Telecom Engine

© Zentel Telecom Ltd, 2009

Table of Contents

The TE Programming Language **12**
Introduction 12
A First Look at the Language 12
Notes on Style 15
Formal Language Definition **16**
Introduction 16
Syntax definitions 17
Program Structure **18**
Program Structure 18
Declaration Block **19**
Declaration Block Definition 19
Declaration Block Examples 19
Function Block **21**
Function Block Definition 21
Onsignal Declaration Definition 22
Function Declaration Definition 22
Function Block Examples 22
Statement Block **23**
Statement Block Definition 23
Expression Statement **24**
Expression Statement Definition 24
Assignment Expression 26
Function Expression 26
Example Expression statements 27
Goto Statement **27**
Goto Statment Definition 27
Goto Statement Example 27
Jump Statement **28**
Jump Statement Definition 28
Jump Statement Example 29
Label Statement **29**
Label Statement Definition 29
Label Statement Example 30
Return Statement **30**
Return Statement Definition 30
Return Statement Example 30
Restart Statement **31**
Restart Statement Definition 31
Restart Statement Example 31
Stop Statement **32**
Stop Statement Definition 32
Stop Statement Example 32
While Statement **32**
While statement Definition 32
While Statement Example 33
Do Statement **33**
Do Statement Definition 33
Do Statement Example 34

For statement 34
For Statement Definition 34
For Statement Example 35
Break Statement 36
Break Statement Definition 36
Break Statement Example 36
Continue Statement 37
Continue Statement Definition 37
Continue Statement Example 37
If statement 37
If Statement Definitions 37
If Statement Example 38
Switch Statement 39
Switch Statement Definition 39
Switch Statement Example 39
More on Expressions 40
Expression Types 40
Arithmetic Expressions 40
Logical Expressions 42
Assignment Expressions 44
Constant Expressions 45
Function Expressions 47
More on Operators 47
TE Language Operators 47
Arithmetic Operators 48
Comparison Operators 48
Logical Operators 49
Assignment Operators 49
Indirection Operators 49
Miscellaneous Operators 52
Ambiguous Operators 52
String Concatenation Operator 52
The Conditional Expression Operator 53
More on Functions 53
More on Functions 53
More on Variables 55
More on Variables 55
More on Arrays 56
Compiler Directives 57
Compiler Directives 57
The $include Directive 57
The $if Directive 58
The TE Compiler 59
Introduction 60
Compiler Options 60
Environment Variables 64
Loading DLLs and .DEF files 65
Function Name Resolution 66
The TE Run-Time Engine 67
Introduction 67
Command Line Options 68
Registry Settings 69

Scrolling Log Tabs 70
Library Configuration Tab 73
Tools Tab 75
The TE Standard Library Set 76
Introduction 76
Manual Conventions 77
Task Management Library 78
Introduction 78
Library Quick Reference 79
Task Management Library Function Reference 80
task_spawn 80
task_chain 81
task_exec 81
task_parentid 82
task_return 82
task_sleep 83
task_hangup 83
task_defersig 84
task_clrdefer 85
task_getpid 86
task_arg 86
task_kill 86
System library 87
Introduction 87
Library Limits and Defaults 87
Side Effects From Signals 88
System Library Quick Reference 88
System Library Function Reference 89
Buffer Manipulation Functions 89
sys_bufuse 89
sys_bufrls 90
sys_bufrlsall 90
sys_bufcopy 90
sys_bufmove 91
sys_bufget 91
sys_bufset 91
File Handle Functions 92
sys_fhopen 92
sys_fhclose 93
sys_fhcloseall 93
sys_fhseek 94
sys_fhreadbuf 94
sys_fhwritebuf 95
sys_fhgetline 96
sys_fhputline 96
sys_fhwrites 97
sys_fheof 97
sys_fhlock 98
sys_fhunlock 98
sys_fhsetsize 98
File System Functions 99
sys_fcopy 99

sys_dirremove 99
sys_frename 100
sys_dirfirst 100
sys_dirnext 101
sys_diskfree 101
sys_fdelete 102
sys_dirmake 102
sys_gethandle 102
sys_finfo 103
Date and Time Functions 103
sys_date 103
sys_time 104
sys_ticks 104
sys_timeadd 104
sys_timesub 105
sys_dateadd 105
sys_tmrstart 106
sys_tmrsecs 106
sys_settime 106
sys_datecvt 107
Other System Functions 107
sys_exit 107
sys_getenv 108
Terminal Console Library 108
Introduction 108
Terminal Console Library Quick Reference 111
Terminal Console Function Reference 112
applog 112
syslog 113
errlog 114
tracelog 114
term_errctl 115
term_log 115
term_resize 116
term_size 116
term_write 117
term_scroll_area 117
term_cur_pos 117
term_print 118
term_box 118
term_colour 119
term_attr_def 120
term_put_nch 123
term_fill 123
term_clear 124
term_kbget 125
term_kbgetx 125
term_kbqsize 127
term_kbedit 128
ActiveX Data Ojects (ADO) Database Library 129
Introduction 129
Some Simple Examples 130
Blocking or non-blocking mode 133

Performance and blocking calls 133
Private and Public Objects 134
Error Codes 135
ADO Library Function Quick Reference 138
ADO Function Reference 139
adoTrace 139
adoErrVerbose 139
adoBlockMode 140
adoLastError 140
adoBusyState 140
adoConnection 141
adoConnOpen 142
adoConnParmGet 143
adoConnParmSet 145
adoConnClose 145
adoConnGetHandle 146
adoConnState 146
adoConnTransBegin 147
adoConnTransCommit 147
adoConnTransCancel 148
adoRecordSet 148
adoRSetQuery 149
adoRSetCmd 152
adoRSetResync 152
adoRSetRequery 153
adoRSetParmGet 154
adoRSetParmSet 156
adoRSetClose 156
adoRSetGetHandle 156
adoRSetRecCount 157
adoRSetMove 157
adoRSetMoveFirst 158
adoRSetMoveLast 159
adoRSetMoveNext 159
adoRSetMovePrev 159
adoRSetAddNew 160
adoRSetUpdate 160
adoRSetCancelUpd 161
adoRSetUpdBatch 161
adoRSetCancelBatch 161
adoRSetDelete 162
adoRSetState 162
adoRSetIsBOF 163
adoRSetIsEOF 163
adoFldCount 164
adoFldGetName 164
adoFldGetValue 165
adoFldSetValue 165
adoFldParmGet 166
adoFldParmSet 167
adoErrCount 167
adoErrMessage 168
adoErrValue 168
adoErrNative 169

adoErrClear 169
String Manipulation Library 169
Introduction 169
String Library Quick Reference 170
String Manipulation Function Reference 171
strtok 171
strlen 172
strsub 172
strcnt 173
strstrip 173
strend 174
strpos 175
strupr 175
strlwr 175
strcmp 176
strindex 177
strselect 177
strltrim 178
strrtrim 178
strrjust 179
strljust 179
itoc 180
ctoi 180
itox 181
xtoi 181
strtohexi 182
inttohexi 182
unstohexi 183
hexitostr 184
hexitoint 184
hexitouns 185
Inter-task Messaging Library 186
Introduction 186
Inter-task Messaging Libary Quick Reference 187
Inter-task Messaging Function Reference 188
msg_setname 188
msg_read 188
msg_send 189
msg_flush 189
msg_senderid 190
msg_sendername 190
msg_freecount 191
Global Array Library 191
Introduction 191
Global Array Library Quick Reference 192
Global Array Function Reference 192
glb_set 192
glb_get 192
array_dim 193
array_free 194
array_set 194
array_get 195
array_search 195

array_srchset 196
Semaphore Library 197
Introduction 197
Semaphore Library Quick Reference 198
Semaphore Function Reference 198
sem_test 198
sem_set 198
sem_clear 199
sem_clrall 199
Clipper Database Library 200
Clipper Database Library Quick Reference 200
Introduction 200
Clipper Database Function Reference 203
db_open 203
db_ixopen 203
db_get 204
db_append 204
db_fget 205
db_fput 205
db_rls 206
db_close 206
db_nrecs 206
db_nfields 207
db_fwidth 207
db_fname 207
db_rlsall 208
db_first 208
db_next 209
db_prev 210
db_key 210
db_recnum 211
db_flock 211
Floating Point Library 211
Introduction 212
Floating Point Library Quick Reference 212
Floating Point Library Reference 212
fp_decs 212
fp_add 212
fp_sub 213
fp_mul 213
fp_div 214
fp_pow 214
fp_rnd 215
Sockets Library 215
Introduction 215
Sockets Library Quick Reference 216
Sockets Function Reference 216
Sconnect 216
Sclose 217
Srecv 218
Slisten 219
Saccept 220

Ssend 220
Scheck 221
Shostname 222
SopenDGRAM 222
SsendDGRAM 222
SrecvDGRAM 223
Strace 224

Aculab E1/T1 Card Library 224

Introduction 224
The ACUCFG.CFG Configuration file 224
Run-time Initialisation and configuration 228
Some Simple Examples 230
Simple VOIP -> TDM example 233
Aculab Call Control Quick Reference 236

Aculab Call Control Function Reference 237

CCnports 237
CCsigtype 237
CCsiginfo 238
CCtrunktype 239
CCwatchdog 240
CCalarm 240
CCtrace 241
CCgetslot 241
CClisten 242
CCunlisten 243
CCstate 243
CCuse 244
CCwait 244
CCabort 245
CCenablein 246
CCaccept 246
CCmkcall 247
CCdisconnect 247
CCrelease 248
CCsetparm 249
CCclrparms 262
CCgetparm 263
CCalerting 267
CCgetcause 267
CCoverlap 268
CCgetcharge 268
CCsetupack 269
CCproceed 270
CCprogress 270
CCgetaddr 271
CCanscode 271
CCputcharge 272
CCnotify 272
CCkeypad 273
CChold 273
CCreconnect 274
CCenquiry 274
CCsetparty 275
CCtransfer 275

CCgetxparm 276
CCsetxparm 278
CCclrxparms 278
CCgetcnctless 279
CCmkxcall 279
CCsendfeat 280
CCsndcnctless 280
CCstrtohex 281
CCinttohex 281
CCunstohex 282
SWmode 282
SWquery 283
SWset 283
CCcreateTDM 283
Aculab Prosody Card Library 283
Introduction 284
Some Simple Examples 284
Simple VOIP example 286
Board Opening Order 288
Nailing transmit timeslots to H.100 or SCBUS 288
Indexed Prompt Files (IPFs) 289
Teminating Events 290
Blocking and Non-Blocking Mode 292
Aculab Prosody Speech Functions Quick Reference 293
Aculab Prosody Speech Function Reference 294
SMgetmodules 294
SMgetchannels 294
SMgetcards 295
SMcardinfo 295
SMmodinfo 296
SMplay 296
SMplayh 298
SMrecord 299
SMsetrecparm 300
SMgetrecparm 302
SMabort 302
SMgetslot 303
SMlisten 304
SMunlisten 304
SMctlDtmf 305
SMctlPulse 305
SMctlCPtone 305
SMctlGrunt 306
SMtoneint 306
SMwaittones 307
SMgetttones 308
SMclrtones 308
SMplaytone 309
SMplaydigits 309
SMplayptone 309
SMgetrecrognised 310
SMmode 312
SMtrace 312
SMaddASRvocab 313

© Zentel Telecom Ltd, 2009

SMclrASRvocabs 313
SMsetASRchanparm 313
SMaddASRitem 314
SMclrASRitems 315
SMctlASR 315
SMconfstart 316
SMconfjoin 316
SMconfleave 318
SMconfend 319
SMdump 319
SMstate 319
SMdetected 320
SMword 320
SMplayph 321
SMplaypr 322
SMplaystrph 322
SMplaystrphm 323
SMcreateVMP 324
SMtraceVMP 325
SMdestroyVMP 325
SMsetcodec 326
SMclrcodecs 327
SMcreateTDM 328
SMtraceTDM 328
SMdestroyTDM 329
SMfeedlisten 329
SMfeedunlisten 331

The TE Programming Language

Introduction

The Telecom Engine (TE) Language is a high level functional programming language that provides rapid application development and easy code maintenance for large and complex telecommunications applications. The syntax of the language has some similarity to the 'C' programming language but has been simplified to provide protection against common bugs that can have a catastrophic effect on critical telecommunications systems. Random system crashes caused by memory overwrites or invalid pointer accesses in 'C' or 'C++' can be a nightmare scenario for a system that is taking thousands of calls a day or passing millions of minutes of voice traffic a month.

All TE applications are compiled to byte-code, in a similar way to the Java language, and this byte code is then executed by the TE Runtime engine. This run-time engine provides a self contained and rubust environment within which the byte code can be run, providing protection from system crashes and bugs to ensure a stable and crash free system.

The Runtime Engine has been designed to be extremely fast, borrowing techniques from PC games programming to allow high density systems to be run in a single PC chassis. This means that many thousands of tasks/byte code programs can be running simultaeously in a single PC chassis allowing for high density solutions to be implemented.

The source code for an application should be created using a text editor in a file with the extension .TES (Telecom Engine Source file). Additional source files with the extension .FUN can be provided which contain the source code for individual functions (the .FUN file should have the same name as the function it contains). The TES file is then compiled using the TCL.EXE compiler to create a file with the same name but with a .TEX extension (Telecom Engine eXcecutable). This .TEX file contains the byte code operations that are executed by the TE Runtime Engine.

The Telecom Engine is a multi-tasking environment where many hundreds of byte code applications can be running simultaneously. Typically for Telecommunications applications there will be a single task running for each network channel which will handle the processing of inbound and outbound calls and the playing of speech files and received DTMF etc on that channel. Many other tasks could also be running to manage the allocation of various resources, updating the screen, communicating with remote host systems etc. The Telecom Engine is designed so that once started each task requires little or no operating system resources and the engine can switch between tasks extremely quickly creating a real-time multitasking environment that can process hundreds (or thousands) of simultaneous calls.

-o-

A First Look at the Language

An important point to note regarding the Telecom Engine language is that there are no 'built-in' functions. All external functionality for operations such as console input/output, database access, speech and network card functionality, inter-process communications etc. are all implemented as DLLs. The language itself does not include any of these functions as part of the

language definition, which means that the language is not tied down to any particular hardware vendor or to any specific implementation.

However without any of the above functionality the language would be pretty useless, therefore a set of DLLs is provided that offer most of the functions that are required for the implementation of telecommunication applications. This set of functions will be referred to as the TE standard function set. However, since these functions are implemented as DLLs there is no reason why these could not be extended or even completely replaced with another implementation if so desired.

In the following description of the TE language, all external functions used in the examples will be one of those from TE Standard function set. For a full description of these functions see the TE Standard Library Set reference manual.

As is usual will all programming guides we will start with an example 'Hello World' application:

```
main
   applog("Hello World");
endmain
```

All Telecom Engine programs consist of a pair of **main … endmain** statements between which the application statements are written. In the above application the applog() function writes the string "Hello World" to the application console and to the application log file . This function is provided by the CXTERMX.DLL function library (See TE Standard Library Set Manual).

This is not a very useful program for a telecommunications system so lets move quickly to another example using the Aculab functions provided by the CXACULAB.DLL and CXACUDSP.DLL function libraries:

```
$include "aculab.inc"

int port, chan, vox_chan, x;
var filename:64;

main
   port=0
   chan=1;
   vox_chan=1;
   filename="hello.vox";

   // Make full duplex H.100 bus routing between vox channel and network port/channel
   CClisten(port,chan,SMgetslot(vox_chan));
   SMlisten(vox_chan, CCgetslot(port, chan));

   // Enable inbound calls on this port/channel
   CCenablein(port,chan);

   // loop waiting for incoming call
   while(1)
       x=CCwait(port,chan,CC_WAIT_FOREVER);
      if(x == CS_INCOMING_CALL_DETECTED)
         break;
      endif
   endwhile

   // Answer the call
   CCaccept(port,chan);

   // Play a vox file to caller
```

```
   SMplay(vox_chan,filename);

   // Hangup the call
   CCdisconnect(port,chan,CAUSE_NORMAL);

   // Wait for state to return to IDLE the release call
   while(1)
     x=CCwait(port,chan,CC_WAIT_FOREVER);
    if(x == CS_IDLE)
       break;
    endif
   endwhile

   // release the call
   CCrelease(port,chan);

   // restart the application to wait for another call..
   restart;

endmain
```

This program initialises a port/channel on an Aculab E1/T1 board for incoming calls (CCenablein()) , then enters a loop waiting for inbound calls (CCwait()). Upon receiving an inbound call it answers the call (CCaccept()), plays a speech file (SMplay()), then finally hangs-up the call and releases the call handle before restarting the program to wait for the next call.
This could be considered the 'Hello World' of telecommunications programming.

Readers with some familiarity with 'C' will notice a few similarities between the TE language and 'C'. However where 'C' uses curly braces ({ and }) to delimit statement blocks, the Telecom engine used **if…else…endif, while…endwhile, for…endfor, do…until(), switch…endswitch,** etc.

The first statement in the above application is as follows:

```
$include "aculab.inc"
```

This statement instructs the compiler to include the contents of the file "aculab.inc" into the program file as though the statements contained in "aculab.inc" had been typed directly into the program file. These files are usually used to define common global variables or constant definitions (for example the CS_INCOMING_CALL_DETECTED, CC_WAITFOREVER, CS_IDLE are all defined in the "aculab.inc" file.) and are often known as 'header' files (usually with ".inc" or ".h" extensions).

The statements after the $include in the program show the declaration of some global variables. There are two types of variables allowed in the TE language: **'var'** and **'int'** types. Actually all variables in the Telecom Engine are stored as ASCII strings and the only difference between the type **'var'** and the type **'int'** is that the TE run-time engine will automatically convert the values stored into **'int'** types to a numeric representation of the string. So if you assign the string "abc" to a variable of type **'var'** then "abc" is what would be stored in the variable, whereas if you attempt to assign "abc" to a variable of type **'int'** then this will be evaluated to 0 and the string "0" will be stored into the variable. This is described in more detail later.

At the top of the **main** program section you will see the global variables being initialised then the functions from the CXACULAB.DLL and CXACUDSP.DLL libraries are called to wait for a call, accept the call, play a speech file ("hello.vox") then hang-up and release the call.

Finally the **restart** statement instructions the TE runtime engine to clear all variables and start again from the top.

Throughout the code you will notice that comments are included by prefixing the comment by // characters. All text from the // marker to the end of the current line are ignored by the compiler. Alternatively comments may also be prefixed with the # character depending on the preference of the programmer.

-o-

Notes on Style

Since the language is similar in many ways to the 'C' programming language then a number of coding style suggestions can be taken from the common 'C' programming style. Programming style is important since it makes a program more readable to a human, thus making debugging and code maintenance easier for the programmer.

For example, the following code is perfectly valid and will compile successfully, but is difficult to read by a human programmer:

```
main int i; int j; for(i=0;i<10;i++) for (j=0;j<10;j++) int tot; tot=i*j; applog("tot=",tot); endfor endfor endmain
```

Whereas the following code is much easier to read:

```
main
int i;
int j;
	for(i=0;i<10;i++)
		for (j=0;j<10;j++)
			int tot;
			tot=i*j;
			applog("tot=",tot);
		endfor
	endfor
endmain
```

As you can see, by using appropriate indentation and having only one statement per line the code becomes much more readable. Typically another level of indentation should be used for each new statement block, and the **if...else..endif**, **while..endwhile, for..endfor** statements should all have the same level of indentation.

Also, since it is possible to nest **if..else..endif** statements within each other, then a decision needs to be made about how to indent these nested statements. It is suggested that when several **if..else..endif** statements are nested within each other then those statements which are at the *same logical level* should be indented to the same extent. What is meant by the *same logical level* will depend upon the context, but will typically be where a certain variable is been tested against various values.

For example in the following code there are a number of tests being made against the variable *x* in a set of nested **if..else..endif** statements, and it is tempting to indent each of these nested

statements one indent further for each statement block as follows:

```
if(x==0)
	applog("Got x=0");
else
	if(x==1)
		applog("Got x=1");
	else
		if(x==2)
			applog("Got x=2");
		else
			if(x==3)
				applog("Got x=3");
			else
				applog("Got something else");
			endif
		endif
	endif
endif
```

The above code is perfectly valid but as you see the nested **if..else..endif** statements are being indented further and further to the right, making the code more difficult to interpret (especially as the code gets larger).

Since the test of each value of *x* is at the *same logical level* then a better coding style would be to keep the indent of each **if..else..endif** statement at the same indentation as follows:

```
if(x==0)
	applog("Got x=0");
else if(x==1)
	applog("Got x=1");
else if(x==2)
	applog("Got x=2");
else if(x==3)
	applog("Got x=3");
else
	applog("Got something else");
endif endif endif endif
```

Notice that the number of **endif** statements at the end then just needs to match the number of **if** statements that have previously appeared at the *same logical level* and indentation.

-o-

Formal Language Definition

Introduction

Now that a couple of "Hello World" applications have been given inorder to provide a feel for the language, a more formal definition of the language and program structure is given below.

There are a few things that should be noted before we start:

a) The language is case sensitive. This means that and identifier AbC is different to identifier aBc. Also all language keywords are in lower-case, and use of upper-case characters would result

in the compiler reporting a syntax error.

b) The compiler makes no distinction between tabs, space and newline characters. These 'white space' characters are all equivalent and statements can span multiple lines so long as they conform to the rules of syntax provided (E.g. terminated with a semi-colon).

The exceptions to the above rule are:

i) Comments only apply until the end of the current line
ii) Constants strings (stings of characters between double quotes) cannot span across multiple lines.

c) Similarly multiple statements may be included in a single line, although this is discouraged because it generally makes the source code harder to read.

d) Identifiers (variable names, constant names and function names) must begin with an alphabetic character (A-Z, a-z) or an underscore character (_). This may be followed by zero or more alphabetic character (A-Z, a-z), numbers (0-9) or underscores (_). Indentifiers may be up to 127 characters long.

e) The scope of constants and variables is as follows:

- Variables and constants defined before the **main** statement are global in scope.
- Variables defined between the **main…endmain** statement block or between a **func…endfunc** block only have scope from the line they are declared on to the following **endmain** or **endfunc** statement.
- Variables defined outside of a **func...endfunc** statements only have scope from the point from where they are declared to the end of the source file the were declared in.

f) Variable of type **int** are similar to having a **var** type of length 21. This provides enough space for a 64 bit integer value and sign to be stored as a null terminated string (i.e 22 bytes are set aside, 21 for the numeric string and sign and one for the null terminator). **Note however that in the current version of the TE all integer arithmetic is signed 32 bit.**

-o-

Syntax definitions

The syntactical structure of the TE language is defined in the following sections. Below is a description of the form that these syntactical definitions will take.

Telecom Engine Language keywords and symbols are highlighted in **bold** font.

Indentifiers are names of constants, variables, arrays and functions are specified by the keyword

identifier where the type of identifier will depend on the context.

Curly braces {} are used to define optional keywords or items in the syntax definition.

Three dots: ... (otherwise known as elipses) means "... and so on" which indicates that the previous definition can be repeated.

Angled brackets <> are included where a constant value is required with a description of the constant inside the the brackets, E.g. *<length>*

Where several definitions are listed one line after another, then it can usually be taken that the word *or* can be implied between the definitions (if *or* is not expicitely used).

Thus, if you see

Definition1
Definition2
Definition2

This can be read as

Definition1
or
Definition2
or
Definition2

-o-

Program Structure

Program Structure

The structure of a TE program is as follows:

{ declaration_block }

main
{ statement_block }
endmain

{ function_block }

All programs must have a **main...endmain** block and include optional decaration_block, statement_block and function_block**.**

The smallest and most useless (but syntactically correct) program is the following:

```
// Totally useless but syntactically correct
main
endmain
```

-o-

Declaration Block

Declaration Block Definition

A *declaration_block* is where variables, arrays and constants are declared. *declaration_blocks* can appear before the **main…endmain** section in which case it declares *global* variables and constants, or it can appear as a statement anywhere inside a statement_block in which case the variables and constants are local to the function in which they are declared. They can also appear in a function_block in which case they are local to the *source file* in which there appear (from the point of declaration to the end of the current source file).

The syntax of a *declaration_block* is as follows:

{**dec**}

var identifier **:** <*length*> {**, var** identifier **:** <*length*> {,…}} **;**
int identifier {**,** identifier {, … } } **;**
var identifier **[** <*start_offs*> **..** <*end_offs*> **] :** <*length*> {**,** identifier **[** <*start_offs*> **..** <*end_offs*> **] :** <*length*> {**,** ... } } **;**
int identifier **[** <*start_offs*> **..** <*end_offs*> **]** {**,** identifier **[** <*start_offs*> **..** <*end_offs*> **]** {, ... } } **;**
const identifier = <*value*> {**, const** identifier = <*value*> {,…} } **;**
…

{**enddec**}

-o-

Declaration Block Examples

Declarations can be surrounded by optional **dec…enddec** statements (which sometimes help with the readability of the code). However the choice of whether to include **dec..enddec** keywords is up to the programmer:

This:

```
dec
   var a:10;
   var b:10;
enddec
```

is equivalent to:

```
  var a:10;
 var b:10;
```

Wherever a declaration_block is valid.

The declarations themselves can be single declarations like this:

```
var a:10;
var b:10;
var c:10;
var d[1..10]:8;
```

Or can be included in a single line as a comma separated list terminated by a semi-colon:

```
var a:10, b:10, c:10, d[1..10]:8;
```

Similiarly **int** types can be declared one to a line or listed on a single line separated by comma.

```
int z;
int y;
int a,b,c,d[1..10];
```

and also the same for **const** types which can be decared:

```
const z= "abc";
const y=4;
const a= "123", b=3, c=-14;
```

int types, **var** types or **const** types cannot be mixed in a single comma separated list.

Arrays are declared by specifying the start and end indexes of the array in square brackets after the array variable name (separated by double dots). For example :

```
var myarray1[1..10];
int myarray2[61..70];
```

This declares two arrays: myarray1 which has ten elements with indexes ranging from 1 to 10 and myarray2 which has ten elements with indexes ranging from 61 to 70. Elements from the array can be referenced in the code by specifying the index in square brackets:

```
a=myarray1[3];
myarray2[62]=4;
```

If an index is specified to an array that is outside of its index range then this will not be picked up during the compile, it will only be spotted at run-time when an error message will be logged to the error screen and to the error logs. The reason for this is that the index to an array can be any expression, for example:

```
a=myarray[a*b+4];
```

It cannot be established at compile time whether any expression is going to be a valid index to an array and therefore these errors are only picked up at run-time.

Constant declarations allow an identifier to be assigned a constant value which can be used instead of that value anywhere in the code. This is typically used to make programs more readable and to

allow easier maintenance of code.

For example, in the following code it is clear that we are waiting for the event CS_INCOMING_CALL_DETECTED which is easier to read than if we were checking that CCwait() was returning the value 2:

```
const CS_INCOMING_CALL_DETECTED=2;
int port,chan,x;

main
  port=0; chan=1;
  while(1)
    x=CCwait(port,chan,0);
    // This is easier to understand than using 'if(x == 2)'
    if(x == CS_INCOMING_CALL_DETECTED)
       break;
    endif
  endwhile


endmain
```

-o-

Function Block

Function Block Definition

The function block consists of zero or one onsignal_declaration and zero or more function_declarations. The onsignal function is the function that is jumped to when a signal (usually a hangup signal) is received. Additional declaration_blocks may also appear between the function_declarations and onsignal_declaration

{ declaration_block }
{ onsignal_declaration }
{ declaration_block }
{ function_declaration }
{ declaration_block }
{ function_declaration }
{ declaration_block }
…
etc

The order of the onsignal_declaration and the function_delarations is not important but the onsignal_declaration must reside in the same source file as the **main…endmain** statements (The main .TES source file), whereas the function_declarations can optionally reside in separate (.FUN) source files (or .TEL library file).

The syntax of the onsignal_declaration and function_declaration are defined in the following sections.

-o-

Onsignal Declaration Definition

The syntax of the onsignal_declaration required consists of a statement_block between **onsignal...endonsignal** keywords. The **onsignal** is jumped to when a hangup signal is received by the program (usually when a call disconnect signal is received from the calling party or called party in a telecommunications application).

onsignal

{ statement_block }

endonsignal

-o-

Function Declaration Definition

The sytax of a function declartion is the **func** keyword followed by the function name identifier followed by parenthesis containing zero or more argument names. After the statement_block the **endfunc** keyword terminates the function declaration.

func identifier **(** { identifier { **,** identifier { **,** … } **)**

{ statement_block }

endfunc

-o-

Function Block Examples

The function_block always follows the **main..endmain** section and may contain an optional onsignal_declaration and zero or more function_declarations and declaration_blocks. If an onsignal_declaration exists is must reside in the main .TES source file, however the rest of the function_declarations can optionally reside in other .FUN source files.

Below is a small program which calls two functions which are declared after the **main…endmain** section. This program also has an onsignal_declaration and another declaration_block:,

```
main

   my_function1("hello");
   my_function2("world");

endmain
```

```
// *** The function block starts here....

func function1(str)
   applog(str);
endfunc

func function2(str)
   applog(str);
endfunc

int a,b;

onsignal
  a=1;
  b=1;

  applog("We have received a signal");

  restart;
endonsignal
```

As mentioned, functions can be declared in a separate source file (or in a TEL library file). The source file that contains the function declaration should have the same name as the function with a ".fun" extension. For example a function called MYfunction() would need to reside in a file called myfunction.fun. Unfortunately windows file names are not case sensitive so it is not possible to declare two different functions like ABC() and abc() in two separate .FUN files and be able to distinguish between them using the file name. For this reason it is advisable not to distinguish between function names by character case alone (i.e. give all functions a unique name), or otherwise include these functions in the main (.TES) source file.

Multiple functions can be declared within a single .FUN file but only the function that has the same name as the .FUN file can be called from another source file (all other functions can only be called from within that particular .FUN file).

Functions can also be combined into a library file (with extension .TEL) using the MKTEL.EXE utility (See Telecom Engine Compiler, and Telecom Engine Utility programs), and these can be linked into the application by providing the name of the TEL file using the –L option of the TCL compiler (See Telecom Engine Compiler).

-o-

Statement Block

Statement Block Definition

The statement_block can consist of zero or more statements of the following types:

[declaration_block]
[empty_statement]
[expression_statement]
[goto_statement]
[jump_statement]

[label_statement]
[return_statement]
[restart_statement]
[stop_statement]
[break_statement]
[continue_statement]
[while_statement]
[do_statement]
[for_statement]
[if_statement]
[switch_statement]

The syntax of the declaration_block has already been described, and the empty_statement consists of a single semi-colon. E.g.

```
func doesnothing()
  ;  // this is the empty statement that does nothing
  ;;; // here are three more all doing lots of nothing
endfunc
```

In the following sections are the descriptions of the other statement type listed above:

-o-

Expression Statement

Expression Statement Definition

An *expression_statement* consists of an expression (expr) followed by a semi-colon:

expr **;**

Where expr can be one of the following:

rvalue
(expr **)**
expr + expr
expr **-** expr
expr ***** expr
expr / expr
expr % expr
'-' expr
expr '&' expr
expr == expr
expr eq expr
expr != expr
: expr <> expr
expr > expr

© Zentel Telecom Ltd, 2009

expr < expr
expr <= expr
expr >= expr
expr '?' expr ':' expr
expr streq expr
expr strneq expr
expr || expr
expr or expr
expr && expr
expr and expr
! expr
not expr

This is a recursive definition so might require some thinking about, but it accurately describes the syntax of all possible combinations of expressions and defines a 'tree' structure where the 'leaves' of the trees are all rvalues.

An rvalue is a 'right hand value' i.e. something that can appear on the right hand side of an equals sign such as a number, a string, a variable name, a pointer to a variable etc. By comparison an lvalue is something that can appear on the left hand side of an equals sign and does not include all things that are rvalues. For example, a variable name is both an rvalue and a lvalue since it can appear on either side of an equals sign:

```
a=b;
b=1;
etc
```

But it doesn't make sense to have a string or a number on the left hand side of an equal sign:

```
// Makes no sense!
"abc"=a;
5=c;
```

The full list of valid rvalues is as follows:

<number>
<string>
indentifier
indentifier **[** expr **]**
-- indentifier
++ indentifier
indentifier --
indentifier ++
& indentifier
* indentifier
asignment_expression
function_expression

Where *<number>* is any signed integer (E.g 10, -115, 121111111) or a hexadecimal value preceded by 0x (e.g. 0xff).

<string> is any string of characters surrounded by double quotes (E.g. "This is a string", "100"

etc)

identifier is the name of a variable and identifier **[** expr **]** is the name of an array and the index into the array (E.g. myarray[12], myarray[6*2]).

-- and ++ are decrement and increment operators (see arithmetic expressions) and **&** and ***** are the indirection operators (**&** - obtain pointer to a variable, ***** - get the contents of the pointer to a variable).

-o-

Assignment Expression

assignment_expression is any expression where a variable or array is assigned a value. It also included assignments that combine the assignment with an arithmetic expression (+=, *= etc). The full list of assignment expressions is as follows:

identifier = expr
* identifier = expr
identifier [expr] = expr
identifier += expr
identifier -= expr
identifier *= expr
identifier /= expr

For example:

```
a=5*5;    // assign variable a to 25
*b=34;    // assign contents of variable pointed to by b to 34
c+=25;    // equivalent to c=c+25;
d-=34;    // equivalent to d=d-34;
e*=25;    // equivalent to e=e*25
f/=34;    // equivalent to f=f/34;
```

-o-

Function Expression

A function_expression is simply a call to a function – passing any arguments as required by the function (E.g. myfunction(1,2,b,b)). The result of the expression is the value returned by the function.

Function expressions can be used anywhere in a expression where an rvalue is valid.

For example:

```
a=myfunc(1,2) * yourfunc(3,2,3) - (a > compfunc(4,2));
```

-o-

Example Expression statements

The following are all valid expression statements:

```
5;       // valid but does nothing
(5);     // valid but does nothing
3*2;     // valid but does nothing
3+(a*4); // valid but does nothing
a=5;      // assignment expression
a++;     // increment rvalue
a=myfunction(1,2,3,4); // Assignment_expression with function_call
myfunction(1,2,3,4);  // function_call
a=b=(5*4)+2*(b=3);    // Nested assignment expressions as rvalues.
a[a=b=(5*4)+2*(b=3)]=(1 and not (a*b) or c); // getting silly
```

More information about expressions will be given later – here we are primarily interested in the syntax of expressions (See More On Expressions).

-o-

Goto Statement

Goto Statment Definition

The *goto_statement* has the following syntax:

goto identifier **;**

Where indentifier is the name of a label defined in a label_statement . This causes the program execution to jump to the specified label which must reside within the same function_declaration or with the **main…endmain** block if that is where the **goto** was called from.

-o-

Goto Statement Example

An example of the goto statement is shown below:

```
var a:10;

main

 // Here is a label statement
 top:

    // do something
    a++;
```

```
    // Here are some goto statements
    if(a == 10)
        goto bottom;
    else
        goto top;
    endif

 // Here is another label
 bottom:

endmain
```

Note that it is illegal to try to jump to a label defined in a different function:

```
main
  f();
end

func f()
 // Here is the label
 mylabel:
    g();
endfunc


func g()
    // This is illegal!
    goto mylabel;
end
```

It is possible however to jump from a function to a label in the **main..endmain** section using the jump_statement.

-o-

Jump Statement

Jump Statement Definition

The jump statement is similar to the goto_statement and has the the following syntax:

jump identifier **;**

Here the identifier is a label defined in a label_statement, but in this case the label_statement *must* reside in the **main…endmain** block. However the jump_statement can be inside any other function_declaration. A jump_statement results in the stack being cleared so that all function return values are lost.

-o-

Jump Statement Example

Below is an example of a jump statement being used to jump from a function back to a label in the **main..endmain** section:

```
main

   // Call the function
  f();

:end_of_program

  restart;
endmain


func f()

     // call another function
    g();
end


func g()

     // Jump back to labale in main program (stack is cleared)
      jump end_of_program;
end
```

Note that is is illegal to jump to a label that is not in the main..endmain section:

```
main
  f()
endmain


func f()
int a;

mylabel:

     a++;
    // This is illegal!
     jump mylabel;

endfunc
```

-o-

Label Statement

Label Statement Definition

A label_statement consists of a label identifier followed by colon. The syntax definition is as follows:

identifier **:**

-o-

Label Statement Example

An example of a label statement is as follows:

```
func f()

// This is a label statement
mylabel:

    // ..then a goto statement
   goto mylabel;
endfunc
```

Labels can be used in goto_statements and well as jump_statements

-o-

Return Statement

Return Statement Definition

The *return_statement* is used to return from a function or to return control to the main program from an onsignal_function. To return a value from a function the syntax is:

return expr **;**

To return from onsignal or to return an empty value (“”) from a function the syntax is:

return ;

-o-

Return Statement Example

Below are some examples of the return statement:

```
func f()
   // This returns the result of the expression 3*2
   return 3*2;
endfunc


func g()
   // This returns the string "123"
   return "123";
endfunc
```

```
func h()
   // This returns the empty str ""
  return;
end
```

Note it is not illegal to return a value from the onsignal function, but the return value will be ignored lost when control is returned to the main program.   Therefore it is more correct to just use a return statement without a return expression:

```
onsignal
   return;
endonsignal
```

-o-

Restart Statement

Restart Statement Definition

The *restart_statement* causes all variables to be cleared (set to “”) and the program is restarted (i.e. control is returned to the top of the **main…endmain** block).   Other side effects may occur when this keyword is encountered since it is equivalent to the task being killed and then restarted again with the same Task ID (for example some DLL function libraries carry out certain actions (like releasing resources) when a task is killed).

The syntax of the restart statement is as follows:

restart ;

-o-

Restart Statement Example

Below is an example of the restart_statement

```
main
   f();
end

func f()
  int x;
  x=do_something();
  // Check for error return
  if(x < 0)
    // Carry out some action
    hangup_call();

    // this will clear all variables and restart the program
   restart;

  endif
```

```
   ...
end
```

-o-

Stop Statement

Stop Statement Definition

The *stop_statement* causes the current task to end and the task is removed from the active list (i.e. the task ID becomes invalid). This has is the same effect as if the task were killed or if then end of program is reached. This statement may cause certain side effects in DLL function libraries (for example certain task related resources may be released).

The syntax is:

stop ;

-o-

Stop Statement Example

Below is an example of a stop_statement:

```
main
   int a

   while(1)
       applog("a=",a);
      a++;
       // Stop the program when a reaches 10
      if (a==10)
          // Stop the task
         stop;
      endif
    endwhile
endmain
```

-o-

While Statement

While statement Definition

The while_statement is one of the three loop statements in the TE language (along with the do_statement and the for_statement). The syntax of the while_statement is as follows:

while (expr)
  statement_block
endwhile

The statements in the statement_block will be repeated continuously while the expression (expr) evaluates to a non-zero value. This expression expr is known as the *conditional expression* for the loop and would usually be a logical expression

-o-

While Statement Example

Below is an example of a simple while_statement

```
// This will loop ten times
i=0;
while(i < 10)
   applog("We're still in the loop since i=",i);
   i++;
endwhile
```

However any expr can be used – the loop will continue until the expression evaluates to 0 (zero):

```
// This will loop forever
while(1)
   applog("Looping forever");
endwhile
```

Similarly the following will result in an infinite loop and using assignment expressions like this is often the cause of bugs where in fact the *comparison_expression* was meant.

```
// a is assigned the value 10 conditional expression, thus 10 is the result of this expression
// so this will loop forever
a=0;
while(a=10)
   applog("Looping forever");
endwhile
```

-o-

Do Statement

Do Statement Definition

The *do_statement* is similar to the while_statement except that the conditional expression is carried out at the end of the loop, guaranteeing that the loop will excecute the main statement_block at least once (whereas for a while loop, if the expression evaluates to zero then the loop may not execute any of the statements in the statement_block). The sytax of the do_statement is as follows:

do
statement_block
until (expr **) ;**

-o-

Do Statement Example

Below is an example of a simple do_statement - note that the statement_block will always be excecuted at least once in a do_statement since the conditional expression is always checked at the end of the loop:

```
// The do loop will always execute its statement_block at least once
i=1;

// this is the do statement
do
  applog("You should see at least one of these messages..");
until(i==1);
```

-o-

For statement

For Statement Definition

The *for_statement* is similar to a while_statement except that a comma separated list of expressions are allowed for initialiation of the loop variables (*start_expr_list*) and another comma separated list of expressions which are excecuted at the end of each loop iteration (the *end_expr_list*). The syntax of the for loop is as follows:

for ({start_expr_list } **;** { expr }**;** { end_expr_list }**)**
statement_block
endfor

The *start_expr_list* is a comma separated list of zero or more expressions that are evaluated before the first iteration of the loop, and the *end_expr_list* is a comma separated list of zero or more expressions that are evaluated at the end of each iteration of the loop. The sytax definition for *start_expr_list* and *end_expr_list* is as follows:

expr {**,** expr {,...}}

The *start_expr_list* is usually used to initialise some variables that are tested in the conditional expression of the loop, and the *end_expr_list* is usually used to increment or decrement these variables so the loop only executes a certain number of times.

