The Austin Protocol Compiler
Reference Manual
Version 1.0

Tommy M. McGuire

May 7, 2004

Abstract

This manual provides a reference for the Austin Protocol Compiler, the TAP language, and the runtime system. It describes the basic usage of the compiler, the fundamentals of the execution of the TAP language, the syntax and semantics of the language, and the C interface provided by the runtime system and the generated code.

Contents

Chapter 1

The TAP language and the apcc compiler

TAP, the language provided by the Austin Protocol Compiler, is based on AP, a formal notation for designing network protocols created by Mohamed G. Gouda.[1] The syntax of TAP is intended to be very similar to that of AP, but TAP is not identical to AP. There are some syntactic and many semantic differences.

TAP is intended to be used by designing and (if needed) formally verifying a protocol using an abstract execution model and then translating that protocol into C using the Austin Protocol Compiler.

The Austin Protocol Compiler is made up of two parts, a compiler and a runtime library. The compiler is mostly written in Python[2], a high-level, object-oriented, interpreted language. The compiler uses a parsing toolkit written in C that interfaces Python with a parser built using the Bison[3] parser generator and the Flex scanner generator[4]. The runtime library is written in C.

For information on getting started and suggestions on using the Austin Protocol Compiler, see *The Austin Protocol Compiler Tutorial*.[1]

For background information about TAP and the abstract and concrete execution models, as well as a general overview of the APC system, see *Correct Implementation of Network Protocols*.[5]

[1] In production.

1.1 Basic operation

The TAP language is not intended to be a complete systems programming language. It relies on C to provide basic input/output and other services not related to network protocols. In particular, function calls (page 12) are passed on untranslated to the generated C code, allowing the use of any C library facility.

What TAP provides is the infrastructure for network protocols—sending and receiving messages, basic calculations and protocol logic, and timeout handling.

1.2 Compiling, linking, and running

To compile an TAP process specification (from a file ending in ".ap".), run the command

```
apcc file.ap
```

This command will produce "file.c" and "file.h". Using the functions described in section 4, the C interface, create a file containing the C main function which includes file.h. Compile file.c and any other files needed by the application and link with the TAP runtime libarary, libAPC.a.

Chapter 2

TAP language behavior

Each TAP program consists of any number of message definitions (section 3.1) and one or more TAP process definitions (section 3.2).[1] The message specifications describe the fields and on-the-wire format of any messages sent or received in the protocol. The process specification describes the state of the protocol process (in the form of variables) and the actions taken by the process in response to its local state, messages it receives, or timeouts.

In execution, a process begins by checking its local actions, as described below, and then waiting for either a message to be received or a timeout to occur. If a message is received, the message is tested against the process's receive actions—if a receive action matches the message, the statements of the corresponding action are executed and if no action matches the message, it is discarded. If a timeout occurs, the statements of the timeout action are executed. In either case, the local actions are again checked, and then the process waits again.

Local actions (page 10) have guards that are predicates referencing only local variables of the process. During execution, the guards of the local actions are evaluated, and if a guard evaluates to true, the statements associated with that action are executed. This process is repeated until all of the guards of local actions have evaluated to false. At that point, the process cannot change state again without outside events and so the process waits for either incoming messages or timeouts.

Receive actions (page 10) have guards that are **rcv** expressions. During

[1] Do not confuse this process with an operating system process. The TAP process is a translation of the AP process formalism and not necessarily related to an operating system process.

execution, when a message has been received, each receive guard is evaluated against it until one of them returns true—a receive guard is evaluated by attempting to parse the message mentioned in the receive guard and succeeds if the constant fields in the message parse to their constant values. (For more information, see sections 3.1 on messages, 3.3 on actions, and 4 on the C interface, below.)

Timeout actions (page 10) have guards that are timeout declarations—these provide only a name for the action. Timeout actions are invoked by act statements. An act statement specifys a timeout action name and a delay. Sometime after the delay has expired (after the execution of an act statement), the statements of the timeout action are executed.

When a statement "**act** A **in** 50000" is executed, it sets a *time variable* associated with the action named A to 50000. If the process immediately begins waiting and no other events occur, A will be executed after a delay of at least 50000 ms. If another **act** statement is executed in the mean time, the new delay value will replace the 50000 ms.

For more information about TAP's conceptual, abstract, behavior and implemented, concrete behavior, see *Correct Implementation of Network Protocols*[5], which should be available from the APC web page.

