line } { … }
```

and

```
proc my_msg_pcsr { code ref msg } { … }
```

In the first case, the argument is the input line (string) entered by the user in the input widget of the script's terminal window. For the message processor, the arguments are: the *code* and *ref* attributes from the message header, and the message data block, i.e., the contents of the message packet following the header passed as a (binary) Tcl string.

4.1 The toolkit

The global script provides a number of built-in functions facilitating the implementation of non-default (custom) parser. The names of all those functions begin with *oss_*. So far we have seen three library functions: *oss_interface*, *oss_command*, and *oss_message.*

4.1.1 Conversions and transformations of data

Owing to the fact that information is sent to (received from) the node in packets carrying binary data, while the script mostly deals with a symbolic representation of the data, we often need to convert from one form to the other. Binary data is represented by strings interpreted as blocks of bytes. There exist functions converting between such blocks and symbolic representations.

```
proc oss_bintobytes { block } { … }
```


This function takes a string as the argument, interpreted as a block of binary bytes, and transforms those bytes into symbolic hexadecimal representations in the form 0x*dd* (where *d* is a hexadecimal digit). The values are returned as a Tcl list whose length is exactly the same as the number of bytes in the block (the length of the argument string).

```
proc oss_blobtovalues { blob } { … }
```

The function takes a (binary) block of bytes interpreted as a blob and returns a list of byte values stored in the blob. The first two bytes (word) is interpreted as the blob's size. If the number of remaining bytes in the block is less than the size, an error (data too short) is triggered. Otherwise, that many subsequent bytes are transformed into 2-digit hex values of the form 0xdd. Those values are returned by the function as a list. Note that a blob can have zero size (in which case the function will return an empty list), but even then the argument to *oss_blobtovalues* must consist of at least one word (containing zero).

```
proc oss_blobtostring { blob } { … }
```

The function takes a (binary) block of bytes interpreted as a blob (as for *oss_blobtovalues*) and returns a string represented by the blob. It is to be used for those occasion when the blob is known to carry a string. The action consists of removing the size word and truncating the remaining portion of the block to *size* bytes. An error is triggered (data too short), if that portion does not have enough bytes. If the string is terminated by a zero byte on the right, that byte is removed. The function returns the extracted string.

```
proc oss_getvalues { block msgid { bstr 0 } } { … }
```

This function performs the composite action of extracting values from a message received from the node. The first argument is a string interpreted as the message block (a binary sequence of bytes) with the header removed (i.e., the first byte of the string is the third byte of the complete message).

The second argument is the message identifier, which can be the name or the code (Section 2.2). The last (optional) argument is a flag indicating whether any blob found in the message should be interpreted as a string (by default it interpreted as a list of values).

The function gets hold of the layout of the specified message and uses that layout to interpret the binary chunks in the message data block. The returned list consists of as many elements as many attributes there are in the message layout. A blob is a single item in the list: it can be a string (if *bstr* is nonzero) or a list of values, otherwise.

The function throws an error (command not found) if the second argument does not identify a message layout known to the script. An error is also triggered if the data block does not match the message layout, e.g., is too short.

```
proc oss_getmsgstruct { msgid { nam "" } { cod "" } { len "" } }
                                                        { … }
```

This function locates the message layout identified by the first argument (the name or the code) and returns a list of records (lists) identifying the consecutive components of the layout.

Each of the last three arguments is optional. If not empty, the argument should be a variable name to return, respectively, the command's name, code, and the length of the data block. If the layout includes a blob, the minimum length is returned, assuming that the blob size is zero.

One item of the list returned by the function is itself a list of these elements:

- *type* (which can be one of *lword*, *lint*, *word*, *sint*, *byte*, *char*, *blob*)
- *name* (which the name assigned to the attribute)

- *tsize* (which is the type-implied size of the attribute: *lword* and *lint* : 4, *word* and *sint* : 2, *byte*, *char*, and *blob* : 2; note that the amount of space occupied by a blob may vary, so its size is formal)
- *dimension* (which is zero for a scalar; note that a blob cannot be an array, so its dimension is always zero)
- *offset* (which is the byte offset of the attribute from the beginning of the data block)

If the specified layout identifier does not point to a message layout known to the script, the function returns an empty string.