The *for_statement* is similar to the while_statement in that the statement_block will continue to be executed until the the conditional expression evaluates to 0 (zero) .

However, notice from the for_statement definition that the *start_expr_list*, *conditional expression* (

expr) and *end_expr_list* are all optional. If the *conditional expression* is left blank in a for_statement then this always equates to true. For example the following is perfectly valid and creates an infinite loop:

```
for(;;)
  applog("Looping to infinity and beyond");
endfor
```

This is not the same for a while_statement or do_statement. In those loops it is illegal to have a blank *conditional expression.*

```
// This is illegal!
while()
   do_something();
endwhile


// This is illegal!
do
   something();
until();
```

-o-

For Statement Example

A for_statement it is usually used to iterate over a range of values where the start_expr_list to initialises some variables that are then checked in the *conditional expression*. The end_expr_list is evaluated at the end of each iteration of the loop and usually increments or decrements the variables that are checked in the *conditional expression.*

For example:

```
myarray[0..9];
// Loop from i=0 through to i=9 setting all elements of array to -1
for(i=0;i<10;i++)
   // Set element i of array to -1
   myarray[i]=-1;
endfor

//i will be equal to 10 at this point in the code..
...
```

Here's another example where the start_expr_list contains two expressions (separated by comma) but there is no expression in the end_expr_list

```
// start and end expressions can have multiple expressions or none at all...
for(i=0,j=10; i <= 100; )
    applog("i=",i," j=",j);
    j++;
    i=i+j;    // This could have gone into the end_expr_list
endfor
```

If the *conditional expression* of a for_statement is left blank then the *conditional expression* will always equated to true so the loop will loop indefinitely. For example:

```
// this will loop forever
for(i=0;;i++)
   applog("i=",i);
endfor
```

Note than this is special syntax for the for_statement only, empty *conditional expressions* are not allowed in while_statements or do_statements.

-o-

Break Statement

Break Statement Definition

The *break_statement* causes a loop to exit immediately (i.e. a jump is made to the statement immediately after the **endwhile**, **endfor** or **until(**expr**);**

The syntax of the *break_statement* is as follows:

break ;

-o-

Break Statement Example

The break statement is used to break out of a loop before the *conditional expression* is met.

For example:

```
while(i < 10)
   if(i == 5)
      // Break prematurely from the loop
      break;
   endif
endwhile
```

Here's an example ising the CXACULAB.DLL function library for waiting for an inbound call to be detected:

```
// Loop indefinitely
for(;;)
   x=CCwait(port,chan,0);
   if(x==CS_INCOMING_CALL_DETECTED)
     applog("Received incoming call on port=",port," chan=",chan);
     // break from the loop
     break;
   else
     applog("Unexpected event x=",x, " whilst waiting for incoming call");
   endif
endfor
```

-o-

Continue Statement

Continue Statement Definition

The *continue_statement* causes the rest of the statements inside the loop to be skipped and the program jumps to the condition which decides whether to repeat the loop. In the case of a for-loop, the end_expr_list is executed before the *conditional expression* is tested. The syntax is:

continue ;

-o-

Continue Statement Example

An example of the continue statement is shown below:

```
// Loop until the afternoon (The sys_time() function returns HHMMSS)
while(1)
   if(sys_time() < 120000)
     // Jump to the top of the loop
     continue;
   else
     break;
   endif
endwhile
```

-o-

If statement

If Statement Definitions

The *if_statement* allows code to be executed only if a certain condition holds true. The syntax of the if_statement is as follows:

if (expr **)**
   statement_block
{ **else**
  statement_block }
endif

If the *conditional expression,* expr, evaluates to a non-zero value then the statement_block after the **if** is executed. If the expr evaluates to zero then if there is an **else** statement then the statement_block after the else is executed otherwise execution jumps to the **endif** statement.

-o-

If Statement Example

An *if_statement* can either have an **else** clause or just have the **if** clause without an **else.** Both of the following statements are valid:

```
if(i == 10)
   applog("Woohoo i equals 10");
else
   applog("Doh! i does not equal 10");
endif
```

```
if(i == 10)
   applog("Woohoo i equals 10");
endif
```

Any expression can be used for the *conditional expression* and the statement_block for the **if** clause will be executed if this expression evaluates to a non zero value. For example all of the following are valid:

```
// this will alway be true
if(1)
  applog("You will always see this")
endif

// this will always be true unless a==1
if(a-1)
  applog("If you see this then a isn't 1");
endif

// this will always be true - but might be a BUG!
if(a=3)
  applog("You will always see this - but did you want to?")
endif
```

The last example shows that an assignment expression is a prefectly valid expression for a conditional statement but is also a common cause of bugs in applications since what is usually meant is:

```
// This is what was probably meant.
if(a==3)
  applog("if you see this then a must be 3");
endif
```

To prevent this kind of bug it might be better to use the TE style comparison operator **eq.** For example:

```
if(a eq 3)
  applog("if you see this then a must be 3");
```

```
endif
```

(see More on Operators)

-o-

Switch Statement

Switch Statement Definition

A switch statement allows a TE program to make a decision based on the value of an expression. Several choices are specified, when a matching choice is found, the following statements are executed. The syntax of the switch statement is as follows:

switch (expr **)**
{ **case** expr **:** statement_block
{ **case** expr **:** statement_block …
{ **default** : statement_block } } }
endswitch

After the switch statement there are zero or more case statements followed by an optional default statement. The way it works is that the value of the switch expr is evaluated then whichever **case** expr matches this value has its statement_block executed then control jumps to the **endswitch** statement. If none of the **case** expr values match then the **default** statement_block is executed if one is present.

-o-

Switch Statement Example

Below is an example of a simple switch statement which is checking for which DTMF digits have been entered:

```
// Switch example using Aculab speech functions from CXACUDSP.DLL
switch (Digit)
     case "1":
          SMplay(voxchan, "one.vox");
     case "2":
          SMplay(line, "two.vox");
     case "#":
          SMplay(line, "pound.vox");
     default:
          SMplay(line, "invalid.vox");
endswitch
```

There are two important things to note about the switch statement:

a) The comparison between the **switch** expr and the **case** expr values is carried out using the **streq** comparator NOT the == (or **eq**) comparator. This means that the values are compared as *strings* not as integer values. So in the following example the second case will be carried out NOT the first:

```
switch ("01")
```

```
   // This is actually doing a streq between "1" and "01"
   case 1:
      applog("This won't be executed");
   case "01":
      applog("This will be executed");
endswitch
```

b) Unlike in the 'C' programming language you do not need to include a break_statement between each **case** statement. In the TE language the **case** statements do NOT drop through – after a **case** has been matched and the statement_block for that case has started execution, as soon as the next **case** statement is encountered the program will jump to the **endswitch** statement. In 'C' if a break_statement is not encountered then the statement_block for the next **case** will be executed as well. If you want to 'drop through' in the TE language then you should explicitly use a goto_statement to do so.

-o-

More on Expressions

Expression Types

There are various types of expressions which can be categorised as follows:

a) Arithmetic expressions - expressions that are use arithmetic operators.
b) Logical expressions - expressions that evaluate to true ("1") or false ("0").
c) Assignment expressions - expression that assign a value to a variable.
d) Constant expressions - expressions that have a contant value.
e) Function expressions - function calls as expressions.

These are described in more detail in the following sections.

-o-

Arithmetic Expressions

Arithmetic expressions include the following types:

Expression	**Result**
exprA + exprB	Add exprA to exprB
exprA – exprB	Subtract exprB from exprA
exprA * exprB	Mulitply exprA by exprB
exprA / exprB	Divide exprA by exprB
exprA % exprB	Modulo division of exprA by exprB (i.e the remainder)

- exprA Change the sign of exprA

Some examples of arithmetic expressions are as follows:

```
1+2
a-2
4/a+b
```
etc.

However the above definition leaves some ambiguity about the order of evaluation. For example in the following expression is ambiguous:

```
1-2+4
```

It is not clear whether this should be evaluated as (1-2)+4=3 or as 1-(2+4)=-5. To clear up this ambiguity then all mathematical operators are given a level of precedence (i.e. a definition of the order of evaluation of operators). The order of precedence for mathematical operators is as follows (strongest first):

Operator	**Associativity**	**Name**
-	Left (unary)	Change sign
+ -	Left (binary)	Add and Subtract
* / %	Left (binary)	Mulitply, divide and modulo

The operators that appear on the same line have equal precedence to each other and it is their associativity that defines the order that they are evaluated. Left associativity means that they are evaluated in order from left to right in the expression (right associativity means that they are evaluated from right to left in the expression (non of the arithmetic operators have this)).

This clears up the ambiguity of expressions like:

```
1-2+4*3
```

since the precedence and associativity defines that this will be evaluated as

```
(1-2)+(4*3)
```

The higher precedence of the multiply operator forces the (4*3) to be carried out first and the left associativity forces the (1-2) to be carried out before the addition.

However it is often better to use brackets to explicitly define the evaluation order than to try and remember the precedence and associativity of all the operators.

The result of an arithmetic expression is always a signed 32 bit integer value. Provision has been made to allow for 64 bit integer expressions in the future (hence the fact that 21 bytes is set aside for an **int** variable type).

32 bit signed integers range from -2147483648 to 2147483647 so no value higher or lower than these two limits can be returned from an arithmetic expression. The values will ‘wrap around’

so for example if you add 1 to 2147483647 you will get -2147483648 and if you subtract 1 from -2147483648 you will get 2147483647.

The result of an arithmetic expression is always evaluated to a null terminated ASCII string so it is important to make sure that a variable is long enough to store the result.

For example in the following code:

```
var a:2;

main
  a=99+1;
end
```

Since a is only two bytes long and the expression 99+1 evaluates to a three character string "100" then the result will be truncated to "10" when it is stored in the variable **a.** Care should always be taken that data is not lost when assigning values or strings to variables if the variable is not long enough to hold the value and it is advisable to assign at least 22 bytes for any variable that will hold a numeric value (or define it as type **int**).

Note that if a non-numeric string is used in an arithmetic expression the string will be converted to a numeric value using the following rules:

a) Trailing spaces are ignored.
b) Conversion starts from the first numeric character (or sign (+ or -)) and stops when the first non-numeric character is encounter.

Below are some examples of what each of the following strings is evaluated to:

String	**Evaluated to**
" 12"	12
"12 13"	12
"12abc"	12
" -12a"	-12
"abc"	0

Notice that any string whose first character (after any trailing spaces) is a non-numeric value is always evaluated as zero. As well as for arithmetic expressions these conversion are always carried out before assigning a value to a **int** type variable.

-o-

Logical Expressions

Logical expressions are those expressions that give a logical (or Boolean) result – i.e. TRUE or FALSE where TRUE is represented as "1" and false is represented as "0".

Logical operators include the **and, or, not** statements (and their symbolic equivalents: &&, ||, !) as well as the comparison operators (greater than, less than, greater than or equal to, less than or equal to , equals, string equals, string not equal).

The language allows for ‘C’ style logical and comparison symbols as well as additional TE language equivalents where appropriate. It is up to the programmer whether to use the ‘C’ style symbols or to adopt the TE language versions depending upon personal preference. The ‘C’ style symbols were included in the language since these symbols are well known and used by a large number of programmers and it is often difficult to remember to swap to the TE language operators causing an annoying number of syntax errors to be introduced to the code by the absent minded programmer (i.e. the author).

The list of Logical and comparison operators is as follows:

‘

‘C’ style operator	TE Language equivalent	Description
&&	and	Logical AND
||	or	Logical OR
!	not	Logical NOT
>	>	Greater Than
>=	>=	Greater or equal
<	<	Less Than
<=	<=	Less or equal
==	eq	Equals
!=	<>	Not equal
n/a	streq	String equals
n/a	strneq	String not equal

The ‘C’ style operators may be useful for ‘C’ programmers who are used to using these, but there are certain disadvantages. The first is that the symbolic nature of all the symbols makes the code more difficult to read and ends up looking like some kind of alien hyroglyphics:

```
if((!a && !b) || c==d)
```

is harder to read than:

```
if((not a and not b) or c eq d)
```

Also the ‘C’ style equals operator (==) is easy to mix up with the assignment operator (=). Statements like the following are common causes of bugs in C programs:

```
// This conditional expression is always true
// This is a bug if 'if(a==1)' was what was meant..
if(a=1) ...
```

The use of **eq** rather than **==** reduces the chance of introducing this kind of bug. Therefore for beginners or non C programmers it might be advisable to use the TE language versions in preference to the ‘C’ style versions if possible, but it is purely down to programmer preference.

Also notice that in the TE language there are two additional comparison operators: **streq** and **strneq.** All the other comparison operators carry out a numeric comparison of the terms,

whereas **streq** (string equals) and **strneq** (string not equal) carry out a *string* comparison. There is a subtle difference between being equal as a number and equal as a string. A string is converted to a number by taking all the characters up to the first non-digit. So, "123", "0123", and "123ABC" are all equal numerically but are different when compared as strings.

This is especially important to remember when making checking for DTMF input using the pound or star keys:

```
"*" eq "*"          // This gives True
"#" eq "*"          // This also gives True!
```

because we are comparing zero with zero. Be sure to use streq, not eq, when comparing non-numeric values.

Logical conditions can include arithmetic operators and parentheses just like other expressions. In fact, there is no distinction between logical conditions and other expressions except to convert the final result to a logical value (“1” or “0”) rather than a numerical value. The following are all valid expressions:

```
a=123 + (5 > 3);    // a will be set to 124 since (5 > 3) evaluates to TRUE ("1")
a=1000* (not 234)   // a will be set to 0 since (not 234) evaluates to FALSE ("0")
```

However logical conditions are normally used in if-statements and to control for, while and do..until loops.

-o-

Assignment Expressions

Assignment expressions assign value to a variable, an array or the contents of a pointer to a variable. The simplest type of assignment expression is as follows:

```
a=1
```

where the variable on the left hand side is set to be equal to the value of the expression on the right hand side (in this case 1). This expression can be made into a statement by adding a semi-colon to the end (see rules of syntax above):

```
a=1;
```

or it can by part of a larger expression

```
if(b eq (a=1)) ...
```

Assignment expressions are examples of expressions that have ‘side effects’ since they alter the value of a variable. Some assignment operators cause the side effect to happen before the expression is evaluated and some cause the side effect to happen after the expression has been evaluated.

Below is the set of assignment expressions supported by the TE language:

Expression	**Effect**	**Side Effect**

A = B	Put the value of B into A.	After
A += B	A = A + B	After
A -= B	A = A – B	After
A /= B	A = A / B	After
A *= B	A = A * B	After
A++	A = A + 1	Before
++A	A = A + 1	After
A--	A = A – 1	Before
--A	A = A – 1	After

Here's some examples to show the difference between those side effects that happen before and after the expression has been evaluated.

```
a=1;
b=++a; // b will be set to 2
```

In the above example **b** will be set to 2 since the **++a** happens before the expression is evaluated (so **a** is incremented then assigned to **b**)

```
a=1;
b=a++;   // b will be set to 1
```

Here **b** will be set to 1 since the **a++** happens after the assignment expression is completed.

-o-

Constant Expressions

There are four types of constant values that can be used in the source code of the TE language:

Decimal Integer constants
Hexadecimal Integer constants
String constants
Predefined constant identifiers.

Decimal integers can be used wherever a **rvalue** is valid (E.g. in expressions and assignments. These numbers may optionally be prefixed with a + or – sign, for example:

```
a=123;
b=+47+12;
c=-123456576;
```

Decimal number constants are converted to null terminated ASCII strings before they are stored or used by the TE language.

Similarly ***hexadecimal integers*** can be used wherever a **rvalue** is valid and must be prefixed with the characters 0x, for example:

```
a=0xff;
b=0xEE7;
```

Either upper or lower case letters can be used in the hexadecimal number and the compiler will convert the values to a decimal numeric string before using or storing the value. Note that this is a compile time conversion and hexadecimal ***string*** values will NOT be converted at run-time. Therefore the following statements are not valid if you are assuming a will evaluate to decimal 15:

```
a= "0xf"    // Compiler will not convert this as it's a string
b=a+1;      // This evaluates to 1 since "0xf" evaluates to 0
```

Since strings are evaluated as decimal values "0xf" will be evaluated as 0 since the first non-numeric character ('x' in this case) stops the conversion.

The difference between the following two statements should be understood:

```
a=0xf;   // Compiler will convert to the decimal string "15"
b= "0xf"; // Compiler will store the string in quotes 'as is'
        //i.e "0xf" will be stored and any arithmetic operation
        // or assignment to int type at run-time will
        // evaluate this to "0"
```

string constants must be surrounded by double quotes and are stored in **var** types 'as is' without conversion. Here is an example of a string:

"this is a string"

Note however that if the **var** type is not long enough to hold the string then it will be truncated before storing in the variable. For **int** type variables or in arithmetic expressions the string will be converted to a numeric string before being stored or used.

Within a string a backslash character has special meaning and is known as an *escape* character since it allows non-printable an other special characters to be included in the string. The following *esccape sequences* are interpreted by the compiler as follows:

Sequence	**Puts this single character into the stored string**
\\	Single backwards character
\q	Double-quote "
\r	Carriage-return (hex 0A)
\n	New-line (hex 0D)
\t	Tab (hex 09)
\xx	Byte with hex value xx, eg \09 would have the same effect as \t. It is illegal to use `00.

Predefined constant identifiers are constant that have been defined using the **const** declaration in a declaration_block. For example, if the following constants will all evaluate to the same string "255" and can be used anywhere that an **rvalue** is valid.

```
//All three of these constants will evaluate to the same sting: "255"
const a=255;
const b=0xff;
const c= "255";
```

```
...

total=a+b+c;    // = 255+255+255
```

-o-

Function Expressions

A function expression is where a function is called and the return value of the function is used as the result of the expression.

For example, if a function is defined that returns the product of two numbers:

```
func product(a,b)
   return (a*b)
end
```

Then this function can be used in an expression wherever an rvalue is valid:

```
a=product(2,3);   // a is set to 6
b=product(2,2)*product(product(2,2),2);   // b is set to 32
```

A function expression is also an expression that can be said to have ‘side effects’ since it can alter the values of variables.

-o-

More on Operators

TE Language Operators

Apart from the logical, assignment and arithmetic operators there are a number of other operators used by the TE language.   Below is the full list of TE operators showing their precedence (highest first) and their associativity:

Operators	Associativity
++ --	Left
- & *	Left (unary)
&	Left (binary)
* /	Left (binary)
+ -	Left (binary)
> < >= <=	Left
== eq <> streq strneq	Left
! not	Left
&& and	Left
|| or	Left

© Zentel Telecom Ltd, 2009

?:	Right
= += -= *= /=	Right

This set of TE language operators can be categorised into the following groups:

Arithmetic Operators
Comparison Operators
Logical Operators
Assignment Operators
Indirection Operators
Miscellaneous Operators.

The following sections describe these types of operators in more detail.

-o-

Arithmetic Operators

Arithmetic Operators:

Expression	Result
A + B	Add A and B.
A – B	Subtract B from A.
A / B	Divide A by B and produce an integer by dropping all decimals (for example: 9/4 gives 2).
A * B	Multiply A and B.
-A	Change the sign of A (same result as 0-A).

-o-

Comparison Operators

Comparison Operators:

Expression	Result
A > B	TRUE if A greater than B.
A >= B	TRUE If A greater than or equal to B.
A < B	TRUE if A less than B.
A <= B	TRUE if A less than or equal to B.
A eq B	TRUE if A equal as number to B.
A <> B	TRUE if A not equal as number to B.
A streq B	TRUE if A is same string as B. See also String Values.

A strneq B	TRUE if A is not the same string as B. See also String Values.

-o-

Logical Operators

Logical Operators:

Expression	Result
A and B	TRUE if A is TRUE and B is TRUE.
A && B	TRUE if A is TRUE and B is TRUE (C style).
A or B	TRUE if either A or B or both are TRUE.
A || B	TRUE if either A or B or both are TRUE (C style).
not A	TRUE if A is FALSE; FALSE is A is TRUE.
! A	TRUE if A is FALSE; FALSE is A is TRUE (C style)

-o-

Assignment Operators

Assignment Operators:

Expression	Result
A = B	Value of B is assigned to A.
A += B	Same as A = A + B
A -= B	Same as A = A - B
A *= B	Same as A = A * B
A /= B	Same as A = A / B
A++	A + 1 (result is A before adding 1)
++A	A + 1 (result is A after adding 1)
A--	A - 1 (result is A before subtracting 1)
--A	A - 1 (result is A after subtracting 1)

-o-

Indirection Operators

Indirection Operators:

Expression	Result
&A	Pointer to variable A.

*A The variable pointed to by A

C programmers will be familiar with the idea of indirection from the unary & and * operators. The TE language provides a similar mechanism whereby the unary & operator can be applied to a variable name and it returns a pointer to the variable (actually it returns the offset of the variable in the internal variable table assigned by the Telecom Engine).

Once a pointer to a variable has been obtained using the & operator then the unary * operator can then be used to access the variable the this pointer refers to.

For example,

```
int a, b, c;

main
   b=123;   // assign a value to b
   a = &b;  // a holds the pointer to variable b
   c=*a;    // c is assigned the value of b
   *a=10;   // b is set to 10
endmain
```

In the above program the variable **b** is assigned the value 123. Then the variable **a** is set to hold a pointer to **b.** This will actually be the integer offset into the internal variable table held by the Telecom Engine, so in the above case this will be 1 (**a** would be at offset 0, **b** at 1, **c** at 2). Don't rely on these offset values though since the way variables are stored in the Telecom Engine may change in the future.

The next line sets **c** to the contents of the variable pointed to by **a** which in this case is the contents of variable b – i.e. 123.

Finally the contents of the variable pointed to by **a** is set to the value 10, so in fact variable **b** is now set to 10.

The unary * operation is called "dereferencing".

The unary * operator may only be applied to variables or function arguments. The expression (*2), though is illegal -- it does not mean "variable number two".

The main use this mechanism is to allow a function to alter the value of a variable passed as an argument to a function. In the TE language function arguments are passed "by value". This means that when a function is called, each function argument is evaluated to a string, and the string values copied to an area of memory that the function can access. The function therefore only has a copy of the values passed to it and can't alter the values held in the original variables.

For example, in the following program:

```
var x:3;

main
   x = "ABC";
   f(x);
   applog("x=",x);
endmain
```

```
func f(argument)
   argument="ZYX";  // this is illegal
endfunc
```

This would result in a compiler error since the function is attempting to change the value of a function argument which is in fact just a copy of the original string that is now stored on the program stack. If the compiler allowed this value to be changed then it could result in the stack being overwritten if a longer string was assigned. In fact it doesn't usually make sense to try and change the value of arguments like this so it has been made illegal in the TE language.

What is usually required is for the original value of the variable passed to the function to be changed and this is what the indirection operators are for. Instead of passing a copy of the variable to the function, a pointer to the variable can be passed instead. For example:

```
main
   var x:20;

   x= "ABC"
   f(&x);   // pass a pointer to the variable x
   applog("x=",x);   // x has now been changed by f()
endmain

func f(arg)
   *arg = "ZYX";   // Use the dereference operator to change the variable pointed to by arg
endfunc
```

This program will assign the string "ZYX" to variable **x**, and write this string to the screen.

Note that the unary & operator cannot be applied to function arguments. The following is illegal:

```
func f(a)
   g(&a);     // Illegal
endfunc
```

However, variable numbers may (like any other string value) be passed through any number of levels of function calls:

```
main
var x:50;
   f(&x);
   applog("x=",x); // value of x changed by function g()
endmain

func f(af)
   g(af);
endfunc

func g(ag)
  *ag = "New value";
endfunc
```

Caution needs to be used when using the indirection operators. The Telecom Engine provides no protection from meaningless code like:

```
x = 36;
```

```
*x = "ABC";
```

which would unexpectedly over-write the variable at offset 36 in the TE variable table.

-o-

Miscellaneous Operators

Miscellaneous Operators:

Expression	**Result**
A & B	String made by adding B to the end of A (string concatenation).
A ? B : C	B if A is TRUE; C if A is FALSE.

-o-

Ambiguous Operators

It can be seen from the above tables that the three symbols

& - *

can each stand for two different operators, depending on how the symbol is used. One usage is as a binary operator, where two different values are combined into one. The other usage is as a unary operator, where a single value is changed to a new value.

The context that the operator is used in (unary or binary) defines the meaning of these symbols in each situation, thus clearing up any ambiguity.

The meaning of these symbols in each situation is:

Symbol	**Example**	**Meaning**
Unary -	-A	Change sign of A
Unary &	&A	Pointer to variable A
Unary *	*A	The variable pointed to by A
Binary -	A – B	Subtract B from A
Binary *	A * B	Multiply A and B
Binary &	A & B	Concatenate strings A and B

-o-

String Concatenation Operator

There is a special operator that can be used with strings, which is the *concatenation operator*. This is the & symbol which if used is used to join two or more strings together:

```
a="1234";
b="567";
c=a & b & "89";   // c becomes "123456789"
```

The result above is that the variable **c** will be set to “123456789” which is a concatenation of the strings stored in variable **a** and **b** and the string constant “89”.

-o-

The Conditional Expression Operator

The Conditional Expression uses the ? and : operators and takes the form:

A ? B : C

Which can be read as: If A is TRUE then the expression takes the value B else it take the value C.

For example:

```
b=(a<0)?-a:a;   // Sets b to the positive value (modulus) of a
b=(a > 255)?255:a; // set b to the value of a up to the limit 255
```

-o-

More on Functions

More on Functions

There are two types of functions that can be called from TE programs:

a) TE language functions (function declared in a **func..endfunc** block).
b) External DLL functions.

There are very few differences between calling these two types of function and the calling syntax is the same.   Both type of functions return *string* values.   However there are a few differences that should be noted:

i) Some DLL functions can take a variable number of parameters whereas TE functions can only take a fixed number of parameters.
ii) DLL functions can result in the calling task becoming suspended (i.e the calling task will block until the calling libary wakes the task up at which point the function will return).
iii) A maximum of 32 arguments can be passed to a DLL function, whereas for a TE language function there is no limit to the number of arguments that can be passed to a function (except for a limit imposed by the stack depth or stack size which might be exceeded if not big enough to handle all of the function parameters).

For example the CXTERMX.DLL library has functions for displaying information to a terminal console and for accepting input from the keyboard.   Some of these functions accept a variable number of parameters.   For example the applog() function which displays a message to the

terminal screen (and writes the message to the application log) can take any number of parameters between 1 and 16. Each of the parameters passed is concatenated together to make the final message string. For example the following two calls would display the same message:

```
applog("This is a string");
applog("This"," is ","a string");
```

This allows the values of variable to be easily displayed:

```
a=3;
b=34;
applog("a=",a," b=",b);
```

The kb_get() function provided by the CXTERMX.DLL library is an example of a function that causes the calling task to be suspended. A call to kb_get() will not return control to the task until a key has been pressed on the keyboard (i.e. the function will block). Of course this only blocks the calling task, all other tasks running on the Telecom Engine will continue to process commands(unless they are also in blocking functions):

```
var ch:1;
ch=kb_get();   // this will 'block' until a key is pressed
applog("Got key=",ch);
```

As mentioned in the discussion about indirection operators, arguments are passed to functions by value which means that a copy of the original value is pushed onto the program stack which can then be referenced by the function. It is illegal to try and alter the value of an argument to a function:

```
func f(arg)
   arg=2;   // illegal
endfunc
```

If you wish to alter the value of a variable inside a function then a pointer to that variable should be passed rather than the value itself and the dereference operator should then be used inside the function. It is advisable to indicate in the name the arguments of a function that it will be a pointer (e.g. by prefixing ptr_ or _p_ or something similar)

```
main

var str:127;

  f(&str);  // Pass pointer to variable

endmain

func f(_p_arg)
   * _p_arg= "New value"
endfunc
```

To return from a TE language function the return_statement can be used which can take one of two forms:

return ;

or

return expr ;

A **return** on its own will return the empty string "" as the result of the function, otherwise the value of the expr will be returned if one is given. If the **endfunc** statement is reached before encountering an explicit return statement then the empty string "" will be returned by the function.

TE language functions can be declared inside the main .TES source file, or else they can be declared in a separate .FUN source file where the name of this source file is the same as the function itself. For example a function called myfunction() would be declared in a .FUN file called myfunction.fun.

Also TE language functions can be merged into a TEL library file and linked into the main application by using the –L option with the TCL compiler (See TCL compiler reference).

-o-

More on Variables

More on Variables

All variables in the TE language are stored as null terminated character strings. Variables can be declared with lengths up to 255 characters long:

```
var a:255;   // This is OK
var b:256;  // illegal..
```

In fact an additional byte is set aside by the Telecom Engine to allow for the terminating null character, so a **var** of length 255 would take up 256 bytes.

Integer type variables are also stored as character strings and the TE language sets aside 22 bytes for each variable declared as type **int** (1 bytes for the sign, 20 bytes for the digits and one byte for the null terminator). Before assigning a value to a variable of type **int** the Telecom Engine will convert the assigned string to a numeric string value.

At the start of an application all **var** type variables are initialised to the blank string "" and all **int** type variables are initialised to 0.

The variables declared in the **declaration_block** before the **main..endmain** section are *global* variables that can be accessed by any function in the program (this applies to contants as well).

All variables declared inside functions are *static* variables that can only be accessed from within that particular function. The fact that these variables are *static* means that they will retain their values between function calls.

For example:

```
main
```

```
  f();
  f();
  f();
endmain

func f()
int a;

   a++;
   applog("a=",a);
endfunc
```

Each time the function f() is called then the variable **a** is increased by one so by the third call **a** will have the value 3. If this behaviour is not what is required then it is up to the programmer to make sure all variables are initialised to the correct value at the top of each function.

Variables can be declared anywhere within a statement_block and can then be accessed from anywhere from that point on in the code to the end of the **func..endfunc** or **main..endmain** block. Variables declared within a function cannot have the same name as a function argument, global variable or global constant:

```
int a;
main
   f("Hello");
end

func f(str)
  var a:20;  // illegal - same name as global
  var str:10;  // illegal -  same name as function argument
endfunc
```

Different functions may declare variables of the same name so long as they don't conflict with another variable, argument or constant that is in scope.

-o-

More on Arrays

The TE language allows for **var** type and **int** type arrays of up to 256 elements long. The declaration of an array requires that the first and last index values be specified. For example:

```
var a[1..20];
```

Defines an array with indexes ranging from 1 through to 20. Any attempt to access an element beyond this range will result in an error at runtime being written to the log. An attempt to access an array element out of range will result in the empty string "" being returned.

```
var a[1..20];

b=a[0];   // b will be set to "" and an error will appear in the error log screen.
c=a[21];   // c will be set to "" and an error will appear in the error log screen.
```

The maximum index range allowed in the declaration of and array is 0 through to 255. Also the lower index range must be less than the upper index range. Therefore the following declarations all illegal:

```
var a[-1..20];    // Illegal: Lower index less than 0
var b[25..256];  // Illegal: Upper index greater than 255
var c[50..1];    // Illegal: Upper index range less than lower
```

-o-

Compiler Directives

Compiler Directives

Compiler directives instruct the compiler to carry out certain actions at compile time and are meta-commands, not really part of the programming language itself. All compiler directives start with a **$** symbol.

The compiler directives recognised by the TCL compiler are as follows:

$include <*filename*>
$if (*constant_expression*) … **$else** … **$endif**

-o-

The $include Directive

The **$include** directive informs the compiler to include the contents of the *filename* into the source code as though the text in *filename* was typed directly into the source.

This is usually used for including files containing common global variable and constant declarations associated with particular function libraries (header files).
For example if the header file MYVARS.INC contains the following declarations:

myvars.inc:
var a:10;
var b:10;
const c="123";

This could be included in a global declaration_block at the top of the program:

```
$include "myvars.inc"

main
  // these are decared in the header file..
  a=1;
  b=c;
endmain
```

Although the **$include** directive is most commonly used to include header files that declare global variable and constants there is no reason why the **$include** directive can;t be used anywhere in the code to include text from another file as though it had been typed directly into the source file.

For example in the following program

```
main
    $include "stuff.txt"
endmain
```

The STUFF.TXT file can contain all the program statements for the application like this:

Stuff.txt

```
   applog("This comes from the stuff.txt file");
   applog("Hello again, world");
```

For header files the convention is to use the extension ".teh" (short for Telecom Engine Header), or if you prefer ".inc" or ".h" are other common extensions used.

An environment variable called INCDIR specifies a semi-colon delimited list of directories where the compile will search for include files (See **TCL Compiler Reference**).

-o-

The $if Directive

The $if (constant_expression) directive allows for conditional compilation. This is useful for allowing the same code to be compiled in two or more different forms (say for different hardware implementations). There are three types of $if statement allowed:

$if (<x> streq <y>)
$if (<x> strneq <y>)
$if (<x>)

<x> and <y> represent constant strings, integers or declared constant identifiers. In the first two forms a string comparison is carried out between <x> and <y> and the compiler will compile the following code (or not) depending on the result of the comparison. For example:

```
$if(HARDWARE_TYPE streq "ACULAB")
    SMplay(vox_chan, "test.vox");
$else
   $if(HARDWARE_TYPE streq "DIALOGIC")
     dx_play(vox_chan,"test.vox");
  $else
      applog("Application compiled for unsupported hardware type=",HARDWARE_TYPE);
  $end
$endif
```

The constant identifier HARDWARE_TYPE is assumed to be declared in a **const** declaration somewhere in scope, else specified using the –D option of the TCL compiler (See TCL compiler reference). In the above application only one line will get compiled depending on the value of HARDWARE_TYPE.

The code in the lines that are not compiled do not even need to conform completely to the TE language syntax, however the text is still parsed *token by token* and so there are some restrictions as to what can be contained in the non-compiled side of the $if..$else..$endif statement. Most

notably a string token cannot span more than one line of the source code.

For example the following will compile successfully and the the compiler will simply excluding all text for the $else side of the following code:

```
const NO_NONSENSE= "1";

$if (NO_NONSENSE streq "1")
    // This code must be valid since it will get compiled
    applog("Not nonsense");
$else
    blah blah this doesn't need to make any sense since it will not get compiled!
    akjsd;lksad;kds;
    ;laksd;lask;laskd'
    No syntax errors to be seen anywhere..
    but don't change NO_NONSENSE to 2 or the compiler will explode
$endif
```

However in the following excerpt a string in the $else side of the $if directive spans more than one line and will result in a syntax error:

```
const NO_NONSENSE= "1";

$if (NO_NONSENSE streq "1")
    // This code must be valid since it will get compiled
    applog("Not nonsense");
$else
    " This string will cause the compiler to fail with
        a sytax error because a string token cannot span
        more than one line of the source code"
$endif
```

The final form of the $if statement is $if(<x>). This requires that a constant name be given and the compiler simply checks if this name has been defined. Even if the value of the constant is blank or zero this statement will hold true. For example:

```
const X_IS_DEFINED = "";

$if(X_IS_DEFINED)
   applog("X_IS_DEFINED is defined");
$endif
```

The applog statement would get compiled since X_IS_DEFINED is defined. The same effect could be achieved by compiling with the –D option in the TCL compiler.

$if …$else…$endif directives can be nested to any level..

-o-

The TE Compiler

Introduction

The Telecom Engine Compiler (TCL.EXE) is a command line compiler that compiles TE source code files (.TES extension) into the TE byte-code (.TEX extension), which can then be executed by the TE Run-Time Engine.

To use the compiler it is necessary to open a command prompt (DOS box) and then to change to the directory where the source code resides. Make sure that the directories that contain the compiler executable and any required DLLs (E.g. Telecom Engine Library DLLs) are all specified in the PATH environment variable (Control Panel->System->Advanced->Environment Variables).

To compile a source code file, the command line format is as follows:

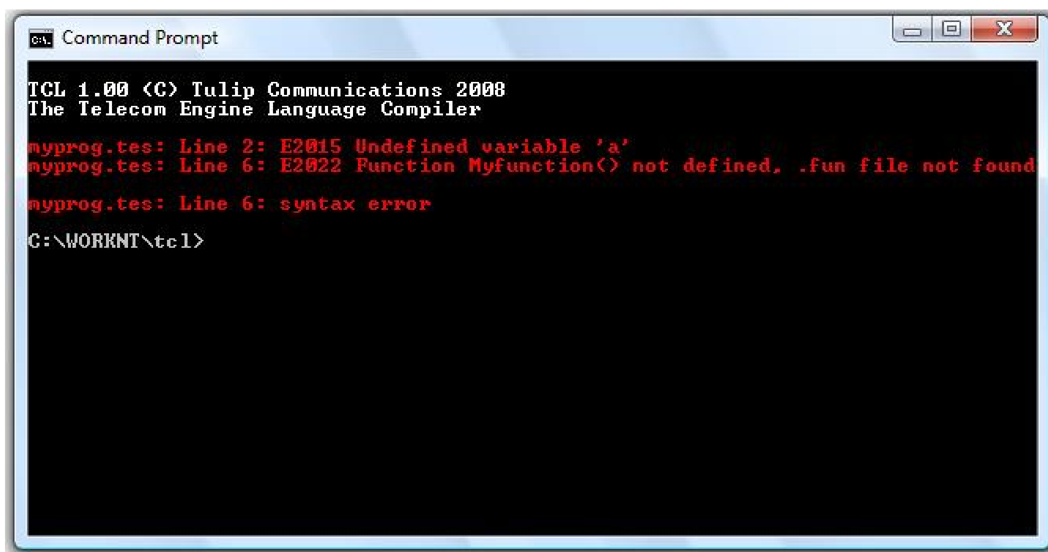
```
TCL [options] SourceFile[.TES]
```

For example:

```
TCL Mysource
```

If the source code file compiles sucessfully, without errors, then Mysource.TEX will be created which contains the byte-code for the program.

If any errors are encountered when compiling then these errors will be printed to the console window in RED (see below):



-o-

Compiler Options

All compiler options should be specified one after another (separated by spaces) and preceded with a '-' character. For example:

```
TCL -u -f -s120,2048 MyProg
```

Some compiler options are specified on their own without additional arguments or values (such as

© Zentel Telecom Ltd, 2009

the **-u** and **-f** options above), whereas others require one or more additional arguments/values to complete the option specification (such as the **-s120,2048** option shown above).

Below is a description of all of the command line options that can be specified with the Telecom Engine Compiler (TCL.EXE):

Option	Additional Arguments	Description.
-u	None	Prevent 'Variable not used' Warnings.
-f	None	Prevent Debug information being written to TEX file.
-v	<level>	Specify verbose mode (<level>=0..9).
-L	<Libname[.TEL]>	Specify a TE libary file to search for functions to compile and link to.
-r	<DLL1[;DLL2[;DLL3...]]]>	Specify one or more TE DLL or DEF files to load.
-s	<Stack Depth,Stack Size>	Specify the stack depth and stack size for the program
-d	<name>	Define a constant and set its value to 1
-d	<name=value>	Define a constant and set its value to <value>
-e	None	Output all errors to TCL.ERR file.
-n	None	Don't load the default TCL.DEF file.
-c	<flag>	Set flag to 0 to disallow 'C' style operators (&&,||,==,!,!=), set to 1 to allow these
-z	<definition file name>	Create library definition file from loaded DLLs and DEF files.