Chapter 3

Details of TAP

The discussion of the complete of the TAP language follows the structure of the TAP grammar. The grammar is described using the Extended Backus-Naur Format, with the following conventions:

- {...} indicates zero or more copies of the contained elements.
- [...] indicates zero or one copy of the contained elements; i.e. the contents are optional.
- (...|...) indicates a choice between the contained elements.
- Literal text is presented in quotation marks.
- Non-literal token elements are in italics. There are three of these:
  - A *string* is a quote-delimited string of characters which does not span lines. Internal quotes and newlines can be escaped by a backslash. These strings cannot be manipulated in TAP, but can serve as arguments to functions as well as to directives as described later.
  - A *number* is one or more decimal digits, indicating a non-negative number.
  - An *id* is an identifier, made up of a letter followed by any number of letters or numbers.

Parsing of each source file begins with the start symbol:

start ::= elements
elements ::= {element}

element ::= “**import**” *string*
| “**include**” *string*
| message
| process

The source file given to the compiler consists of a sequence of elements. Each element is either an import directive, an include directive, a message definition, or a process definition.

The import directive looks for the file named in the *string*. The contents of this file are read and processed by the compiler before any subsequent elements in the current source file.

The include directive inserts a C include directive in the output file, calling for the file named by the *string*. These included files form part of the interface between the APC-generated C module and external C code.

3.1 Message syntax

message ::= m-header m-body
m-header ::= [“**external**”] “**message**” m-name [m-functs]
m-name ::= *id*
m-functs ::= “**(**” m-in “**,**” m-out “**)**”
m-in ::= *id*
m-out ::= *id*
m-body ::= “**begin**” fields “**end**”
fields ::= {field “**,**”} field

Each message definition consists of a header and a body. The message header primarily provides a name for the message. The body of the message is a sequence of fields, separated by commas. The message definition is used by the compiler to produce:

1. A C structure with records for each field in the message.
2. Parsing and marshalling functions, which interpret and recognize received messages and convert a message structure to a sequence of bytes for transmission, respectively.

Optionally, the message can be marked as external, in which case the compiler does not generate the C functions for marshalling and parsing the message. This allows the programmer to provide such functions, in order to handle more

complex messages than those that can be described by TAP. Also optionally, two functions can be identified which process the message immediately after the fields in the message have been parsed (m-in) and immediately before the message is sent (m-out). These functions receive the message buffer as well as the structure describing the fields of the message, allowing them to compute a checksum for the message, for example.

field ::= f-name "**:**" f-type ["**=**" f-value]
f-name ::= *id*
f-value ::= expression

Each field definition consists of a field-name, a field-type, and optionally, a field-value. If the field-value is present, the field is considered constant; the field is automatically set to that value before the message is sent and received messages are checked to ensure the field contains the proper value as part of the process of recognizing messages. In these expressions, the only allowable values are constants and the names of previous fields.

f-type ::= f-size ("**bits**" | "**bytes**")
f-size ::= expression

A field-type describes the size and type of the contents of the field. The expression describing the size can contain literal values or the names of previous fields in the message. The type of the field is implied by the use of bits or bytes to describe the field.[1]

- A bit field contains an unsigned integer value. The size expression describes the size of the field in bits; it must not be larger than 32 bits.
- A byte field contains a sequence of data bytes. The size expression describes the size of the field in 8-bit bytes. For a received message, the value of the record for the field in the structure generated by the compiler will be a pointer to the data in the original message buffer. When building a message to be sent, the value of the record should be set to a pointer to a sequence of bytes which will remain valid until the message is sent.

Each message has an additional field, named size, which indicates the overall size of the message in bytes. When receiving a message terminating with an arbitrary-length data field, the size field (minus the size of any previous fields)

[1] For grammatical correctness, "**bit**" is allowed as a synonym for "**bits**" and likewise, "**byte**" for "**bytes**".

provides the length of the final field. When sending such a message, assigning to the size field allows the message marshalling functions to copy the appropriate number of bytes from the array pointed to by the data field.

3.2 Process syntax

process ::= p-header p-body
p-header ::= "**process**" p-name [constants] [variables]
p-name ::= *id*
constants ::= "**const**" declarations
variables ::= "**var**" declarations
declarations ::= {declaration "**;**"} declaration
p-body ::= "**begin**" actions "**end**"

Each process definition also consists of a header and a body. The process header provides a name for the process as well as the optional declarations for the process's constants and variables. The process's body contains a sequence of actions.

declaration ::= ids "**:**" type ["**=**" initial-value]
ids ::= {*id* "**,**"} *id*
type ::= "**integer**"
| *number* "**..**" *number*
| "**boolean**"
| "**address**"
| "**array**" "[" array-size "]" "**of**" type
array-size ::= *number*
initial-value ::= (*number* | "**true**" | "**false**")

In each declaration, a sequence of identifiers which name constants or variables are associated with a type and optionally an initial value. The basic types allowed by TAP are 32-bit integers, booleans, and addresses. The integer type can be specified as either a general integer or as a range of allowed values.