```
proc oss_bytestobin { vals } { … }
```

This function takes a list of *Tcl* values as the argument and transforms those values into a block of binary bytes which is returned as the function's value. The specified values are interpreted as expressions that must evaluate to integers. They are (quietly) truncated to 8 bits and stored into the bytes. The length of the returned block (string) is equal to the length of the list of values.

```
proc oss_valuestoblob { vals } { … }
```

This function takes a list of values as the argument and transforms those values into a blob (a block of bytes) which is returned as the function's value. The first two bytes (word) of the block contains the blob *size*, i.e., the number of values in the argument list which also equals the number of the remaining bytes in the block. Those remaining bytes are produced by invoking *oss_bytestobin*.

Formally, the maximum blob size is 65535 bytes. If the length of the list of values is greater than that, only the first 65535 values from the list are written to the blob. In reality, the blob size is constrained by the limitation of packet length (Section 2.1) and how much room in the packet has been taken by preceding attributes.

Note that the argument list can be empty in which case a zero-length blob is created.

```
proc oss_stringtoblob { str } { … }
```

The function takes a string as the argument and transforms it into a blob which is returned as the function's value. If the string does not end with a zero sentinel, one is added as the last byte. The blob size is equal to the string length.

If the argument string is longer than 65535 characters (including the sentinel), only the first 65534 characters of the argument string + the sentinel (which is always present) will be stored in the blob.

```
proc oss_setvalues { vals cmdid { bstr 0 } } { … }
```

This function performs the composite action of transforming the list of values specified as the first argument into a command data block, i.e., the command part following the header. The second argument identifies the command layout (it can be the name or the code), and the last (optional) argument is a flag indicating whether any blob expected by the layout and specified in the list of values should be interpreted as a string (by default it is interpreted as a list of values).

The function gets hold of the layout of the specified command and uses that layout to fill in the binary chunks in the command data block by the values from the specified list. The function returns a block of bytes that should be appended to a header to comprise a complete message packet. A blob is specified as a single item in the list of values: it can be a string (if *bstr* is nonzero) or a list of byte values, otherwise.

The function throws an error (command not found) if the second argument does not identify a command layout known to the script. Errors are also triggered if the command layout does not match the list of values or an illegal value occurs in the list.

```
proc oss_getcmdstruct { cmdid { nam "" } { cod "" } { len "" } }
                                                        { … }
```

This function locates the command layout identified by the first argument (the name or the code) and returns a list of records (lists) identifying the consecutive components of the layout. If the layout is not found, the function returns an empty string.

The interpretation of the remaining arguments as well as the value returned by the function is exactly as for *oss_getmsgstruct* (see above).

4.1.2 Parsing strings

The most powerful of the library functions available to the command parser is *oss_parse*. The arguments of *os_parse* are *selectors*, represented by keywords preceded by – (akin to the *Tcl* convention for internal procedures) indicating actions to be performed by the function. The actions indicated by the selectors are carried out in sequence from left to right. The first action that fails aborts the processing (the remaining actions are not performed). Typically, the function returns a list of items extracted from the input line. When the function (any of its actions) fails, the returned value is an empty string. There is no explicit limit on the number of selectors in a single invocation of the function.

Some selectors take specific arguments. Sometimes those arguments are optional. The presence or absence of an optional argument is determined based on whether the item following the selector begins with –. Also, when the selector is the last item on the list, then, clearly, it has no argument.

Here is the list of selectors in alphabetical order: *-checkpoint, -match, -number, -restore, -return, -save, -skip, -start, -string, -subst, -then*. They can be arbitrarily abbreviated as long as the abbreviation uniquely identifies a selector. Here are the minimum abbreviations for the selectors (in the above order): *-c, -m, -n, -res, -ret, -sa, -sk, -sta, -str, -su, -t*.

Typically, the first selector ever applied when parsing a line is *-start*. It must be followed by an argument which is the input string for the parser. Thus, this call:

```
oss_parse -start $line
```

initializes the parser with the contents of variable *line* as the new string to process.

Selector *-match* looks up a regular expression in the parsed string. The selector must be followed by a regular expression, e.g.,

```
oss_parse -match {^-([[:alpha:][:alnum:]*)}
```

With the above call, the function will try to match a sequence consisting of a minus sign followed by a letter, followed in turn by an optional sequence of letters and/or digits, at the beginning of the current string. If it succeeds (i.e., such a sequence does in fact occur), the function will return a list consisting of two items: the complete matched string and the keyword (the part marked by the parentheses in the expression). Note that this is exactly what *regexp* would return when called with the specified pattern and the input string as the arguments. One difference with respect to pure *regexp* is that our function removes the matched sequence from the input string.

If the looked up sequence is permitted to include spaces in front, you can modify the pattern, e.g.,

```
oss_parse -match {^[:blank:]*-([[:alpha:][:alnum:]*)}
```

or do something like this:

```
oss_parse -skip -match {^-([[:alpha:][:alnum:]*)}
```

The *-skip* selector tells the function to skip all white space characters (if any), removing them from the front of the input string, before proceeding to the next selector. The difference between the two invocations of *oss_parse* is that if *-match* fails, then, in the

first case, the input string will be left completely intact, while in the second case it will be stripped of any initial white space characters (because the failure will concern the second selector only; the first one always succeeds). Another difference is that the list returned by the function in the second case (assuming *-match* has been successful) will consist of three items: the *-skip* selector causes the function to return the first non-skipped character as the first item. The idea is that those selectors that return something (not all of them do) append their items to the list. However, if any selector fails, then the entire list is cleared (the function returns an empty string), even if some preceding selectors have managed to accomplish their goals.

If you want to make sure that a failure of a selector (preceded by some other selectors) does not cause the input string to be modified at all, use *-checkpoint*, e.g., in this case:

```
oss_parse -checkpoint -skip -match {^-([[:alpha:][:alnum:]*)}
```

the input string is guaranteed to be completely untouched if *-match* fails. You can insert *-checkpoint* anywhere into the sequence of selectors: a subsequent failure of a selector will leave the input string in the state from the last checkpoint.

As we said earlier, some selectors, e.g., *-match*, always require an argument. Some other selectors, e.g., -checkpoint, never require an argument. Some selectors, e.g., *-skip*, take an optional argument (which can never begin with –), In the case of *-skip*, the argument can be a string of characters to be skipped, e.g.,

```
oss_parse -skip " \t," -match {^-([[:alpha:][:alnum:]*)}
```

will cause the function to skip any blanks, tabs, and commas before proceeding with *-match*.

Note that unless the pattern of *-match* begins with ^, the regular expression will be matched anywhere in the input string, not necessarily at the beginning. If found, it will be removed from the input string, but any characters preceding it will be left. So you can use *-match* to yank strings from the middle of the parsed string. For example, suppose that this string is being parsed:

```
radio state=on power=6 channel=33
```

When you execute:

```
set res [oss_parse -match {power=([[:digit:]]+)}]
```

You will get *{ "power=6" "6" }* as the result and the input string will be left as:

```
radio state=on  channel=33
```

Sometimes, having matched something in the middle of the string you may want to continue parsing from the place where the match was found, rather than from the beginning of the input string, e.g., look at this call:

```
oss_parse -match {power[[:blank:]]*=} -then -skip -then -number
```

The role of *-then* (an argument-less selector) is to tell the function that the next selector is to be applied to the string following the place where the last selector has ended, rather than to the entire string (from the beginning). So what will happen is that the function will locate the sequence *power=* (allowing for blanks around =), then it will remove any blanks starting from the first character after =, then it will look up a number, again starting from the place where *-skip* has finished. The advantage of using *-number*, rather than a simple pattern, is that the selector will parse an expression (see Section 3.1), e.g.,