Each of these options is described below in more detail:

-u Prevent 'Variable not used' Warnings.

Under normal operation the compiler will issue warnings when a variable has been declared but not used in the application. However in a large application (particularly when using 'include' files to define constants and global variables) it might be desirable to switch these warnings off if too many are being generated.

-f Prevent Debug information being written to TEX file.

By default the compiler will write various debugging information to the excecutable binary output file (.TEX file). This information includes the source code file names and the line numbers within the source code for each statement. By specifying the **-f** option then the compiler will not include this information into the .TEX executable file (making the .TEX file smaller).

-v<level> Specify verbose mode (<level>=0..9).

This allows the compiler to be run in verbose mode which is usually used for debugging. Setting verbose level 1 or above will increase the amount of output from the compiler for each increasing level. Verbose more 0 turns off the verbose mode for the compiler.

-L<libname.[TEL]> Specify a TE libary file to search for functions to compile and link to.

A number of function source code files can be combined into a singel TE library file (.TEL extension) using the Telecom Engine Utility Program: MKTEL.EXE. This option allows these library files to be searched for functions that will then be compiled and linked into the program in the same way as normal functions found in .FUN files. Note that .FUN files wlil always override any files defined in a .TEL library file.

-r<DLL1[;DDL2[;DLL3...]]] Specify one or more TE DLL or DEF files to load.

In order to link to the TE DLL libary functions then the compiler must load these DLLs to obtain the function definitions (or the .DEF file that represents the function definitions for the DLL). The **-r** option allows for a semi-colon separated list of DLLs (or .DEF files) to be loaded at compile time so that the function definitions can be accessed by the compiler. For example:

```
TCL -rCXTERMX;CXSYS;CXTASK.DEF MyProg
```

The above statement would attempt to load the TE Standard library DLLs: CXTERMX.DLL, CXSYS.DLL and CXTASK.DEF in order to resolve all the function names defined in these libaries. Note that if no extension is specified then it is assumed that it is a DLL that is to be loaded. See Loading DLLs and .DEF files for more information about this.

-s<Stack depth,Stack Size> Specify the stack depth and stack size for the program.

The header of the .TEX executable file contains two fields that defines the stack depth and stack size to allocate at run-time for the application. By default these are set to ***Stack Depth=256*** and ***Stack Size=4096*** which should be sufficient for nearly all but the largest and most complex applications, however this option provides the means to increase or decrease these values if required.

The *Stack* is a special area of memory which is used by the program at run-time to evaluate the results of expressions and to store the values of any arguments that are passed to functions and to hold the return value.

The ***Stack Depth*** defines the number of items that can be *pushed* onto the stack at any one time. The main thing that uses up stack space is when a function is called since being passed to the function are pushed onto the stack before calling the function (plus the return address and number of arguments is also puched onto the stack) and these are not removed until a return statement is encountered. Therefore if your application calls functions with a large number of arguments and/or if you have functions nested deeply in your application (functions that call functions that call functions etc for many nested levels), or if you call functions recursively (i.e functions that call themselves), then an analysis might need to be carried out to see if the stack depth needs to be increased. As a rule of thumb, find the part of you program that makes the most nest function calls (or calls functions with a large number of arguments) and then add up the number of arguments and add two for each function called (the extra two are for the return address and the number of arguments which are also pushed onto the stack). If this could exceed the stack depth you have specifed then you should increase this.

The other thing the stack is used for is for holding intermediate values when evaluating

expressions. This might become significant if you have very long and complicated expressions and you are already deep into nested or recursive functions, but usually and expression will not need more the four or five entries on the stack before the expression is resolved to a single value and removed from the stack.

The ***Stack Size*** is the physical number of bytes that the stack contains. Usually variables pushed onto the stack will range in length from only a few characters (E.g. when a number is being used) or several hundred characters for long strings. If your application passes alot of long string values as arguments to functions then you might want to consider increasing the size of the stack to ensure that the stack space does not run out.

As mentioned previously, the default stack size should be suffient for nearly all applications (I've yet to write an application that required the stack to be increased).

-d<name> Define a constant and set its value to "1".

This option causes the constant to be defined with the name **<name>** and its value will be set to "1". For example:

```
-dISDN
```

The above will define a constant called ISDN and set its value to "1". This is equivalent to the compiler encountering the following statement within the global declaration block of the main source file:

```
const ISDN="1";
```

-d<name=value> Define a constant and set its value to <value>

This is a variation of the previous option and allows for a constant to be defined and its value to be set to some <value> string. For example:

```
-dISDN_SUPPORT=OFF
```

The above will define a constant called ISDN_SUPPORT and set its value to "OFF". This is equivalent to the compiler encountering the following statement within the global declaration block of the main source file:

```
const ISDN_SUPPORT="OFF";
```

The **-d** option is often used in conjuction with the **$if...$else...$endif** compiler directives which allow condition compilation of portions of the source code (see Compiler Directives).

-e Output all errors to TCL.ERR file.

If this option is specified then all compiler errors are written to the TCL.ERR file in the currect working directory (they are still also written to the console window).

-n Don't load the default TCL.DEF file.

By default the TE Standard Libary Set Definition file (TCL.DEF) is loaded by the compiler so that the names of the TE standard library functions can be resolved without havihng to load the actuall

DLLs. This function prevents the TCL.DEF file from being loaded by the compiler by default (See Loading DLLs and .DEF files).

-c<flag> Allow or disallow the use of 'C' style operators.

Set <flag> to 1 to allow 'C' style operators to be used in the program (or 0 to disallow 'C' style operators (this is the default)). The 'C' stlye operators being referred to here are **&&**, ||, ==, **!** and **!=** which are equivalent to the TE operators **and, or, eq, not** and <> respectively.

-z<definition file> Create library definition file from loaded DLLs and DEF files.

When this option is used it is not required that a program source file be specified since it causes the compiler to output a definition file and it will not compile the source. The definition file that is output will be created from all of the other .DEF and DLL files that have been previously loaded (including the default TCL.DEF and any files specified in the **-r** option).

This is the easiest way to create .DEF definition files from the actually TE DLL. If you don't want the functions in the default TCL.DEF file to be included in the output .DEF file then **-n** option should be specifed. See Loading DLLs and .DEF files for more information about this.

-o-

Environment Variables

The TE Compiler relies on a number of environment variables which define the paths to function files, include files, definition files or library files. If any of these environment variables paths are not specified then the compiler will just look in the current working directory for all files of this type (unless a full path was specified in the TCL.EXE command line for any of them).

Note that once a path is specified by an environment variable then the current working directory is not automatically searched by the compiler unless it is included explicitly in the path variable (E.g. as ".\")

All path variables can specify mulitple paths by providing a list of directories separated by semi-colons, for example:

```
set FUNCDIR=\myproject\common;.\;d:\utilities\functions
```

Below are the path variables that are used by the compiler:

FUNCDIR	Specifies the directories where the compiler will search for all function (.FUN) files.
INCDIR	Specifies the directories where the compler will search for all *include* files (see $include Compiler Directive)
TELDIR	Specifies the directories where the compiler will search for TE library (.TEL) files
DEFDIR	Specifies the directories where the compiler will search for DLL definition

© Zentel Telecom Ltd, 2009

	(.DEF) files. (see Loading DLLs and .DEF files)

In addition to the above path definition variables there are two other environment variables shown below:

TELLIBS	Defines a semi-colon separated list of TE library (.TEL) files to load (equivalent to the **-L** compiler option)
TEDLLS	Defines a semi-colon separated list of DLL or .DEF files to load (equivalent to the **-r** compiler option)

These last two environment variables are the equivalent to swetting either the **-L** or the **-r** compiler options.

-o-

Loading DLLs and .DEF files

The **-r** compile time option provides the mechanism to load the function definitions for external TE DLL library functions. However it is not necessary to always load the actual DLL libary when compiling an application, instead the compiler can be instructed to use a .DEF file instead which is simply a text file containing the function definitions from the actual DLL. If a .DEF file is specified then the compiler does not need to load the entire DLL in order to access the function definitions - it will simply read them from ythe .DEF file instead.

This has two advantages over loading the full DLL at compile time:

a) Since the full DLL does not need to be loaded it makes the compilation faster.
b) Sometimes a DLL will automatically attempt to load other DLLs which may only be present on the run-time machine and not on the machine where compilation is taking place (E.g. the ACULUB DLLs libaries (sw_lib.dll, cc_lib.dll etc)).

By default there is one large definition file (TCL.DEF) that holds the definitions for all of the TE Standard Library Set. The compiler will search for this definition file in the current directory first, then it will search in the directory where the TCL.EXE program resides (this is where the TCL.DEF file will reside by default). By searching the currect directory first this allows the default TCL.DEF file to be overridden by a specific TCL.DEF file in the current directory.

Therefore in general it is not necessary to load any of the TE Standard Library Set DLLs at compile time since usually the compiler will load the default TCL.DEF file which will contain these function definitions. If this is not the behaviour that is required then the **-n** option can be specified which prevents the compiler loading the default TCL.DEF file automatically (in which case individual DLL or .DEF files must be specified for the TE Standary Library Set functions.

A definition file is a text file will the following format:

```
<DLL library  name 1>
<DLL compilation date>
<Number of Funtions>
<Function 1 Minimum arguments>,<Maximum Arguments>,<UniqueFunction ID>,<Number of
Synonyms>,<Function name1>[,function name 2[,...]]>
<Function 2 Minimum arguments>,<Maximum Arguments>,<UniqueFunction ID>,<Number of
Synonyms>,<Function name1>[,function name 2[,...]]>
```

```
...
[
<DLL library name 2>
<DLL compilation date>
<Number of Funtions>
<Function 1 Minimum arguments>,<Maximum Arguments>,<UniqueFunction ID>,<Number of Synonyms>,<Function name1>[,function name 2[,...]]>
<Function 2 Minimum arguments>,<Maximum Arguments>,<UniqueFunction ID>,<Number of Synonyms>,<Function name1>[,function name 2[,...]]>
...
...]
```

For example, below is the definition file for the CXSEM.DLL library:

```
cxsem
Nov 10 2006 11:39:08
4
1,1,115,1,sem_test
1,1,116,1,sem_set
1,-1,117,1,sem_clear
1,-1,16,1,sem_clrall
```

(Note that a maximum or minimum number of arguments of -1 means that a variable number of arguments is allowed).

Definition files can be specified in the **-r** option by specifying the full name of definition file. For example:

```
TCL -r CXTERMX.DEF;C:\DEFFILES\CXSEM.DEF MyProg
```

It is OK to mix DLLs and .DEF files in the **-r** option library list.

Note that if the same function name is defined more than once either in multiple DLL libaries or .DEF files then the one with the lowest unique function ID will be the one that is used.

Also if two functions have the same unique function ID then the one that was defined last in the **-r** option will overwrite the previous definition and will be the one that is used.

-o-

Function Name Resolution

If a conflict occurs between function names between functions defined in .FUN files. .TEL library files or within external DLLs then the order that the compiler will search for functions is as follows:

1) Functions defined in TE DLLs.
2) Functions defined in the main source (.TES) file.
3) Functions defined in .FUN files.
4) Functions defined in TE library files and specified using the -L compiler option.

If two functions have the same name then the above order will define which function will be compiled/linked into the application.

-o-

The TE Run-Time Engine

Introduction

The Telecom Engine Run-Time consists of four parts as follows:

a) The Graphical User Interface (GUI) front-end (TEX.EXE).
b) The Telecom Engine Scheduler (CXDLL.DLL).
c) The Telecom Engine DLL back-end libraries (CXTASK.DLL, CXTERMX.DLL, CXSYS.DLL etc.).
d) The executable byte code files that are run by the scheduler (*.TEX) .

When the GUI front-end (TEX.EXE) is started it first loads the back-end Telecom Engine Libary DLLs that have been specified (either on the command line or in the registry). The initialisation functions for all of these libaries are then called to ensure that the back-end is fully initialised and ready.

You should make sure that the PATH environment variable includes the locations of all the Telecom Engine executables and DLLs.

The TEX.EXE program then loads the Telecom Engine Scheduler (CXDLL.DLL) in a separate thread and starts the scheduler engine.

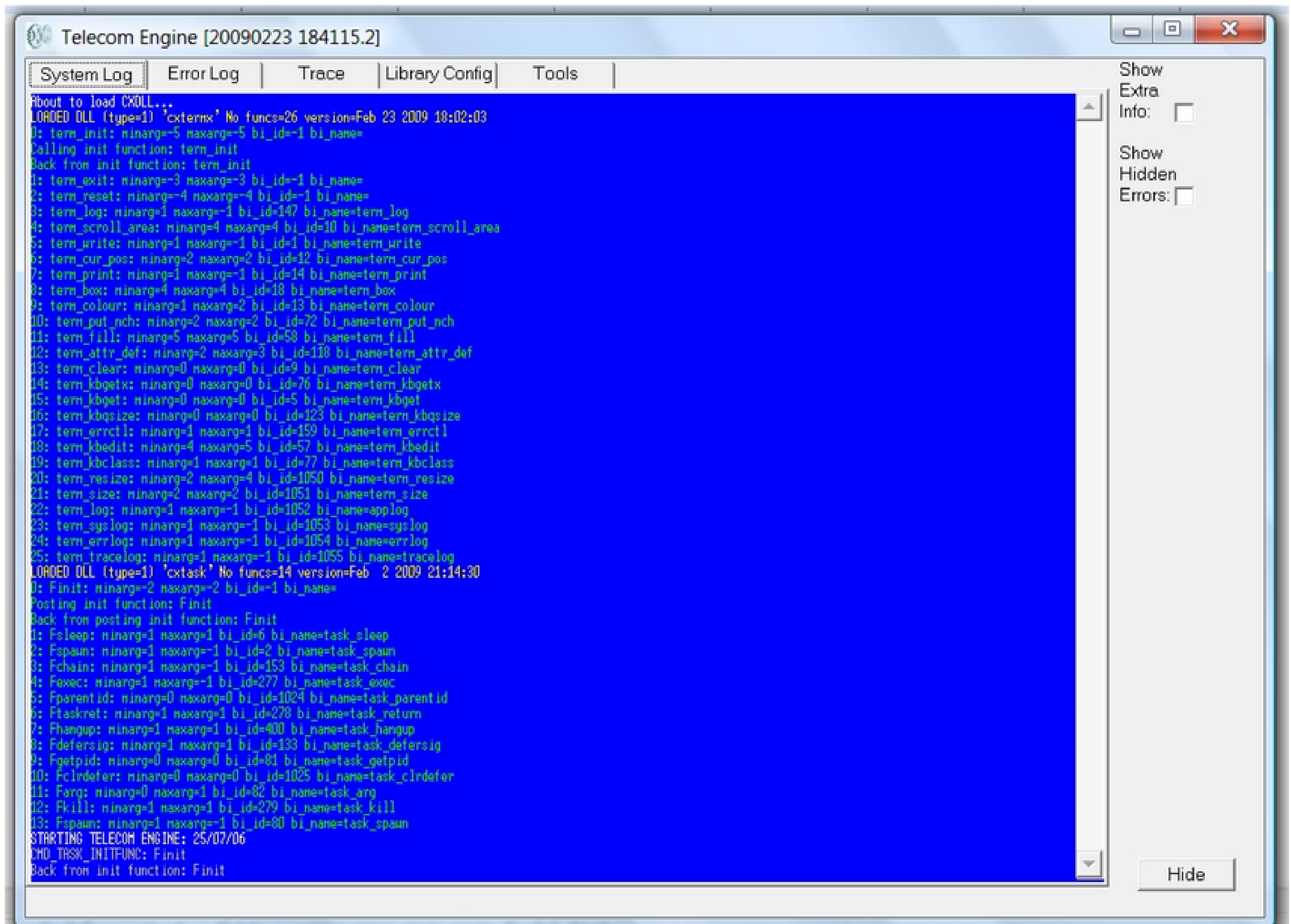
Finally, any Telecom Executable applications that have been specified on the command line are then loaded into the scheduler to begin execution.

Execution will continue until a system exit request is received either from one of the byte-code application tasks or by attempting to close the Telecom Engine GUI appplication window.

The Telecom Engine front-end application has a set of tabbed windows for each of the following:

a) The System Log tab.
b) The Error Log tab.
c) The Trace Log tab.
d) The Library Configuration tab
e) Tools Tab for managing the Telecom Engine.

Below is a screenshot showing the system log tab on a typical system start-up:



-o-

Command Line Options

The Telecom Engine front-end front-end application (TEX.EXE) should be started from the command line. The format required for running the TEX.EXE is as follows:

TEX [options] [List of applications]

The optional [options] can be one of the following start-up options:

Option	Description
-n	Prevent the default back-end libraries from loading at startup
-r<libary list>	Load the following semi-colon separated list of back-end libaries.

The optional [List of applications] is the list of .TEX files to load at start-up (each one separated by a space). Note that the TEX.EXE can be run without loading any byte-code applications for execution if required.

Thus the simplest form of the command line is as follows:

```
tex
```

This will load the default list of back-end libaries and start the Telecom Engine Schedule thread, but no byte-code (.TEX) applications will be loaded for execution (byte code applications can then be loaded from the Tools Tab).

Below is an example where the default libaries specified by the Libary Configuration Tab are prevented from loading by using the -n option, and a new set specified by the -r option are loaded instead. This will also attempt to load the byte-code application TEST1.TEX and start executing it:

```
tex -n -rcxtermx;cxtask;cxsys;cxstring TEST1
```

If necessary, more than one byte code application can be loaded upon start-up by specifying more than one file on the command line (separated by spaces):

```
tex TEST1 TEST2 TEST3
```

The above command line will load the default libaries from the Registry and attempt to load and execute the three byte-code (.TEX) files TEST1, TEST2 and TEST3.

-o-

Registry Settings

A number of settings for the front-end GUI application (TEX.EXE) are stored in the registry under the following key:

HKEY_LOCAL_MACHINE\Software\Telecom Engine\TEX

The following entries can be found under this key:

Entry Name	Description
AlphaSpeed	This is the speed that the Telecom Engine Splash window will fade upon start-up. The AlphaBlend value of the splash screen form will start at 0 (transparent) and every 100th of a second the value of AlphaSpeed will be added to the AlphaBlend property until it reaches 255. The default value of AlphaSpeed is 3 which means it will take 255/3=85 ticks to become completely opaque (i,e just less than 1 second). After the alphablend (fade-in) part of the splash screen cycle the splash screen will display on the screen for one more second before disapearing. To prevent the fade-in part of the cycle from occuring set the AlphSpeed value to 0 (in which case the splash screen will just display for 1 second with no fade-in). To prevent the splash screen from displaying at all set the AlphaSpeed value to -1 (0xFFFFFFFF).
FontSize	The font size is used to calculate the size of the scrolling log

	window area. If the screen resolution on the system you are using is low then you should reduce the FontSize so that the TEX.EXE windows can fit onto the screen properly. However on high reolution screens a small font-size might will make for a small TEX.EXE window size which might be difficult to read. Suggested values of FontSize between 8 (for low res screens) and 16 (for high res screens) are suggested. The default value of FontSize is 10.
HideTools	If this is set to a non-zero value then the 'Tools' tab will not be shown on system startup.
CXLibraries	This is the list of Telecom Engine Libaries to load upon start-up. This entry is a string type and the list of libaries should be written as a semi-colon separated list of libary names.
OldLibraries	This is the list of Old-Style Telecom Engine Libraries to load upon start-up. None of the standard library set supplied with the Telecom Engine are in this format any more and this type of libarary has now been depreciated and may soon become obsolete.

-o-

Scrolling Log Tabs

The Grpahical front end interface (TEX.EXE) has three tabs for the three different scrolling log windows as follows:

a) The system log.
b) The error log.
c) The trace log.

The system log displays initialisation information showing the library DLLs that have been loaded and the libary initialisation information, as well as other general system messages whilst the Telecom Engine is running. The error log displays any error or warning messages that have been generated whilst the system has been running. And the trace log shows all trace messages that have been generated whilst the Telecom Engine has been running.

Each of the log windows also generates log files on disk. The name of the log files depends on which scrolling log window generateed the log message.

The system log window will cycle through two log files: SYSLOG0.LOG and SYSLOG1.LOG. When the Telecom Engine starts it will always start writing from fresh into SYSLOG0.LOG truncating the file (thuis overwriting any previous log). Once the SYSLOG0.LOG file has reached 4MB in size it will close the file and start writing to SYSLOG1.LOG (truncating the original contents). Once this has reached 4MB in size then the cycle will start back at SYSLOG0.LOG (truncating the original file). Therefore a maximum of 8MB of system log will be retained on in the two log files before the original data will be overwritten.

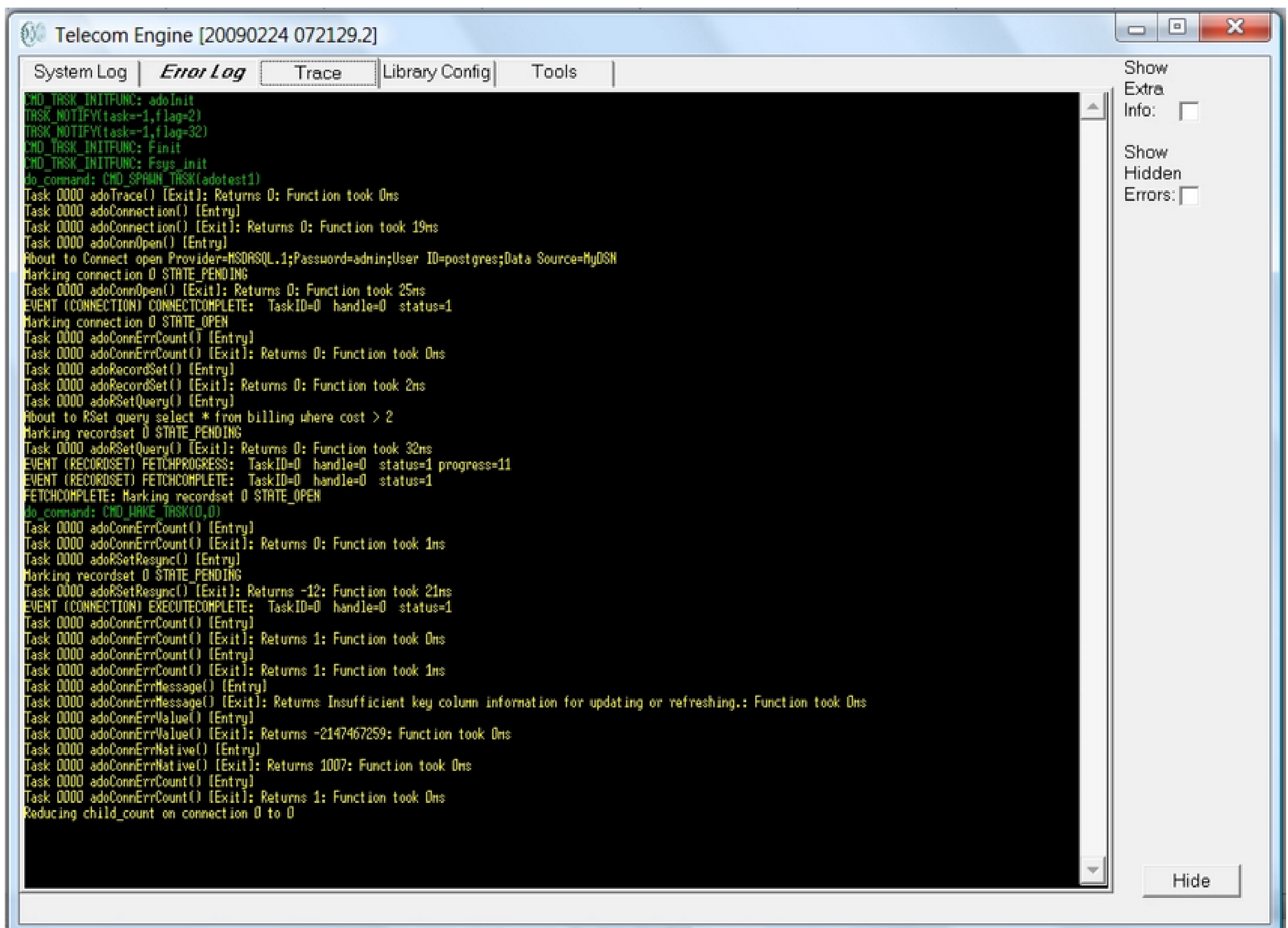
Similarly fvor the error log there will be two log files that get cycled through: ERRLOG0.LOG and ERRLOG1.LOG each of which will grow to a maximum size of 4MB before cycling to the other log file.

For the trace log, since alot more trace is likely to be generated than for the system and error logs, the trace log will cycle through 20 trace log files before cycling back and overwriting the original log files.   Therefore the trace log files will cycle through TRACELOG0.LOG through to TRACELOG19.LOG, each file growing to a maximum of 4MB.

Note: The application log file is created by the CXTERMX.DLL application terminal libary.   This differs slighly in how it generates the application log in that upon startup it will not always start at APPLOG0.LOG but will continue on from the last log file it wrote to.   Also there are up to 10 application log files written (APPLOG0.LOG to APPLOG9.LOG) and each can grown up to 10MB in size before cycling to the next file.

It might be useful to start the Telecom Engine from a batch file so that the log files from the previous run can be backed up (if necessary) and then deleted so that the system always starts from a known state.

Below is a screen shot showing the trace log scrolling window from a typical run-time session (The trace seen here has been generated by the ActiveX Data Objects (ADO) libary:



Notice that in the above screen shot the ***Error Log*** tab is shown in ***Bold Italic*** font to show that there are unviewed messages in the error log.   This will occur for all scrolling log windows if another tab is being viewed when a message arrives at on that scrolling log window.

Also notice on the right there are two check boxes:

i) Show extra info.
ii) Show hidden errors.

The show extra info allows for the full log message to be displayed in the scrolling window area exactly as it appears in the log file on disk. By default only the text part of the log file is shown in the scrolling log window whereas on disk the full log format is written.

The full log format is as follows:

<YYYYMMDD HHMMSS.mmm> [<Application Task Info>] <Log MessageText>

The [<Application Task Info>] field is only written if the log message can be traced to a specific running task (as opposed to a background thread or other system generated log message). The format of the <Application Task Info> is as follows:

<Program name>:<Task ID>:<Program Counter>

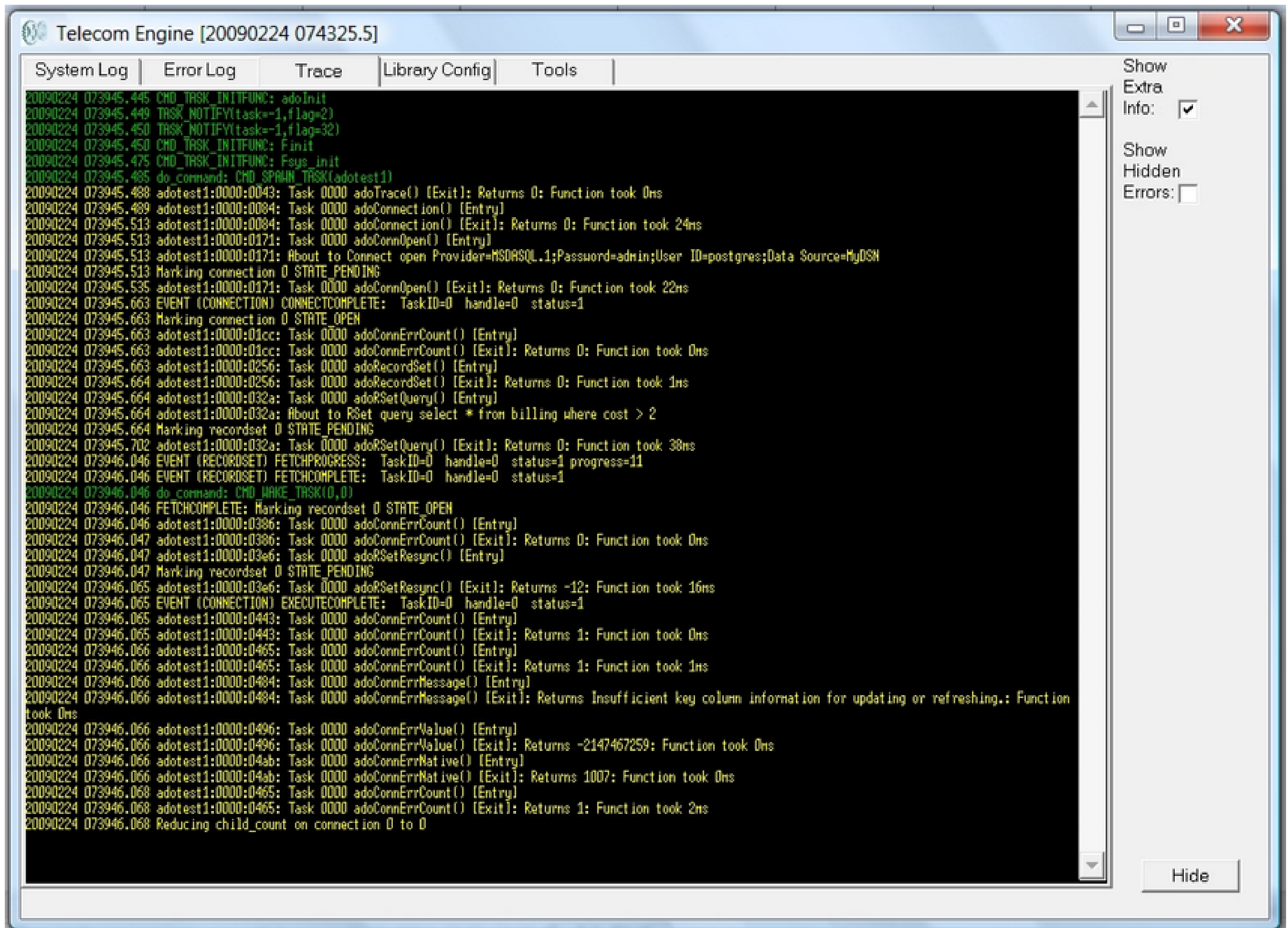
For example the following excerpt is taken from an ERRLOG0.LOG file and shows both forms of the log message:

```
20090224 072033.068 adotest1:0000:03e6: @E Exception caught: Unspecified error (0x80004005)
20090224 072033.072 adotest1:0000:03e6: @E ... Caused by: adoRSetResync(0,1):
20090224 072033.076 adotest1:0000:04b2: Err: Insufficient key column information for updating or
refreshing. number=80004005 native=3ef
20090224 072033.083 KillTask ID=0 found in connection handle 0
20090224 072033.083 KillConnection handle 0 found in recordset handle 0
```

The first three lines were generated by the asotest1.tex application running as taskId 0000. The program counters give the position of the program counter in the byte code that caused the log message to be generated.

The last two lines were generated by the background thread of the CXADO.DLL libary and are some diagnostic messages generated for debug purposes.

Below is a screen shot of the same application that has been run with the 'Show Extra Info' check box ticked..



The 'Show Hidden Errors' check-box allows errors that have been suppressed by a call to term_errctl() to be displayed. Sometimes an application may wish to suppress certain error messages from being displayed to stop the scrolling logs from being overrun with unecessary error messages. By ticking this box then these messages will be displayed regardless of whether the application tried to suppress them with a call to term_errctl() or equivalent.

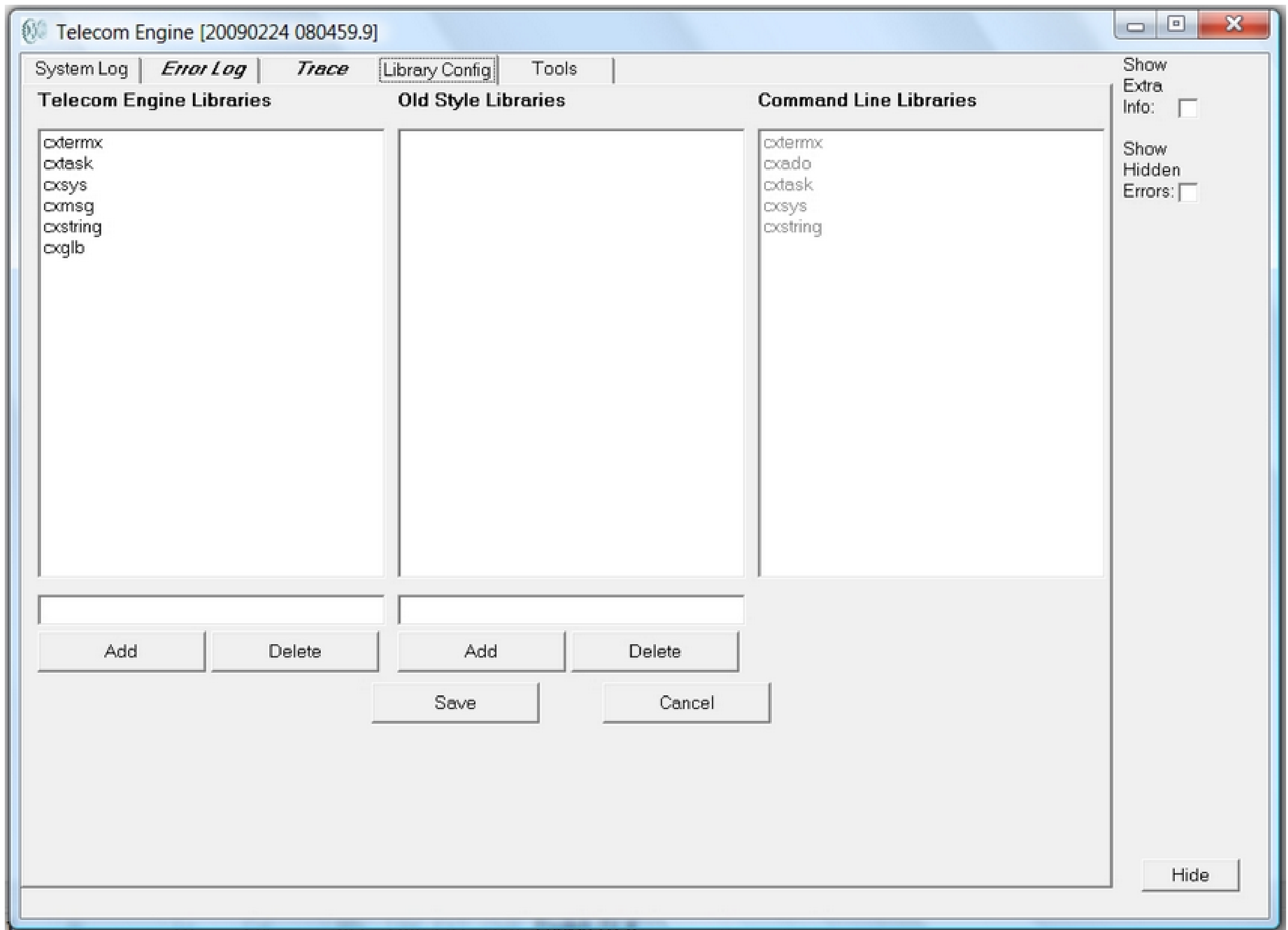
The 'HIDE' button down the bottom right causes the TEX.EXE application to shrink to an icon the system tray. Note that this icon will turn red if any error messages are written to the error log after the application has been minimised to the system tray to alert the user that errors have occurred.

-o-

Library Configuration Tab

The Library Configuration Tab allows the Telecom Engine to be configured to load the same set of back-end libaries each time it is run.

Below is a screen shot of the Libary Configuration Window:



The left pane displays (and allows editing of) the list of back-end Telecom Engine libaries that will be loaded by default upon start-up (unless the **-n** command line option is specified (see Command Line Options)). The middle pane displays (and allows editing of) the older stlyle Telecom Engine libraries (now depreciated and will possibly be made obsolete in the future). The right hand pane diaplys the libaries that were specified for loading on the command line (using the **-r** command line option (see Command Line Options)).

In the above example the application was loaded with the following command line options:

tex -n -rcxtermx;cxado;cxtask;cxsys;cxstring <application name>

Note that the list of libaries that are loaded by default (those in the left and middle pane) are written and read from the following registry key:

HKEY_LOCAL_MACHINE\Software\Telecom Engine\TEX

and are stored in the string type registry entrys:

CXLibraries
OldLibraries

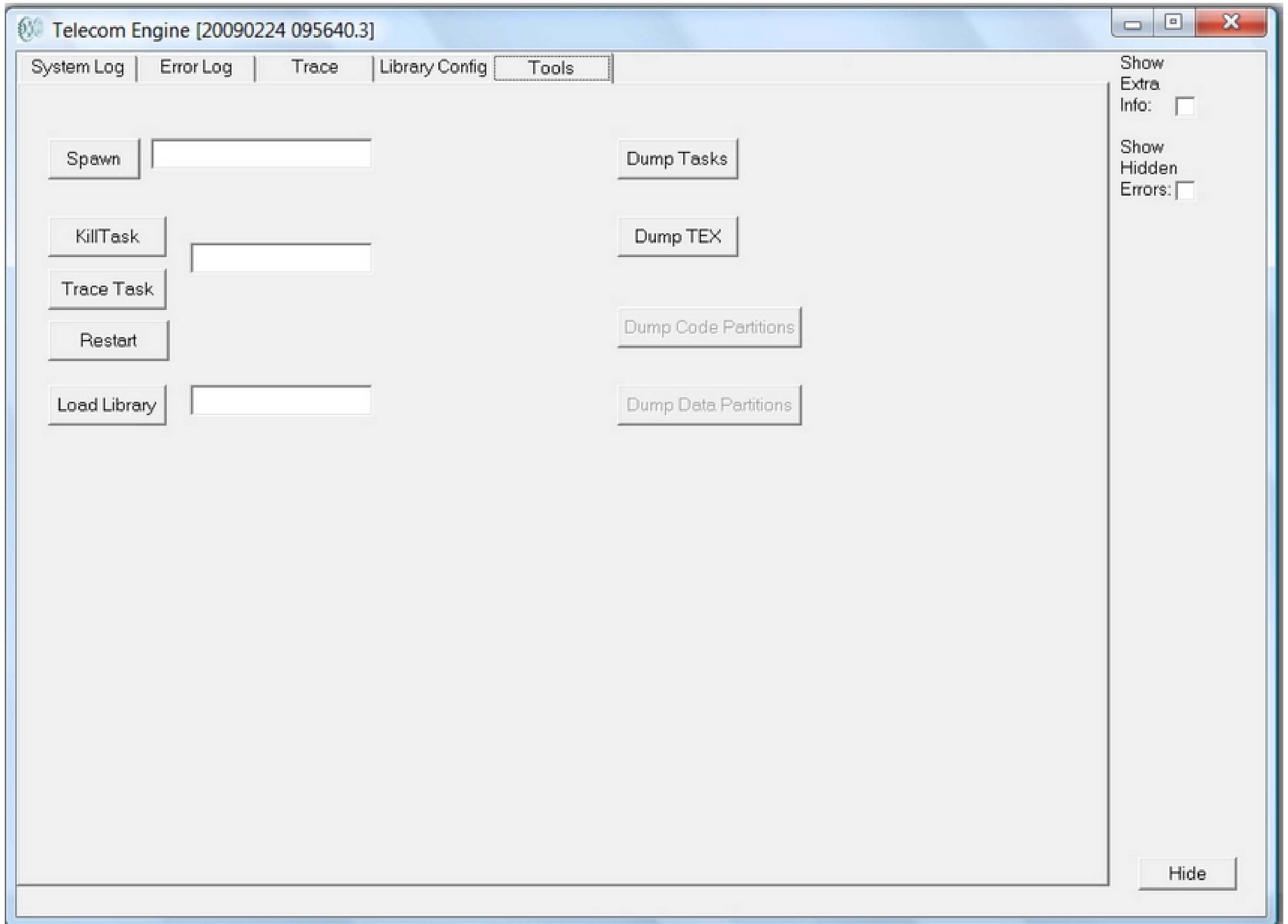
The list is simply stored as a semi-colon separated list.