The initial values for variables or constants must match the type of the variable or constant; the value of an integer is a number, and the value of a boolean is either true or false. Addresses may not be given an initial value in TAP. (The initial value of an address can be given via the C interface while initializing the APC runtime system.)

The only complex type supported by TAP is the array, with any number of dimensions. The allowed indices of each array dimension is given by the array-size value; indices range from 0 to the array-size−1. If an initial value is given for an array, each element of the array is set to the value.

3.3 Action syntax

actions ::= {action "[]"} action
action ::= guard "**->**" statements
guard ::= (local-guard | receive-guard | timeout-guard)
local-guard ::= expression
receive-guard ::= "**rcv**" m-name "**from**" address
address ::= *id*
timeout-guard ::= "**timeout**" t-name
t-name ::= *id*

In a TAP process, actions are separated by a box, written as two square brackets: `[]`. Each action consists of a guard and a sequence of statements. There are three forms of guards: local, receive, and timeout.

Local guards are made up of a predicate, a boolean expression. The action is enabled when the guard evaluates to true.

Receive guards specify a message accepted by the action and an address. The guard may be enabled if and only if the received message matches the message specified by the receive guard. If the address is a constant, then the action will only be enabled if the message is from the process identified by the address. If the address is a variable, then the action will be enabled no matter where the message is from and the address will be set to the source of the message.

Timeout actions provide a name, t-name, for the action for use with the activation statement; the behavior of such actions is described in the next chapter.

3.4 Statement syntax

statements ::= {statement "**;**"} statement
statement ::= "**skip**" | function-call | assignment | send
| conditional | loop | activate

In any sequence of statements, the individual statements are separated by semicolons. The two fundamental statements are skip, which does nothing,

and a function call, which invokes a C function and is more fully described on page 12.

assignment ::= left-sides "**:=**" expressions
left-sides ::= {left-side "**,**"} left-side
left-side ::= (*id* | field-reference | array-reference)
expressions ::= {expression "**,**"} expression

TAP assignment statements allow multiple values to be assigned simultaneously; in the code generated by the APC compiler, each expression is evaluated independently and stored in a temporary location. Subsequently, the values are assigned to the left-hand-side locations. Locations which can be assigned values are either variables, message fields, or array elements.

send ::= "**send**" m-name "**to**" address

A fundamental operation in TAP is sending a message, identified by m-name, to a process, identified by the address. Any necessary fields in the message should be set before executing the send statement.

conditional ::= "**if**" guarded-statements "**fi**"
guarded-statements ::= {guarded-statement "[]"} guarded-statement
guarded-statement ::= expression "**->**" statements

TAP provides a conditional statement with guarded branches separated by the box. Each branch consists of a boolean expression guarding a sequence of statements. In execution, one branch with a true-valued expression is chosen and executed. If no branches are enabled, execution continues with the next statement after the conditional.

loop ::= "**do**" expression "**->**" statements "**od**"

The iteration statement in TAP is made up of a single guarded statement, which provides a sequence of statements which are executed repeatedly as long as the expression evaluates to true.

activate ::= "**act**" t-name "**in**" delay
delay ::= expression

In order to handle message loss, TAP has an additional action guard and an additional statement:

- The timeout guard, **timeout** t, provides a name, t, for the action. This name is used by the activation statement.
- The activation statement, **act** t **in** d, provides a delay, d, between the activation statement being executed and the timeout action t becoming enabled. For the request/reply protocol in this abstract model, the delay is essentially arbitrary—any non-zero delay will have the same behavior. However, in a protocol with multiple timeout actions or multiple delays, the delay values will describe the relative behavior of the timeouts.

Every timeout guard has a *time variable* associated with it, which either is null or has a numeric delay. Initially, the value of every time variable is null. The execution of an activation statement with a timeout guard name t sets the value of the time variable associated with the timeout guard t to the delay given in the activation statement.