```
power = 1 + (2*2)
```

will also work. Normally *-number* looks for the expression at the beginning of the input string; *-then* tells it to start at the place where the previous selector has ended. In fact, *-number* will skip any initial blanks on its own, so *-skip* is superfluous in the above

example. One problem, though, is that if *-match* succeeds, but *-number* fails, the input string will be left with the matched string removed. This may be OK (the failure of *-number* is probably indicative of an error in the input line), but if it isn't, use *-checkpoint*, e.g.,

```
oss_parse -checkpoint -match {power[[:blank:]]*=} -then -number
```

Note that *-then* can be applied across multiple invocations of the function, e.g.,

```
oss_parse -checkpoint -match {power[[:blank:]]*=} -then
oss_parse -number
```

will work as intended, but, for example, if you precede *-number* by *-skip* (and do not insert *-then* after it), the number will be sought from the beginning of the input string (even though the *-skip* will be applied after the matched text).

Another structural matching selector is *-string*. It will try to match a quoted sequence of characters. Normally, for the selector to succeed, the first character of the input string must be “. Then, the function will scan all subsequent characters until a matching “ is found and combine them into a list of numerical (ASCII) values. This is the item that will be contributed by the selector to the list of values returned by the function. Standard UNIX-style escapes apply. Consider the following input:

```
,  "abc \"\t\x1f\n"
```

When you execute:

```
set res [oss_parse -skip ", \t" -string]
```

you will get this list: *{ “ { 97 98 99 32 34 9 31 10 } }*. Note that the first item is the first non-skipped character (contributed by *-skip*) and the second item is the list of character values representing the string parsed by *-string*.

Needless to say, *-then* applies to *-string* in the same manner as to *-number*. One general difference between *-string* and *-number* is that *-string* does not automatically remove blanks in front of the parsed string, i.e., it will fail if the first character to parse is not “.

The selector accepts an optional argument. If *-string* is followed by a number between 0 and 1024, then the number indicates the length of the string to parse. No delimiter is expected in that case: the function will copy from the input string the specified number of characters (or less if the input string ends sooner). For example, for the same input string as above, this call:

```
oss_parse -string 10
```

will produce *{ { 44 32 32 34 97 98 99 32 34 9 } }*. Note that escapes are processed exactly as before.

If only one item of the list procured by the function is interesting, you can use *-return* to indicate which one. The selector takes one argument, which is an index into the list of values. For example,

```
oss_parse -match {power[[:blank:]]*=} -then -number -return 1
```

will return the second element (number 1) from the list that would be returned otherwise, i.e., the value parsed by *-number*. The selector also has the effect of terminating the chain of selectors (forcing a return) wherever it occurs, so perhaps it doesn't make a lot of sense to use it anywhere other than the end of the chain (maybe with the exception of tests and debugging). If the argument of *-return* is absent, the full list is returned as usual, so argument-less *-return* at the very end of the chain is exactly void.

Selectors *-save* and *-restore* can be used to preserve the current shape of the input string with the possibility of restoring it later, say, when a multi-stage attempt at parsing fails and something else has to be tried. Each of them accepts an optional argument (a name) identifying the save. The name, if present, must begin with a letter and consist of

letters and digits. If no name is specified, then a default (unnamed) save is assumed. When the parser is initialized with *-start*, all the previous saves are erased.

The action of *-subst* is to carry out command and variable substitution on the input string. When this selector is processed, the input string is Tcl-expanded by evaluating all variable references and *proc* invocations. Note that the parser may introduce variables and/or functions to be referenced from command lines.

4.1.3 Command parser interface

If a user command parser function is defined (Section 2.1) then it will be called whenever the user enters a line in the input widget of the script's terminal window. Note that the function takes a single argument which is the Tcl string entered by the user (no newline character at the end). The function will parse the string (e.g., taking advantage of *oss_parse*) and transform it into a command data block (e.g., using the functions described in Section 4.1.1). When the command block is ready, the function will call:

```
proc oss_issuecommand { code block } { … }
```

to send the command to the node. The first argument of *oss_issuecommand* is the command code, the second argument is the command's data block, e.g., as procured by *oss_setvalues* (Section 4.1.1). The *ref* field in the command's header will be filled in automatically by the global script.

When interpreting user input, the parser function can take advantage of the full power of the underlying Tcl shell. For example, it can assume that some chunks of text entered by the user are Tcl statements to be evaluated internally. The function executes in the private *namespace* of the user parser named *USER*. This is where it should evaluate statements typed by the user and store its variables. This is best accomplished by referring to this function:

```
proc oss_evalscript { script } { … }
```

which evaluates the argument string as a Tcl script within the *USER namespace*.

To display a line of text in the terminal window, the parser function can invoke:

```
proc oss_ttyout { line } { … }
```

The argument string needs no newline at the end, but any newlines in the middle of the string will result in multiple lines being shown.

The command parser can fall down to the default parser by executing:

```
proc oss_defparse { line } { … }
```

where the argument is the line to be parsed, e.g., this user-provided parser:

```
proc my_cmd_prsr { line } {
      oss_defparse $line
}
```

is perfectly void, i.e., equivalent to its absence.

If the user-provided parser throws an error (before dispatching the command), the error message will show up on the screen and the input line will be ignored.

4.1.4 Message interpreter interface

If a message interpreter (the second function of the user parser) is defined, then it will be called for every message received from the node by the global script, except for heartbeat (autoconnect) polls (Section 3.2). Recall that the function takes three arguments: the *code* and *ref* attributes from the message header, and the message data block following the header. By using the functions described in Section 4.1.1, the message interpreter can decode the data block into values, format them into one or more lines, and send those lines to the terminal window via *oss_ttyout*.

The message interpret can fall down to the default interpreter by calling:

```
proc oss_defshow { code ref block { bstr 0 } } { … }
```

The first three arguments are exactly those passed to the user interpreter; the last one (optional) is a flag telling whether the blob (if one occurs in the data block) should be shown as a string (1) or a list of values (0). For example, this user-provided interpreter:

```
proc my_msg_pcsr { code ref block } {
       oss_defshow $code $ref $block
}
```

is perfectly void and equivalent to its absence.

If the message interpreter throws an error, the error string will be written to the terminal window followed by a byte dump of the message.

4.1.5 A few auxiliary functions

There exist a few more functions in the *oss_...* collection that the user parser functions may find useful. They are:

```
proc oss_keymatch { key klist } { … }
```

The first argument is a string, the second argument is a list of strings. The function selects the single unique string from *klist* that begins with the string in *key*. If no such string is found, or if the match isn't unique (more than one string in *klist* begins with *key*), the function throws an error. For example, assume that *klist* is:

```
{ match number skip string start save restore then return subst }
```

If *key* is any of *sa*, *sav*, *save*, the function will return *save*; if it is *s* or *a*, the function will trigger an error. In the first case, the error will be “multiple matches for s: skip string start save”, in the second “a not found”.

```
proc oss_isalnum { str } { … }
```

The function returns 1 if the argument string is alphanumeric, i.e., it begins with a letter or _ and contains no other characters than letters, digits, and _. Otherwise, the function returns 0.

```
proc oss_valint { n { min "" } { max "" } } { … }
```

The functions checks if the first argument is an integer expression whose value is between *min* and *max* exclusively. Note that *min* and *max* are independently optional (if any of them is empty, then the corresponding bound is not checked). If the validation succeeds, the function returns the simple decimal value of the first argument. Otherwise it triggers an error.

5 Illustration

Praxis *Apps/EXAMPLES/OSS/* provides a sample interface and a simple parser with two commands and two message types. File *ossi.tcl* in the praxis directory contains the specification. File *ossi.h* is the header automatically generated from the specification.

5.1 Quick start

Compile the praxis for VUEE (or for a real [WARSAW] node, for that matter, but VUEE is simpler). Start the praxis, then execute *oss* in the praxis directory (we assume that the global script has been moved to the user's *bin* directory, e.g., by *deploy*).

You will see a window similar to *piter's*. When you press the *Connect* button, you will get a simple dialog giving you a VUEE option. Click on the *VUEE* checkbox (assuming you are running the praxis under VUEE) then click *Proceed*. In a short while the script will connect to node 0 of the model.

The praxis is basically a variant of *RFPing* (or the starting point for the new edition of *TESTS/WARSAW*). It responds to two types of commands: *radio* (to start and stop RF

activities and set up their parameters), and *system* (to query and set some "system" parameters).

Enter:

```
system -query
```

into the input area at the bottom of the window. You will get a message in response showing you the current parameters of RF, memory statistics, and the power (UP/DOWN) mode of the CPU. Note that you can abbreviate the command as:

```
s -q
```

To start the radio, enter:

```
radio -state on
```

or the abbreviated version:

```
r -s on
```

You will see messages showing packets going out.

Note that you can open another window connecting to another node of the network. Execute *oss* again and select another node (say 1) from the VUEE connection options. When you switch on the radio in the other node, you will see received packets as well.

The commands accept more arguments. The full set of arguments of the *radio* command is:

```
radio -state st -power p -channel c -interval imin imax
                                          -length lmin lmax cnt
```

where *st* can be *on*, *off*, *rx*, *tx*, *wor*, *rx+tx*, *wor+tx*, and so on; *p* is a value between 0 and 7, *c* is a value between 0 and 255. The arguments of *-interval* and *-length* are pairs of values (ranges). If only one value is specified, the other is assumed to be the same. In the first case, the range concerns the transmission interval (with the minimum of 1 and the maximum of 65534). For *-length*, the range describes the transmitted packet length with the minimum of 1 and the maximum of 60. If both values of a range are specified then the second one cannot be less than the first.

The last argument, *cnt*, has no selector: anything that follows the last selector argument is interpreted as the contents (payload) of transmitted packets. If the first character of *cnt* is ", then the argument is assumed to be a character string; otherwise, it is a list of values to be assigned to consecutive bytes, e.g.,

```
radio -state on -power 6 -length 32 "baca is fine"
radio -interval 1024 2048 1 2 3 4 5 + 3 6
```

Note that in the first case the range of *-length* consists of a single number, while in the second case the sequence of payload bytes begins with 1 (2048 is interpreted as the second value of the *-interval* range).

For *system*, the full selection of arguments looks like this:

```
system -reset -spin s -leds l -blinkrate b -power p -query msg
```

with the restriction that *-reset* can only be specified alone (and it causes the node to reset). The argument of *-spin* is a value between 1 and 65534 indicating the number of milliseconds for which the CPU is supposed to enter a spin loop (e.g., for current measurements). The argument of *-leds* can be an arbitrary-length sequence of constructs l*ed=state* where *led* is a led number (from zero up) and *state* is *on*, *off*, or *b[link]*. For *-blinkrate*, the argument can be *f[ast]* or *s[low]*, and for *-power* it is *u[p]* or *d[own]*. Contrary to the first impression, *-query* is argument-less (the option causes a status message to be sent back by the node) while *msg* stands for the optional (selector-less) argument representing a *diag* message to be issued by the node (if the argument

is present). Similar to *radio*, the blob of the message can be written as a string or as a list of values.

5.2 Comments on the parser

Look into *ossi.tcl* in the praxis directory to see the parser (as well as message processor). Note that a command beginning with : is interpreted as a Tcl scrip to be executed internally. According to what we said in Sections 4.1.2 and 4.1.3, that script is evaluated in the *USER namespace*. To realize the power of this feature, try this:

```
:set cmd "radio"
:set status "on"
$cmd -state $status
```

Needless to say, you can define scripts in *ossi.tcl* and make them callable from command lines as Tcl procedures. As we explained in Section 4.1.2, a call to *oss_parse* with selector *-subst* causes the current version of the parsed line to be expanded for variables and function calls. Note that the parser in *ossi.tcl* performs this action as the first step of interpreting the input line.