-o-

Tools Tab

The tools tab provides a few mechanisms for dumping internal Telecom Engine Scheduler information or for manually killing or loading byte-code tasks and manually loading back-end libaries.

Below is a screen shot from the Tools Tab Window:



Extreme care should be taken when using any of the options on the left hand side of this form, particularly on a running system as it could disrupt the execution of important tasks. The registry entry HideTools can be set to prevent the Tools tab from being shown on a live system to prevent accidental disruption of the running tasks (see Registry Settings).

Below is a description of the buttons shown on this form:

Button	Description
Spawn Task	This button will cause the byte-code file entered into to the adjacent edit box to be loaded into the Telecom engine Scheduler.
Kill Task	This button will kill the task ID entered into the adjacent edit box
Trace Task	The button will trace the task ID entered into the adjacent edit box

Restart Task	This button will restart the task ID entered into the adjacent edit box
Load Libary	This button will load the back-end Telecom Engine Libary file specified in the adjacent edit box.
Dump Tasks	This button will dump a list of all of the tasks current running in the system to the trace log.
Dump TEX	This button will dump a list of all the byte code (*.TEX) files that have been loaded
Dump Data Partition	Internal use only (disabled)
Dump Code Partition	Internal use only (disabled)

-o-

The TE Standard Library Set

Introduction

The main functionality of the Telecom Engine is provided by external DLL function libraries which allows the Telecom Engine to interface with the operating system and various telecommunications hardware as well as providing sets of functions for task management, console terminal and logging, inter-task communications, string manipulation, TCP/IP socket connectivity, database access etc.

The language is fully extendable and new DLLs can be written to offer new functionality or to extend or replace the existing standard function sets.

The libraries belonging to the TE standard library set all have a name beginning with CX.

The full list of libraries currently supplied is as follows:

Library	DLL Name	Description
Task Management Library	CXTASK.DLL	Starting and stopping tasks (task_spawn(); task_chain(); task_sleep() etc.)
Terminal Console Library	CXTERMX.DLL	Console terminal, logging and keyboard input functions (applog(); errlog(); kb_get() etc)
System library	CXSYS.DLL	File, directory, date and time functions..
String Manipulation Library	CXSTRING.DLL	Functions for manipulating strings (strlen(); substr() etc)
Inter-task Messaging Library	CXMSG.DLL	Functions to all messages to be exchanged between tasks. (msg_put(); msg_get() etc)
Global Variable Library	CXGLB.DLL	System wide global variables and large array handling

Semaphore Library	CXSEM.DLL	Semaphore functions for mutual exclusion.
Clipper Database Library	CXDBF.DLL	Dbase3 DBF file handling and Clipper (NTX) indexing functions (db_open(); db_get() etc)
Floating Point Library	CXFP.DLL	Floating point arithmetic functions (fp_add(); fp_mul() etc).
Sockets Library	CXSOCK.DLL	Socket handling (TCP/IP and DataGram) – Slisten(); Sconnect() etc
Aculab E1/T1 Card Library	CXACULAB.DLL	Aculab Network card functions (CCwait(); CCmkcall() etc)
Aculab Prosody Card Library	CXACUDSP.DLL	Aculab prosody speech card functions (SMplay(); SMrecord() etc).
Dialogic DTI Card Library	CXDTI.DLL	Dialogic DTI card functions (DTI_wait(); DTI_setsig() etc)
Dialogic Speech Card Library	CXDLGC.DLL	Dialogic speech card functions (dx_play(); dx_record() etc)
Dialogic Global Call Library	CXGCALL.DLL	Dialogic Global Call function (GCwait(); GCaccept() etc.)
ActiveX Data Objects (ADO) Database Library	CXADO.DLL	Advance Data Objects (ADO) database libary for SQL queries and commands on Database servers. (adoConnection(), adoRecordSet() etc).

-o-

Manual Conventions

The entries in this manual all conform to the same conventions as follows:

Synopsis: Provides a definition of the function and its arguments.
Arguments: Provides a description of the function arguments.
Description: Provides a detailed description of the function.
Returns: Describes the possible return values.

For each entry a **synopsis** is provided showing the function name, its return value and the function arguments as follows:

Synopsis:

```
handle=getHandle(arg1,arg2[,arg3[,arg4]]);
```

In the above example the function *getHandle()* is defined with two compulsory arguments *arg1, arg2* and a two optional arguments *arg3* and *arg4*. Optional arguments are specified by showing the arguments in square bracketss ([]).

Below is another slightly different example:

Synopsis:

handle=getHandle(arg1,arg2[,arg3[,...]]);

In this example the function *getHandle()* is defined with two compulsory arguments *arg1, arg2* and an unspecified number of optional arguments all of a similar type to *arg3* (if it is specified). By specifying elipses inside the square brackets this indicates that the previous argument can be repeated an unspecified number of times.

The argument description and detailed description will then provide clarification about what arguments are actually required.

All example code shown in the manual will be shown in the 'Courier' font as shown below:

```
main
    applog("Goodbye Cruel World");
   sys_exit(1);
endmain
```

-o-

Task Management Library

Introduction

This library provides the task management functions for starting and stopping tasks, sleeping, retrieving task arguments etc.

A new task can be started in the Telecom Engine either by naming a program in the TEX command line, using the TOOLS tab in the TE Run-Time Engine Window or by using the task_spawn**(),** task_exec**()** or task_chain**()** functions in the CXTASK.DLL. The task_spawn**()** function starts a new task and the original task continues processing. The task_exec**()** function starts a new task and the calling task is suspended until the new task stops. The task_chain**()** function starts a new task and kills the calling task.

If spawn successfully launches a new task, then the return value is the task ID of the new task. Otherwise a negative number is returned to indicate an error:

```
task_id = task_spawn("CHILD");
if (task_id < 0)
    errlog("Error spawning CHILD.TEX, Cannot continue ");
   exit (1);
endif
```

Arguments may be optionally passed to the new task.

```
task_spawn("SLAVE", port,chan);
```

The new task can retrieve these arguments by using the task_arg() function. A typical application will have a single 'MASTER' program that then spawns all other tasks in the system. Usually there would be one task in charge of each network channel and the port and channel number (amongst other parameters) would be passed to the slave task. Here is a simple example using the CXACULAB.DLL library:

```
// MASTER.TEX program:
int port, chan;
main
   port=0;   // The first E1
   // Spawn a slave task for each channel of the E1
    for (chan = 1; line <= 30; chan++)
       task_spawn("SLAVE", port,chan);
   endfor
endmain

// SLAVE.TEX program:
int port, line;
main
   // Get the port and chan that this task is in charge of ..
   port = task_arg(1);
   chan=task_arg(2);

   // Enable inbound calls on chan
   CCenablein(port,chan)

   // Wait for incoming call
    while(CCwait(port,chan,0) != CS_INCOMING_CALL_DETECTED)
     ;
   end

    // Answer the call
   CCaccept(port,chan);

  …

endmain
```

The MASTER.TEX program (which would be specified on the command line to the TEX.EXE run-time program) spawns 30 SLAVE.TEX tasks, each one in charge of a single channel (the port and channel being passed as arguments to the SLAVE task). The SLAVE task the waits for and incoming call on the channel and answers the phone when one arrives. This task could then start playing speech files or could call task_chain() to launch another application (depending of the received DID digits for example).

-o-

Function Quick Reference

A summary of the full set of functions in this library is as follows (optional arguments are show in curly braces {}):

task_id=task_spawn(task_name{, arg1 {, arg2 {,..arg15 }}})
task_id=task_chain(task_name{, arg1 {, arg2 {,..arg15}}})
return_value=task_exec(task_name{, arg1 {, arg2 {,..arg15}}})
parent_id=task_parentid()
task_return(value)
task_sleep(tenths)
task_hangup()
task_defersig(flag)
task_id=task_getpid()
task_clrdefer()
arg_value=task_arg(arg_num)
task_kill(task_id)

-o-

Task Management Library Function Reference

task_spawn

Synopsis:
task_id=task_spawn(task_name[, arg1 [, arg2[,..arg15]]]])
Arguments:
task_name – The name of the task to spawn (with or without the .TEX) extension)
arg1..arg15 – Optional arguments to pass to spawned task (up to 15)

Description: This function creates a new task from the compiled byte code file given in *task_name*.

If the .TEX extension is omitted it will be added automatically.

If the TEXDIR environment variable is set, then the semi-colon delimited list of directories specified in this environment variable will be searched in order for the file specified in *task_name*. If TEXDIR is not set then the only current directory will be searched.

If any of *arg1..arg15* are specified then these arguments will be passed to the new task which can then be read using the **task_arg()** function.

The function call will always return immediately regardless of whether the new task was started successfully or not.

Return Value:

The function will return the task ID of the spawned task, or else a negative error code.

-o-

task_chain

Synopsis:
ret_code=task_chain(task_name[, arg1 [, arg2[,..arg15]]]])
Arguments:
task_name – The name of the task to spawn (with or without the .TEX) extension)
arg1..arg15 – Optional arguments to pass to spawned task (up to 15)

Description: This function creates a new task from the compiled byte code file given in *task_name* and kills the calling task if the new task is started successfully.

If the .TEX extension is omitted it will be added automatically.

If the TEXDIR environment variable is set, then the semi-colon delimited list of directories specified in this environment variable will be searched in order for the file specified in *task_name*. If TEXDIR is not set then the only current directory will be searched.

If any of *arg1..arg15* are specified then these arguments will be passed to the new task which can then be read using the **task_arg()** function.

The function call will only return if it cannot start the new task for any reason (e.g. invalid file name or file not in TEXDIR directory path). If the task starts successfully then the calling task is immediately killed.

Return Value:

The function will only return if it fails in which case it returns a negative error code.

-o-

task_exec

Synopsis:
ret_code=task_exec(task_name{, arg1 {, arg2 {,..arg15}}}})
Arguments:
task_name – The name of the task to spawn (with or without the .TEX) extension)
arg1..arg15 – Optional arguments to pass to spawned task (up to 15)

Description: This function creates a new task from the compiled byte code file given in *task_name*. If the new task starts successfully then the calling task will be suspended (i.e. will block) until the new task stops (either by encountered a **stop** statement, or reaching the end of the program instructions, or by calling a **task_return()** function, or if it explicitly killed (say by a **task_kill()** call) or if the child task chains to another task..

The calling task If the .TEX extension is omitted it will be added automatically.

If the TEXDIR environment variable is set, then the semi-colon delimited list of directories specified in this environment variable will be searched in order for the file specified in *task_name*. If TEXDIR is not set then the only current directory will be searched.

If any of *arg1..arg15* are specified then these arguments will be passed to the new task which can then be read using the **task_arg()** function.

If the child task chains to another task then the child is deemed to have stopped and the parent will be woken. If the child task encounters a **restart** statement then the parent task will not be woken up ans the child is deemed to be still active.

Return Value:

If the new task could not be started then the function will return a negative error code. If the new task was started successfully then the function will only return once this new task is stopped or killed. The return values will then be as follows:

Empty string “” is returned if the task encountered a **stop** or the end of program reached or if the task was expicitly killed, or the child chained to another task.

“?” is returned if the task stopped due to a fatal error (like stack overflow).

If **task_return(str)** is called from the new task then the return value given is this call will become the return value of the task_exec() function call. It would be advisable not to return a negative value or one of the above string values from this call to avoid confusion.

-o-

task_parentid

Synopsis:
task_id=task_parentid()
Arguments:
none
Description: This function returns the ID of the task that started this task through a call to **task_spawn()**, **task_exec()** or **task_chain()**. This can be useful for example if a child task needs to communicate with the parent task using Telecom Engine inter-task messaging.

If there is no parent task either because the task was started by the TEX.EXE run-time program or the parent task has stopped or been killed then this function will return -1

Return value:

Either the task ID of the parent task or -1 if there is no parent task or the parent task has stopped or been killed.

-o-

task_return

Synopsis:
task_return(return_str)
Arguments:
return_str = The string value to return to the parent task
Description: This function stops the current task and sets the return value of the parent task to

return_str. If the task was started using the **task_exec()** function then this function will return with the value specified in the *return_str* . If the parent task has been killed, or did not start the task using the **task_exec()** function, then the calling task will still be stopped but the *return_str* argument will be ignored.

Return Value:

This function stops the current task and does not return.

-o-

task_sleep

Synopsis:
task_sleep(tenths)
Arguments:
tenths = The number of tenths of a second to put the task to sleep.
Description: This function puts the calling task to sleep for the specified number of tenths of a second. The function blocks during this time and will only wake up when the specified timeout expires.

Returns Value:

Returns -1 if an invalid time is given (<0) otherwise returns empty string “” when timeout expires.

-o-

task_hangup

Synopsis:
task_hangup(task_id)
Arguments:
task_id = The id of the task to send the hangup signal to. second to put the task to sleep.
Description: This function sends a hangup signal to the specified task ID which will cause the specified task to jump immediately into its **onsignal** function. If the task specified does not have an **onsignal** function declared then the signal will be ignored and the task will carry on processing as normal. If the task specified has called **task_defersig(1)** one or more times then the jump to **onsignal** will be deferred until the task clears all defers (either by calling **task_defersig(0)** the corresponding number of times or by calling **task_clrdefer()**)

Return Value:

Returns 0 is call was succesfull, -1 if invalid task ID is specified. No indication is returned as to whether the specified task actually jumped to its onsignal function.

-o-

task_defersig

Synopsis:
task_defersig(flag)
Arguments:
flag = Either 1 or "(" to defer jump to onsignal, else 0 or ")" to clear previous defer request.
Description: This call is used to prevent a jump to **onsignal** from occurring until a particular block of code has finished executing. Each time **task_defersig()** is called with *flag* set to 1 or "(" a counter is increased. Each time **task_defersig()** is called with *flag* set to 0 or ")" a counter is decreased (but not below 0). If a signal is received whilst the counter is higher than zero then the jump to **onsignal** will not occur until the counter reaches zero again (through corresponding calls to **task_defersig(0)** or a call to **task_clrdefer()**).

Calls to task_defersig(1) can be nested, but the programmer must make sure there is a corresponding task_defersig(0) to match (or a single call to task_clrdefer()) otherwise the task can never respond to signal events. This is a very common bug and can cause channels to get stuck if no other method is employed to detect a hangup event (such a noticing that the caller has stopped responding to menus).

The programmer should use task_defersig() during database updates or file writes or anywhere else where a hangup signal need to be temporarily ignored..

For example:

```
// Prevent jumps to onsignal while we write to a file..
task_defersig(1);

// Carry out some tasks that should not be interrupted..
write_to_log("This cannot be interrupted by a hangup_signal");

// Now allow jumps to onsignal again..
task_defersig(0).
```

Note that instead of passing 1 and 0 to task_defersig() you can pass "(" and ")" instead which reflects the nested nature of the calls to this function, but this is up to programmer preference E.g.

```
// Prevent jumps to onsignal while we write to a file..
task_defersig("(");

// Carry out some tasks that should not be interrupted..
write_to_log("This cannot be interrupted by a hangup_signal");

// Now allow jumps to onsignal again..
task_defersig(")").
```

This maybe more intuitive for some programmers since it reminds the programmer that all "(" must be matched by a corresponding ")".

Again watch out for bugs like this:

```
func f()
  task_defersig("(");

  x=do_something_important();
```

```
   // check for error
   if(x < 0)
        // This looks like a bug since a corresponding
        // task_defersig is not being called before returning
        // from the function.
        return x
   end

   // This is OK but where is the one in the if statement above!!??
   task_defersig(")");
endfunc
```

Return Value:

Returns an empty string "".

-o-

task_clrdefer

Synopsis:

task_clrdefer()

Arguments:

none

Description: This call is resets the task_defersig() counter back to zero thus allowing hangup signals to cause jumps to **onsignal** again. If a hangup signal had already been received then this function will not return and a jump to **onsignal** will occur immediately.

This is often used where an non-recoverable error has been received and the programmer wishes to jump immediately to **onsignal** to disconnect the caller.

For example:

```
func f()

   task_defersig("(");
   x=do_something_important();
   // check for non-recoverable error.
   if (x < 0)
     // can't recover so....
      // First clear all signal defers (there maybe
      // some nested calls outside this function)
      task_clrdefer();
      // send a hangup signal to self to force jump to onsignal
      task_hangup(task_getpid());
   endif
   …
  task_defersig(")");
end
```

Return Value:

Returns the empty string "".

-o-

task_getpid

Synopsis:
task_getpid()
Arguments:
none
Description: This call returns the task ID (otherwise know as the process ID) of the calling task. This can be used so a task can send a hangup signal to itself to force a jump to onsignal:

```
task_hangup(task_getpid());
```

Return Value:

Returns the task ID of the calling task.

-o-

task_arg

Synopsis:
task_getpid(arg_num)
Arguments:
arg_num - The argument number to return.
Description: This call returns the value of the argument number *arg_num* passed to the task by the parent task through a call to task_spawn(), task_exec() or task_chain().

Arguments are numbered from 0 through to 15, where argument 0 returns the name of the calling task (i.e. the name of the .TEX byte code file). Arguments 1 through to 15 are the arguments passed the call to task_spawn(), task_exec() or task_chain(). If an attempt is made to get the value of an argument that was not passed by the parent task then the function will return an empty string "".

Return Value:

Will return the name of the TEX byte code file if argument 0 is specified, otherwise will return the argument 1 through to 15 passed to the task from the parent task (or an empty string "").

-o-

task_kill

Synopsis:
task_taskkill(task_id)
Arguments:
task_id - The task ID of the task to kill.
Description: This function will instantly kill the specified task. The specified task will stop processing immediately and the task ID will become invalid (until reused by another task).

Return Value:

Will return -1 if an invalid task ID was given, else will return 0 if the task was successfully stopped.

-o-

System library

Introduction

The System library provides a set of functions to interface with the operating system and provides the following sets of function:

Buffer Manipulation Functions	The functions allow for manipulation of binary data which would otherwise be difficult due to the fact that TE variables store data as null terminated ascii strings. Buffers are used mainly for reading and writing binary data to and from files.
File Handle Functions	The file handle functions allow for files to be opened and read from and written to. This functions can be used in conjuction with the buffer manipulation functions to read and write binary files, or for reading and writing Ascii text files.
File System Functions	The file system function offer the ability to manaipulate the file system and the directory and file level (E.g copying or renaming files and searching directories)
Date and Time Functions	The date and time functions allow access to the system date and time and offer various date and time manipulation functions (such as adding or subtracting time or finding out the day of the week)
Other System Functions	these functions provide other miscellaneous system functions such as exiting the Telecom Engine or reading environment variables.

-o-

Library Limits and Defaults

In this version of the library the following limits and default values are hardcoded. To change any of these values will require recompilation of the library. In future versions these values may become configurable.

Description	**Limit**
Size of Buffers	1024 Bytes
Maximum number of Buffers	32
Maximum Open Files	2048
Maximum Number of File Locks	2048

-o-

Side Effects From Signals

When a task stops or is killed then a notification signal is sent to the libary notification function which will automatically the following actions:

All Buffers allocated to the task are released
All file locks made by the task are released
All File handles allocated to the task are closed
Any file copy initiated by the task is cancelled.

NOTE: The above actions will also be carried out if a **restart** statement is encountered or if the task chains to another task (since this is equivalent to starting a new task then killing the calling task).

-o-

System Library Quick Reference

Buffer Manipulation Functions

buf_handle=sys_bufuse()
sys_bufrls(buf_handle)
sys_bufrlsall()
sys_bufcopy(buf_dest,buf_source)
sys_bufmove(buf_dest,dest_offs,buf_source,source_offs,num_bytes)
sys_bufset(buf_handle,offs,string)
sys_bufget(buf_handle,offs {,num_bytes})

File Handle Functions

file_handle=sys_fhopen(filename,flags)
sys_fhclose(file_handle);
sys_fhcloseall()
sys_fhseek(file_handle,offset,fromwhere)
sys_fhreadbuf(file_handle,buf_handle,bytes)
string=sys_fhgetline(fil_handle)
sys_fhwritebuf(fil_handle,buf_handle,bytes)
sys_fheof(fil_handle)
sys_fhputline(fil_handle,string)
sys_fhlock(fil_handle,offset,bytes)
sys_fhunlock(fil_handle,offset)
sys_fhwrites(file_handle,string)
sys_fhsetsize(fil_handle,size)
hex_handle=sys_gethandle(file_handle)

File System Functions

sys_fcopy(source_filename,dest_filenname)
sys_fdelete(filename)
sys_finfo(filename,infotype)

sys_frename(source_filename,new_filename)
sys_dirmake(dir_name)
sys_dirremove(dir_name)
sys_dirfirst(path,attributes)
sys_dirnext()
sys_dirend()
sys_diskfree()

Date and Time Functions

{YY}YYMMDD=sys_date({long_format_flag})
HHMMSS=sys_time()
ticks=sys_ticks(bios_or_ms_flag)
sys_tmrstart()
secs=sys_tmrsecs()
{YY}YYMMDDHHMMSS=sys_timeadd({YY}YYMMDD,HHMMSS,secs)
{YY}YYMMDDHHMMSS=sys_timesub({YY}YYMMDD,HHMMSS,secs)
{YY}YYMMDD=sys_dateadd({YY}YYMMDD,days)
sys_settime({YY}YYMMDD,HHMMSS)
day=sys_datecvt({YY}YYMMDD,convert_type)

Operating System Functions
sys_exit(exit_value)
sys_getenv(variable_name)

-o-

System Library Function Reference

Buffer Manipulation Functions

sys_bufuse

Synopsis:
buf_handle=sys_bufuse()
Arguments:
None

Description: This function returns a handle to one of the pool of 1024 byte buffers which can then be used by the task in the sys_fhread() and sys_fhwrite() functions.

Return Value:

The function will return a unique handle to a 1kb buffer, else wioll return -1 if there are no free bufferes available.

-o-

sys_bufrls

Synopsis:
sys_bufrls(buf_handle)
Arguments:
buf_handle - The handle of the buffer to release.

Description: This function releases the specified buffer handle back to the pool of free buffers. Note that all buffers allocated to a task will be automatically released if the task stops or is killed. This includes if the task chains to another task or if the task restarts.

Return Value:

Will return 0 if the buffer is released or -1 if an invalid handle is given.

-o-

sys_bufrlsall

Synopsis:
sys_bufrlsall()
Arguments:
None

Description: This function releases all buffers allocated to the calling task back to the pool of free buffers.

Note that all buffers allocated to a task will be automatically released if the task stops or is killed. This includes if the task chains to another task or if the task restarts.

Return Value:

Will return 0

-o-

sys_bufcopy

Synopsis:
sys_bufcopy(buf_dest,buf_source)
Arguments:
buf_dest - The destination buffer handle
buf_source - The source buffer handle

Description: This function copies the entire contents of the buffer specified by *buf_source* into the buffer specified by *buf_dest*

Return Value:

Will return 0 on success or -1 if an invalid handle is given.

-o-

sys_bufmove

Synopsis:
sys_bufcopy(buf_dest,dest_offs,buf_source,source_offs,num_bytes)
Arguments:
buf_dest - The destination buffer handle
dest_offs - The byte offset in the destination buffer
buf_source - The source buffer handle
source_offs - The byte offset in the source buffer
num_bytes - The number of bytes to copy

Description: This function copies *num_bytes* bytes from the buffer specified by *buf_source* at offset *source_offs* to the buffer specified by *buf_dest* at offset *dest_offs*.

Return Value:
Will return 0 on success or -1 if an invalid handle or and attempt is made to read or write beyond the start or end of either buffer.

-o-

sys_bufget

Synopsis:
sys_bufget(buf_handle,offs {,num_bytes})
Arguments:
buf_handle - The buffer handle
offs - The byte offset into the buffer
num_bytes - Optional argument specifying the number of bytes to get.

Description: If called with two arguments, returns the given byte in the buffer at byte offest *offs* as a one-character string (unless the byte is zero, in which case the string is empty). The first byte in the buffer is numbered zero.

A third argument bytes may be given specifying a number of bytes to extract from the buffer. The return value is then a string of length bytes (unless one of the bytes in the specified range is zero, in which case the zero byte terminates the string).

Return Value:

Will return 0 on success or -1 if an invalid handle is given or an attempt is made to read beyond the end or before the beginning of the buffer

-o-

sys_bufset

Synopsis:

sys_bufset(buf_handle,offs,string)

Arguments:

buf_handle - The buffer handle
offs - The byte offset into the buffer
string - The string to write

Description: This function writes the given *string* into the specifier buffer at byte offset given by *offs*. The terminating null byte of *string* is not written to the buffer.

Return Value:

Will return 0 on success or -1 if an invalid handle is given or an attempt is made to write beyond the end or before the beginning of the buffer.

-o-

File Handle Functions

sys_fhopen

Synopsis:

file_handle=sys_fhopen(filename,flags)

Arguments:

filename - The path of the file to open
flags - Defines how the file is to be opened.

Description: This function opens the file specified by *filename* and returns a file_handle to the file. *Flags* is a string that defines how the file should be opened. Each character of the string in *flags* has a different meaning as follows:

r - Open the file for reading
w - Open the file for writing
s - Open the file in shared mode
c - Create the file if it deosn't exist
t - Truncate the file to 0 bytes when it is opened.

At least one of the flags **r** or **w** should be specified (or both), whereas the other flags (**s**, **t** or **c**) are optional.

For example, the following code specifies that "myfile.txt" should be opened for ***reading*** and ***writing*** in ***shared*** mode and it should be ***created*** if it doesn't exist:

```
file_open("myfile.txt","rwsc");
```

The following specifies that "myfile.txt" should be opened for ***reading*** in exclusive mode (non shared).

```
file_open("myfile.txt","r");
```

The file handle returned will actually be an integer between 0 and the maximum number of open files allowed by the library - it will not be an operating specific file handle. To obtain the operating system file handle (say for use by other DLL libaries) then the function **sys_gethandle()** can be used.

Note that if the same file is opened by multiple tasks then each task will receive its own unique handle to the file.

Return Value:

The function will return a unique file handle or an negative error value if the file could not be opened. The value returned will be that which is returned by the Windows GetLastError() function.

-o-

sys_fhclose

Synopsis:
sys_fhclose(file_handle)
Arguments:
file_handle - The file handle to close

Description: This function closes a handle previously opened by a call to sys_fhopen(). Any locks placed by fil_lock() using this handle will be unlocked. Only files that have been opened by a specific task can be closed by that task - if an attempt is made to close a file opened by another task then an error is returned.

Note that if a task is stopped or is killed then all files opened by that task are automatically released.

Return Value

Returns 0 if the file was closed successfully or -1 if an invalid file handle is given.

-o-

sys_fhcloseall

Synopsis:
sys_fhcloseall()
Arguments:
None

Description: This function closes all files that were opened by the calling task. Any file locks established on the open files will be released.

Note that if a task is stopped or is killed then all files opened by that task are released automatically.

Return Value

Returns 0

-o-

sys_fhseek

Synopsis:

sys_fhseek(file_handle,offet,fromwhere)

Arguments:

file_handle - The file handle
offset - The byte offset in the file to seek to
fromwhere - Where to seek from

Description: This function moves the current file pointer of the specified *file_handle* to the byte position in the file specifed by *offset.* The *fromwhere* argument defines where *offset* is being measured from:

0 move relative to start of file
1 move relative to current position
2 move relative to end of file

A negative value of *offset* indicates to move the pointer backwards towards the start of the file.

Return Value

Returns the new file position relative to the start of the file, or a negative error code.

Since the function returns the new position in the file after the move has been carried out, the current file position can be establish by calling the function but by specifying a move of zero bytes from the current position. The following example uses he sys_fhseek() function to find the length of a file by moving to the last byte. Once the length has been found it then moves back to the previous position in the file:

```
// Get our current position
curr_pos=sys_fhseek(fhandle,0,1);

// Move to end of file to find the file size
file_size=sys_fhseek(fhandle,0,2);

// Move back to where we started from
sys_fhseek(fhandle,curr_pos,0);
```

-o-

sys_fhreadbuf

Synopsis:

sys_fhreadbuf(file_handle,buf_handle,num_bytes)

Arguments:

file_handle - The file handle to read from
buf_handle - The buffer handle of the buffer to read the data into
num_bytes - The number of bytes to read.

Description: This function attempts to read *num_bytes* bytes from the given *file_handle* into the given buffer. The *buf_handle* is a handle to a 1Kbyte buffer returnedd from a call to sys_bufuse ().

The maximum number of bytes that can be read is 1024 (the maximum size of a buffer).

Return Value

Returns 0 the number of bytes actually read or a negative error code (as returned by the windows function GetLastError()). If an invalid argument is given then the function will return -1.

For eample the following code copies the contents of one file to another by reading 1024 bytes at a time:

```
main
  int fh1,fh2,bytes;
  int bufh;

  fh1=sys_fhopen("source.txt","rs");
  fh2=sys_fhopen("dest.txt","rwsct");
  bufh=sys_bufuse();


  do
    bytes=sys_fhread(fh1,bufh,1024);
    if(bytes > 0)
       sys_fhwrite(fh2,bufh,1024);
    endif
  until(bytes < 1024);
endmain
```

-o-

sys_fhwritebuf

Synopsis:

sys_fhwritebuf(file_handle,buf_handle,num_bytes)

Arguments:

file_handle - The file handle to write tom
buf_handle - The buffer handle of the buffer to write the data from
num_bytes - The number of bytes to write.

Description: This function attempts to write *num_bytes* bytes to the given *file_handle* from the specified. The *buf_handle* is a handle to a 1Kbyte buffer returned from a call to sys_bufuse().

The maximum number of bytes that can be written is 1024 (the maximum size of a buffer).

Return Value
Returns the number of bytes actually written or a negative error code.

-o-

sys_fhgetline

Synopsis:
sys_fhgetline(file_handle,pVar)
Arguments:
file_handle - The file handle to read from
pVar - Pointer to the variable that will hold the returned string

Description: This function is used to read strings from ASCII text files. It will read the string from the current file position up to the end of the current line (i.e until a carriage-return/line-feed is encountered) and write that string to the variable pointed to by *pVar*. Carriage-return and Line-feed characters are not included in the returned string.

An empty will be returned if the line only contains a carriage-return/line-feed.

Return Value:
The function returns 0 on success or a negative error code.

The following example opens a text file and writes each line of the file to the application log terminal.

```
int file_handle;
var str:255;

file_handle=sys_fhopen("textfile.txt","rs");

while(!sys_fheof(file_handle))
   // Read the next line from the file..
   sys_fhgetline(file_handle,&str);
   //write line to the application log and terminal console
   applog(str);
endwhile
```

-o-

sys_fhputline

Synopsis:
sys_fhputline(file_handle,string)
Arguments:
file_handle - The file handle to write to

string - The string to write to the file

Description: This function appends the given *string* to the file specified by *file_handle* and automatically writes a Carriage-return/Line-Feed cobination to terminate the line. Before writing the text the function will seek to the end of the file before writing the text line.

If you are appending to a log file that might be written to across a network by several machines it is advisable to use file locks to make sure that only one machine is appending at any one time (otherwise garbled data can occasionally appear in the file).

Return Value
The function will return 0 in sucess or an negative error code.

-o-

sys_fhwrites

Synopsis:
sys_fhwrites(file_handle,string)
Arguments:
file_handle - The file handle to write to
string - The string to write to the file

Description: This function writes the given *string* to the file specified by *file_handle.* Unlike sys_fhputline() this function will not append to the end of the file, it will write the string from the current file position and it will NOT append a carriage return or line-feed character.

Return Value
The function will return 0 in sucess or an negative error code.

-o-

sys_fheof

Synopsis:
sys_fheof(file_handle)
Arguments:
file_handle - The file handle

Description: This function checks whether the end of file has been reached when reading from files. If the current file position is past the end of file the this function will return 1, otherwise it will return 0.

Return Value: Will return 1 if end of file has been reached, otherwise will return 0 or a negative error code.

-o-

sys_fhlock

Synopsis:
sys_fhlock(file_handle,position,bytes)
Arguments:
file_handle - The file handle
position - The byte offset in the file to lock
num_bytes - The number of bytes to lock.

Description: This function attempts to lock the number of bytes specified by *num_bytes* at byte offset *position* in the file specified by *file_handle*. It is possible to lock a region that is beyond the end of the file.

Note that all file locks will be release when a fil_handle is closed.

Return Value:

The function will return 0 if the lock was sucessfully obtained, otherwise it will return -33 if it could not obtain the lock or another negative error code.

-o-

sys_fhunlock

Synopsis:
sys_fhunlock(file_handle,position)
Arguments:
file_handle - The file handle
position - The byte offset in the file to unlock

Description: This function allows a task to release a lock it previously obtained by a call to sys_fhlock(). The current task must have made a sucessful call to sys_fhlock() for a region starting at the same byte *position*.

Note that locks are automatically removed and files all closed if a task stops or is killed (including if it encounters a **restart** statement or chains to another task).

Return Value:
Returns 0 on sucess or a negative error code.

-o-

Synopsis:
sys_fhsetsize(file_handle,size)
Arguments:
file_handle - The file handle
size - The size to set the file to

Description: This function will set the file specified by *file_handle* to the size specified by *size.* If the *size* specified is less than the current file size, the file will be truncated. If the *size* specified is longer than the current file size, the file will be extended with "random" data (i.e., data currently residing in currently unassigned sectors)

Return Value:

Returns 0 on sucess else a nga

-o-

File System Functions

sys_fcopy

Synopsis:
sys_fcopy(source_filename,dest_filename{,non_block_flag})
Arguments:
source_filename - The name of the source file
dest_filename - The name of the destination file
non_block_flag - Optional flag to specify if the calling task should block until copy completes.

Description: This function attempts to copy the file specified in *source_filename* to the file specified in *dest_filename.* Since a file copy can take a substaintial amount of time (depending on the file size) the copy is carried out in a separate thread and the calling task is blocked (unless the *non_block_flag* is specified and set to a non zero value). When the copy has complete then the calling task is woken up.

If the optional *non_block_flag* is set to a non-zero value then the function will return immediately. Only one file copy can be carried out by a task at any one time, although multiple file copies can be being carried out by multiple tasks

Return Value:
Returns 0 on sucess or a negative error value is the copy fails.

Return Value: Will return 1 if end of file has been reached, otherwise will return 0 or a negative error code

-o-

sys_dirremove

Synopsis:
sys_dirremove(directory_name)

Arguments:

directory_name - The name of the directory to remove

Description: This function attempts to remove the *directory_name* specified. The function will fail if the specifed directory is not empty.

Return Value:
Returns 0 on success or a negative error code.

-o-

sys_frename

Synopsis:

sys_frename(filename,new_filename)

Arguments:

filename - The name of the file to rename
new_filename - The new name of the file

Description: This function attempts to rename the file specified by *filename* to the *new_filename* specified. The directory path of the *new_filename* can be different to the original filename in which case the file will moved to the new directory path

Return Value:
This function will return 0 on success or a negative error code.

-o-

sys_dirfirst

Synopsis:

sys_dirfirst(path,attributes)

Arguments:

path - The path to search for
attributes - The attributes of the file or directory to search for.

Description: This function starts a search for the first occurance of the file or directory that matches the *path* and *attributes* specified. Wildcard characters (* and ?) can be used in the *path.*

The *attributes* argument can include one or more of the following characters indicating the attribute(s) of the files to be found:

n Normal file
r Read-only file
h Hidden file
s System file

v Volume label
d Sub-directory
a Archive bit set

If successful, dir_first returns the file name (only the name.ext part, without any preceding directory path or drive). If the path or file given was not found the return value is "?".

Return Value:
Returns "?" or the file name found or a negative error code.

-o-

sys_dirnext

Synopsis:
sys_dirnext()

Arguments:
None

Description: Returns the next matching file as given to sys_dirfirst(), or "?" if no further files can be found or if no valid call has been made to sys_dirfirst(). See description of sys_dirfirst() for an example.

Return Value:

Returns the next matching file or "?".

-o-

sys_diskfree

Synopsis:
sys_diskfree(driveID)

Arguments:
driveID - The ID of the drive (0=A:, 1=B: etc)

Description: Returns the amount of disk space available on the specified *driveID* where driveID=0 represents A:, driveID=1 represents B: etc

Return Value:

Returns the amount of space left on the specified disk drive or a negative error value if an invalid disk is specified.

-o-

sys_fdelete

Synopsis:
sys_fdelete(filename)
Arguments:
filename - The name of the file to delete.

Description: This function attempts to delete the file specified in *filename.*

Return Value:
Returns 0 on sucess or a negative error value if the delete fails.

-o-

sys_dirmake

Synopsis:
sys_dirmake(directory_name)
Arguments:
directory_name - The name of the directory to make

Description: This function attempts to make the *directory_name* specified.

Return Value:
Returns 0 on success or a negative error code.

-o-

sys_gethandle

Synopsis:
sys_gethandle(file_handle)
Arguments:
file_handle - The file handle

Description: This function is used to convert the internal *file_handle* returned from sys_fhopen() into a hex-string representation of the actual operating system handle. Hex strings are used to represent binary values as strings where each byte of the binary value is represented by a two charater hexidecimal string.

The value returned in this case will be an 8 character hex-string representing the 4 byte Windows file handle.

This function is typically used by other DLL libraries to access a real operating system file handle when passed the internal file handle returned by sys_fhopen().
(For example the function SMplayh(chan,filehandle) in the CXACUDSP library uses this function)

It is unlikely that this funtion will be used from your code.

Return Value:

Returns the hexidecimal string representation of the actual Window file handle.

-o-

sys_finfo

Synopsis:
sys_finfo(filename,info_type)
Arguments:
filename - The name of the file to delete.
info_type - The type of information to retrieve
Description: This function returns various file system information about the file specified in *filename.* The *info_type* can be one of the following:

1 File size
2 Modified Date YYMMDD
3 Modified Time HHMMSS
4 Directory/file ("D" or "F")
5 Read only? ("Y" or "N")

Return Value:
Returns the specified information or a negative error value

-o-

Date and Time Functions

sys_date

Synopsis:
sys_date({long_format_flag})

Arguments:
long_format_flag - if this optional argument is set to a non zero value then date will be returned in long format (YYYYMMDD)

Description: This function returns the current system date in the form YYMMDD unless the optional *long_format_flag* is set to a non-zero value in which case it will return the date in YYYYMMDD format.

Return Value:
Returns the current date.

-o-

sys_time

Synopsis:
sys_time()

Arguments:
None

Description: This function returns the current system time in the form HHMMSS.

Return Value:
Returns the current time.

-o-

sys_ticks

Synopsis:
sys_date({ms_flag})

Arguments:
ms_flag - if this optional argument is set to a non zero value then the function returns the number of ticks in millseconds.

Description: This function returns the number of system ticks since midnight. By default this returns the number of BIOS ticks which occur at a rate of 18.2 per second. However if the *ms_flag* argument is set to a non-zero value then this will return the number of ticks in millseconds.

At midnight the number of ticks will reset back to zero so care should be taken when comparing two tick values that might span midnight.

Return Value:
Returns the number of system ticks since midnight.

-o-

sys_timeadd

Synopsis:
sys_timeadd(start_date,start_time,seconds)

Arguments:

start_date - The starting date (either YYYYMMDD or YYMMDD)
start_time - the starting time (HHMMSS)
seconds - The number of seconds to add to the start_date and start_time

Description: This function adds the specified number of *seconds* to the *start_date* and start_time and returns a new date and time as a single string with the format YYMMDDHHMMSS or YYYYMMDDHHMMSS depending on how the *start_date* was originally specified.

Return Value:
Returns the new date and time string after adding *seconds* to the original date and time.

-o-

sys_timesub

Synopsis:

sys_timesub(start_date,start_time,end_date,end_time)

Arguments:

start_date - The starting date (either YYYYMMDD or YYMMDD)
start_time - The starting time (HHMMSS)
end_date - The starting date (either YYYYMMDD or YYMMDD)
end_time - The starting time (HHMMSS)

Description: This function returns the number of seconds that have elapsed between the *start_date* and *start_time* and the *end_date* and *end_time*.

Note that a negative value will be returned if the start date and time is later than the end date and time..

Return Value:
Returns the seconds elapsed between the two dates and times.