3.5 Expression syntax

In order to simplify the description of the TAP expression, the grammar rule is broken into a number of sub-rules below. The expression rule is the combination of all of the individual sub-rules.

expression ::= (*id* | *number* | "**true**" | "**false**" | *string*)

The fundamental expressions in TAP are variable names, numbers, true and false, and strings (which may only be used as arguments to function calls).

expression ::= field-reference
| array-reference
| function-call
field-reference ::= m-name "**.**" (f-name | "**size**")
array-reference ::= (array-reference | *id*) "[" expression "]"
function-call ::= function-name "**(**" [expressions] "**)**"
function-name ::= *id*

Further expressions are field references, array references, and function calls. Field references are described by a message name and either a field within the message or the special field, "size", which contains the overall size of the message in bytes.

Array references follow the traditional syntax, with a numeric expressions describing the element within the array.

A function call identifies a C function by name and executes it with the arguments given by the expressions. The C type of the return value of the function should be one of:

- void, for functions called as statements,
- unsigned long, for integer values, or
- unsigned char *, for an assignment to a message's data field.

expression ::= "(" expression ")"
| expression binary-operator expression
| unary-operator expression
binary-operator ::= "=" | ">" | "<" "<=" | ">=" | "<>"
| "|" | "&" | "+" | "-" | "*" | "/"
unary-operator ::= "~" | "-"

The next group of general expressions include the normal binary and unary operators. The binary operators are equality, inequality, boolean operators, and arithmetic operators. Unary operators are boolean and arithmetic negation.

5	"=" ">" ">=" "<" "<=" "<>"
4	"&" "|"
3	"+" "-"
2	"*" "/"
1	"~" "-" (unary)

Figure 3.1: TAP operator precedence, from lowest to highest.

These operators have the precedence described in Figure 3.1.

expression ::= "**size**"

The final form of expression, a bare reference to a size message field, is only valid in an expression that is part of a message definition. The value of the "**size**" expression is the overall size of the message in bytes.

Chapter 4

The C interface

The Austin Protocol Compiler is intended to be used similarly to the Lex and Yacc compiler construction tools—to generate code that will be embedded in another program. The output of the compiler is C source code that must be linked with other C functions in order to create a working program.

4.1 Fundamental operations

The C code which uses the output of the compiler must do five things:

1. Initalize the TAP runtime engine.
2. Initialize each process.
3. Add each process to the runtime engine.
4. Set any constant (or variable) addresses needed by the processes.
5. Invoke the TAP runtime engine.

Additionally, the external code can provide functions to assist in encoding and decoding messages, or to replace the message parsing and writing entirely, if the message handling provided by the compiler is not sufficient.

For all of the functions, if an error occurs (the exact notification method for errors is described below with the individual functions), the variable APC_error is set to a character string describing the error.

4.2 The TAP engine

The first and last steps involve direct interaction with the TAP runtime engine.

The current runtime library provides support for protocols sending and receiving UDP messages. Initalizing the TAP engine is handled by the function call

int UDP_initalize_engine (int port)

This function returns true in case of error. The port is a UDP port number, at which the engine will listen for incoming messages.

Invoking the engine is handled by the function call

int APC_engine ()

This call does not return until the protocol engine has either failed or terminated. Termination is indicated by a false return value.

4.3 Process initialization

Once the runtime engine is initialized, it is necessary to initialize the process that the engine will be executing. This initialization is handled by code generated by the compiler, but a special function must be called. This function has the form

int process_*process* (char *name, *process*_state_t *state,
*process*_process_t *process)

where the "process" suffix of the function name is replaced by the process name specified in the TAP code. For example, if a process v is specified in TAP, the function will be called process_v.

A C declaration is provided in the *file*.h generated by the compiler. This function takes no arguments, and returns true in the case of an error or otherwise false.

4.4 Message addressing

In the code generated by the compiler, a variable of type address will be assigned a value by the receive action when a message arrives. This allows the process to respond to messages coming from any source, including those which are not previously known. On the other hand, constants of type address are treated as

parts of the tests for receive action guards—if the source address of the message does not match the constant address, the receive action is not enabled. For constant addresses, as well as for variable addresses which are used to send messages before any are received, an initial value must be provided.

Since the value of an address depends on the communication methods underlying the TAP runtime system, these values are not handled by the TAP language or compiler. Instead, addresses are manipulated by the identifier provided in the TAP source, which is referenced as a C character string. The C code which uses the functions produced by the compiler should use the APC_set_address function to assign an address's initial value.

```
int APC_set_address (APC_process_t process, char *name,
    APC_address_type_t type, char *address)
```

For generality here, as well, the address is also treated as a C character string. For UDP addresses, the address string should be in the form

```
host-name:port-number
```

where the host name is optional and defaults to the local host and the port is the UDP port number on the specified host. The name given is the identifier used in the TAP program to send and receive messages.

