The Austin Protocol Compiler Reference Manual

Tommy M. McGuire

September 20, 2002

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

Contents

1 The APT language and the apcc compiler **2**
- 1.1 Basic operation . . . 2
- 1.2 Compiling, linking, and running . . . 2
- 1.3 Language behavior . . . 3
- 1.4 Includes and imports . . . 4
- 1.5 Messages . . . 4
- 1.6 Processes . . . 6
- 1.7 Declarations . . . 6
- 1.8 Actions . . . 6
- 1.9 Statements . . . 7
- 1.10 Expressions . . . 8

2 The C interface **9**
- 2.1 Basic operation . . . 9
- 2.2 The APT engine . . . 10
- 2.3 Process initialization . . . 10
- 2.4 Message addressing . . . 10
- 2.5 Message handling . . . 11

A Reference grammar **12**
- A.1 Lexical elements . . . 12

1 The APT language and the apcc compiler

APT, 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]

APT is intended to be used with AP, by designing and (if needed) formally verifying a protocol using AP and then translating that protocol into APT for compilation. As a result, the syntax of APT is intended to be very similar to that of AP, but APT is not identical to AP. There are some syntactic and many semantic differences.

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 parser generator and the Flex scanner generator. The runtime library is written in C.

The compiler is about 1100 lines of Python, 1100 lines of C for the parsing tool kit, and 475 lines of Bison/Flex specifications. (In this Reference Manual, the grammar in Appendix A is generated from the Bison/Flex specifications; that grammar describes most of the 475 lines.)

The runtime libarary is about 1200 lines of C code, divided into 900 lines of relatively generic runtime engine code and 300 lines of UDP specific code.

For a simple protocol specification, the vending machine process is 30 lines in APT, of which 10 lines define the messages sent between processes. The vending machine process compiles into about 200 lines of C. Of the 200 lines, about 100 are message handling (the message structures and functions to read/write them to the network format), 55 are the actions from the process, and the remainder are used for record keeping and initialization.

When compiled on Linux and the debugging information removed, the vendor process is 9KB, including the runtime library functions.

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

1.1 Basic operation

The APT 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 9) are passed on untranslated to the generated C code, allowing the use of any C library facility.

What APT 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 APT process specification (from a file ending in ".ap".), run the command

```
apcc file.ap
```

[1]In production.

This command will produce “file.c” and “file.h”. Using the functions described in section 2, 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 APT runtime libarary, libAPC.a.

1.3 Language behavior

Each APT program consists of any number of message definitions (section 1.5) and a single[2] APT process definition (section 1.6).[3] 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 7) 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 7) have guards that are **rcv** expressions. During 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 1.5 on messages, 1.8 on actions, and 2 on the C interface, below.)

Timeout actions (page 7) 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.

The timeout actions are executed “in order.” When a statement

act A **in** 50000

is executed and followed by the execution of a statment

act A **in** 10000

[2]This limitation may be removed in future versions.

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

before the 50 millisecond delay has expired, the first execution of timeout A will be approximately 50 ms after the execution of the first act statement no matter how much time passes between the execution of the two act statements; the second execution of timeout A may happen immediately after the first if the execution of the second act statement happened within 40 ms of the execution of the first act statement.

1.4 Includes and imports

Before examining the APT language more closely, there are two features of the language which should be mentioned. APT allows two directives for including text into either the protocol being read by the APT compiler or the C code produced by the compiler.

import "file" The import directive textually inserts the contents of a file at the point of the directive, before the directive has been read by the compiler.

include "file" An include directive is passed along to the C file that is generated by the compiler, for example allowing a C header file to be included to declare functions, constants and other C identifiers.

This directive can be placed anywhere at the top level of the file, before the process.

1.5 Messages

[**external**] **message** message-name [(in-funct, out-funct)]
begin
  field-name: size [= value] [, field-name: size [= value]]...
end

The first elements of an APT file are message definitions. The message-name is used by receive expressions and statements as described below.

The identifiers in-funct and out-funct provide the names of two C functions that optionally process the message buffer. In-funct is called immediately after the message's fields have been parsed from the buffer, and can be used to verify a checksum, for example. Out-funct is called after the message's fields have been written to a message buffer, but before it is sent. It can be used to compute a checksum. For more information on these two functions, see section 2.5.

The fields, which should be separated by semicolons, consist of a field-name, a size (in bits or bytes), and an optional constant value. There are two kinds of fields:

integer fields

field-name: constant **bits** [= value]

An integral field has a constant width, measured in bits. It is transmitted on the wire in network byte order. The size should not exceed the smaller of either 32 bits, for compatibility with other systems, or the size of an unsigned long integer, for the generated code to be correct.

If the field has a value, the field is set to that value immediately before the message is sent. The value also identifies received messages; see section 1.8.

data fields

field-name: expression **bytes**

A data field can have a constant or variable width, which is measured in bytes. A field of this type is transmitted as a byte array.