-o-

sys_dateadd

Synopsis:

sys_dateadd(start_date,days)

Arguments:

start_date - The starting date (either YYYYMMDD or YYMMDD)
days - The number of days to add

Description: This function adds the specified number of *days* to the *start_date* and returns a new date in the same format that the original was specified in.

If a negative value is given for *days* then the returned date will be before the start date specifed.

Return Value:

Returns the new date after adding the specified number of days to the start date.

-o-

sys_tmrstart

Synopsis:
sys_tmrstart()

Arguments:
none

Description: This function starts a timer that can then be periodically checked by subsequently calling sys_tmrsecs(). This timer has a one second resolution.

Return Value:
Returns 0

-o-

sys_tmrsecs

Synopsis:
sys_tmrsecs()

Arguments:
none

Description: Returns the number of seconds that have elapsed since the last call to sys_tmrstart().

Return Value:
Returns 0

-o-

sys_settime

Synopsis:
sys_settime(date,time)

Arguments:
date - The starting date (either YYYYMMDD or YYMMDD)
time - the starting time (HHMMSS)

Description: This function sets the system date and time to the values specified.

Return Value:
Returns 0 or a negative error code if an invalid date/time is specified.

-o-

sys_datecvt

Synopsis:
sys_datecvt(date,type)

Arguments:
date - The date to convert (YYYYMMDD or YYMMDD)
type - How to convert the date (0=day of week, 1=day of year)

Description: This function converts the specified date into a number that represents either the day of the week or the day of the year.

If *type* is set to 0 then the function will return a value that represents the day of the week 0 - 6 for Sunday - Saturday.

If *type* is set to 1 then the function will return a number between 0 and 364 (or 365 on a leap year) representing the day of the year.

Return Value:
Returns the day of the week or day of the year.

-o-

Other System Functions

sys_exit

Synopsis:
sys_exit(exit_value)

Arguments:
exit_value - The value which the Telecom Engine will exit with.

Description: This function will cause the Telecom Engine to quit with the exit code specified by *exit_value.* Note that this is not a graceful shutdown - the system will exit immediately so caution should be taken when using this call.

Return Value:
none

-o-

sys_getenv

Synopsis:
sys_getenv(var_name)

Arguments:
var_name - The name of the environment variable to return.

Description: This function returns the value of the specified environment variable *var_name* or an empty string "" if the variable does not exist.

Return Value:
returns the value of the specified environment variable *var_name* or an empty string "" if the variable does not exist.

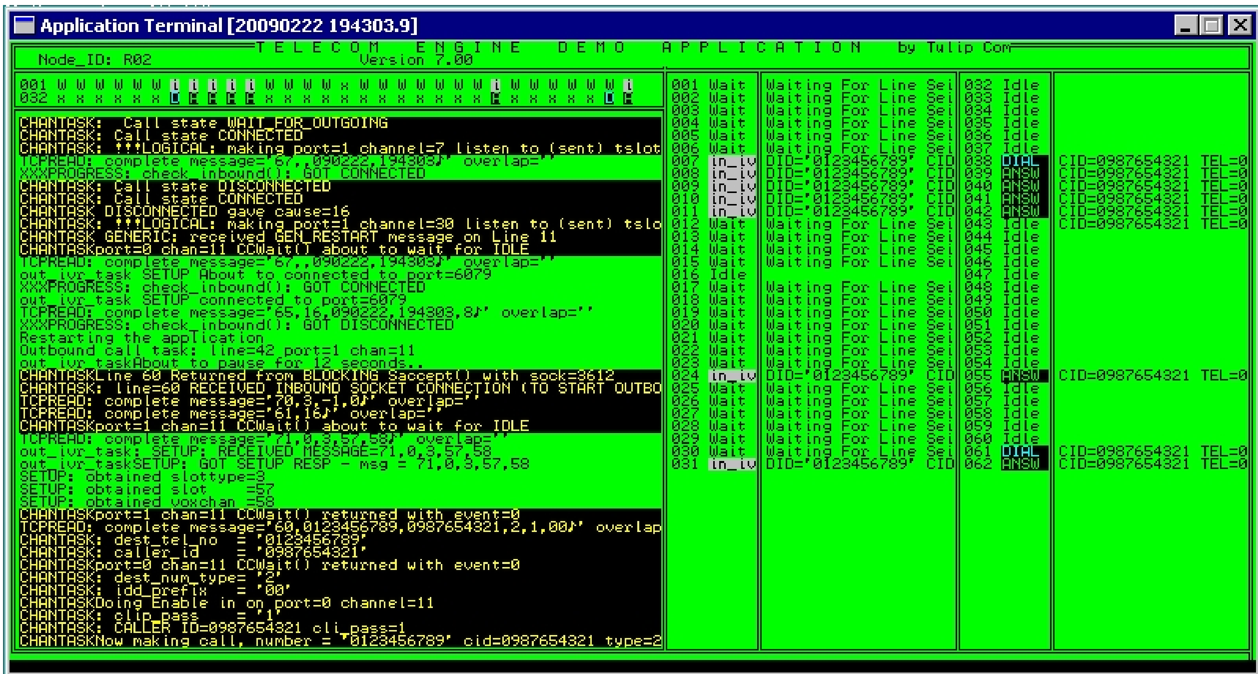
-o-

Terminal Console Library

Introduction

The Application Terminal Library provides a text based application console based around the functionality of an old 16 colour VGA graphics monitor. For critical applications like IVR and other Telecomunications systems, which typically run without much user interference, a simple text based console terminal is usually sufficient to show the status of channels and to provide a scrolling log of events. A more complex graphical user interface (GUI) is not necessarily required on the live system (although a GUI interface for managing resources and database tables from a separate workstation might be required as a separate application).

Thus the Terminal Console Library is offered as part of the standard TE libary set and provides functions for simple terminal input, output and logging to offer the above basic functionality. When the Terminal Console Library DLL (CXTERMX.DLL) is loaded by the run-time Telecom Engine, the Application Console window is displayed on the screen that is 80 columns wide by 25 rows high, however this can be resized (either by the term_resize() function call or by setting some registry keys). Below is a screen shot from a typical application terminal screen that might be used for a running system:



In the above example the top right shows single character channel status for each channel on each E1 trunk.   Down the right hand side is a more detailed channel status and additional channel/call specific information.   The main area rectangular area on the bottom left is the scrolling log window, showing all calls to the applog(), errlog(), tracelog() and similar functions to display scrolling messages.

Obviously the programmer is free to design a screen that is suitable for the needs of the application and may simply consist of a simple black and white scrolling log without any of the above channel status areas etc.

If an application requires something more sophisticated than a text based terminal window, then the programmer should consider writing a window GUI application and communicating with their program using the TCP/IP socket library (CXSOCK.DLL) to receive the appropriate information, or otherwise another more sophisticate terminal libary could be developed.

The font used by the Application Terminal console is 'Terminal' with character size 10.   This can be changed by a call to term_resize() or by setting up some entries in the registry.   The registry information that is used by the library is found under the following registry key:

HKEY_LOCAL_MACHINE\Software\Telecom Engine\CXTERMX

and the entries found under this key are as follows:

SCREEN_HEIGHT   - The number of character rows for the terminal screen (Default 25)
SCREEN_WIDTH   - The number of character columns for the terminal screen (Default 80)
FONT_SIZE   - The size of the font used (Default 10)

Since the application terminal is modelled on an old VGA 16 colour terminal we have the following 16 colours defined:

Colour ID	24 Bit definition	Colour Name	VGA Code

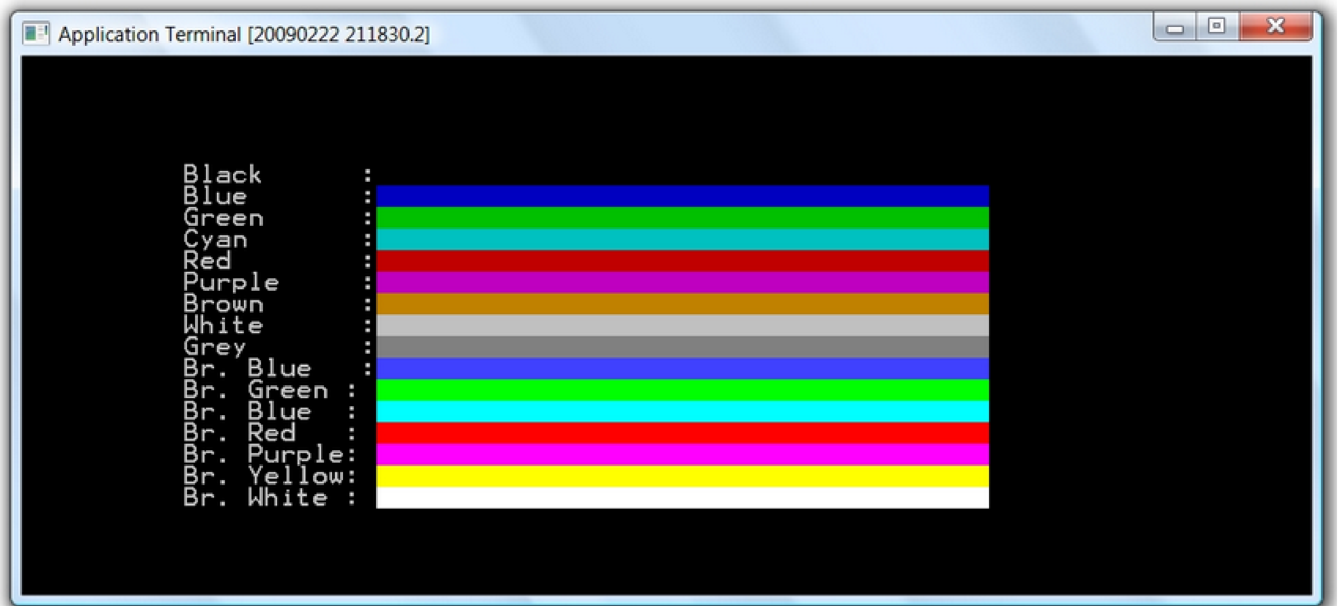
CL_BLACK	0x000000	Black	0
CL_BLUE	0xC00000	Blue	1
CL_GREEN	0x00C000	Green	2
CL_CYAN	0xC0C000	Cyan	3
CL_RED	0x0000C0	Red	4
CL_PURP	0xC000C0	Purple	5
CL_BROWN	0x0080C0	Brown	6
CL_WHITE	0xC0C0C0	White	7
CL_GREY	0x808080	Grey	8
CL_B_BLUE	0xFF4040	Bright Blue	9
CL_B_GREEN	0x00FF00	Bright Green	10
CL_B_BLUE	0xFFFF00	Bright Blue	11
CL_B_RED	0x0000FF	Bright Red	12
CL_B_PURP	0xFF00FF	Bright Purple	13
CL_B_YELLOW	0x00FFFF	Bright Yellow	14
CL_B_WHITE	0xFFFFFF	Bright White	15

The following program prints out the 16 colours used by the application terminal libary:

```
const CL_BLACK =0;
const CL_BLUE  =1;
const CL_GREEN =2;
const CL_CYAN  =3;
const CL_RED   =4;
const CL_PURP  =5;
const CL_BROWN =6;
const CL_WHITE =7;
const CL_GREY  =8;
const CL_B_BLUE=9;
const CL_B_GREEN=10;
const CL_B_BLUE=11;
const CL_B_RED =12;
const CL_B_PURP=13;
const CL_B_YELLOW=14;
const CL_B_WHITE=15;

main

  int i;
    term_set_attr(CL_B_WHITE,CL_BLACK);
   term_clear();

   for(i=0;i<=15;i++)
       term_cur_pos(5+i,10);
        term_set_attr(CL_WHITE,CL_BLACK);
      switch(i)
         case 0:        term_print("Black     : ");
         case 1:        term_print("Blue      : ");
         case 2:        term_print("Green     : ");
         case 3:        term_print("Cyan      : ");
```

```
        case 4:       term_print("Red       : ");
        case 5:       term_print("Purple    : ");
        case 6:       term_print("Brown     : ");
        case 7:       term_print("White     : ");
        case 8:       term_print("Grey      : ");
        case 9:       term_print("Br. Blue  : ");
        case 10:      term_print("Br. Green : ");
        case 11:      term_print("Br. Blue  : ");
        case 12:      term_print("Br. Red   : ");
        case 13:      term_print("Br. Purple : ");
        case 14:      term_print("Br. Yellow : ");
        case 15:      term_print("Br. White  : ");
      endswitch
      term_set_attr(i,i);
      term_cur_pos(5+i,22);
      term_print("                    ");
    endfor
endmain
```

When this program is run the following output is displayed on the terminal screen:



-o-

Terminal Console Library Quick Reference

applog(str1[,str2],...]]);
syslog(str1[,str2],...]]);
errlog(str1[,str2],...]]);
tracelog(str1[,str2[,...]]);
term_log(str1[,str2[,...]]);
term_write(str1[,str2[,...]]);
term_scroll_area(row,column,width,height)

© Zentel Telecom Ltd, 2009

term_cur_pos(row,column);
term_print(str1[,str2[,...]]);
term_box(row,column,width,height)
term_colour(forground,background)/term_colour(user_defined_colout)
term_put_nch(character,number_of_times)
term_fill(row,column,width,height,character)
term_attr_def(attribute_num,fgcolour_24bit,bgcolour_24bit)
term_clear()
term_kbgetx()
term_kbget()
term_kbqsize()
term_errctl(on_off)
term_edit(row, column, width, initial, attribute)
term_resize(rows,columns[,FontSize[,FontName]])
term_size(pRows,pColumns)

-o-

Terminal Console Function Reference

applog

Synopsis:
applog(string1,[string2[,...]])

Arguments:
string1,string2... - The string arguments provided are concatenated together for form the final log string.

Description: This function concatenates the *string* arguments passed to it into a single string and writes the resulting string to the Application Terminal scrolling area. The function also writes the log string to the application log file, which resides in the directory where the Telecom Engine run-time program was started from.

The application log files cycle though ten application logs: APPLOG0.LOG, APPLOG1.LOG...APPLOG9.LOG before starting back at the beginning and overwriting the previous application logs again. Each application log file will reach a maximum size of 2MB before moving on to the next file - so a total of 20MB of log data will be stored until it starts to get overwritten. Upon start-up of the Terminal Console DLL, the last log file that was written to is located (by modified date) and the next application log file is then used and overwritten. Therefore each time the Telecom Engine is started the Terminal Console DLL will start at the next log file after the last one that was written. This way a history of log files is maintained which may be useful for debugging purposes. If you want to always start at APPLOG0.LOG then you should delete this log file (say in a batch file) each time the system is restarted.

The log file messages that are written to the application logs have some information prefixed to the string before writing to the log. This information has the following format:

<date (YYYYMMDD)>:<time (HHMMSS.ms)>: <Message Type>:<Task_name (Tex file name)>: <Task ID>: <Program Counter (Hex)>: *string*

The <Message Type> Field will be one of the following:

L - Normal Log message
E - Error Message
D - Debug Message
T - Trace Message

For example in the following program in TES file **MYAPP.TES**

```
int a,b;
main
   a=123;
   b=45678;
   applog("The value of a=",a);
   applog("The value of b=",b);
endmain
```

Would result in something similar to this being written to the application log:

APPLOG0.LOG

20060413:151334.567:L:myapp:0:000a: The value of a=123
20060413:151334.567:L:myapp:0:0023: The value of b=4567

Return Value:
Returns 0

-o-

syslog

Synopsis:
syslog(string1,[string2[,...]])

Arguments:
string1,string2... - The string arguments provided are concatenated together for form the final log string.

Description: This function concatenates the *string* arguments passed to it into a single string and writes the resulting string to the system log screen provided by the front-end (TEX.EXE) application . NOTE that unlike the errlog() and tracelog() functions the message is not copied to the application terminal or application log.

The system log files (provided by the TEX.EXE front-end application) cycle through just two files SYSLOG0.LOG and SYSLOG1.LOG.

Return Value:
Returns 0

-o-

errlog

Synopsis:
errlog(string1,[string2[,...]])

Arguments:
string1,string2... - The string arguments provided are concatenated together for form the final log string.

Description: This function concatenates the *string* arguments passed to it into a single string and writes the resulting string to the Application Terminal scrolling area in red. The function also writes the log string to the application log file and system error file, which resides in the directory where the Telecom Engine run-time program was started from.

The application log files cycle through files APPLOG0.LOG through to APPLOG9.LOG, whereas the system error log files (provided by the TEX.EXE front-end application) cycle through just two files ERRLOG0.LOG and ERRLOG1.LOG. See applog() for a description of the application log format.

Return Value:
Returns 0

-o-

tracelog

Synopsis:
tracelog(string1,[string2[,...]])

Arguments:
string1,string2... - The string arguments provided are concatenated together for form the final log string.

Description: This function concatenates the *string* arguments passed to it into a single string and writes the resulting string to the Application Terminal scrolling area in yellow. The function also writes the log string to the application log file and system trace file, which resides in the directory where the Telecom Engine run-time program was started from.

The application log files cycle through files APPLOG0.LOG through to APPLOG9.LOG, whereas the system trace log files (provided by the TEX.EXE front-end application) cycle through just two files TRACELOG0.LOG and TRACELOG1.LOG. See applog() for a description of the application log format.

Return Value:
Returns 0

-o-

term_errctl

Synopsis:
term_errctl(off_or_on_)

Arguments:
off_or_on - Set to 0 to turn error supress off, or 1 to turn error supress on.

Description: This function allows the application to suppress the printing of error messages to the error log and application log. This is typically used when the programmer knows that an error message might be generated but this would be an expected behaviour that doesn't need to be printed to the log. A typical example of when term_errctl() could be used to suppress an error message is shown below:

```
# Check if the control file exists..
term_errctl(1);    # Prevent 'file not found' error message from printing..
fh=sys_fhopen("control.txt","rs");
term_errctl(0);    # switch error suppression off again

# if it does exist then do something
if(fh > 0)
    sys_fhclose(fh);
    # Do something....
    etc.
 endif
```

In the above example we are checking for the existance or non-existance of a control file to adjust the program flow. Since the non-existance of the file is a perfectly valid event then we don't want to print out the 'file not found' error that the sys_fhopen() would print out.

Another example is shown below:

```
term_errctl(1);   # Suppress error messages
SMplay(vox_chan,"advert.vox");    # play the optional advert if it exists
term_errctrl(0);  # allow error logging again
```

In the above example the program attempts to play 'advert.vox', but if this doesn't exist then the program will simply return immediately and drop through. Since the 'advert.vox' is an option prompt then we supress the printing of the 'file not found' mesage that SMplay() would normally generate.

Return Value: Returns 0

-o-

term_log

Synopsis:
term_log(string1,[string2[,...]])

Arguments:

string1,string2... - The string arguments provided are concatenated together for form the final log string.

Description: This function concatenates the *string* arguments passed to it into a single string and writes the resulting string to the Application Terminal scrolling area and application log file. This function is identical to the applog() function.

Return Value:
Returns 0

-o-

term_resize

Synopsis:
term_resize(rows,columns[,FontSize[,FontName]])

Arguments:
rows - The number of rows high the terminal window should be
columns - The number of columns wide the terminal window should be
FontSize - The font size of the text (def
FontName - The Name of the font to use (defaults to Terminal)

Description: This function allows the size of the Application Terminal window to be changed and/or the character font or font size to be changed. The number of *rows* and *columns* specified are the number of character *rows* and *columns,* so the actual window size will depend on the *FontName* and *FontSize* used. By default the FontName is "terminal" and the FontSize is 10.

Return Value:
Returns 0 on success or -1 if an invalid argument is supplied.

-o-

term_size

Synopsis:
term_size(&pRows,&pColumns)

Arguments:
pRow - Pointer to a variable that will hold the number of rows of the application terminal window
pColumn - Pointer to a variable that will hold the number of columns of the application terminal window

Description: This function returns the current size of the application terminal window. By default the terminal window will be 25 rows high by 80 columns wide, but this can be changed by the registry variables or by calling the term_resize() function. The number of rows and columns are written to the variables pointed to by the

Return Value:
Returns 0

-o-

term_write

Synopsis:
term_write(string1,[string2],...]])

Arguments:
string1,string2... - The string arguments provided are concatenated together for form the final log string.

Description: This function concatenates the *string* arguments passed to it into a single string and writes the resulting string to the Application Terminal scrolling area, but does NOT write the string to the application log file.

Return Value:
Returns 0

-o-

term_scroll_area

Synopsis:
term_scroll_area(row,column,width,height)

Arguments:
row - The top left row of the scroll area
column - The top left column of the scroll area
width - The width of the scroll area
height - The height of the scroll area

Description: This function defines the scrolling area of the Application Terminal where strings are written from calls to term_write(), applog(), errlor() etc. By default the entire Application Terminal window is the scrolling area until a call is made to this function to modify it.

Return Value:
Returns 0 or -1 if an invalid scroll area is defined.

-o-

term_cur_pos

Synopsis:
term_cur_pos(row,column)

Arguments:
row - The row position to set the cursor
column - The column to set the cursor to

Description: This function moves the *task specific* cursor to the position specified by *row* and *column*. Any subsequent term_print() function will begin from the position specified by this call. The top left row of the application terminal is row 0, column 0.

Note that after a term_print() function call the cursor position does not change. Another call to term_cur_pos() is needed to change the cursor position.

The cursor position of other task's cursors will not be affected by this call.

Returns: 0 for success or -1 for error (e.g attempt to postion cursor off screen)

-o-

term_print

Synopsis:
term_print(string1,[string2[,...]]))

Arguments:
string1,string2... - The string arguments provided are concatenated together for form the final string to print to the screen.

Description: This function prints the given strings to the application terminal at the current cursor position. The cursor position remains unchanged after a call to term_print() and to move it will require an explicit call to term_cur_pos().

Returns: 0 for success or -1 for error

-o-

term_box

Synopsis:
term_box(row, column, width, height)

Arguments:
row - The top left row of the scroll area
column - The top left column of the scroll area
width - The width of the box
height - The height of the box

Description: This function draws a rectangular box bounded by double lines and blanks out the interiior of the rectangle.
The width and height include the border, so the blanked area is (height - 2) lines by (width - 2)

columns. The border and interior blanks are written with the current terminal attributes set by the term_set_attr() function .

Returns: Empty string ("")

-o-

term_colour

Synopsis:

term_colour(foreground,background)

or

term_colour(user_defined_colour)

Arguments:

Foreground - The foreground colour (0..15)
background- the background colout (0..15)

or

user_defined_atrribute - One of the 256 colour attributes that can be defined by the user using term_attr_def()

Description: This function sets the current foreground and background colours used by the task in subsequent calls to terminal output functions (such as term_print(), term_write, applog() etc).

The function can take one of two forms, the first of which specifies the foreground and background colours from the predefined set of colours shown below (foreground and backTaken from the Colour code column):

Colour Name	Colour Code
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Purple	5
Brown	6
White	7
Grey	8
Bright Blue	9
Bright Green	10
Bright Blue	11
Bright Red	12

Bright Purple	13
Bright Yellow	14
Bright White	15

The second form of the function takes one of the 256 user defined colour attribute that can be defined by the term_attr_def() function. A colour attribute is a number between 0 and 255 and which indexes a table that provides both the foreground and background colour for that attribute.
Typically this form of the function is used to allow colour combinations outside of the predefined colours defined in the above table.

Return Value: Returns 0 or a negative error value (e.g if an invalid colour is given).

-o-

term_attr_def

Synopsis:
term_attr_def(attribute_id,fgcolour_24bit,bgcolour_24bit)
or
term_attr_def(attribute_id,VGA_256code)

Arguments:
atrribute_id - The atrribute ID for the define colour attribute (0..255)
fgcolour_24bit - The foreground colour (0..15)
bgcolour_24bit - The background colout (0..15)

or
atrribute_id - The atrribute ID for the define colour attribute (0..255)
VGA_256code - A VGA colour code made from a combination of one of the existing 16 predefined colours

Description: This function allows for one of up to 256 colour attributes to be defined by the user for use with the term_colour(user_defined_attribute) function. A colour attribute is an index into table of up to 256 entries which defines both the foreground and background colour in a single attribute ID. This attribute ID can then be passed to the term_colour(attributeID) function to define the current foreground and background colour.

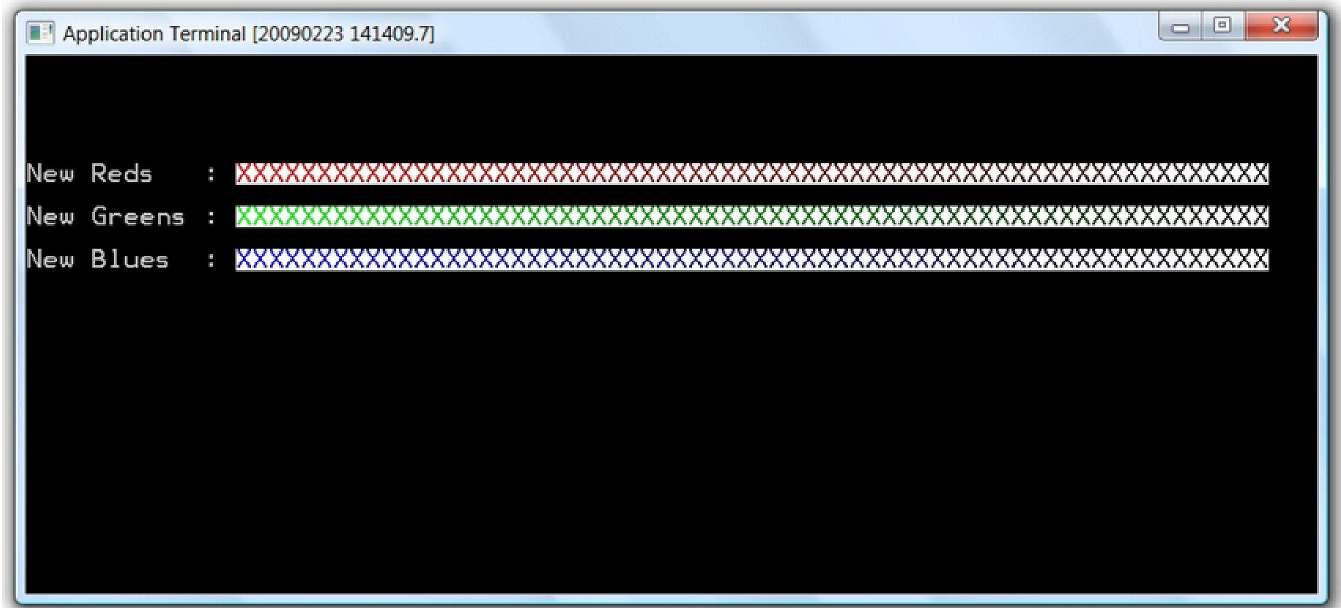
The first form of the function allows for new colours to be defined that are not a combination of any of the 16 predefined colours. In this form the function takes the attribute_id and the 24 bit colour value for the new foreground and background in the *fgcolour_24bit* and *bgcolour_24bit* arguments.

For example, the following program defines in turn 64 different shades of red, green and blue on white (in attributes 0..63) and uses these new attributes to display 64 characters on the screen:

```
const CL_RED=0x0000FF;
const CL_WHITE=0xFFFFFF;

const CL_16_WHITE=7;
const CL_16_BLACK=0;
```

```
main

   int i;

   # Define 64 new shades of Red on white
   for(i=0;i<64;i++)
     vid_attr_def(i,CL_RED-(i*4),CL_WHITE);
   endfor

   // Set the colour to white on black
   term_colour(CL_16_WHITE,CL_16_BLACK);
   term_cur_pos(5,0);
   term_print("New Reds  :");

   // Now print out 64 characters using the new colour attributes (0..63)
   for(i=0;i<64;i++)
     term_cur_pos(5,13+i);
     // Set the colour to one of the new user defined colours
     term_colour(i);
     term_print("X");
   endfor


   # Define 64 new shades of Green on white
   for(i=0;i<64;i++)
     // Multiple by 256 to shift into Green bits of colour field (0x00FF00)
     vid_attr_def(i,256*(CL_RED-(i*4)),CL_WHITE);
   endfor

   // Set the colour to white on black
   term_colour(CL_16_WHITE,CL_16_BLACK);
   term_cur_pos(7,0);
   term_print("New Greens:");

   // Now print out 64 characters using the new colour attributes (0..63)
   for(i=0;i<64;i++)
     term_cur_pos(7,13+i);
     // Set the colour to one of the new user defined colours
     term_colour(i);
     term_print("X");
   endfor


   # Define 64 new shades of Blue on white
   for(i=0;i<64;i++)
     // Multiple by 256*256 to shift into Blue bits of colour field (0xFF0000)
     vid_attr_def(i,256*256*(0xFF-(i*4)),CL_WHITE);
   endfor

   // Set the colour to white on black
   term_colour(CL_16_WHITE,CL_16_BLACK);
   term_cur_pos(9,0);
   term_print("New Blues  :");

   // Now print out 64 characters using the new colour attributes (0..63)
   for(i=0;i<64;i++)
     term_cur_pos(9,13+i);
     // Set the colour to one of the new user defined colours
     term_colour(i);
     term_print("X");
   endfor

endmain
```

The output from the above program is shown below:



The second form of the function:

    term_attr_def(attribute_id,VGA_256code)

Allows an attribute to be defined which is a combination of two of the existing predefined colours. The VGA_256code is a number between 0 and 255 created from the following formula: VGA_256code=16 * Background + Foreground

Where the Background and Foreground colours are taken from the set of 16 predefined colours shown in the following table.

Colour Name	**Colour Code**
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Purple	5
Brown	6
White	7
Grey	8
Bright Blue	9
Bright Green	10
Bright Blue	11

Bright Red	12
Bright Purple	13
Bright Yellow	14
Bright White	15

Below is an example showing attribute ID 1 being set to white on cyan from the existing colour set..

```
// White on Cyan from the existing colour set
term_attr_def(1,7*16+3);
// Change to the new colour..
term_colour(1);
```

-o-

term_put_nch

Synopsis:
term_put_nch(character, number_of_times)

Arguments:
character - The character to write
number_of_times - the number of times to write the character

Description: This function writes the specified *character* to the screen at the current cursor position the given *number_of_times*. The cursor position remains unchanged after a call to term_put_nch() and to move it will require an explicit call to term_cur_pos().

The characters are written in the current active colour for the task as set by term_colour().

Returns: 0 for success or -1 for error

-o-

term_fill

Synopsis:
term_fill(row, column, width, height, character)

Arguments:
row -
character - The character to write
number_of_times - the number of times to write the character

Description: This function fills the given rectangular area with the first *character* in the string argument using the current active terminal colour for the task (as set by term_colour()). The *row* and *column* arguments define the top left position of the rectangular area and the *width* and

height arguments define the number of characters across and down to draw the rectangle.

Row and column numbers count from (0,0) at the top left corner of the screen.

The characters are written in the current active colour for the task as set by term_colour().

The following example loops drawing concentric rectangles of different colours:

```
main
int colour;

   # Loop forever..
   while(1)
      for(colour=1;colour<=12;colour++)
         # Set colour to colour on black..
         term_colour(colour,0);
         # Fills a rectangle 2 chars smaller that the current colour
         term_fill(0+colour,0+(colour*2),80-(colour*4),25-colour*2,"X");
         # Sleep half a second
         sleep(5);
      endfor
   endwhile
end
```

Returns: 0 for success or -1 for error

-o-

term_clear

Synopsis:
term_clear()

Arguments:
NONE

Description: This function clears the current terminal screen and writes spaces to every character location on the screen using the current task's colour as specified by term_colour().

The following code causes the application terminal screen to change colour rapidly..

```
main
int colour;


   while(1)
      for(colour=0;colour<16;colour++)
         // Set colour
         term_colour(colour,colour);
         // Fills a rectangle 2 chars smaller that the current colour
         term_clear();
         // Sleep half a second
         sleep(5);
      endfor
   endwhile
end
```

Return Value: Returns 0

-o-

term_kbget

Synopsis:
key=term_kbget()

Arguments:
NONE

Description: This function suspends the calling task until a key has been hit and returns the ASCII value of the key that was pressed. If there are already one or more keys in the keyboard buffer then this function will return immediately with the ASCII value of the first key in the keyboard buffer and will remove that key from the buffer.

Only one task can be calling term_kbget() or term_kbgetx() at any one time. If there is already a task waiting for a key press with this function then any other tasks calling this function will cause the function to display an error message and a blank string ("") will be returned.

It is possible to obtain the number of keys waiting in the keyboard buffer by calling the term_kbqsize() function.

A maximum of 256 keys can be held in the keyboard buffer before they start being overwritten.

Returns: Returns a string containing the string that was pressed.

-o-

term_kbgetx

Synopsis:
key=term_kbgetx()

Arguments:
NONE

Description: This function suspends the calling task until a key has been hit and returns the four digit scan code of the key that was pressed. If there are already one or more keys in the keyboard buffer then this function will return immediately with the scan code value of the first key in the keyboard buffer.

The value is NOT removed from the buffer, so repeated calls to term_kbgetx() will continue to give the same value (if there are type-ahead characters, the returned character is always the earliest in the buffer). A call to term_kbgetx() will generally be followed by a call to term_kbget() to remove the keystroke.

Only one task can be calling term_kbget() or term_kbgetx() at any one time. If there is already a task waiting for a key press with these functions then any other tasks calling this function will cause the function to display an error message and a blank string ("") will be returned.

It is possible to obtain the number of keys waiting in the keyboard buffer by calling the term_kbqsize() function.

A maximum of 256 keys can be held in the keyboard buffer before they start being overwritten.

The first two characters of the a scan code indicate which key has been pressed. The second two characters indicate the ASCII code (as a hexadecimal number), if any, for the pressed key.

For example, [Esc] the first key on the keyboard (in the standard IBM layout), and has ASCII code 27 decimal, 1b hexadecimal; hence "011b". Special keys which have no ASCII code have "00" as the last two characters, for example the function key [F1].

The following tables show the scan codes for the special keys:

Function Keys:

key	Scan Code
[F1]	3b00
[F2]	3c00
[F3]	3d00
[F4]	3e00
[F5]	3f00
[F6]	4000
[F7]	4100
[F8]	4200
[F9]	4300
[F10]	4400
[F11],[F12]	ignored

Special Keys:

[Enter]	1c0d
[↑]	4800
[Backspace]	0e08
[↓]	5000
[Esc]	011b
[→]	4d00
[Tab]	0f09
[←]	4b00
[Ins]	5200
[PgUp]	4900
[Del]	5300
[PgDn]	5100
[Home]	4700

[End]	4f00
[Ctrl]+[Enter]	1c0a
[Shift]+[Tab]	0f00

[Alt]+Key:

[A] 1e00	[B] 3000	[C] 2e00	[D] 2000
[E] 1200	[F] 2100	[G] 2200	[H] 2300
[I] 1700	[J] 2400	[K] 2500	[L] 2600
[M] 3200	[N] 3100	[O] 1800	[P] 1900
[Q] 1000	[R] 1300	[S] 1f00	[T] 1400
[U] 1600	[V] 1f00	[W] 1100	[X] 2d00
[Y] 1500	[Z] 2c00	[1] 7800	[2] 7900
[3] 7a00	[4] 7b00	[5] 7c00	[6] 7d00
[7] 7e00	[8] 7f00	[9] 8000	[0] 8100
[-] 0c00	[=] 0d00		

The following program prints out the scan codes for each key:

```
main
var scan:4;
var ch:1;

   while(1)
      scan=kb_getx();
      ch=kb_get();
      applog("Scab=",scan," ch=",ch);
   endwhile
endmain
```

Returns: Returns a string containing the scan code of the key that was pressed.

-o-

term_kbqsize

Synopsis:
key_count=term_kbqsize()

Arguments:
NONE

Description: This function returns the number of keys that are waiting in the keypress buffer. A maximum of 256 keys will be stored in the buffer after which they will start to be overwritten.
Keys are removed from the buffer by calls to term_kbget() or term_kbedit().

-o-

term_edit

Synopsis:

input_string = term_kbedit(row, column, width, initial_str[, attribute]);

Arguments:

row - The row where the edit will start
column - The column where the edit will start
width - The number of characters to get
initial_str - The initial value of the edit string
[attribute] - Option attribute ID (as defined by term_attr_def())

Description: This function allows for string input to be obtained from the keyboard. The calling task will be suspended while the input is being entered and will only return when one of the input termination keys is pressed. Only one task at a time can be calling a blocking keyboard input function so if any other task is currently executing term_kbget(), term_kbgetx() or term_kbedit() then the function will return immediately with a bloank string and will output an error message

The editing area on the screen is defined by the row, column and width arguments and the *initial_str* value will be displayed in this area when the function is first called with the cursor positioned at the end for the *initial_str*.

Editing keys are the following:

[Enter] - Terminates the input and returns the edited string
[Esc] - Aborts the edit and returns the initial string
[Backspace] - Deletes the character at the previous position
[Del] - Deletes the character at the current position
[←] [→] - Moves the cursor left or right through the editing string.

If any other special (non-ascii key) is pressed then this will terminate the edit as if Enter had been pressed.

Note: The key that terminated the input is not returned with the input string and is left in the keyboard buffer. This key nust then be removed from the keyboard buffer using term_kbget() before term_kbedit() is called again (otherwise the next and subsequent calls to term_kbedit() will return immediately as though enter had been hit). This is a common cause of bugs when using this function.

Below is an example program which loops continuously returning input from the term_kbedit() function and printing the returned string to the scrolling log area:

```
main

  var input_str:60;

  term_box(5,0,80,20);
  term_scroll_area(6,1,78,18);
  term_cur_pos(3,0);
  term_print("Enter a string:");
```

```
  while(1)
      input_str=term_kbedit(3,16,50,"This is the initial value");
      term_kbget();
      applog("Input=",input_str);
  endwhile
endmain
```

-o-

ActiveX Data Ojects (ADO) Database Library

Introduction

The ActiveX Data Object (ADO) Library (CXADO.DLL) provides an easy-to-use, be a high-level interface to provide ease of access to data stored in a wide variety of database sources. ADO is a Microsoft technology which stands for **A**ctiveX **D**ata **O**bjects ADO and is a Microsoft Active-X component. The CXADO Libary provides a layer of abstraction between your application and the low-level OLE DB interfaces so that data access can be acheived without having to learn the intricacies of COM or OLE DB.

The current version of the CXADO.DLL library provides the functions that allow a Telecom Engine to establish **connections** to various database sources and then to excecute queries and commands (including invoking stored procedures) against the tables using Standard Query Language (SQL) statements and to browse and manipulate the returned **recordsets**.

Not all of the functionality of the ADO is exposed by the CXADO library, but the functions that are provided should provide the mechanisms needed for the vast majority of applications. If any additional functionality is required that is not currently supported by the CXADO.DLL library (suchs as streams, stored procedures that return multiple datasets etc.) then it is suggested that the developer write their own middleware application in the language of their choice and communicate with that using a client-server model using the CXSOCKETS.DLL libary.

The entire CXADO.DLL library is implemented using just two ADO objects: The **Connection** object and the **Recordset** object.

Represents a unique session with a data source. In the case of a client/server database system, it may be equivalent to an actual network connection to the server.

An ADO **Connection** object represents a unique session with a data source, including a DBMS, a file store, or a comma-delimited text file. In the case of a client/server database system, the ADO connection can be an actual network connection to the server. Data providers represent diverse sources of data such as SQL databases, indexed-sequential files, spreadsheets, document stores, and mail files. Providers expose data uniformly using a common abstraction called the rowset.

ADO is powerful and flexible because it can connect to any of several different data providers and still expose the same programming model, regardless of the specific features of any given provider. However, because each data provider is unique, how your application interacts with ADO will vary by data provider.

Under the CXADO.DLL library, commands and queries are not executed through the connection object itself (which is possible under ADO), instead all commands and queries must be executed

through a **Recordset** object. Some SQL queries return data as a set of rows in a table (E.g. SELECT query), whereas other queries excecute commands that do not return data (such as the CREATE TABLE query). The CXADO.DLL library provides functions that enable both types of queries to be executed on a **Recordset** object.