The APC_set_address function returns true and sets APC_error in case of an error. It returns false otherwise.

4.5 Message handling

By default, based on the message definitions provided to the compiler, the code generated by the compiler includes functions for marshalling and unmarshalling messages to and from the network format. These functions, called a writer and a reader respectively, move the data of the message between the C structure representing the message fields in the generated code and a character array buffer used to send and receive messages.

A message definition can optionally identify two C functions to be called by the reader and writer, respectively, in order to perform any processing that requires the actual message buffer.

```
int in_funct (unsigned char *in, int in_length, message *msg)
int out_funct (unsigned char *out, int out_length, message *msg)
```

(The actual functions should have "message" replaced by the name of the message. The structure has a type definition allowing the dst and src pointers to reference the structure to be read into or sent from.)

The function in_funct will be called after the fields of the message have been parsed from the buffer, but before the statements associated with the receive action are executed. If it returns 0, the statements will not be executed, just as if the values in the message did not match the constant fields in the message definition.

The out_funct will be called after the fields of the message have been written to the buffer, but before the message is sent. If it returns 0, the message will not be sent.

For more information, see section 3.1.

For messages marked as external, the compiler does not generate the reader and writer functions. The user is expected to provide the functions, matching the following declarations:

```
int read_message (unsigned char *in, int in_len, message *dst)
int write_message (message *src, unsigned char *out, int *out_len)
```

The reader function should initialize the structure pointed to by dst and then read the information from the incoming buffer in, which has a length specified by in_len bytes. If the buffer contains a correct message, the function should return true. Otherwise, it should return false.

The writer function should initialize the outgoing buffer out, which has out_len bytes, and then write the fields specified by the structure pointed to by src to the buffer. It should return false in case of error and true otherwise.

The fields of the message can be referenced in the structure by the same names as given in the TAP message definition.

Appendix A

Reference grammar

This grammar is generated by y2l, the Yacc to LaTeX utility by Kris Van Hees, from the Bison grammar used by the compiler. y2l is included in the source distribution's doc directory to make rebuilding this document easier. y2l is copyright © 1994-2000 by Kris Van Hees, Belgium.

The conventions used in the grammar are:

- {...} indicates zero or more copies of the contained elements.
- [...] indicates zero or one copies of the contained elements.
- (...|...) indicates a choice between the contained elements.
- Literal text is between quotation marks.
- The [] box is the two characters "`[]`".

A.1 Lexical elements

The elements unspecified by the grammar are

- *ID* A letter, followed by any number of letters or numbers.
- *STRING* A quote-delimited string which does not span lines. Internal quotes and newlines can be escaped by a backslash, however. (Strings are not capable of being manipulated in TAP, but can be used as arguments to C functions.)
- *NUMBER* One or more decimal digits.

start	::=	toplevel
toplevel	::=	elements
elements	::=	{ element }
element	::=	“**include**” STRING
	|	message
	|	process
message	::=	external “**message**” ID (messagebody | “**(**” ID “**,**” ID “**)**” messagebody)
external	::=	[“**external**”]
messagebody	::=	“**begin**” fields “**end**”
fields	::=	{ field “**,**” } field
field	::=	ID “**:**” fieldtype [“=” expression]
fieldtype	::=	expression (“**bit(s?)**” | “**byte(s?)**”)
process	::=	“**process**” ID constants variables “**begin**” actions “**end**”
constants	::=	[“**const**” declarations]
variables	::=	[“**var**” declarations]
declarations	::=	{ declaration “**;**” } declaration
declaration	::=	ids “**:**” type [“=” const_value]
const_value	::=	NUMBER
	|	“**true**”
	|	“**false**”
ids	::=	{ ID “**,**” } ID
type	::=	“**integer**”
	|	“**boolean**”
	|	NUMBER “**..**” NUMBER
	|	“**address**”
	|	“**array**” “[” NUMBER “]” “**of**” type
actions	::=	{ action “‖” } action
action	::=	(expression | “**rcv**” ID “**from**” ID | “**timeout**” ID) “−>” statements
expression	::=	“**(**” expression “**)**”
	|	expression “=” expression
	|	expression “>” expression
	|	expression “<” expression
	|	expression “>=” expression
	|	expression “<=” expression
	|	expression “<>” expression
	|	expression “|” expression