The expression describing the width can only use constants or the names of previous fields in the message. The field should also be byte-aligned in the message.

A message declared as "external" will not have the reader and writer functions defined in the C code produced by the compiler. For more information, see section 2 on the C interface, below.

For example, the following message skeleton defines a message, "msg":

```
message msg
begin
  field1: 8 bits = 12;
  field2: 15 bits;
  field3: 1 bit;
  field4: field2 bytes
end
```

The first field defines a message field called field1 which is 8 bits long and has the constant value 12. When a message with this field is sent, the 8 bits at this location in the message will have the value 12. When any message is received, each receive action will be tested against it, and a receive action with a message specifier (see section 1.8) which has this field will not match if the 8 bits at this location do not have the value 12.

The next field,

field2: 15 **bits**

can be assigned a value in an action (see section 1.6 below), and when the message is sent in that action, the 15 bits at this location will have that value. When a message with this field is successfully received, the value of this field can be accessed in the receiving action.

The last field,

field4: field2 **bytes**

describes a variable-width data field whose size is defined by the value of the second field. This field will have a C type of "char *" in the generated code. See section 1.10 for more information.

1.6 Processes

process process-name
[**const** declarations. . .] [**var** declarations. . .]
begin action [[] action]... **end**

The **const** and **var** keywords precede lists of constant and variable declarations used by the process, separated by semicolons. Both constants and variables are optional. The actions, separated by boxes ("[]"), define the code of the process.

1.7 Declarations

identifier [, identifier]... : type [= initial-value]

The available types are

lower..upper Integer ranges. These are translated to unsigned long integers in the generated code. Since the actual values are limited to the range of unsigned long integers, a range is preferred over the integer type below for variables.

integer Unsigned long integers.

address The address of a process. Address do not allow initializers; constant addresses can be set using the C interface (see section 2 below) before starting the engine. Variable addreses are set when messages are received or by statements of the process.

Receive actions referencing constant and variable addresses differ in that constant addresses are tested against the source of the message; if the source is not the same as the address, the action is not enabled. Variable addresses match any source address and are assigned the source value.

array [size] of type Declares an array of size elements of the specified type. The elements are referenced from 0 to size-1.

1.8 Actions

Actions in APT have the following basic format:

guard $\rightarrow$ statements

(where the arrow is the two characters "–>"). In AP, an action is considered enabled when the guard would evaluate to true. When an enabled action is choosen, its statements are executed. In APT, the execution is somewhat more complicated (see section 1.3), but is intended to behave similarly. There are three types of actions.

local actions Local actions have guards which are boolean expressions. When the guard evaluates to true, the statements of the action are executed.

receive actions Receive actions have guards of the form

> **rcv** message **from** address

When a message is received by the process, the reader function for each of the messages used in a receive guard is called with the message. If the reader function is successful in parsing the message, the source of the message is tested against a constant address, or the source of the message is assigned to a variable address. If the message and the source address allow, the statements of the action are executed.

timeout actions Timeout actions have guards of the form

> **timeout** identifier

where the identifier is unique among the actions of the process. The timeout action is initially disabled. When an action of the process executes an **act** statement (described below in section 1.9) referencing the identifier, the timeout action will become enabled after the delay specified in the act statement.

1.9 Statements

There are seven types of statements. The statements in actions or other blocks are separated by semicolons.

skip The skip statement does nothing.

assignment

> variable [, variable]... := expression [, expression]...

The expressions of the assignment statement are evaluated and then the values are assigned to the variables.

Possible left-hand-sides are variable identifiers, array references (page 9), and field references (page 8).

send

> **send** message **to** address

This statement causes the writer function for message to be called, and the resulting buffer to be sent to the address.

conditional

> **if** condition $\rightarrow$ statements [$[\!]$ condition $\rightarrow$ statements]... **fi**

Each condition is evaluated. The first which evaluates to true results in the execution of the statements associated with it.

loop

do condition $\rightarrow$ statements **od**

The boolean condition is evaluated, and if it evaluates to true the statements are executed and the condition is evaluated again. If the condition is false, the do statment ends execution.

timeout activation

act identifier **in** delay

Execution of this statement will enable the timeout action associated with the identifier after the delay. The delay is specified in microseconds.

The timer used by the timeout mechanism is somewhat unusual; if a timeout action is activated more than once, the executions occur in order of activation—the delay used by an activation of a timeout is the maximum of the delay specified by the act statement and any currently outstanding delays.

function call A function call can be used as a statement.

1.10 Expressions

Integer variables in APT are represented as C unsigned longs. The operators for expressions are the usual suspects: =, $<$, $>$, $<=$, $>=$, $<>$ (not equal), $|$ (boolean or), & (boolean and), +, -, *, /, and unary˜(boolean negation), and - (arithmetic negation).

Additional expressions are message field references, array references, and function calls.

message field references

message.field

When used in expressions, the value of this expression is the value of the field—either from a received message or a message that will be sent by a later statement. Message fields can also be assigned to by being mentioned on the left-hand-side of assignment statements.