The ADO **Command** object is currently not implemented by the library as most functionality can be acheived through executing commands through the **Recordset** object (except for executing stored procedures that return multiple recordsets, or that return both recordsets and parameter values). *Note that the **Command** object may be implemented in future versions if there is a pressing need for it.*

-o-

Some Simple Examples

The best way to get a feel for the capabilities of the library is to provide some simple examples.

In the following code a connection is established to an ODBC provider which establishes a connection to a remote datasource. This example assumes that ODBC has configuration has been configured to provide a Data Source Name (DSN) called MyDSN which connects to the database server. This server could be MS-SQL, Oracle, MySQL, PostGres (or whatever). All connections are made using 'Connection Strings' which will vary from provider to provider, and an understanding of what the provider requires in this connection string will be needed by the programmer.

Once the connection has been established then the program excecutes a simple query on the database by creating a recordset object then opening the object with an SQL query string. In this case it assumes there is a table called 'billing' that has the columns: 'date','time','duration','telno','rate','cost'.

The program then steps through the returned recordset data using the cursor manipulation functions (adoRSetMoveFirst(), adoRSetMoveNext() etc), and prints out the values of the various fields, before closing the recordset and the connection.

The code is as follows:

```
// This include file is provided with the libary and contains standard constant definitions.
$include "ado.inc"

main

    int conHandle;
    int setHanldle;
   int x;

    // Turn on trace of ado function entry, function exit and events
```

```
    adoTrace(1);

    // Get a private adoConnection object handle (named "MyConn")
    conHandle=adoConnection("MyConn",0);
    if(conHandle < 0)
        errlog("Error getting connection handle...err=",conHandle);
        stop;
    endif

    // Open the connection...
    x=adoConnOpen(conHandle,"","","Provider=MSDASQL.1;Password=admin;User ID=postgres;Data
Source=MyDSN");
    if(x < 0)
        errlog("Error opening connection...err=",x);
        stop;
    endif

    // Now get a private adRecordset object (named "MySet")
    voslog("About to get adoRecordset() handle...");
    setHandle=adoRecordset(conHandle,"MySet",0);
    if(setHandle < 0)
        errlog("Error getting recordset handle...error=",setHandle);
        stop;
    else

    // Now execute a query on the recordset
    x=adoRSetQuery(setHandle,adOpenStatic,adLockReadOnly,adCmdText,"select * from billing where cost
> 1.0");
    if(x < 0)
        errlog("Error executing query...err=",x);
        stop;
    endif

    // Move to first record (this is done implicitly by the adoRSetQuery() .. but lets make it explicit for the
example..)
    // ... now that the query has completed successfully, to stop the example growing too long I have stopped
checking
    //        errors for every ado call (although in your application you should really keep checking for errors..)
    adoRSetMovefirst(setHandle);

    // If we get here then the query complete sucessfully...
    while(not adoRSetEOF(setHandle))
        var date:20,time:20,duration:10,cost:10,telno:50;
        adoFldGetValue(setHandle,"date",&date);
        adoFldGetValue(setHandle,"time",&time);
        adoFldGetValue(setHandle,"duration",&duration);
        adoFldGetValue(setHandle,"telno",&telno);
        adoFldGetValue(setHandle,"cost",&cost);

        applog("Date=",date," Time=",time," Dur=",duration," Telno=",telno," cost=",cost);

        // MoveNext
        adoRSetMovefirst(setHandle);
    endwhile

    adoRSetClose(setHandle);
    adoConnClose(contHandle);
endmain
```

In the following example I create a table called 'billing' using the adoRSetCmd() function...

```
// This include file is provided with the libary and contains standard constant definitions.
```

```
$include "ado.inc"

main

    int conHandle;
    int setHanldle;
   int x;

    // Turn on trace of ado function entry, function exit and events
    adoTrace(1);

    // Get a private adoConnection object handle (named "MyConn")
    conHandle=adoConnection("MyConn",0);
    if(conHandle < 0)
        errlog("Error getting connection handle...err=",conHandle);
       stop;
    endif

    // Open the connection...
     x=adoConnOpen(conHandle,"","","Provider=MSDASQL.1;Password=admin;User ID=postgres;Data
Source=MyDSN");
   if(x < 0)
       errlog("Error opening connection...err=",x);
      stop;
   endif

    // Now get a private adRecordset object (named "MySet")
    voslog("About to get adoRecordset() handle...");
    setHandle=adoRecordset(conHandle,"MySet",0);
    if(setHandle < 0)
        errlog("Error getting recordset handle...error=",setHandle);
       stop;
   else
create table customers (Name Text,Balance Float,DOB Date)");
    // Now execute a query on the recordset
     x=adoRSetCmd(setHandle,adOpenStatic,adLockReadOnly,adCmdText,"create table billing (date Date,
time Time, duration Integer, telno Text, cost Float");
   if(x < 0)
       errlog("Error executing command...err=",x);
      stop;
   endif

    applog("Billing table has been created successfully!");
    adoRSetClose(setHandle);
    adoConnClose(contHandle);
endmain
```

There is a subtle difference between the adoRSetQuery() and adoRSetCmd() functions in how the ado events are handled in order to wake up the calling task after the function completes (this only applies in blocking mode (see adBlockMode() function)).

For adoRSetQuery() calls the CXADO.DLL libary ignores the **ExecuteComplete** Event (which always occurs first even on a query that returns data) and instead waits for the **FetchComplete** event before waking up the calling task.   The **FetchComplete** event only triggers on recordsets that return one or more rows of data.   Therefore if a query is executed using the adoRSetQuery() function that does not return any rows of data, then the **FetchComplete** event will not be triggered and so the task will stay blocked indefinitely.   If we tried to wakeup a call to adoRSetQuery() using the **ExecuteComplete** event then any attempt to access the rows of data after it returns are prone to generate errors since the recordset has not completed fetching all of the rows.

In the case of the adoRSetCmd(), the calling task will be woken up as soon as the **ExecuteComplete** event is triggered.

It is possible to carry out a query that returns data using the adoRSetCmd() function, but the application would need to poll the recordset state using the adoRSetState() function to wait for the data to be fully fetched before trying to access it. Similarly one could carry out a query that doesn't return any data using the adoRSetQuery() function so long as non-blocking mode is used (see adoBlockMode()) and the application polls to wait for the execution to complete using the adoRSetState() function.

-o-

Blocking or non-blocking mode

The functions adoConnOpen(), adoRSetQuery() and adoRSetCmd() are executed Asynchronously by the CXADO.DLL libary and can be called in blocking or non-blocking mode.

In blocking mode, the task will block until the appropriate event has been triggered in the libary, after which the task will be woken up and the function will return with an appropriate error code.

In non-blocking mode the function will always return immediately and it is up to the application to poll the state of the **connection** or **recordset** to determine when the execution of the connection or query has completed (see adoConnState(), adoRSetState()).

To change the state of a task to non-blocking mode then call the adoBlockMode(blocking_flag) function. Set the blocking_flag argument to 0 for blocking mode and a non-zero value for non-blocking mode. By default a task will start in blocking mode.

-o-

Performance and blocking calls

Every attempt has been made to ensure that all function calls are made aynchronously and that control is returned to the Telecom engine Scheduler as quickly as possible. However, even though all connection opens and recordset queries are specified to be carried out asynchronously, the various calls still seem to block briefly. The adoConnOpen(), adoRSetQuery() and adoRSetCmd() functions can block for something in the order of 100 - 200ms on a local area network before returning control back to the calling task. This may seem a fairly short time, but considering that the Telecom Engine task scheduler will swap tasks at a rate of many tens of thousands of times per second, then this delay can cause a significant performance drop if many tasks are attempting connections or queries at the same time.

A solution is to establish all connections at start-up and try to limit the number of queries being made by the application (do not poll continuously on a database without a significant delay between polls for example). Alternatively the use of stored procedures should increase the performance of queries, and it might be useful to carry out performance tests on a datasource to understand where performance problems may arise.

The adoTrace() function allows for tracing of the entry and exit points of all ado functions and this

also provides information about how long the function took to complete (in ms). Note that the time shown by the trace will be the time taken to execute the actual ado command and not the time it took for the complete asyncronous action to complete.

For example the following trace is taken from a call to adoConnOpen():

```
20090218 135453.162 Task 0000 adoConnOpen() [Entry]
20090218 135453.162 About to Connect open Provider=MSDASQL.1;Password=admin;User ID=postgres;Data Source=MyDSN
20090218 135453.162 Marking connection 0 STATE_PENDING
20090218 135453.363 Task 0000 adoConnOpen() [Exit]: Returns 0: Function took 203ms
```

Even though the adoConnOpen() function is still blocking waiting for the **ConnectionComplete** event to be triggered it can be seen that the underlying _ConnectionPtr->Open() function took 203ms to complete (even though adAsyncConnect is automatically specified in the options for the Open() call). Later in the same trace log one can see the **ConnectionComplete** event triggered which wakes up the calling task some 352ms later:

```
20090218 135453.715 EVENT (CONNECTION) CONNECTCOMPLETE:  TaskID=0  handle=0  status=1
20090218 135453.715 Marking connection 0 STATE_OPEN
```

Similarly in the trace below for an adoRSetQuery() call which attempts to select all the records from the billing table (>9000), the initial call to adoRSetQuery() takes 74ms, which is actually quite quick and would not present a great deal of problems performance-wise as long as there wasn't a continous loop calling these queries very frequently:

```
20090218 135453.723 Task 0000 adoRSetQuery() [Entry]
20090218 135453.724 About to RSet query select * from billing
20090218 135453.724 Marking recordset 0 PENDING
20090218 135453.795 Task 0000 adoRSetQuery() [Exit]: Returns 0: Function took 74ms
20090218 135516.059 EVENT (CONNECTION) EXECUTECOMPLETE:  TaskID=0  handle=0  status=1
20090218 135536.059 EVENT (RECORDSET) FETCHCOMPLETE:  TaskID=0  handle=0  status=1
20090218 135536.060 Marking recordset 0 STATE_OPEN
```

The actual query took quite a long time since it selected all records from the billing table which contained about 9000 records.
Notice that the adoRSetQuery() triggers two events. First the **ExecuteComplete** event is triggered first after about 23 seconds (which is ignored because this is an adoRSetQuery() call), then the **FetchComplete** is triggered 20 seconds later which is the one we are actually interested in and which wakes up the calling task.

An understanding of the trace output will be helpful in identifying potential performance issues and designing appropriate mechanisms to ensure that they don't effect the running of the system. Be aware that different providers function in different and unpredictable ways and it might be necessary to experiment with the ADO library function calls to understand the behavior fully,,

-o-

Private and Public Objects

Both **connection** and **recordset** objects can be created either as private or public objects. If a

task creates a private object then the created object is only accessible from that task. If any other task attempts to access the object then the function will generate an error.

The connection and recordset objects are created with the following functions:

conHandle=adoConnection([name[,priv_or_pub]]);
setHandle=adoRecordSet(conHandle[,name[,priv_or_pub]]);

If a task ends by encountering a stop or restart statement, or by chaining to another task, or by reaching the end of the program statements, or by an explicit kill command, thenany private objects that the task has created will be closed and released.

Public objects created by a task can be accessed by any other task, and if the creating task ends the object is not closed or released.

It is a good idea to give names to public objects so that other tasks can obtain the handle to that object through the adoConnGetHandle(name) or adoRSetGetHandle(name) functions.

There are two constants defined in the ado.inc file for the private or public types which can be used in the above functions as follows:

```
## Private and public type constants
const TYPE_PRIVATE       = 0;
const TYPE_PUBLIC        = 1;
```

-o-

Error Codes

The CXADO.DLL library functions typically return one of the internal error values as defined in ADO.INC. These are defined as follows:

```
const ADOERR_NRARGS    ="-1";       # Bad number of arguments passed to the function
const ADOERR_BADPARM    ="-2";      # One of the arguments passed to the function was invalid
const ADOERR_TOOMANY    ="-3";      # Exceeded the maximum number of concurrently open
connection or recordset handles
const ADOERR_ALLOC     ="-4";       # Memory allocation error (out of memory)
const ADOERR_NAMEDUP    ="-5";      # Name has already been used for connection or recordset
const ADOERR_TOOLONG    ="-6";      # Query/command/connection string exceeded the limit of
2047 characters
const ADOERR_CMDCLASH   ="-7";      # Attempt to execute a command whilst another asyncronous
command is in progress
const ADOERR_NOTOWNER   ="-8";      # Attempt to access a private connection or recordset handle
const ADOERR_INVLDNAME ="-9";       # Named connection or recordset not found
const ADOERR_BADHANDLE ="-10";      # Invalid connection or recordset handle
const ADOERR_NOTREADY  ="-11";      #  Attempt to execute a function on an unopened connection or
recordset
const ADOERR_COMERR    ="-12";      #  An ADO error occurred access adoLastError() to obtain error
value
```

If the function returns the code ADOERR_COMERR then the underlying ADO error value is stored in the task structure for the task that made the original call and can be obtained through the adoLastError() function.

adErrBoundToCommand	3707 -2146824581 0x800A0E7B	Cannot change the ActiveConnection property of a Recordset object that has a Command object as its source.
adErrCannotComplete	3732 -2146824556 0x800A0E94	Server cannot complete the operation.
adErrCantChangeConnection	3748 -2146824540 0x800A0EA4	Connection was denied. New connection you requested has different characteristics than the one already being used.
adErrCantChangeProvider	3220 -2146825068 0X800A0C94	Supplied provider differs from the one already being used.
adErrCantConvertvalue	3724 -2146824564 0x800A0E8C	Data value cannot be converted for reasons other than sign mismatch or data overflow. For example, conversion would have truncated data.
adErrCantCreate	3725 -2146824563 0x800A0E8D	Data value cannot be set or retrieved because the field data type was unknown, or the provider had insufficient resources to perform the operation.
adErrCatalogNotSet	3747 -2146824541 0x800A0EA3	Operation requires a valid ParentCatalog.
adErrColumnNotOnThisRow	3726 -2146824562 0x800A0E8E	Record does not contain this field.
adErrDataConversion	3421 -2146824867 0x800A0D5D	Application uses a value of the wrong type for the current operation.
adErrDataOverflow	3721 -2146824567 0x800A0E89	Data value is too large to be represented by the field data type.
adErrDelResOutOfScope	3738 -2146824550 0x800A0E9A	URL of the object to be deleted is outside the scope of the current record.
adErrDenyNotSupported	3750 -2146824538 0x800A0EA6	Provider does not support sharing restrictions.
adErrDenyTypeNotSupported	3751 -2146824537 0x800A0EA7	Provider does not support the requested kind of sharing restriction.
adErrFeatureNotAvailable	3251 -2146825037 0x800A0CB3	Object or provider is not able to perform requested operation.
adErrFieldsUpdateFailed	3749 -2146824539 0x800A0EA5	Fields update failed. For more information, examine the Status property of individual field objects.
adErrIllegalOperation	3219 -2146825069 0x800A0C93	Operation is not allowed in this context.
adErrIntegrityViolation	3719 -2146824569 0x800A0E87	Data value conflicts with the integrity constraints of the field.
adErrInTransaction	3246 -2146825042 0x800A0CAE	Connection object cannot be explicitly closed while in a transaction.
adErrInvalidArgument	3001 -2146825287 0x800A0BB9	Arguments are of the wrong type, are out of acceptable range, or are in conflict with one another.
adErrInvalidConnection	3709 -2146824579 0x800A0E7D	The connection cannot be used to perform this operation. It is either closed or invalid in this context.
adErrInvalidParamInfo	3708 -2146824580 0x800A0E7C	Parameter object is incorrectly defined. Inconsistent or incomplete information was provided.
adErrInvalidTransaction	3714 -2146824574 0x800A0E82	Coordinating transaction is invalid or has not started.
adErrInvalidURL	3729 -2146824559 0x800A0E91	URL contains invalid characters. Make sure that the URL is typed correctly.
adErrItemNotFound	3265 -2146825023 0x800A0CC1	Item cannot be found in the collection that corresponds to the requested name or ordinal.
adErrNoCurrentRecord	3021 -2146825267 0x800A0BCD	Either BOF or EOF is True, or the current record has been deleted. Requested operation requires a current record.
adErrNotExecuting	3715 -2146824573 0x800A0E83	Operation cannot be performed while not executing.

adErrNotReentrant	3710 -2146824578 0x800A0E7E	Operation cannot be performed while processing event.
adErrObjectClosed	3704 -2146824584 0x800A0E78	Operation is not allowed when the object is closed.
adErrObjectInCollection	3367 -2146824921 0x800A0D27	Object is already in collection. Cannot append.
adErrObjectNotSet	3420 -2146824868 0x800A0D5C	Object is no longer valid.
adErrObjectOpen	3705 -2146824583 0x800A0E79	Operation is not allowed when the object is open.
adErrOpeningFile	3002 -2146825286 0x800A0BBA	File could not be opened.
adErrOperationCancelled	3712 -2146824576 0x800A0E80	Operation has been canceled by the user.
adErrOutOfSpace	3734 -2146824554 0x800A0E96	Operation cannot be performed. Provider cannot obtain enough storage space.
adErrPermissionDenied	3720 -2146824568 0x800A0E88	Insufficient permission prevents writing to the field.
adErrProviderFailed	3000 -2146825288 0x800A0BB8	Provider did not perform the requested operation.
adErrProviderNotFound	3706 -2146824582 0x800A0E7A	Provider cannot be found. It may not be correctly installed.
adErrReadFile	3003 -2146825285 0x800A0BBB	File could not be read.
adErrResourceExists	3731 -2146824557 0x800A0E93	Copy operation cannot be performed. Object named by destination URL already exists. Specify adCopyOverwrite to replace the object.
adErrResourceLocked	3730 -2146824558 0x800A0E92	Object represented by the specified URL is locked by one or more other processes. Wait until the process has finished and try the operation again.
adErrResourceOutOfScope	3735 -2146824553 0x800A0E97	Source or destination URL is outside the scope of the current record.
adErrSchemaViolation	3722 -2146824566 0x800A0E8A	Data value conflicts with the data type or constraints of the field.
adErrSignMismatch	3723 -2146824565 0x800A0E8B	Conversion failed because the data value was signed and the field data type used by the provider was unsigned.
adErrStillConnecting	3713 -2146824575 0x800A0E81	Operation cannot be performed while connecting asynchronously.
adErrStillExecuting	3711 -2146824577 0x800A0E7F	Operation cannot be performed while executing asynchronously.
adErrTreePermissionDenied	3728 -2146824560 0x800A0E90	Permissions are insufficient to access tree or subtree.
adErrUnavailable	3736 -2146824552 0x800A0E98	Operation did not complete and the status is unavailable. The field may be unavailable or the operation was not attempted.
adErrUnsafeOperation	3716 -2146824572 0x800A0E84	Safety settings on this computer prevent accessing a data source on another domain.
adErrURLDoesNotExist	3727 -2146824561 0x800A0E8F	Either the source URL or the parent of the destination URL does not exist.
adErrURLNamedRowDoesNotExist	3737 -2146824551 0x800A0E99	Record named by this URL does not exist.
adErrVolumeNotFound	3733 -2146824555 0x800A0E95	Provider cannot locate the storage device indicated by the URL. Make sure that the URL is typed correctly.
adErrWriteFile	3004 -2146825284 0x800A0BBC	Write to file failed.
adWrnSecurityDialog	3717 -2146824571 0x800A0E85	For internal use only. Do not use.
adWrnSecurityDialogHeader	3718 -2146824570 0x800A0E86	For internal use only. Do not use.

-o-

ADO Library Function Quick Reference

// Library level functions
adoTrace(Tracelevel)
adoErrVerbose(Errlevel)

// Task level functions
adoBlockMode(mode)
adoLastError()
adoBusyState()

// Connection functions
adoConnection(name, prv_or_pub)
adoConnOpen(handle,UID,PWD,conn_str1[,conn_str2...])
adoConnParmGet(handle,parmID,&Value)
adoConnParmSet(handle,parmID,Value)
adoConnClose(handle)
adoConnGetHandle(ConnName)
adoConnTransBegin(handle)
adoConnTransCommit(handle)
adoConnTransCancel(handle)
adoConnState(SetHandle)

// Recordset functions
adoRecordSet(name,conHandle,prvpub_flag)
adoRSetQuery(RShandle,cursor_type,lock_type,options,query_str1[,query_str2...])
adoRSetCmd(RShandle,cursor_type,lock_type,options,query_str1[,query_str2...])
adoRSetParmSet(handle,parmID,Value)
adoRSetParmGet(handle,parmID,&Value)
adoRSetClose(handle)
adoRSetGetHandle(RsetName)
adoRSetRecCount(handle)
adoRSetMove(handle,numrec[,start])
adoRSetMoveFirst(handle)
adoRSetMoveLast(handle)
adoRSetMoveNext(handle)
adoRSetMovePrev(handle)
adoRSetAddNew(handle)
adoRSetUpdate(handle)
adoRSetCancelUpd(handle)
adoRSetUpdBatch(handle,affected)
adoRSetCancelBatch(handle,affected)
adoRSetDelete(handle[,affected])
adoRSetState(SetHandle)
adoRSetIsBOF(SetHandle)
adoRSetIsEOF(SetHandle)

// Field level functions
adoFldCount(handle)
adoFldGetName(handle,ix)
adoFldGetValue(handle,ix/name,&Value)
adoFldSetValue(handle,ix/name,Value)
adoFldParmGet(handle,ix/name,ParmID,pValue)
adoFldParmSet(handle,ix/name,ParmID,Value)

-o-

ADO Function Reference

adoTrace

Synopsis:
adoTrace(*TraceLevel*)

Arguments:
TraceLevel - 0=off, 1=trace on

Description: This function switches on or off the tracing of ADO function calls and events.

Returns: 0

-o-

adoErrVerbose

Synopsis:
adoErrVerbose(*Errlevel*)
Arguments:
ErrLevel - 0 to turn off verbose error messages, 1 to switch on verbose error messages (default)

Description: Verbose error messages also show the full function name and all of the aguments passed to an ADO function as part of the error message. With verbose error messages there will be two lines of error message in the log for every error that is generated..

Returns : 0

-o-

adoBlockMode

Synopsis:
adoBlockMode(*block_mode*)
Arguments:
block_mode - Set to 0 for blocking, non-zero for non-blocking mode

Description: This function affects the adoConnOpen(), adoRSetQuery() and adoRSetCmd() functions and defines wheher these functions will block waiting until the asynchronous function completes (blocking mode) , or whether the function return control back to the calling task immediately.

In non-blocking mode it is up to the calling application to ensure that the asyncronous function has completed before attempting any other ado function that attempts to access the object.
There are certain exceptions to this such as the adoConnParmGet(), adoConnParmSet(), adoRSetParmGet(), adoRSetParmSet() which can be called at any time after the object handles have been created.

Returns: 0

-o-

adoLastError

Synopsis:
adoLastError()
Arguments:
NONE

Description: If an ado function results in the error ADOERR_COMERR being returned, then the underlying ADO error code will be stored in the task structure and can be retreived by this function.

Returns: Returns the last ADO error code received by the calling task (see Error Codes).

-o-

adoBusyState

Synopsis:
adoBusyState()
Arguments:
NONE

Description: Returns 1 if the current task is executing an asynchronous function such as adoConnOpen(), adoRSetQuery(), adoRSetCmd(). This can be used by a task in non-blocking mode to poll for the completion of an asynchronous function. Once the asyncronous function has completed then adoBusyState() function will return 0. Note that this will only return the status of the asyncronous functions initiated by the calling task. If another task has started the asynchronous function then adoConnState() or adoRSetState() should be used instead to determine

when the function has completed.

Returns: 1 if there is an asyncronous function still running that was initiated by this task, 0 if there are no asynchronous functions running that we started by this task.

-o-

adoConnection

Synopsis:
handle=adoConnection([name, prv_or_pub])
Arguments:
name - An optional name that will uniquely identify the connect object (mainly for public connections)
prv_or_pub - Set to 0 for Private connection, 1 for public connection (optional - default private)

Description: This function creates a Connection object and returns a unique handle. The connection object can be opened in private or public mode and can be given an optional name which can be used to retreive the handle using the adoConnGetHandle(name) function.

Important Note: Upon creating the underlying _Connection object the ***CusorLocation*** property is hard-coded to to ***adUseClient***. This is necessary in order for the libary to work in asyncronous (non-blocking) mode. If you attempt to change the ***CusorLocation*** to adUseServer in the adoConnParmSet() function then this could cause some of the ado functions to work syncronously, thus blocking the Telecom Engine task scheduler and seriously effecting performance. If you need to use server side cursors for any reason then it is suggested that you carefully trace all function calls to check that they don't block for significant periods of time (see Performance and blocking calls) and, if necesary, consider using a client-server model based on the CXSOCKETS.DLL library instead.

A private **connection** can only be accessed from the task that created it, and any attempt to access a **connection** from another task will result in an error. When a task ends (either through a chain to another task, a stop or restart statement is encountered, the end of program is reached or the task is explicit killed), then all private **connections** that the task created will be closed and released. When a connection is closed then all **recordsets** associated with that **connection** will also be closed and released.

Public connections can be accessed from any task once it has been opened. If the task that created it is ended for any reason then the connection will not be closed. Often it is useful to make a single public connection to a datastore upon start-up, then all other tasks can create and open **recordsets** on that single connection.

The maximum number of simultaneous connections that can be open at any one time 1024 (ie there are 1024 handles available).

If a name is provided for a public connection then other tasks can obtain the connection handle by using adoConGetHandle(name). All names must be unique, although a blank name can be given for multiple public connections, however then the handle cannot be retreived using adoConnGetHandle(name) and must be obtained another way (E.g. global variable). Giving a name to a private connection is of little use since the handle will only be retrievable from the task that called the function (which should already have obtained the handle from the return value.

Both options can be omitted, in which case a private, nameless connection will be created.

Returns: Returns the connection object handle or a negative error code.

-o-

adoConnOpen

Synopsis:
adoConnOpen(handle,UID,PWD,conn_str1[,conn_str2...])

Arguments:

handle - The connection handle.
UID - The user ID (can be left blank if it appears in the connection string)
PWD - The password (can be left blank if it appears in the connection string)
conn_str1 - The first part of the connction string
[conn_str2..] - *(optional)* The remainder of the connection string across the remaining arguments.

Description: This function attempts to open a connection on the given *handle* using the user id (*UID*), password (*PWD*), and connection string (*conn_str1, con_str2....*) provided.

Note that this function will block the calling task, unless block mode is switched off using adoblockMode(), and will remain blocked until the connection completion event is received, after which the task is woken up again with the appropriate return code. Please make sure you are aware of performace issues that may arise with this function (see Performance and blocking calls).

The *UID* and *PWD* are optional arguments that can specify the login user ID and password for the data source. Alternativly these can be specified driectly inside the connection string.

The remaining arguments are concatenated into a single connection string (but must not exceed 2047 characters). For example in the following connection string is made up from a number of arguements that have been concatenated together in this way:

```
userID="postgres"
Passwd="admin"
Datasource="MyDSN";
Provider="MSDASQL.1"

    x=adoConnOpen(handle,"","", "Provider=",Provider,"; User ID=",userID,"; Password=",Passwd,"; Data
Source=",Datasource);
   etc.
```

This is equivalent to :

```
        x=adoConnOpen(handle,"","2,"Provider=MSDASQL.1;Password=admin;User ID=postgres;Data
Source=MyDSN");
```

Which is also equivalent to:

```
userID="postgres"
```

```
Passwd="admin"

        x=adoConnOpen(handle,userID,Passwd,  "Provider=MSDASQL.1;Data  Source=MyDSN");
Source=",Datasource);
```

The connection string that is used will depend upon the data source provider and will differ from source to source. Below are some examples of connection strings for different kinds of data sources:

Oracle using Microsoft OLE provider:

```
"Provider=msdaora;Data Source=MyOracleDB;UserId=myUsername;Password=myPassword;"
```

Microsoft SQL server native client 10.0 OLE DB provider:

```
"Provider=SQLNCLI10;Server=myServerAddress;Database=myDataBase;Uid=myUsername;
Pwd=myPassword;"
```

Microsoft Access Database using ACE OLE DB 12.0:

```
"Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\myFolder\myAccess2007file.accdb;Jet
OLEDB:Database Password=MyDbPassword";
```

A comprehensive list of connection strings for different data sources can be found here:

http://www.connectionstrings.com

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoConnParmGet

Synopsis:

adoConnParmGet(handle,parmID,&pValue)

Arguments:

handle - The connection handle.
parmID - The parameter ID
pValue - Pointer to the variable that will hold the parameter value

Description: This function enables the underlying connection parameters to be examined for the connection defined by *handle*. The parmID argument is the ID of the parameter that is to be read and the *pValue* is a pointer to a variable that will hold the returned parameter value. The parmID should be set to one of the following values as defined in the **ado.inc** file supplied with the library as follows:

```
################# CONNECTION PARAMETERS ###############
const P_CON_ATTRIBUTES     =101;    # Sets or returns the attributes of a Connection object
const P_CON_COMMANDTIMEOUT    =102;    # Sets or returns the number of seconds to wait while
attempting to execute a command
const P_CON_CONNECTIONSTRING  =103;    # Sets or returns the details used to create a connection
to a data source
```

```
const P_CON_CONNECTIONTIMEOUT=104;       # Sets or returns the number of seconds to wait for a
connection to open
const P_CON_CURSORLOCATION   =105;      # Sets or returns the location of the cursor service
const P_CON_DEFAULTDATABASE  =106;      # Sets or returns the default database name
const P_CON_ISOLATIONLEVEL  =107;      # Sets or returns the isolation level
const P_CON_MODE            =108;      # Sets or returns the provider access permission
const P_CON_PROVIDER         =109;      # Sets or returns the provider name
const P_CON_STATE           =110;      # Returns a value describing if the connection is open or closed
const P_CON_VERSION          =111;      # Returns the ADO version number
```

These constants are mapped to the equiivalent properties in the underlying _Connection object as follows:

Property	Readable	Writeable	Description
Attributes			Sets or returns the attributes of a Connection object
CommandTimeout			Sets or returns the number of seconds to wait while attempting to execute a command
ConnectionString			Sets or returns the details used to create a connection to a data source
ConnectionTimeout			Sets or returns the number of seconds to wait for a connection to open
CursorLocation			Sets or returns the location of the cursor service
DefaultDatabase			Sets or returns the default database name
IsolationLevel			Sets or returns the isolation level
Mode			Sets or returns the provider access permission
Provider			Sets or returns the provider name
State			Returns a value describing if the connection is open or closed
Version			Returns the ADO version number

For those parameters that are restricted to a set of enumerated values, then the equivalent constants have been defined in the ***ado.inc*** include file for those enumerations.   For example, the LockType property of the _Connection property can take on one of the following enumerated values:

adLockBatchOptimistic	4	Indicates optimistic batch updates. Required for batch update mode.
adLockOptimistic	3	Indicates optimistic locking, record by record. The provider uses optimistic locking, locking records only when you call the Update method.
adLockPessimistic	2	Indicates pessimistic locking, record by record. The provider does what is necessary to ensure successful editing of the records, usually by locking records at the data source immediately after editing.
adLockReadOnly	1	Indicates read-only records. You cannot alter the data.
adLockUnspecified	-1	Does not specify a type of lock. For clones, the clone is created with the same lock type as the original.

The equivalent constants for these enumerated values are defined in ado.inc as follows:

```
# Lock type constants
const   adLockUnspecified = "-1";
const   adLockReadOnly= 1;
const   adLockPessimistic = 2;
const   adLockOptimistic = 3;
const   adLockBatchOptimistic = 4;
```

[See the Microsoft ADO Reference Libary for more details].

Returns: Returns 0 upon success or a negative error code.   If ADOERR_COMERR is returned,

then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoConnParmSet

Synopsis:
adoConnParmSet(handle,parmID,Value)
Arguments:
handle - The connection handle.
parmID - The parameter ID to set
Value- The value to set the parameter to

Description: This function enables the underlying connection parameters to be set for the connection defined by *handle*. The parmID argument is the ID of the parameter that is to be set (see adoConnParmGet()) and the *Value* is the value to set it to. Note that this function can be used at any time after the **connection** object has been created by a call to adoConnection() and can be used to set up the properties of the **connection** prior to calling the adoConnOpen() function call.

Important Note: Upon creating the underlying _Connection object the ***CusorLocation*** property is hard-coded to to ***adUseClient***. This is necessary in order for the libary to work in asyncronous (non-blocking) mode. If you attempt to change the ***CusorLocation*** to ***adUseServer*** in the adoConnParmSet() function then this could cause some of the ado functions to work syncronously, thus blocking the Telecom Engine task scheduler and seriously effecting performance. If you need to use server side cursors for any reason then it is suggested that you carefully trace all function calls to check that they don't block for significant periods of time (see Performance and blocking calls) and, if necesary, consider using a client-server model based on the CXSOCKETS.DLL library instead.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoConnClose

Synopsis:
adoConnClose(handle)
Arguments:
handle - The connection handle.

Description: This function closes the **connection** specified by *handle* (if it is open) and then releases the underlying ADO _Connection object. Note that a task can only close public connections or those private connections that it opened itself. When a connection is closed then all underlying **recordsets** that rely on this connection will also be closed and released. If any other tasks are using the connection or any recordsets dependant on this connection then any subsequent calls attempting to use these objects will result in an error. Careful co-ordination should be carried out between tasks if a public connection is closed since another adoConnOpen() might result in the same handle being allocated, then any tasks that were using the original handle will suddenly have access to the new connectio data instead, with possible unexpected results.

Note that when a task stops through an explicit kill request, or a stop or restart command, or by reaching the end of the program, or by 'chaining' to another program then all open private connections and recordsets will be automatically closed and released.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoConnGetHandle

Synopsis:
handle=adoConnGetHandle(ConnName)
Arguments:
ConnName - The connection connection name.

Description: If a connection is opened in public mode and is given a unque name (see adoConnection()), then this function enables other tasks to obtain the connection handle by specifying this unque name as the argument. For example if task 1 opens a public, named connection like this:

```
adoConnection("MyConn",TYPE_PRIVATE);
```

Another task can obtain the handle to this public connection through:

```
handle=adoConnGetHandle("MyConn");
```

Returns: Returns the connection handle associated with the given name upon success or ADOERR_INVLDNAME if the name was not found.

-o-

adoConnState

Synopsis:
state=adoConnState(handle)
Arguments:
handle - The connection handle.

Description: This function returns the underlying state of the ADO _Connection object and can be used to determine when an asynchronous function has completed (Especially useful in non-blocking mode (see adoBlockMode()).

The state returned will be one of the enumerated state values as defined in the **ado.inc** file as follows:

Constant	Value	Description
adStateClosed	0	The object is closed
adStateOpen	1	The object is open

adStateConnecting	2	The object is connecting
adStateExecuting	4	The object is executing a command
adStateFetching	8	The rows of the object are being retrieved

Returns: Returns the underlying ***state*** property of the ADO _Connection object or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoConnTransBegin

Synopsis:
adoConnTransBegin(handle)
Arguments:
handle - The connection handle.

Description: This function starts a transaction on the connection specified by *handle*. After you call the adoConnTransBegin() method, the provider will no longer instantaneously commit changes you make until you call adoConnTransCommit() or adoConnTransCancel() to end the transaction.

For providers that support nested transactions, calling the adoConnTransBegin() method within an open transaction starts a new, nested transaction. The return value indicates the level of nesting: a return value of "1" indicates you have opened a top-level transaction (that is, the transaction is not nested within another transaction), "2" indicates that you have opened a second-level transaction (a transaction nested within a top-level transaction), and so forth. Calling adoConnTransCommit() or adoConnTransCancel() affects only the most recently opened transaction; you must close or roll back the current transaction before you can resolve any higher-level transactions.

Calling the adoConnTransCommit() method saves changes made within an open transaction on the connection and ends the transaction. Calling the adoConnTransCancel() method reverses any changes made within an open transaction and ends the transaction. Calling either method when there is no open transaction generates an error.

Depending on the **Connection** object's **Attributes** property (see adoConnParmGet()), calling either the adoConnTransCommit() or adoConnTransCancel() methods may automatically start a new transaction. If the **Attributes** property is set to **adXactCommitRetaining**, the provider automatically starts a new transaction after a adoConnTransCommit() call. If the **Attributes** property is set to **adXactAbortRetaining**, the provider automatically starts a new transaction after a adoConnTransCancel() call.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoConnTransCommit

Synopsis:
adoConnTransCommit(handle)
Arguments:
handle - The connection handle.

Description: This function saves changes made within an open transaction on the connection and ends the transaction. All changes made from the time the adoConnTransBegin() function call is made to the time adoConnTransCommit() is made will be written to the database. See adoConnTransBegin() for a full description of ADO transactions..

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoConnTransCancel

Synopsis:
adoConnTransCancel(handle)
Arguments:
handle - The connection handle.

Description: This function cancels (or Rolls back) a transaction on the connection specified by *handle*. Any changes that have been made to any recordsets associated with this connection between the adoConnTransBegin() and the adoConnTransCancel() function call will be lost.

For a full description of ADO transactions please see adoConnTransBegin().

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRecordSet

Synopsis:
adoRecordset(name,conHandle,prvpub_flag)
Arguments:
conHandle **-** The connection handle of the connection object to associate with this recordset.
name - An optional name that will uniquely identify the recordset object (mainly for public recordsets)
prv_or_pub - Set to 0 for Private recordset, 1 for public recordset(optional - default private)

Description: This function creates a **recordset** object on the **connection** specified by *conHandle* and returns a unique recordset handle to the created object. The **recordset** object can be opened in private or public mode and can be given an optional name, which can be used by other tasks to retrieve the handle using the adoRSetGetHandle(name) function.

A private **recordset** can only be accessed from the task that created it, and any attempt to access a recordset from another task will result in an error. When a task ends (either through a chain to another task, a stop or restart statement is encountered, the end of program is reached or the task is explicit killed), then all private recordsets that the stopped task created will be closed and released. If the **connection** with which the **recordset** is associated with is closed then the created **recordset** will also be automatically closed and released along with it.

Public recordsets can be accessed from any task once it has been created. If the task that created it is stopped for any reason then the public **recordset** will not be closed and released (it must be done explicitly).

There are two constants defined in the **ado.inc** file as follows:

```
const TYPE_PRIVATE=0;
const TYPE_PUBLIC=1;
```

The maximum number of simultaneous recordsets that can be open at any one time 2048 (ie there are 2048 recordset handles available).

If a name is provided for a public **recordset** then other tasks can obtain the recordset handle by using adoRSetGetHandle(name). All names must be unique, although a blank name can be given for multiple public connections, however then the handle cannot be retreived using adoConnGetHandle(name) and must be obtained another way (E.g. global variable). Giving a name to a private recordset is of little use since the handle will only be retrievable from the task that called the function (which should already have obtained the handle from the return value).

Both *name* and *priv_or_pub* arguments can be omitted, in which case a private, nameless recordset will be created.

Returns: Returns the connection object handle or a negative error code.

-o-

adoRSetQuery

Synopsis:
adoRSetQuery(handle,cursor_type,lock_type,options,query_str1[,query_str2...])
Arguments:
handle - The recordset handle.
cursor_type - The type of cursor
lock_type - The locking type
options - Additional options
query_str1 - The query string
[query_str2..] - optional additional text that will be concatenated to make the complete query string..

Description: This function executes a query on the **resordset** specified by the *handle* argument. The *cursor_type* argument can be set to one of the following constants (as defined in **ado.inc**):

Constant	Value	Description
adOpenUnspecified	-1	Does not specify the type of cursor.
adOpenForwardOnly	0	Default. Uses a forward-only cursor. Identical to a static cursor, except that you can only scroll forward through records. This improves performance when you need to make only one pass through a Recordset.
adOpenKeyset	1	Uses a keyset cursor. Like a dynamic cursor, except that you can't see records that other users add, although records that other users delete are inaccessible from your Recordset. Data changes by other users are

		still visible.
adOpenDynamic	2	Uses a dynamic cursor. Additions, changes, and deletions by other users are visible, and all types of movement through the Recordset are allowed, except for bookmarks, if the provider doesn't support them.
adOpenStatic	3	Uses a static cursor. A static copy of a set of records that you can use to find data or generate reports. Additions, changes, or deletions by other users are not visible.