	| expression “&” expression
	| expression “+” expression
	| expression “−” expression
	| expression “∗” expression
	| expression “/” expression
	| “ ” expression
	| “−” expression
	| fieldreference
	| arrayreference
	| functioncall
	| ID
	| NUMBER
	| STRING
	| “**true**”
	| “**false**”
	| “**size**”
fieldreference	::= ID “**.**” (ID | “**size**”)
arrayreference	::= (arrayreference | ID) “[” expression “]”
functioncall	::= ID “**(**” (expressions “**)**” | “**)**”)
statements	::= { statement “**;**” } statement
statement	::= “**skip**”
	| leftsides “**:=**” expressions
	| “**send**” ID “**to**” expression
	| “**if**” guardedstatements “**fi**”
	| “**do**” expression “−>” statements “**od**”
	| “**act**” ID “**in**” expression
	| functioncall
leftsides	::= { leftside “**,**” } leftside
leftside	::= ID
	| fieldreference
	| arrayreference
expressions	::= { expression “**,**” } expression
guardedstatements	::= { guardedstatement “[]” } guardedstatement
guardedstatement	::= expression “−>” statements

Appendix B

Runtime function interface

B.1 Types

- APC_address_t

  C type representing an abstract address.

- APC_address_type_t

  C enumeration used to select the type of an address: either APC_local_address (for an address being set to a process running within the same engine), or APC_lowlevel_address (for a low-level, remote address; currently a UDP address/port number pair).

- APC_process_t

  Generic C type containing process information. The specific sub-types of this type are created from the TAP process definition by the compiler. Contains references to the process's state records, action and timeout records, local message buffer, and other bookkeeping information.

- *process*_process_t

  Specific sub-type of APC_process_t that is generated by the compiler; the *process* prefix is replaced by the name of the process. Each individual value of this type should be set up by the corresponding process_*process* function.

- APC_state_t

  Generic C type describing the state information. Like the APC_process_t, specific sub-types are created by the compiler.

- *process*_state_t

  The C data structure representing the state of a process; contains records for the variables and constants declared in the process; it is generated by the compiler.

- *message*

  A C data structure containing information about a message, used in the code generated by the compiler. The steps of parsing a message include setting up a message structure; the steps of sending a message include filling out a buffer with the information from a message structure. The structure has records for each field of the message as well as a field for the overall size of the message.

B.2 Variables

- char * APC_error

  A C string describing the most recent error condition. Normally, this variable is null.

B.3 Functions

- int int APC_add_process (APC_process_t process)

  Function called by generated code to inform the runtime engine about a newly-initialized process.

- int int APC_set_address (APC_process_t process, char *name, APC_address_type_t type, char *address)

  Set an address named name in process to a local or low-level address (based on type) based on address . For a local address, address is the name of the process; for a low-level, UDP, address, the format of the string is "host-name:port-number". Other low-level formats may vary.

- int int APC_engine ()

  Execute the APC runtime engine, running any configured processes.

- int int UDP_initialize_engine (int port)

  Set up a runtime engine to use UDP low-level message passing and to listen on UDP port port .

- int int process_*process* (char *name, *process*_state_t *state, *process*_process_t *process)

  Configure the process tag structure process , assigning it name within the engine, setting up its local state .

- int int read_*message* (unsigned char *in, int in_len, *message* *dst)
  int int write_*message* (*message* *src, unsigned char *out, int *out_len)

  If *message* is marked as external, these two functions must be provided by the programmer; the first parses the message from the buffer in; the second writes the message to the buffer out.

- int int APC_address_equal (APC_address_t address1, APC_address_t address2)
  int int APC_address_copy (APC_address_t *address1, APC_address_t address2)

  This function and the next are part of the address handling system used internally by the runtime engine. However, if it is necessary for C code to manipulate addresses, these can be used to control the reference counts.

Bibliography

[1] Mohamed G. Gouda. *Elements of Network Protocol Design.* John Wiley & Sons, 1998.

[2] Python language website. http://www.python.org.

[3] The GNU Project. Bison, April 2004. http://www.gnu.org/software/bison/bison.html.

[4] Vern Paxson, *et al.* Flex, April 2004. http://www.gnu.org/directory/text/wordproc/flex.html.

[5] Tommy M. McGuire. *Correct Implementation of Network Protocols.* PhD thesis, The University of Texas at Austin, May 2004. Available from the APC home page.

Index