Integer fields are translated to unsigned longs and data fields are translated to char pointers. In a received message, the value of a data field is a pointer to the actual message buffer; this value will not be valid after the action receiving the message. In a message to be sent, the value assigned to the field should point to an array of characters of the size described in the message definition and this pointer should remain valid at least until the **send** statement.

array references

identifier[expression][[expression]...]

The identifier should be an array and the expression provides an offset into the array. Arrays are indexed from zero.

function calls

identifier(expression [, expression]...)

Function calls can be either expressions or statements, in which case their value is ignored and they act as procedures. These function calls are passed along unchanged to the C code produced by the compiler.

Function arguments are the primary use of quote-delimited strings since there are no string variables and (aside from import and include directives) no other use for them in the language. Strings are also passed along unchanged to the C code produced by the compiler.

2 The C interface

Note: This section, and the runtime system, is subject to change due to planned enhancements, including exploring the use of multiple APT processes using a single protocol engine.

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.

2.1 Basic operation

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

1. Initalize the APT runtime engine.
2. Initialize the process.
3. Set any constant (or variable) addresses needed by the process.
4. Invoke the APT runtime engine.

Additionally, the external code can provide functions to encode and decode messages, 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 prtcl_err is set to a character string describing the error.

2.2 The APT engine

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

The current runtime library provides support for protocols sending and receiving UDP messages. Initalizing the APT 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 engine()
```

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

2.3 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_*identifier*()

where the "identifier" segment of the function name is replaced by the process name specified in the APT code. For example, if a process *v* is specified in APT, 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.

2.4 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 APT runtime system, these values are not handled by the APT language or compiler. Instead, addresses are manipulated by the identifier provided in the APT source, which is referenced as a C character string. The C code which uses the functions produced by the compiler should use the set_address function to assign an address's initial value.

```
int set_address(char *name, 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
```

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 APT program to send and receive messages.

The set_address function returns true and sets prtcl_err in case of an error. It returns false otherwise.

2.5 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 represtenting 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. For more information, see section 1.5.

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

(The actual functions should have “msg” 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.)

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_msg(unsigned char *in, int in_len, msg *dst)
int write_msg(msg *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 APT message definition.

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 APT, but can be used as arguments to C functions.)

NUMBER One or more decimal digits.

start	::=	toplevel
toplevel	::=	messages process
messages	::=	{ message }
message	::=	external “**message**” ID messagebody
	|	external “**message**” ID “**(**” ID “**,**” ID “**)**” messagebody
	|	“**include**” STRING
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 [“=” NUMBER]
ids	::=	{ ID “**,**” } ID
type	::=	“**integer**”
	|	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
fieldreference	::=	ID “**.**” ID

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

References

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

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

Index

prtcl_err, 9

Abstract Protocols, 2
Abstract Protocols-Timed, 2
act, 8
act statement, 3
actions, 6
  local, 3, 6
  receive, 3, 7
  timeout, 3, 7
address, 6, 10
AP, 2, 6
apcc
  running, 2
APT, 2
  execution, 3
  program, 3, 9
array, 6
array expressions, 8
assignment, 8
assignment left-hand-sides, 7
assignment statements, 7
Austin Protocol Compiler Tutorial, 2

Bison, 2

C functions
  engine, 10
  set_address, 10
  UDP_initialize_engine, 10
C, programming language, 2, 3, 9
conditional statements, 7
constant declaration, 6

data field, 5
declarations, 6
  const, 6
  var, 6
directives
  import, 4
  include, 4
do, 8

engine, 10
errors, runtime, 9
expressions, 8
  array, 8
  field reference, 8
  function calls, 9
  integer, 8
  operators, 8
external message, 5

field, message
  data, 5
  expressions, 8
  integer, 4
Flex, 2
function call
  statements, 8
function calls
  expressions, 9

Gouda, Mohamed G., 2
grammar, 12
  ID, 12
  NUMBER, 12
  STRING, 12

ID, 12
if, 7
import directive, 4
include directive, 4
integer, 6
integer expressions, 8
integer field, 4

libAPC.a, 3
Linux, 2
local action, 3
local actions, 6
loop statements, 8

message, 3, 4, 11
  buffer functions, 4, 11
  external, 5, 11
  field, 4, 11
    expressions, 8

NUMBER, 12

operators, 8

port, UDP, 10
process, 3, 6
Python, 2

range, 6
rcv, 7
receive action, 3
receive actions, 7
runtime errors, 9
runtime library, 3

send statements, 7
set_address, 10
skip statement, 7
statements, 7
    act, 3
    assignment, 7
    conditional, 7
    function calls, 8
    loop, 8
    send, 7
    skip, 7
    timeout activation, 8
STRING, 12

timeout, 7
timeout actions, 3, 7
timeout activation statements, 8
types, 6
    address, 6
    array, 6
    integer, 6
    range, 6

UDP port, 10, 11
UDP_initialize_engine, 10

variable declaration, 6