The *lock_type* argument can be set to one of the following constants (as define in **ado.inc**):

Constant	Value	Description
adLockUnspecified	-1	Unspecified type of lock.
adLockReadOnly	1	Read-only records
adLockPessimistic	2	Pessimistic locking, record by record. The provider lock records immediately after editing
adLockOptimistic	3	Optimistic locking, record by record. The provider lock records only when calling update
adLockBatchOptimistic	4	Optimistic batch updates. Required for batch update mode

If **adLockBatchOptimistic** is specified then the recordset is opened in batch mode which enables the use of the adoRSetUpdBatch() and adoRSetCancelBatch() functions to be used (if supported by the data provider).

The options argument allows additional options to be specified and can be one or more of the **CommandTypeEnum** values or the **ExecuteOptionEnum** values shown below (these constants are defined in the **ado.inc** file) . The adoRSetQuery() function will always add the options: **adoAsyncExecute**, **adoAsyncFetch** and **adoAsyncFetchNonBlocking** options, and this functionality cannot be turned off. Care should be taken when adding some of these options as it may effect the way the library reacts to ADO events. for example, setting the adoExecuteNoRecords option will probably stop the **FetchComplete** event being triggered so in blocking mode the adoRSetQuery() function call will block forever and never return.

CommandTypeEnum values:

Constant	Value	Description
adCmdUnspecified	-1	Unspecified type of command
adCmdText	1	Evaluates CommandText as a textual definition of a command or stored procedure call
adCmdTable	2	Evaluates CommandText as a table name whose columns are returned by an SQL query
adCmdStoredProc	4	Evaluates CommandText as a stored procedure name
adCmdUnknown	8	Default. Unknown type of command
adCmdFile	256	Evaluates CommandText as the file name of a persistently stored Recordset. Used with Recordset.Open or Requery only.
adCmdTableDirect	512	Evaluates CommandText as a table name whose columns are all returned. Used with Recordset.Open or Requery only. To use the Seek method, the Recordset must be opened with adCmdTableDirect. Cannot be combined with the ExecuteOptionEnum value adAsyncExecute.

ExecuteOptionEnum values.

adOptionUnspecified	-1	Unspecified command
adAsyncExecute	16	The command should execute asynchronously. Cannot be combined with the CommandTypeEnum value adCmdTableDirect
adAsyncFetch	32	The remaining rows after the initial quantity specified in the CacheSize property should be retrieved asynchronously
adAsyncFetchNonBlocking	64	The main thread never blocks while retrieving. If the requested row has not been retrieved, the current row automatically moves to the end of the file. If you open a Recordset from a Stream containing a persistently stored Recordset, adAsyncFetchNonBlocking will not have an effect; the operation will be synchronous and blocking. adAsynchFetchNonBlocking has no effect when the adCmdTableDirect option is used to open the Recordset
adExecuteNoRecords	128	The command text is a command or stored procedure that does not return rows. If any rows are retrieved, they are discarded and not returned. adExecuteNoRecords can only be passed as an optional parameter to the Command or Connection Execute method
adExecuteStream	256	The results of a command execution should be returned as a stream. adExecuteStream can only be passed as an optional parameter to the Command Execute method
adExecuteRecord	512	The CommandText is a command or stored procedure that returns a single row which should be returned as a Record object

The SQL query string is specified in one or more of the the remaining arguments (*query_str1, query_str2 ...*), which will be concatenated together into one complete string. The concatenated query string must not exceed 2047 characters. For example in the following query string is made up from a number of arguments that have been concatenated together in this way:

```
acc_no="129934";
x=adoRSetQuery(handle,adOpenStatic,adLockReadOnly,-1,"select * from customers where acc_no=",acc_no);
```

which is equivalent to the following:

```
x=adoRSetQuery(handle,adOpenStatic,adLockReadOnly,-1,"select * from customers where acc_no=129934");
```

Remember that the maximum number of arguments that can be passed to a DLL library function in the Telecom Engine is 32.

Note that this function relies on the **FetchComplete** event being triggered in blocking mode in order to wake up the calling task. If for some reason the data provider does not cause the **FetchComplete** task to trigger then use the adoRSetCmd() function instead, which is identical except for it relies on the **ExecuteComplete** event instead.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetCmd

Synopsis:
adoRSetCmd(handle,cursor_type,lock_type,options,query_str1[,query_str2...])
Arguments:
handle - The recordset handle.
cursor_type - The type of cursor
lock_type - The locking type
options - Additional options
query_str1 - The query string
[query_str2..] - optional additional text that will be concatenated to make the complete query string..

Description: This function executes a query on the **resordset** specified by the *handle* argument. This function is identical to the adoRSetQuery() function except that it waits for the **ExecuteComplete** event to trigger before waking up the calling task, rather than the **FetchComplete** event used by the adoRSetQuery() function. The adoRSetCmd() function is usually used for executing queries that do not return any rows of data whereas the adoRSetQuery() function is used when the query will return one or more rows of data.

For a full description of the *cursor_type*, *lock_type* and *options* arguments, see adoRSetQuery().

The actual query string is made up by concatenating all of the remaining arguments (*query_str1, query_str2....*), but the concatenated query string must not exceed 2047 characters. For example in the following query string is made up from a number of arguments that have been concatenated together in this way:

```
table_name="customers";
x=adoRSetCmd(handle,adOpenStatic,adLockReadOnly,"create table ",table_name," (name Text, acc_no Integer)");
```

which is equivalent to the following:

```
x=adoRSetCmd(handle,adOpenStatic,adLockReadOnly,-1,"create table customers (name Text, acc_no Integer)");
```

Remember that the maximum number of arguments that can be passed to a DLL library function in the Telecom Engine is 32.

Note that in blocking mode this function relies on the **ExecuteComplete** event being triggered in blocking mode in order to wake up the calling task.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetResync

Synopsis:
adoRSetResync(handle[,AffectRecord[,ResyncValues]])
Arguments:
handle - The recordset handle.
AffectRecord - Optional flag to specify which records are effected

ResyncValues - Option specifying if underlying values are updated

Description: This function refreshes the data in the **recordset** specified by *handle*. It does not re-execute the query on the recordset therefore new records in the underlying database will not be visible. If the attempt to resynchronize fails because of a conflict with the underlying data (for example, a record has been deleted by another user), the provider returns warnings to the **Errors** collection and a run-time error occurs.

The optional *AffectRecord* argument defined which records are affected by the resynchronization (default adAffectAll) and can be set to one of the following values as defined in the **ado.inc**:

```
const adAffectCurrent     = 1;    # just current record
const adAffectGroup            = 2;    # those made visible by filter
const adAffectAll         = 3;    # all records
const adAffectAllChapters      = 4;
```

The optional *RsyncValues* argument defines whether underlying values are overwritten. It can be set to one of the following values as specified in **ado.inc**:

```
const adResyncUnderlyingValues     =1;         # Does not overwrite data, and pending updates are not canceled
const adResyncAllValues       =2;         # Default. Overwrites data, and pending updates are canceled
```

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetRequery

Synopsis:

adoRSetRequery(handle,options)

Arguments:

handle - The recordset handle.
options - Additional options

Description: This function executes a requery on the **resordset** specified by the *handle* argument in order to update the underlying recordset data from the data source.

The options argument is a combination of the **CommandTypeEnum** and **ExecuteOptionEnum** values as described in adoRSetQuery(). The adoRSetRequery() function will always add the options: **adoAsyncExecute**, **adoAsyncFetch** and **adoAsyncFetchNonBlocking** options, and this functionality cannot be turned off.

Note that in blocking mode this function relies on the **FetchComplete** event being triggered in blocking mode in order to wake up the calling task.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetParmGet

Synopsis:

adoRSetParmGet(handle,parmID,&Value)

Arguments:

handle - The recrodset handle.
parmID - The parameter ID
pValue - Pointer to the variable that will hold the parameter value

Description: This function enables the underlying **recordset** parameters to be examined for the recordset defined by *handle*. The parmID argument is the ID of the parameter that is to be read and the *pValue* is a pointer to a variable that will hold the returned parameter value. The parmID should be set to one of the following values as defined in the **ado.inc** file supplied with the library as follows:

```
const P_RS_ABSOLUTEPAGE =1;   # Sets or returns a value that specifies the page number in the Recordset object
const P_RS_ABSOLUTEPOSITION=2;      #Sets or returns the ordinal position of the current record in the Recordset object
const P_RS_ACTIVECOMMAND     =3;   #Returns the Command object associated with the Recordset
const P_RS_ACTIVECONNECTION=4;     # Unreadable/unsettable from the cxado.dll library
const P_RS_BOF             =5;   #Returns true if the current record position is before the first record, otherwise false
const P_RS_BOOKMARK           =6;   #Sets or returns a bookmark. The bookmark saves the position of the current record
const P_RS_CACHESIZE          =7;   #Sets or returns the number of records that can be cached
const P_RS_CURSORLOCATION     =8;   #Sets or returns the location of the cursor service
const P_RS_CURSORTYPE    =9;   #Sets or returns the cursor type of a Recordset object
const P_RS_DATAMEMBER    =10;  #Sets or returns the name of the data member that will be retrieved
const P_RS_DATASOURCE    =11;  # Unreadable/unsettable from the cxado.dll library
const P_RS_EDITMODE      =12;  #Returns the editing status of the current record
const P_RS_EOF           =13;  #Returns true if the current record position is after the last record, otherwise false
const P_RS_FILTER        =14;  #Sets or returns a filter for the data in a Recordset object
const P_RS_INDEX         =15;  #Sets or returns the name of the current index for a Recordset object

const P_RS_LOCKTYPE      =16;  #Sets or returns the type of locking when editing a record in a Recordset
const P_RS_MARSHALOPTIONS     =17;  # Sets or returns a value that specifies which records are to be returned to the server
const P_RS_MAXRECORDS         =18;  # Sets or returns the maximum number of records to return to a Recordset object from a query
const P_RS_PAGECOUNT          =19;  # Returns the number of pages with data in a Recordset object

const P_RS_PAGESIZE      =20;  # Sets or returns the maximum number of records allowed on a single page of a Recordset object
const P_RS_RECORDCOUNT   =21;  # Returns the number of records in a Recordset object
const P_RS_SORT          =22;  # Sets or returns the field names in the Recordset to sort on
const P_RS_SOURCE        =23;  # Sets a string value or a Command object reference, or returns a String value that indicates the data source of the Recordset object
const P_RS_STATE         =24;  #Returns a value that describes if the Recordset object is open, closed, connecting, executing or retrieving data
const P_RS_STATUS        =25;  #Returns the status of the current record with regard to batch updates or other bulk operations
const P_RS_STAYINSYNC         =26;  # Sets or returns whether the reference to the child records will change when the parent record position changes
```

The above constants map one-to-one with the corresponding underlying properties from the ADO **_Recordset** object as shown below:

Property	Readable	Writeable	Description

AbsolutePage			Sets or returns a value that specifies the page number in the Recordset object
AbsolutePosition			Sets or returns a value that specifies the ordinal position of the current record in the Recordset object
ActiveCommand			Returns the Command object associated with the Recordset
ActiveConnection			Sets or returns a definition for a connection if the connection is closed, or the current Connection object if the connection is open
BOF			Returns true if the current record position is before the first record, otherwise false
Bookmark			Sets or returns a bookmark. The bookmark saves the position of the current record
CacheSize			Sets or returns the number of records that can be cached
CursorLocation			Sets or returns the location of the cursor service
CursorType			Sets or returns the cursor type of a Recordset object
DataMember			Sets or returns the name of the data member that will be retrieved from the object referenced by the DataSource property
DataSource			Specifies an object containing data to be represented as a Recordset object
EditMode			Returns the editing status of the current record
EOF			Returns true if the current record position is after the last record, otherwise false
Filter			Sets or returns a filter for the data in a Recordset object
Index			Sets or returns the name of the current index for a Recordset object
LockType			Sets or returns a value that specifies the type of locking when editing a record in a Recordset
MarshalOptions			Sets or returns a value that specifies which records are to be returned to the server
MaxRecords			Sets or returns the maximum number of records to return to a Recordset object from a query
PageCount			Returns the number of pages with data in a Recordset object
PageSize			Sets or returns the maximum number of records allowed on a single page of a Recordset object
RecordCount			Returns the number of records in a Recordset object
Sort			Sets or returns the field names in the Recordset to sort on
Source			Sets a string value or a Command object reference, or returns a String value that indicates the data source of the Recordset object
State			Returns a value that describes if the Recordset object is open, closed, connecting, executing or retrieving data
Status			Returns the status of the current record with regard to batch updates or other bulk operations
StayInSync			Sets or returns whether the reference to the child records will change when the parent record position changes

[See the Microsoft ADO Reference Libary for more details].

Note that some of the parameters are read-only and some of them can't be read or written at all by the CXADO.DLL libarary.

Also, for some of the parameters, there are utility functions which can provide another mechanism for getting the value of the parameter. specifically for the _Recordset parameters there are the following utility functions defined:

adoRSetIsEOF(handle)
adoRSetIsBOF(handle)
adoRSetState(handle)

Which allow for the equivalent parameters from the set above to be quickly read.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetParmSet

Synopsis:
adoRSetParmSet(handle,parmID,Value)
Arguments:
handle - The connection handle.
parmID - The parameter ID to set
Value- The value to set the parameter to

Description: This function enables the underlying **recordset** parameters to be set for the connection defined by *handle*. The parmID argument is the ID of the parameter that is to be set (see adoRSetParmGet()) and the *Value* is the value to set it to. Note that this function can be used at any time after the recordset object has been created by a call to adoRecordset() and can be used to set up the properties of the recordset prior to calling the adoRSetQuery() or adoRSetCmd() function call.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetClose

Synopsis:
adoRSetClose(handle)
Arguments:
handle - The recordset handle.

Description: This function closes the **recordset** specified by the *handle* argument and releases the underlying ADO _Recordset object.

Any attempt to use the *handle* in any other functions will result in an error.

A task cannot close a recordset that has been opened privately by another task, and any private recordsets that are still open when a task stops will be automatically closed and released.

Note that care should be taken when closing public recordsets because the recordset *handle* might be quickly reallocated to another recordset so any tasks making called to the original *handle* will then be referencing the wrong dataset, with unexpected results.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetGetHandle

Synopsis:
handle=adoRSetGetHandle(SetName)
Arguments:

SetName - The recordset name.

Description: If a **recordset** is opened in public mode and is given a unque name (see adoRecordset()), then this function enables other tasks to obtain the connection handle by specifying this unque name as the *SetName* argument. For example if a task opens a public, named recordset like this:

```
adoRecordSet("MySet",TYPE_PRIVATE);
```

Another task can obtain the handle to this public recordset through:

```
handle=adoRSetGetHandle("MySet");
```

Returns: Returns the **recordset** handle associated with the given name upon success or ADOERR_INVLDNAME if the name was not found.

-o-

adoRSetRecCount

Synopsis:
count=adoRSetRecCount(handle)
Arguments:
handle - The recordset handle.

Description: This function returns the number of records in the **rescordset** returned by a call to adoRSetQuery() or adoRSetCmd().

Returns: Returns the number of records in the recordset or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetMove

Synopsis:
adoRSetMove(handle,numrec[,start])
Arguments:
handle - The recordset handle.
numrecs - The number of records to move the record pointer (negative value moves backwards).
[start] - Optional flag to indicate the start position of the move (default is to move relative to current position).

Description: This function moves the record pointer in a recordset (specified by *handle*) a certain number of records as specified by *numrecs*, forward or backwards through the data relative to a start position specified by the optional *start* argument. If a positive value for *numrecs* is specified then the function will move the record pointer forward in the recordset, if a negative value is specified the the record pointer will attempt to move backwards in the recordset.

The optional *start* argument indicated the starting position, relative to which the record pointer will be moved. This can be set to a bookmark offset position (see adoRSetParmGet (P_RS_BOOKMARK)) or one of the following constants defined in **ado.inc** :

Constant	Value	Description
adBookmarkCurrent	0	Starts at the current record (default)
adBookmarkFirst	-1	Starts at the first record
adBookmarkLast	-2	Starts at the last record

Technical Note: the above constants differ from the underlying equivalent enumeration values used by ADO, as they have been made negative so that they can be distinguished from a valid bookmark value. In the underlying ADO library, the a VARIANT data type is used to make this distinction (bookmarks are type R8, enums are I4). however since the Telecom Engine deals with string types for variables the above constants have been made negative in order to make this same distinction.

If the **Move** call would move the current record position to a point before the first record, ADO sets the current record to the position before the first record in the recordset (BOF is set to **True**). An attempt to move backward when the **BOF** property is already **True** generates an error.

If the **Move** call would move the current record position to a point after the last record, ADO sets the current record to the position after the last record in the recordset (EOF is set to **True**). An attempt to move forward when the **EOF** property is already **True** generates an error.

Calling the **Move** method from an empty **Recordset** object generates an error.

Use adoRSetIsBOF() and adoRSetIsEOF() to test these conditions.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetMoveFirst

Synopsis:
adoRSetMoveFirst(handle)
Arguments:
handle - The recordset handle.

Description: This function moves the record pointer of the underlying recordset data to the first record in the recordset. A call to adoRSetMoveFirst() when the recordset is empty will result in an error. An empty recordset will have EOF and BOF properties set to **true** which can be tested with the adoRSetIsEOF() and adoRSetIsBOF() functions.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetMoveLast

Synopsis:
adoRSetMoveLast(handle)
Arguments:
handle - The recordset handle.

Description: This function moves the record pointer of the underlying recordset data to the last record in the recordset. A call to adoRSetMoveLast() when the recordset is empty will result in an error. An empty recordset will have EOF and BOF properties set to **true** which can be tested with the adoRSetIsEOF() and adoRSetIsBOF() functions.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetMoveNext

Synopsis:
adoRSetMoveNext(handle)
Arguments:
handle - The recordset handle.

Description: This function moves the record pointer of the underlying recordset data to the next record in the recordset. If the last record is the current record and you call adoRSetMoveNext() , ADO sets the current record to the position after the last record in the **Recordset** and sets **EOF** to **True**. An attempt to move forward when the **EOF** property is already **True** generates an error.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetMovePrev

Synopsis:
adoRSetMovePrev(handle)
Arguments:
handle - The recordset handle.

Description: This function moves the record pointer of the underlying recordset data to the previous record in the recordset. The **Recordset** object must support bookmarks or backward cursor movement; otherwise, the method call will generate an error. If the first record is the current record and you call the **MovePrevious** method, ADO sets the current record to the position before the first record in the **Recordset** and sets **BOF** to **True**. An attempt to move backward when the **BOF** property is already **True** generates an error. If the **Recordset** object does not support either bookmarks or backward cursor movement, the **MovePrevious** method will generate an error.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned,

then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetAddNew

Synopsis:
adoRSetAddNew(handle)
Arguments:
handle - The recordset handle.

Description: This function attempts to add a new record to a **recordset** as specified by *handle*.

Use the adoRSetSupports() function with **adAddNew** (a CursorOptionEnum value) to verify whether you can add records to the current **recordset** object.

After you call the adoRSetAddNew() function, the new record becomes the current record and remains current after you call the adoRSetUpdate() function. Since the new record is appended to the **recordset**, a call to adoRSetMoveNext() following the update will move past the end of the **recordset**, making **EOF** True. If the **recordset** object does not support bookmarks, you may not be able to access the new record once you move to another record. Depending on your cursor type, you may need to call the adoRSetRequery() method to make the new record accessible.

If you call adoRSetAddNew() while editing the current record or while adding a new record, ADO calls the underlying **Update** method automatically in this case to save any changes and then creates the new record.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetUpdate

Synopsis:
adoRSetUpdate(handle)
Arguments:
handle - The recordset handle.

Description: This function saves any changes that have been made to the current record of a **recordset** object since calling the adoRSetAddNew() function or since changing any field values in an existing record. The **recordset** object must support updates.

If you move from the record you are adding or editing before calling the **Update** method, ADO will automatically call **Update** to save the changes. You must call the adoRSetCancelUpd() function if you want to cancel any changes made to the current record or discard a newly added record.

The current record remains current after you call the adoRSetUpdate() function.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetCancelUpd

Synopsis:
adoRSetCancelUpd(handle)
Arguments:
handle - The recordset handle.

Description: This function cancels any changes made to the current record or discard a newly added record.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetUpdBatch

Synopsis:
adoRSetUpdBatch(handle,affected)
Arguments:
handle - The recordset handle.

Description: This function allows multiple changes to one or more records to be cached locally until you call the adoRSetUpdBatch() function. If you are editing the current record or adding a new record when you call the adoRSetUpdBatch() function method, ADO will automatically call and **Update** method to save any pending changes to the current record before transmitting the batched changes to the provider. Batch updating should be used with either a keyset or static cursor only and the lock type should be **adLockBatchOptimistic.**

If the attempt to transmit changes fails for any or all records because of a conflict with the underlying data (for example, a record has already been deleted by another user), the provider returns warnings to the Errors collection and a run-time error occurs.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetCancelBatch

Synopsis:
adoRSetCancelBatch(handle,affected)
Arguments:
handle - The recordset handle.

Description: This function cancels any pending batch updates.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned,

then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetDelete

Synopsis:
adoRSetDelete(handle[,affected])
Arguments:
handle - The recordset handle.

Description: This function marks the current record or a group of records in a **recordset** object for deletion. If the **recordset** object doesn't allow record deletion, an error occurs. If you are in immediate update mode, deletions occur in the database immediately. If a record cannot be successfully deleted (due to database integrity violations, for example), the record will remain in edit mode after the call to adoRSetUpdate(). This means that you must cancel the update with CancelUpdate before moving off the current record (for example, with adoRSetClose(), adoRSetMove() etc).

If you are in batch update mode, the records are marked for deletion from the cache and the actual deletion happens when you call the adoRSetUpdBatch() function. Use the Filter property to view the deleted records.

Retrieving field values from the deleted record generates an error. After deleting the current record, the deleted record remains current until you move to a different record. Once you move away from the deleted record, it is no longer accessible.

If you nest deletions in a transaction, you can recover deleted records with the adoConnTransCancel() function. If you are in batch update mode, you can cancel a pending deletion or group of pending deletions with the adoRSetCancelBatch() function.

If the attempt to delete records fails because of a conflict with the underlying data (for example, a record has already been deleted by another user), the provider returns warnings to the **Errors** collection but does not halt program execution. A run-time error occurs only if there are conflicts on all the requested records.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetState

Synopsis:
adoRSetState(SetHandle)
Arguments:
handle - The recordset handle.

Description: This function is a utility function that returns the value of the **State** property of a **recordset** object. This value can also be obtained by calling the adoRSetParmGet() function passing the P_RS_STATE constant to retrieve the value.

The returned value will be one of the adObjectState enumeration values as follows:

Constant	Value	Description
adStateClosed	0	Indicates that the object is closed.
adStateOpen	1	Indicates that the object is open.
adStateConnecting	2	Indicates that the object is connecting.
adStateExecuting	4	Indicates that the object is executing a command.
adStateFetching	8	Indicates that the rows of the object are being retrieved.

Returns: Returns the value of the **State** property of the recordset object, or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoRSetIsBOF

Synopsis:
adoRSetIsBOF(SetHandle)
Arguments:
handle - The recordset handle.

Description: This function is a utility function that returns the value of the **BOF** property of a **recordset** object. This value can also be obtained by calling the adoRSetParmGet() function passing the P_RS_BOF constant to retrieve the value.

The **BOF** property will be set to **true** if the record pointer of the **recordset** is positioned before the beginning of the data, and the function will return "1" (**true**);

If a **recordset** does not contain any records then both **EOF** and **BOF** will be true.

Note: ADO uses the value -1 for **true** and so if adoRSetParmGet() is used then it will be -1 that is returned if the condition is **true**, whereas this function will return "1" instead.

If the record pointer is not before the beginning of the data the function will return "0".

Returns: Returns 1 if the record pointer is positioned before the beginning of the data, otherwise returns 0 or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

Synopsis:
adoRSetIsEOF(SetHandle)
Arguments:
handle - The recordset handle.

Description: This function is a utility function that returns the value of the **EOF** property of a **recordset** object. This value can also be obtained by calling the adoRSetParmGet() function passing the P_RS_EOF constant to retrieve the value.

The **EOF** property will be set to **true** if the record pointer of the **recordset** is positioned after the end of the data, and the function will return "1" (**true**);

If a **recordset** does not contain any records then both **EOF** and **BOF** will be true.

Note: ADO uses the value -1 for **true,** and so if adoRSetParmGet() is used then it will be -1 that is returned if the condition is **true**, whereas this function will return 1 instead.

If the record pointer is not positioned after the end of the data the function will return "0".

Returns: Returns 1 if the record pointer is positioned before the beginning of the data, otherwise returns 0 or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoFldCount

Synopsis:
numfields=adoFldCount(handle)
Arguments:
handle - The recordset handle.

Description: This function returns the number of fields (columns) in the underlying **recordset** data. The names of these fields can then be obtained using the adoFldGetName() function.

Returns: Returns the number of fields (columns) in the underlying **recordset** data or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoFldGetName

Synopsis:
adoFldGetName(handle,ix)
Arguments:
handle - The recordset handle.
ix - The field index (starting from 0)

Description: This function returns the field name of the field specified by the index value *ix* for the recordset specifed by *handle*. Field index values start from 0 up to one less that the total number of fields in the recordset.

Returns: Returns the field name upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoFldGetValue

Synopsis:
adoFldGetValue(handle,ix/name,&pValue)
Arguments:
handle - The recordset handle.
ix/name - The field index number or field name
pValue - Pointer to the variable that will hold the returned field value

Description: This function returns the value of the field specified by the field index number (*ix*) or the field *name* for the current record in the recordset specifed by *handle*. The function will detect automatically whether a field index number or field name has been specified. The value of the uinderlying field will be returned to the variable pointed to by the *pValue* argument.

Below is an example showing this function in use. This example assumes that the underlying recordset has fields called 'date','time','telno','duration','rate' and 'cost':

```
 rec_count=adoRsetRecCount(SetHandle);

for(i=0;i<rec_count;i++)
     x=adoRSetMove(SetHandle,-i,adBookmarkLast);
    if(adoRSetIsBOF(SetHandle))
      break;
   endif

    adoFldGetValue(SetHandle,"date",&date);
    adoFldGetValue(SetHandle,"time",&time);
    adoFldGetValue(SetHandle,"telno",&telno);
    adoFldGetValue(SetHandle,"duration",&duration);
    adoFldGetValue(SetHandle,"rate",&rate);
    adoFldGetValue(SetHandle,"cost",&cost);
    applog("Rec ",rec_count-i,": date=",date," time=",time," telno=",strrtrim(telno)," dur=",duration,"
rate=",rate," cost=",cost);
endfor
```

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoFldSetValue

Synopsis:
adoFldSetValue(handle,ix/name,Value)
Arguments:
handle - The recordset handle.
ix/name - The field index number or field name
pValue - Pointer to the variable that will hold the returned field value

Description: This function sets the value of the field specified by the field index number (*ix*) or the field *name* to the value specified by *Value* for the current record in the recordset specifed by *handle*. The function will detect automatically whether a field index number or field name has been specified.

This only updates the fields values in the current copy of the record. In order to save the values to the underlying database table then it is necessary to call the adoRSetUpdate() function. Also if you

move from the record you are adding or editing before calling the **Update** method, ADO will automatically call **Update** to save the changes. You must call the adoRSetCancelUpd() function if you want to cancel any changes made to the current record.

Below is an example showing this function in use. This example assumes that the underlying recordset has fields called 'date','time','telno','duration','rate' and 'cost':

```
// Add a new record to the record set
   x=adoRSetAddNew(SetHandle);
 if(x<0)
   stop;
 endif

  // Set the field values for the new record
   adoFldSetValue(SetHandle,"date","20090101");
   adoFldSetValue(SetHandle,"time","120000");
   adoFldSetValue(SetHandle,"telno","971414292929");
   adoFldSetValue(SetHandle,"duration",26);
   adoFldSetValue(SetHandle,"rate","0.08");
   adoFldSetValue(SetHandle,"cost","0.035");

  // Update the underlying record set
   adoRSetUpdate(SetHandle);
```

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoFldParmGet

Synopsis:

adoFldParmGet(handle,ix/name,ParmID,pValue)

Arguments:

handle - The recordset handle.
ix/name - The field index number or name
parmID - The parameter ID
pValue - Pointer to the variable that will hold the parameter value

Description: This function enables the underlying field parameters to be examined for the field defined by the field index number (*ix*) or field *name* in the recordset defined by *handle*. The parmID argument is the ID of the parameter that is to be read and the *pValue* is a pointer to a variable that will hold the returned parameter value.

The parmID should be set to one of the following values as defined in the **ado.inc** file supplied with the library as follows:

```
const P_FLD_ACTUALSIZE          =201; #Returns the actual length of a field's value
const P_FLD_ATTRIBUTES          =202; #Sets or returns the attributes of a Field object
const P_FLD_DEFINEDSIZE         =203; #Returns the defined size of a field
const P_FLD_NAME                =204; #Sets or returns the name of a Field object
const P_FLD_NUMERICSCALE        =205; #Sets or returns the number of decimal places allowed for
numeric values in a Field object
const P_FLD_ORIGINALVALUE       =206; #Returns the original value of a field
const P_FLD_PRECISION                =207; #Sets or returns the maximum number of digits allowed
when representing numeric values in a Field object
```

```
const P_FLD_STATUS              =208; #Returns the status of a Field object
const P_FLD_TYPE                =209; #Sets or returns the type of a Field object
const P_FLD_UNDERLYINGVALUE     =210; #Returns the current value of a field
const P_FLD_VALUE               =211; #Sets or returns the value of a Field object
```

The above constants map one-to-one with the corresponding underlying properties from the ADO **_Field** object as shown below:

Property	Readable	Writeable	Description
ActualSize			Returns the actual length of a field's value
Attributes			Sets or returns the attributes of a Field object
DefinedSize			Returns the defined size of a field
Name			Sets or returns the name of a Field object
NumericScale			Sets or returns the number of decimal places allowed for numeric values in a Field object
OriginalValue			Returns the original value of a field
Precision			Sets or returns the maximum number of digits allowed when representing numeric values in a Field object
Status			Returns the status of a Field object
Type			Sets or returns the type of a Field object
UnderlyingValue			Returns the current value of a field
Value			Sets or returns the value of a Field object

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoFldParmSet

Synopsis:

adoFldParmSet(handle,ix/name,ParmID,Value)

Arguments:

handle - The recordset handle.
ix/name - The field index number or name
ParmID - The parameter ID
Value - The value to set

Description: This function enables the underlying **field** parameters to be set for field specified by the field index (ix) or field name in the **recordset** defined by *handle*. The parmID argument is the ID of the parameter that is to be set (see adoFldParmGet()) and the *Value* is the value to set it to. Note that this function can be used at any time after the **recordset** has been returned by a call to adoRSetQuery() and can be used to set up the properties of the field.

Returns: Returns 0 upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoErrCount

Synopsis:

numErrors=adoErrCount(handle)

Arguments:

handle - The Connection handle.

Description: This function returns the number of errors in the **Errors** collection of underlying **Connection** object data. Information about the errors in the collection can can then be obtained using the adoErrMessage(), adoErrValue() and adoErrNative() functions. The **Errors** collection provides the means to get more specific and detailed data about errors which have occured on a connection.

The **Errors** collection is maintained on a per **connection** basis and multiple errors may be inserted into the **Errors** collection by the data provider for a single function call.

The adoErrClear() function allows the error collection to be cleared for a particular connection.

Returns: Returns the number of errors in the **Errors** collection of underlying **connection** object, or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoErrMessage

Synopsis:

message=adoErrMessage(handle,ix)

Arguments:

handle - The recordset handle.
ix **-** The index into the **Errors** collection**.**

Description: This function returns the text of an error in the **Errors** collection of the underlying **connection** object specified by *handle*. The *ix* argument is the index number of the error (starting from 0) in the **Errors** collection to return.

Returns: Returns the Error message upon success or a negative error code. If ADOERR_COMERR is returned, then the underlying ADO error can be obtained by calling adoLastError()

-o-

adoErrValue

Synopsis:

value=adoErrValue(handle,ix)

Arguments:

handle - The recordset handle.
ix **-** The index into the **Errors** collection**.**

Description: This function returns the error value of an error in the **Errors** collection from the underlying **connection** object specified by *handle*. The *ix* argument is the index number of the error (starting from 0) in the **Errors** collection to return.

Returns: Returns the Error code value of the item in the **Errors** collection.

-o-

adoErrNative

Synopsis:
NativeValue=adoErrValue(handle,ix)
Arguments:
handle - The recordset handle.
ix **-** The index into the **Errors** collection**.**

Description: This function returns the native error code (provider specific) of an error in the **Errors** collection from the underlying **connection** object specified by *handle*. The *ix* argument is the index number of the error (starting from 0) in the **Errors** collection to return.

Returns: Returns the Native Error code value of the item in the **Errors** collection.

-o-

adoErrClear

Synopsis:
adoErrClear(handle)
Arguments:
handle - The recordset handle.

Description: This function clears the the **Errors** collection from the underlying **connection** object specified by *handle,* and sets the count to 0.

Returns: Returns 0 upon success or a negative error code.

-o-

String Manipulation Library

Introduction

The String Manipulation Library (**CXSTRING.DLL**) provides various functions for the manipulation of character strings, such as token extraction, partial string extraction, case changing etc.

In addition it provides a number of conversion routines to allow strings to be converted from and to hexadecimal or ascii values.

It also provides functions for the manipulation of so called hexi-strings. Hexi-strings are strings of hexidecimal characters where two hexidecimal characters of the string represent a single byte value. These strings are used in some of the function libraries where it is necessary to set abitary binary values for various parameters in the library.

Here is an example of a hexi-string that is the 4 byte long integer representation of the number 255 in little endian byte order:

"FF000000"

The hexi-strings are ofter used by function libraries when it is necessary to access the low-level fields of a communications protocol (such as the information elements of ISDN). See the Aculab Call Control Library (CXACULAB.DLL) for examples where hexi-strings are put to use (e.g in the CCsetparm() function).

-o-

String Library Quick Reference

token=strtok (str,tok_delimiter)
len=strlen(str)
partial_string=strsub(str,start_pos[,end_pos])
true_false=strcnt(str1,str2)
new_str=strstrip(str[,char])
tail=strend(str,count)
offset=strpos(str1,str2)
upper_str=strupr(str)
lower_str=strupr(str)
hi_same_low=strcmp(str1,str2)
index=strindex (str,str1[,str2[,str3...]])
selected_str=strselect (index,str1[,str2[,str3....])
new_string=strltrim(str[,char])
new_string=strrtrim(str[,char])
new_string=strrjust(str,char, tot_chars)
new_string=strljust(str,char, tot_chars)
char=itoc(ascii_val)
ascii_val=ctoi(char_str)
hex_val=itox(int_val)
value=xtoi(hex_str)
hexstr=strtohexi(string)
hexstr=inttohexi(unsigned_val,num_bytes)
hexstr=unstohexi(int_val,num_bytes)
string=hexitostr(hexi_str)
int_val=hexitoint(hexi_str,num_bytes)
unsigned=hexitoint(hexi_str,num_bytes)

-o-

String Manipulation Function Reference

strtok

Synopsis:

token=strtok (str,tok_delimiter)

Arguments:

str - The string containing the list of tokens to extract
tok_delimiters - The set of token delimiters to look for.

Description: This function searches through the specified *str* and extracts the tokens one at a time that are separated the specified *token_delimiter.* The first call to strtok() should me made with both *str* and *tok_delimeter* set to empty strings (""). This causes all internal counters and data to be reset. Thereafter the function can be called any number of times to extract the tokens from the string one at a time until there are no tokens left to extract (after which the function will return and empty string ("")).

For example, lets say that we have a string of values speparated by commas and we want to extract the values from this string one at a time (this is often the case when reading Call Data Records for billing purposes). E,g,

```
CDR="20080320,120603,02082073435,0014153325678,0,360,0.025";

// Reset the strtok() function
strtok("","");
// Extract the fields
Date=strtok(CDR,",");
Time=strtok(CDR,",");
callerID=strtok(CDR,",");
DestNum=strtok(CDR,",");
CallStat=strtok(CDR,",");
Duration=strtok(CDR,",");
Charge=strtok(CDR,",");
```

Note that the token_delimiter can be a multiple character delimeter, for example:

```
  // Notice that the fields are separated by two charatecrs ("::")
CDR="20080320::120603::02082073435::0014153325678::0::360::0.025";

// Reset the strtok() function
strtok("","");
// Extract the fields
Date=strtok(CDR,"::");
Time=strtok(CDR,"::");
... etc
```

Note that for fields separated by tabs, newlines or carriage returns then the escape characters "`t", "`n" and "`r" can be used as the *tok_delimiter.*

Returns: Returns the next token from the string or an empty string ("") if there are no more tokens to extract.

-o-

strlen

Synopsis:

len=strlen(str)

Arguments:

str - The string whose length to return

Description: This function returns the number of characters in the given *str*.

For example:

```
string="The cat sat on the mat";
// Count how many times "t" occurs
int i;
int tot;
for (i=1,i<=strlen(string);i++)
        if(strsub(string,i,1) streq "t")
                tot+=1;
        endif
endif
applog("Number of t's in string=",tot);
```

Returns: Returns the number of characters in the given *str*.

-o-

strsub

Synopsis:

partial_string=strsub(str,start_pos[,end_pos])

Arguments:

str - The string to extract partial string from

start_pos - The starting position from which to extract the partial string (first character is numbered 1)

[end_pos] - Optional ending position for the partial string extraction

Description: This function extracts a partial string from the given *str* using the *start_pos* and *end_pos* to specified the offsets in the *str* from which to obtain the partial string to return. The *start_pos* must hold a value between 1 and the total number or characters in the string. The optional *end_pos* should hold a value between 1 and the total number or characters in the string and must be greater than or equal to the *start_pos*. If *end_pos* is not specified then it defaults to the offset of the last character in the given string and so the partial string returned will be from the *start_pos* to the end of the given *str*.

If invalid *start_pos* or *end_pos* values are given (such as negative values, start_pos > end_pos, start_pos past end of string etc) then the function will return an empty string.

For example:

```
string="ABC123ZYX"
// This will return ABC
part_str=strsub(string,1,3);

// This will return 123
part_str=strsub(string,4,6);

// This will return A
part_str=strsub(string,1,1);

// This will return 123ZYZ
part_str=strsub(string,4);
```

Returns: Returns the partial string as sepcifed by the *start_pos* and *end_pos* or an empty string if invalid parameters are passed.

-o-

strcnt

Synopsis:
true_false=strcnt(str1,str2)

Arguments:
str1 - The string to search
str2 - The sub-string to search for in str1

Description: This function searchs through *str1* for the first occurance of the sub-string *str2*. If *str2* is found in *str1* then this function returns 1, otherwise it will return 0 if *str2* was not found in *str1*.

For example:

```
string="the cat sat on the mat";

// This will return 1
true_flase=strcnt(string,"cat");
```

Returns: Returns 1 if *str1* contains *str2* otherwise returns 0

-o-

strstrip

Synopsis:
new_str=strstrip(str[,char])

Arguments:
str - The string from which to strip the characters

[char] - optional argument specify the character to strip from *str* (defaults to space character)

Description: This function removed all occurances of the character *char* from the specified string *str*. If *char* is not specified then it defaults to the space character.

For example:

```
string="the cat sat on the mat";

// This will return "thecatsatonthemat"
new_str=strstrip(string);

// This will return "cat sat on mat"
new_str=strstrip(string,"the ");
```

Returns: Returns the new string with specified character stripped out.

-o-

strend

Synopsis:
tail=strend(str,count)

Arguments:
str - The string whose last *count* characters to return.
count - the number of characters to return at the end of the string

Description: This function returns the tail end of the specified string *str* as defined by the argument *count*. If *count* is more than the number of characters in the string then the entire string will be returned.

For example:

```
string="the cat sat on the mat"

// This will return "mat"
tail=strend(string,3);

// This will return "on the mat"
tail=strend(string,10);

// This will return "the cat sat on the mat"
tail=strend(string,200);
```

Returns: Returns the tail end of the string *str* with the number of characters specified by *count*.

-o-

strpos

Synopsis:
offset=strpos(str1,str2)

Arguments:
str1 - The string to search
str2 - The sub-string to search for in str1

Description: This function searchs through *str1* for the first occurance of the sub-string *str2*. If *str2* is found in *str1* then this function returns the character offset in *str1* where the first occurance of *str2* was found, otherwise it will return 0 if *str2* was not found in *str1*.

For example:

```
string="the cat sat on the mat";

// This will return offset 5
offset=strcnt(string,"cat");
```

Returns: Returns the offset of the first occurance of *str2* within *str1*, or 0 if *str2* is not found within *str1*.

-o-

strupr

Synopsis:
upper_str=strupr(str)

Arguments:
str - The string to convert to upper case

Description: This function converts the specified string *str* to upper case characters.

For example:

```
string="the cat sat on the mat"

// This will return "THE CAT SAT ON THE MAT"
upper_str=strupr(string);
```

Returns: Returns the *str* with all lower case characters converted to upper case.

-o-

strlwr

Synopsis:
lower_str=strupr(str)

Arguments:

str - The string to convert to lower case

Description: This function converts the specified string *str* to lower case characters.

For example:

```
string="THE CAT SAT ON THE MAT"

// This will return "the cat sat on the mat"
upper_str=strlwr(string);
```

Returns: Returns a string with all upper case characters converted to lower case.

-o-

strcmp

Synopsis:

hi_same_low=strcmp(str1,str2)

Arguments:

str1 - The first string to compare
str2 - The second string to compare

Description: This function carries out an alphabetical comparison between *str1* and *str2* and will return the following values based upon the result of the comparison:

If *str1* > *str2* the function will return 1
If *str1* = *str2* the function will return 0
If *str1* < *str2* the function will return -1

For Example:

```
str1="ABC";
str2="BBC"
// This will return 1
hi_same_low=strcmp(str,str2);

str1="BBC"
str2="BBC"
// This will return 0
hi_same_low=strcmp(str,str2);

str1="BBC"
str2="ABC";
// This will return -1
hi_same_low=strcmp(str,str2);

// Note that because the comparison is done alphabetically then str1 < str2 in the following
str1="01999";
str2="1";
// This will return -1
hi_same_low=strcmp(str,str2);
```

Returns: Returns 1, 0 or -1 depending on whether str1 > str2, str1 = str2 or str1 < str2.

-o-

strindex

Synopsis:
index=strindex (str,str1[,str2[,str3...]])

Arguments:
str - The string to match
str1 - The first comparison string
[str2] - The second comparison string
[str3] - The third comparison string
...

Description: This function compares the given string, *str*, against the set of other strings, *str1, str2, str3... etc* and will return the index number of the first string that matches *str.* The index ranges from 1 upwards for *str1* onwards. If none of the strings, *str1, str2, str3... etc* match the given string, *str,* then the function will return 0.

For example:

```
string="MAR";
// This will return 3

which_month=strindex(string,"JAN","FEB","MAR","APR","MAY","JUN","JUL","AUG","SEP","OCT","NOV","DE
C");
```

Returns: Returns the index 1, 2, 3.. of the first string *str1, str2 ,str2…* to match *str*, or 0 if the string is not found in the list.

-o-

strselect

Synopsis:
selected_str=strselect (index,str1[,str2[,str3....])

Arguments:
index - The index (starting from 1) of the string to select
str1 - The first string
[str2] - The second string
[str3] - The third string
... etc

Description: This function will select and return the string *str1, str2, str3...* depending upon the value of *index.* If *index* is 1 then *str1* will be returned, if *index* is 2 then *str2* will be returned etc. If an invalid index is given then the function will return an empty string ("").

Returns: Returns str1 if index is 1, str2 if index is 2, and so on. If index is < 1 or > number of string arguments given, then an empty string is returned.

-o-

strltrim

Synopsis:
new_string=strltrim(str[,char])

Arguments:
str - The string from which to strip the given character
[char] - Optional argument giving the character to strip from the left side of the string (defaults to space character).

Description: This function strips the given *char* from the left side of the given string, *str*. If *char* is not specified then it defaults to the space character.

For example:

```
string="          the cat sat on the mat";
// This will return "the cat sat on the mat"
new_str=strltrim(str);

string="00001234";
// This will return "1234"
new_string=strltrim(string,"0");
```

Returns: Returns the new string with all *char* removed from the beginning of the string.

-o-

strrtrim

Synopsis:
new_string=strrtrim(str[,char])

Arguments:
str - The string from which to strip the given character
[char] - Optional argument giving the character to strip from the right hand side of the string (defaults to space character).

Description: This function strips the given *char* from the right hand side (the end) of the given string, *str*. If *char* is not specified then it defaults to the space character.

For example:

```
string="the cat sat on the mat          ";
```

```
// This will return "the cat sat on the mat"
new_str=strltrim(str);

string="1234!!!!!!!!!";
// This will return "1234"
new_string=strltrim(string,"!");
```

Returns: Returns the new string with all *char* removed from the end of the string.

-o-

strrjust

Synopsis:
new_string=strrjust(str,char, tot_chars)

Arguments:
str - The string to pad with *char* at the beginning
char - The character to insert at the beginning of the string
tot_chars - How big the string should be after padding

Description: This function right justifies the given string to the length specified by *tot_chars* by padding the beggining of the string, *str,* with *char.* If the string, *str*, already contains the same or more characters that specified in *tot_chars* then the sting *str* will be returned unchanged.

For example:

```
// This will return "00000023"
new_str=strrjust("23","0",8);

// This will return ***ABC
new_str=strrjust("ABC","*",6);

// This will return ABCDEF
new_str=strrjust("ABCDEF","*",6);
```

Returns: Returns the given string right justified with with the character *char* up to the number of characters specified by *tot_chars*

-o-

strljust

Synopsis:
new_string=strljust(str,char, tot_chars)

Arguments:
str - The string to pad with *char* at the end
char - The character to insert at the end of the string
tot_chars - How big the string should be after padding

Description: This function left justifies the given string to the length specified by *tot_chars* by padding the end of the string, *str,* with *char.* If the string, *str*, already contains the same or more

characters that specified in *tot_chars* then the sting *str* will be returned unchanged.

For example:

```
// This will return "ABC   "
new_str=strljust("ABC"," ",6);

// This will return "ABC***"
new_str=strljust("ABC","*",6);

// This will return ABCDEF
new_str=strljust("ABCDEF","*",6);
```

Returns: Returns the given string left justified with with the character *char* up to the number of characters specified by *tot_chars*

-o-

itoc

Synopsis:
char=itoc(ascii_val)

Arguments:
ascii_val - The ascii value to convert into a character

Description: This function converts the ascii value passed to the function into a string containing a single character which is the ASCII character represesent by the value given by *ascii_val.* If zero or an invalid ascii value (< 0 or > 255) is given then the function will return an empty string.

For example:

```
// This will return " " (space)
char=itoc(32);

// This will return "A"
char=itoc(65);

// This will return "0"
char=itoc(0x30);
```

Returns: Returns a single character string which is the ASCII character represented by the value *ascii_val.*

-o-

ctoi

Synopsis:
ascii_val=ctoi(char_str)

Arguments:
    char_str - The character to convert to an ASCII value.

Description: This function returns the ASCII value of the first character of the string, *char_str*.

For example:

```
// This will return 65
ascii_val=ctoi("A");

// This will return 48
ascii_val=ctoi("0123");
```

Returns: Returns the ASCII value of the first character of *char_str*.

-o-

itox

Synopsis:
    hex_val=itox(int_val)

Arguments:
    int_val - The value to convert to hexadecimal

Description: This function converts the integer value represented by *int_val* into a string containing the hexadecimal equivalent.

For example:

```
// This will return 10
hex_val=itox(16);

// This will return "FF"
hex_val=itox(255);

// this will return "101"
hex_val=itox(257);
```

Returns: Returns atring containing the hexadecimal equivalent of the *int_val* vlaue.

-o-

xtoi

Synopsis:
    value=xtoi(hex_str)

Arguments:
    hex_str - The hexadecimal string to convert to a decimal integer

Description: Converts the hexadecimal value in the string *hex_str* to a decimal integer.

For Example:

```
// Returns 255
value=xtoi("FF");

// Returns 11
value=xtoi("b");
```

Returns: Returns the decimal integer equivalent of the hexadecimal number passed in the *hex_str* argument.

-o-

strtohexi

Synopsis:
hexstr=strtohexi(string);

Arguments:
string - The string to convert to a hexadecimal string

Description: This function converts the *string* argument into the corresponding hexadecimal string. Each character of the *string* will be converted to exactly two hexadecimal characters in the returned string corresponding to the ASCII values of the *string.* This function is useful in a number of other library functions (such as the CCsetparm() function) where a hexadecimal string is required for a parameter value.

Examples:

```
// This will return hex_str="3434323038323232"
hex_str=CCstrtohex("442082222");

//This will return hex_str="010fffde"
hex_str=CCstrtohex("`01`0f`ff`de");
```

Returns: Returns the hexadecimal string representation of the ASCII values of the characters in the *sting* argument.

-o-

inttohexi

Synopsis:
hexstr=inttohexi(unsigned_val,num_bytes);

Arguments:

int_val - The integer value to convert to a hexadecimal string
num_bytes - Set to 1, 2 or 4 for byte, short integer or long integer value

Description: This function converts the integer value *int_val* argument into the corresponding hexadecimal string . The *num_bytes* argument defines whether the integer should be treated as a 1 byte (char), 2 byte (short) or 4 byte (long) integer and will thus be converted into a 2,4 or 8 character hexidecimal string respectively.

Each byte of the integer will be converted to exactly two hexadecimal characters in the returned string in *little endian byte order* (i.e byte order will be low to high). This function is useful in some of the other library functions (such as the CCsetparm() function) where a hexadecimal string is required for a parameter value.

Examples:

```
// This will return hex_str="ff"
hex_str=CCinttohex(255,1);

// This will return hex_str="ff00"
hex_str=CCinttohex(255,2);

// This will return hex_str="ff000000"
hex_str=CCinttohex(255,4);

// This will return hex_str="80000000"
hex_str=CCinttohex(-128,4);
```

Returns: Returns the hexidecimal string representation of the given integer

-o-

unstohexi

Synopsis:

hexstr=unstohexi(int_val,num_bytes);

Arguments:

unsigned_val - The unsigned integer value to convert to a hexadecimal string
num_bytes - Set to 1, 2 or 4 for byte, short integer or long integer value

Description: This function converts the unsigned integer value *unsigned_val* argument into the corresponding hexadecimal string. The *num_bytes* argument defines whether the integer should be treated as a 1 byte (char), 2 byte (short) or 4 byte (long) integer and will thus be converted into a 2,4 or 8 character hexidecimal string respectively. Note that if a negative number is passed to the function then this will be converted to an unsigned integer first which will result in an interger overflow.

Each byte of the integer will be converted to exactly two hexadecimal characters in the returned string in *little endian byte order* (i.e byte order will be low to high). This function is useful in some other library functions (such as the CCsetparm() function) where a hexadecimal string is required for a parameter value.

Examples:

```
// This will return hex_str="ff"
hex_str=CCunstohex(255,1);

// This will return hex_str="ff00"
hex_str=CCunstohex(255,2);

// This will return hex_str="ff000000"
hex_str=CCunstohex(255,4);

// This will return hex_str="ffffffff" since a -ve integer will overflow when converted to unsigned
hex_str=CCunstohex(-1,4);
```

Returns: Returns the hexidecimal string representation of the given unsigned integer

-o-

hexitostr

Synopsis:
string=hexitostr(hexi_str)

Arguments:
hexi_str - The hexadecimal string of ASCII values to convert to a character string

Description: This function converts a hexadecimal string (where two hex characters represent a single byte) into a character string.

For Example:

```
// This will return "012345678"
string=hexitostr("303132333435363738 39")

// This will return "ABCDEF"
string=hexitostr("414243444546")
```

Returns: Returns the character string represented by the hexadecimal string of ASCII values..

-o-

hexitoint

Synopsis:
int_val=hexitoint(hexi_str,num_bytes)

Arguments:
hexi_str - The hexadecimal string to convert
num_bytes - The number of bytes to convert (either 1,2 or 4)

Description: This function converts a hexidecimal string into its decimal equivalent. The *hexi_str* value must be specified in *little endian* format (i.e lowest to highest byte order). The

num_bytes argument specifies whether 1, 2 or 4 bytes are to be converted. If an invalid *num_bytes* value is given then it is assume to be a 1 byte conversion.

For example:

```
// This returns -1
int_val=hexistoint("FF",1);

// This returns 255
int_val=hexistoint("FF00",2);

// This returns -1
int_val=hexistoint("FFFF",2);

// This returns 65535
int_val=hexistoint("FFFF0000",4);


// This returns        -268435456
int_val=hexistoint("00000010",4);
```

Returns: Returns the decimal integer equivalent of the given *hexi_str*.

-o-

hexitouns

Synopsis:

unsigned=hexitoint(hexi_str,num_bytes)

Arguments:

hexi_str - The hexadecimal string to convert
num_bytes - The number of bytes to convert (either 1,2 or 4)

Description: This function converts a hexidecimal string into its unsigned decimal integer equivalent. The *hexi_str* value must be specified in *little endian* format (i.e lowest to highest byte order). The *num_bytes* argument specifies whether 1, 2 or 4 bytes are to be converted. If an invalid *num_bytes* value is given then it is assume to be a 1 byte conversion.

For example:

```
// This returns 255
int_val=hexistoint("FF",1);

// This returns 255
int_val=hexistoint("FF00",2);

// This returns 65535
int_val=hexistoint("FFFF",2);

// This returns 65535
int_val=hexistoint("FFFF0000",4);


// This returns 268435456
int_val=hexistoint("00000010",4);
```

Returns: Returns the unsigned decimal integer equivalent of the given *hexi_str*.

-o-

Inter-task Messaging Library

Introduction

The Inter-task Messaging library (**CXMSG.DLL**) provides the means to pass messages between Telecom Engine tasks. Either the process ID of the task or the task name (if set by msg_setname() is called) can be used to identify a particular task to send a message to.

The sending side would then use the msg_send() function to send a string message to the recieving task. The receiving task must use msg_read() to then retrieve the message. All messages will be received in the order that they are send and they will be added sequentially to a queue if the receiving task is not already waiting for the message by a call to msg_read().

It is important that the receiving task retrives the incoming messages before the the incoming message queue gets full. By default the maximum number of messages that can be in the incoming message queue for all tasks is 512 messages. If this limit is reached then an error message is generated and the current message queue is dumped to the system tracelog, however on this first occasion the message queue is automatically increased to hold 1536 messages and the msg_send() function will still complete sucecssfully.

If however this new message queue limit is reached then all further calls to msg_send() will fail and an error message will be sent to the system error log.

Note that when a task is killed (either by a specific task_kill() function call, or implicitly after a *stop* or *restart* statement or a call to task_chain()) then all messages destined for that task that are waiting in the message queue will be deleted.

Note that is is possible that one task may attempt to send a message to another task using the task ID, but the task that the message was destined for has recently been stopped and another task has started running using the same task ID. In this situation an unexpected message may appear in the message queue for the newly started task. Programmers should take care to handle this situation by calling msg_flush() prior to calling msg_send()/msg_read() for the first time to ensure that there are no stry messages in the tasks message queue. This still could leave a small window of opportunity for a stray message to arrive after the call to msg_flush() and so other mechanisms may be taken (such as using a session number in the send/received messages) to cover this rare eventuality if it could cause problems/bugs to occur .

For example here is some code that requests a server task to return some kind of resource (E.g. an outbound channel).. This code covers all the eventuallities mentioned above about stray messages and uses a session number of correlate sent and received messages and the programmer should implement similar mechanisms to ensure correct code for this kind of functionality.

```
// Clear any stray messages in the message queue
msg_flush();

// Get a unique session ID
session_id=get_unique_number();

// Send a message to the resource handler to get the resource
x=msg_send("resouce_task","GET," & session_id);
if(x < 0)
	errlog("Failed to send message: err=",x);
	// Force jump to onsignal function to clear down call...
	task_hangup(task_getpid());
endif

// Now wait up to 10 seconds for response
resp_str=msg_get(10);

// resp should be in the form RSP,session_id,resource
strtok("","");
resp=strtok(resp_str,",");
resp_sess=strtok(resp_str,",");
resp_resource=strtok(resp_str,",");

// Check that this message is a valid response to our request
if(resp strneq "RSP" or resp_sess!=session_id)
	errlog("Got a bad message!  msg was ", resp_str);
	// Force jump to onsignal function to clear down call...
	task_hangup(task_getpid());
endif

ETC...
```

-o-

Inter-task Messaging Libary Quick Reference

result=msg_setname(name)
msg_str=msg_read(timeout[,ms_flag]);
result=msg_send(task_name/task_id,msg_str);
msg_flush()
task_id=msg_senderid()
task_id=msg_sendername()
free_count=msg_freecount()

-o-

Inter-task Messaging Function Reference

msg_setname

Synopsis:

result=msg_setname(name)

Arguments:

name - The name to associate with the calling task

Description: This function sets a name to associate with the calling task for use in calls to msg_send(). Note that msg_send() can use either the Telecom Engine Task ID or this name after this call has been made. The name used must be a unique name and an error will be generated if more than one task try to use the same task name.

The task name must begin with a non-numeric character, since calls to msg_send() with a task_id/name starting with a numerical character will be assumed to be passing the task ID rather than the name.

The maximum length of the task name is 32 characters. IF more characters than this are given for the task name then it will be truncated.

For example

```
// This sets the name for the calling task to "resource_handler"
x=msg_setname("resource_handler");
```

Now other tasks can send messages to the task that made the above call using the task name, e.g.

```
// Send a message to the task whose name was set to "resource_handler"
msg_send("resource_handler","GET");
```

Returns: Returns 0 if successfull or -1 if the task name already exists

-o-

msg_read

Synopsis:

msg_str=msg_read(timeout[,ms_flag]);

Arguments:

timeout - This is the time to wait for a message to arrive in seconds (or millisecs if ms_flag is set)

[ms_flag] - If this is set to a non-zero value then the timeout

Description: This function blocks and waits for an incoming message. The *timeout* argument specifies the time that the function should wait for a message to arrive in seconds (or if *ms_flag* is set then the *timeout* is specified in milliseconds). If there is already a message waiting for the

task in the message queue then the function will return immediately with the received message. If the *timeout* period is exceeded before a message is received then the function returns an empty string.

For example:

```
// Loop forever waiting for messages
while(1)
      // Wait 60 seconds for a message to arrive
      msg_str=msg_read(60);
      if(msg_str streq "")
            applog("No message received yet!");
            continue;
      else
            // we have got a message so process it..
            .... ETC
      endif
endwhile
```

Returns: Returns the received message or a blank string it timeout occurs

-o-

msg_send

Synopsis:
result=msg_send(task_name/task_id,msg_str);

Arguments:
task_name/task_id - Either the task name or the Telecom Engine Task ID of the task to send the message to
msg_str - The message to send

Description: This function sends a message to the task specified be the *task_name/task_id*. If a task calls the msg_setnname() function then this name can be used to identify the task, otherwise the Telecom Engine task ID (as returned from task_spawn() or task_getpid()) should be used.

If the *task_name/task_id* does not exist then the function will return -1.

Returns: Returns 0 if successful or -1 if an invalid *task_name/task_id* wa specified.

-o-

msg_flush

Synopsis:
msg_flush()

Arguments:
none

Description: This function flushes all pending messages from the message queue for the calling

task.

Returns: Nothing

-o-

msg_senderid

Synopsis:
task_id=msg_senderid()

Arguments:
none

Description: This function returns the Telecom Engine task id for the task that sent the last message to the calling task. This function can be used to determine where to send a return message to after receiving a message.

For example:

```
msg_str=msg_read(30);
// If a message was received..
if(msg_str strneq "")
      // Get the task ID of the sending task
      return_id=msg_senderid();
      // Send back an ACK message
      msg_send(return_id,"ACK");
endif
```

Returns: Returns the task ID of the task that sent the last message received by the calling task or -1 if no messages have been received.

-o-

msg_sendername

Synopsis:
task_id=msg_sendername()

Arguments:
none

Description: This function returns the task name (as set by a call to msg_setname()) for the task that sent the last message to the calling task. This function can be used to determine where to send a return message to after receiving a message.

For example:

```
msg_str=msg_read(30);
// If a message was received..
if(msg_str strneq "")
      // Get the task ID of the sending task
```

© Zentel Telecom Ltd, 2009

```
        return_name=msg_sendername();

        // make sure the return name is not blank
        if(return_name streq "")
              // use the task ID instead
              return_name=msg_senderid();
        endif


        // Send back an ACK message
        msg_send(return_name,"ACK");
endif
```

Returns: Returns the task name of the task that sent the last message received by the calling task, a blank string if no name was set or a -1 if no messages have been received.

-o-

msg_freecount

Synopsis:

free_count=msg_freecount()

Arguments:

none

Description: This function returns the number of free slots in the message queue. If messages are sent to a task faster than they can be picked up (or if the receiving task is not picking up messages at all), then the number of free message slots will gradually reduce until the message queue is full. If this is the case then this function will return 0.

Sometimes during load testing and diagnostic stages of development it might be useful to print the result of this call to the screen to check that message queues are being serviced quickly enough and that the number of free slots does not drop to zero or a low level due to some kind of bottleneck is the program.

Returns: Returns the number of free message slots in the message queue.

-o-

Global Array Library

Introduction

The Global Variable Library (**CXGLB.DLL**) provides a set arrays which are accessible to all tasks to allow global data to shared.

There are two types of Global array that are made available by this library. There is one static global array that contains 1024 elements, each of length 255 characters that can be retreived or set by the glb_get() and glb_set() functions.

Secondly, there are a set of functions that provide the means to dynamically create and access multi-dimensional arrays of strings.

All the arrays handled by this library a globally accessible from all tasks in the Telecom Engine.

-o-

Global Array Library Quick Reference

glb_set(index,string)
string=glb_get(index)
result=array_dim(array_name,dim0[,dim1,dim2..],size)
result=array_free(array_name)
result=array_set(array_name,index0[,index1,...],value)
value=array_get(array_name,index0[,index1,...])
result=array_search(array_name,search_order,value)
result=array_srchset(array_name,search_order,value,set_value)

-o-

Global Array Function Reference

glb_set

Synopsis:
glb_set(index,string)

Arguments:
index - This is the index into the static global array (from 0 to 1023)
string - This is the string value to set the specified global array element to.

Description: This function allows for an element of the static global array to be set. The index to the global array runs from 0 to 1023 (giving a maximum of 1024 elements). The maximum length of each string in the global array is 255 characters.

Returns: 0 if successful, -1 if an invalid element is specified.

-o-

Synopsis:
string=glb_get(index)

Arguments:

index - This is the index into the static global array (from 0 to 1023)

Description: This function returns the value from the static global array. If the element of the array has not previously been set by a call to glb_set(), or if an invalid index is given then the function will return an empty string. The index to the global array runs from 0 to 1023 (giving a maximum of 1024 elements). The maximum length of each string in the global array is 255 characters.

Returns: Returns the value of the global array element specified by the *index.*

-o-

array_dim

Synopsis:
result=array_dim(array_name,dim0[,dim1,dim2..],size)

Arguments:
array_name - The name of the array
dim0 - The size of the first dimension of the array
[dim1] - Optional size of second array dimension
[dim2] - Optional size of the third array dimension
.. ETC
size - Size of the string element of the array

Description: This function allows for a multi-dimensional array (with the name *array_name)* to be created where the dimensions are specified by *dim0, dim1, dim2... etc.* The *size* argument defines the size of the string for the array elements.

The *array_name* can be up to 32 characters long and is used to reference the array in the other array function calls such as array_set(), array_get() etc.

Arrays of any number of dimensions can be defined by specifying more dimension parameters in the *dim0, dim1, dim2... etc* list.

Below is an example of a one dimensional array:

```
// Creates a one dimensional array for resource management (element string length =1)
x=array_dim("RESOURCE_ARRAY",1024,1);

// Set an element of this array to 1 to mark the resource as "in-use"
array_set("RESOURCE_ARRAY",0,"1");
```

Below is an example of a two dimesional array:

```
// Create an array to hold data for a text screen console (25 rows x 80 cols)
x=array_dim("SCREEN1",25,80,1);

// Set the characters of the top line of the screen array to "_"
for(i=0;i<80;i++)
	array_set("SCREEN1",0,i,"_");
endfor
```

Below is an example of a three dimensional array

```
// Create an array to hold 3d co-ordinate space (100x100x100)
x=array_dim("3DSPACE",10000,100,1);

// Set co-ordinate (10,20,5) to 1
array_set("3DSPACE",10,20,5,"1");
```

Returns: Returns 0 if successful or -1 if the array failed to be allocated (E.g. out of memory error)

-o-

array_free

Synopsis:
result=array_free(array_name)

Arguments:
array_name - The name of the array to free

Description: This function releases a previous allocated array and free up the dynamically created memory.

Returns: 0 if successful or -1 if an invalid *array_name* was specified.

-o-

array_set

Synopsis:
result=array_set(array_name,index0[,index1,...],value)

Arguments:
array_name - The array name as previouslydefined by a call to array_dim()
index0 - The first index specifier
[index1] - The Optional second index specifier
[index2] - The Optional third index specifier
.. ETC
value - The value to set the array element to

Description: This function allows an element of an array previously created by a call to array_dim() to be assigned. The *array_name* specifies the name of the array and must correspond to the name given when the array was created. The *index0[, index1[, index2...etc]]* indexes specify the element of the array to set and the number of indexes given must correspond to the number of indexes specified when the array was created.

The *value* argument is the string value that will be set in to the array.

Returns: Returns 0 on success or -1 if an invalid *array_name* was given or an invalid number of

indexes.

-o-

array_get

Synopsis:
value=array_get(array_name,index0[,index1,...])

Arguments:
array_name - The array name as previouslydefined by a call to array_dim()
index0 - The first index specifier
[index1] - The Optional second index specifier
[index2] - The Optional third index specifier
.. ETC

Description: This function allows the value from an element of an array previously created by a call to array_dim() to be retrieved. The *array_name* specifies the name of the array and must correspond to the name given when the array was created. The *index0[, index1[, index2...etc]]* indexes specify the element of the array to get and the number of indexes given must correspond to the number of indexes specified when the array was created.

Returns: Returns the value of the element at the specified indexes or a blank string if if an invalid *array_name* was given or an invalid number of indexes was specified.

-o-

array_search

Synopsis:
result=array_search(array_name,search_order,value)

Arguments:
array_name - The array name as previouslydefined by a call to array_dim()
search_order - Search order (-1 for bottom up, -3 for search next)
value - The value to search for in the array

Description: This function allows for a one-dimensional array to be searched for a particular value. Note that only one-dimensional arrays are supported by this function. The search order defines how the array is to be searched and can be set to one of the following values:

-1 - Search for the first entry from the bottom up that matches the value
-2 - Top down search (NOT CURRENTLY SUPPORTED)
-3 - Search for then next match starting from the last match found
-4 - search for the previous value (NOT CURRENTLY SUPPORTED)

If -1 is specified then the search will always begin at element 0 of the index of the array and then cycle through the last index of the array until the end of the array is reached looking for a match for the specified value.

IF -2 is specified then the last position where a match was found will be remembered and the next

time the function is call the search will continue from that position and then wrap around to the beginning again.

The function will return the index number of the element that matches the value specified by the *value* argument. otherwise it will return -1

For example:

```
        // Create a one dimensional array
        x=array_dim("RESOURCE",100,1);

            // Set some elements of the array to 1.
        array_set("REOURCE",10,"1");
        array_set("REOURCE",20,"1");
        array_set("REOURCE",30,"1");


        // Search the RESOURCE array for the first element set to 1
        // This will return 10
        index=array_serch("RESOURCE",-1,"1");

        // This will return 10 since as well since we specified -1 for the search order which always searches
from 0 upwards
        index=array_serch("RESOURCE",-1,"1");

        // This will return 20 since we are now specifying to search for the next entry that matches after the
last match (ie. search order -3)
        index=array_serch("RESOURCE",-3,"1");

        // This will return 30 since we are now specifying to search for the next entry that matches after the
last match (ie. search order -3)
        index=array_serch("RESOURCE",-3,"1");
```

Returns: Returns the index where a match was found or -1 is no match was found

-o-

array_srchset

Synopsis:
result=array_srchset(array_name,search_order,value,set_value)

Arguments:
array_name - The array name as previouslydefined by a call to array_dim()
search_order - Search order (-1 for bottom up, -3 for search next)
value - The value to search for in the array
set_*value* - The vlaue to set the element to that matched the search for *value*

Description: This function allows for a one-dimensional array to be searched for a particular value and if an entry is found that matches the given *value* then that element is set to the new *set_value* value. Note that only one-dimensional arrays are supported by this function. The search order defines how the array is to be searched and can be set to one of the following values:

-1 - Search for the first entry from the bottom up that matches the value
-2 - Top down search (NOT CURRENTLY SUPPORTED)
-3 - Search for then next match starting from the last match found
-4 - search for the previous value (NOT CURRENTLY SUPPORTED)

If -1 is specified then the search will always begin at element 0 of the index of the array and then cycle through the last index of the array until the end of the array is reached looking for a match for the specified value.

IF -2 is specified then the last position where a match was found will be remembered and the next time the function is call the search will continue from that position and then wrap around to the beginning again.

The function will return the index number of the element that matches the value specified by the *value* argument. otherwise it will return -1.

This function is often used for maintaining resources where it is necessary to search for and set an array element in a single atomic operation.

For example:

```
// Create a one dimensional array
x=array_dim("RESOURCE",100,1);


// Search for the first element that has a blank string and set it to 1
index=array_srchset("RESOURCE",-1,"",1);


if(index >= 0)
   // we have found a free resource in the array (and it is now set to 1 to mark it as busy..
      ... etc
endif
```

Returns: Returns the index where a match was found or -1 is no match was found

-o-

Semaphore Library

Introduction

The Semaphore Library (**CXSEM.DLL**) provides semaphore functional to provide for critial sections within the code where it is necessary to ensure that only one task is accessing a particular data object or excecuting a particular set of code.

A semaphore is a special type of variable that can hold one of two values (set or unset) and can protect a section of code from being excecuted when another task has set the semaphore by providing functions that wait, check and set a semaphore and/or wait until a semphore is clear before allowing a process to enter the critical section.

The functions in this library rely on a fixed set of 1024 semaphores numbered 0 through to 1023. Each function takes one of these semaphore IDs as a function argument.

Note that the functions in this library are sensitived to the kill signal. If a task is killed (either by

an expicit external task_kill() call, by exceuting a *stop* or *restart* statement or by chaining to another task) then any semaphores that have been set by the task will be cleared and it will be removed from any queues wiating for semaphores.

-o-

Semaphore Library Quick Reference

sem_test(sem_id)
sem_set(sem_id)
sem_clear(sem_id)
sem_clrall()

-o-

Semaphore Function Reference

sem_test

Synopsis:
sem_test(sem_id)

Arguments:
sem_id - The semaphore ID.

Description: The first task that calls this function will cause the value of the specified semaphore (*sem_id)* to be set and the function will return 1 to indicate that the calling task sucessfully caused the semaphore to be set. If the semaphore had already been set by another task then the semaphore will remain unchanged and the return value form the function will be 0.

This allows for a quick check of the status of a semaphore to be carried out, but does not provide any kind of waiting or queuing mechanism. To wait in a queue for a semaphore to be released then the sem_set() function should be used.

Returns: 1 if the semaphore was successfully set by the calling task, 0 if another task has already set the semaphore, -1 if an invalid semaphore ID is specified.

-o-

sem_set

Synopsis:
sem_set(sem_id)

Arguments:
sem_id - The semaphore ID.

Description: The first task that calls this function will cause the value of the specified semaphore (*sem_id)* to be set and the function will return 1 to indicate that the calling task sucessfully caused the semaphore to be set. If the semaphore had already been set by another task then the semaphore will remain unchanged and the calling task will block and will be entered into a queue waiting for the semaphore to be cleared. When the calling task reaches the top of the queue and sets the semaphore the task will be unblocked and the function will return 1.

Note that this function also carries out a rudimentary check for deadlock. If an attempt to set a semaphore is carried out by a task (task 1), but the semaphore has already been set by another task (task 2), then before putting task 1 into a blocking state a check is made to ensure that task 2 isn't already waiting on a semaphore that has been set previously by task 1 (deadlock condition). If this condition is found the then function will output an error message and will return -2.

Note that if a task is killed (either through an external kill command, a stop statement, a restart statemetn or by chaining to another task) then all semaphore that the task had previously set will be cleared and it will be removed from any queues waiting for semaphores.

Returns: 1 if the semaphore was successfully set by the calling task, -1 if an invalid semaphore ID is specified, -2 if a deadlock condition would have occurred if the task had blocked.

-o-

sem_clear

Synopsis:
sem_clear(sem_id)

Arguments:
sem_id - The semaphore ID.

Description: This function clears a semaphore that was previous set by the calling task. If an attempt is made to slear a semephore that was not previously set by the calling task then a error message is generated and the function will return -3.

Returns: 0 if the semaphore was successfully cleared by the calling task, -1 if an invalid semaphore ID is specified, -3 if the semaphore was not set by the calling task.

-o-

sem_clrall

Synopsis:
sem_clrall()

Arguments:
NONE

Description: This function clears all semaphores that have been previous set by the calling task.

Returns: Always returns 0.

-o-

Clipper Database Library

Clipper Database Library Quick Reference

db_handle=db_open(filename,type,mode)
retval=db_ixopen(db_handle,field_nr/name,ntxfilename)
rec_handle=db_get(db_handle,recno[,lock])
rec_handle=db_append(db_handle)
value=db_fget(rec_handle,fieldnr/name[,pad])
ret_val=db_fput(rec_handle,fieldnr/name,data)
db_rls(rec_handle)
db_close(db_handle)
num_recs=db_nrecs(db_handle)
num_recs=db_nfields(db_handle)
no_chars=db_fwidth(db_handle,field_nr/name)
field_name=db_fname(db_handle,field_nr)
db_rlsall()
rec_num=db_first(db_handle,fieldnr/name,search_term)
rec_num=db_next([exact_flag])
rec_num=db_prev([exact_flag])
key_value=db_key()
rec_num=db_recnum(rec_handle)
db_flock(db_handle,on_or_off)

-o-

Introduction

This CLIPPER Database Llibrary (CXDBF.DLL) allows for searching and manipulating DBaseIII tables using CLIPPER indexes. In many cases it is simpler and faster to have a flat DBF table with CLIPPER indexes to carry out simple table lookup and index searching, rather than have a separate SQL database server. Not only does this often provide a quicker and faster method of implementing an application, but it reduces the costs and maintenance required to implement a full SQL server solution.

Database files in DBF format are supported by many PC applications, including dBase, Clipper, FoxPro and others. These files have the .DBF file extension. All DBF database file consists of a header followed by a number of fixed length records.

The header contains a list of the field names and types, the record length and the number of records stored in the DBF file.

All fields are stored as characters strings and can be one of the following field types:

Type	**Name**	**Description**
C	Character	An ASCII character string
N	Numeric	A numerical value with a fixed number of places. The field is right justified and is padded with blanks on the left and the there are *n* decimal places then the last *n* characters are assumed to be following the decimal point. For example if there a 3 decimal places then 123456 represents 123.456
L	Logical	Contains a single character T or F for TRUE or FALSE. A blank character can also represent FALSE.
D	Date	An Eight character string YYYYMMDD

All fields are numbered from zero for the first field and all TE functions which require a field as an argument will accept either field number or a field name.

All records witrhin a DBF table are numbered from one as the first record.

When a database is opened by a call to db_open() then this function will return a handle to the database table which is then used in subsequent calls to other db_xxx() functions. Zero or more associated indexes can then also be opened with calls to db_ixopen() for any indexes associated woth the DBF file.

When a record is read from the table using db_get() or a new record is appended using db_append() then a copy of the record is placed in memory and is referenced by a record handle. All changes made to the record using db_fput() and any field values retrieved by db_fget() are actually done using the copy that is held in memory. The underlying record on the disk will not be updated with any changes until a call is made to db_rls().

Note however that if an index key field is changed using db_fput() then the underlying index key value will be change immediately. It is important to make use of record locks when accessing and updating shared database tables to ensure that data integrity is maintained.

Another issue that should be considered when updating DBF records is to ensure that a hangup signal does not cause a jump to the onsignal program before all of the data in a record and been updated correctly. Use of functions such as CCsigctl() (from the Aculab function library) or sc_sigctl() (from the dialogic function libary) should be used to ensure that sections of code cannot be interrupted by hangup signals.

Index files are files that contain a list of <index key values> and the corresponding <DBF record number>. For example if we had a DBF table with the following entries:

Record Number	**Index Field Value**
1	London
2	Birmingham
3	Leeds
4	Manchester

5	Newcastle

The corresponding index file would have the following index key list

Index Key Value	**Record Number**
Birmingham	2
Leeds	3
London	1
Manchester	4
Newcastle	5

Notice that the Index file holds the records in alphabetical order. If a call is made to db_first() with a *search_term* set to "Aberdeen" then the db_first() function will fail to find an exact match and will return 0 and will position the index pointer to a position just before the first index entry. A call to db_next() will then cause the index pointer to move to the first index key entry ("Birmingham") and the db_next() call will return 2 for the corresponding record number.

If a call is to db_first() with the *search_term* set to "London" then and exact match will be found and the function will return the record number 1 (and the index pointer will be set to the "London" index entry. However if we set the *search_term* to "Lond", then an exact match will not be found and the db_first() function will return a record number of 0 and the index pointer will be set to point between the "Leeds" and "London" key entries. A subsequent call to db_next() would then return recrod number 1 (for "London") or a call to db_prev() will return record number 3 (for "Leeds").

Note that when a call to db_first() is made the *search_term* is always padded with spaces on the right up to the full width of the index key field.

All database handles and record handles (and their associated locks) are owned by the TE task that first obtained those handles. If an attempt is made by another task to release a record handle or close a database handle that it does not own then an error will be generated. Also if a task is killed for any reason (e.g. a *stop, restart, endmain, endsignal* statement is encountered or task_kill() or task_chain() causes the task to end) then all records owned by the task will be released (as if a call to db_rls() had been made) and all database handles will be closed (as if a call to db_close() had been made).

If an error is encountered by the library then an error message is written to the error log and a negative error value is returned. Below is a list of the possible error values that can be returned by the functions:

-1 General Error (see error message for reason)
-5 No free database handles. (see limits)
-6 File system error (E.g. bad path or file name)
-9 Invalid database handle given
-10 Invalid record handle given
-11 Invalid record number given

-13 No free record handles (see limits)
-14 Invalid field name or number given
-22 Index error
-24 Locking conflict

There are certain limits that have been hard-coded into the library with respect to the maximum number of open database jhandles or record handles. these limits are shown below:

Maximim database handles 2048
Maximum open indexes 4096
Maximum record handles 4096

-o-

Clipper Database Function Reference

db_open

Synopsis:
db_handle=db_open(filename,type,mode)
Arguments:
filename - The full path to the DBF file
type - Future Use (set to 0)
mode - Open mode (0=excl,1=shared)

Description: This function opens the DBF file file specifed by *filename*. The *type* argument is for furture use and should be set to 0. The *mode* argument specifies whether the DBF file should be opened in shared or exclusive mode.

If the DBF file is successfully opened then the function returns a file handle to the DBF file otherwise a negative error code is returned.

Returns: Returns the database handle or a negative error code.

-o-

db_ixopen

Synopsis:
retval=db_ixopen(db_handle,field_nr/name,ntxfilename)

Arguments:
db_handle - database handle returned from db_open()
field_nr/name - The field number or name of the field
ntxfilename - Full path to the NTX filename

Description: This function opens an NTX clipper index file associated with the DBF file specified by the *db_handle.* The *field_nr/name* specifies either the field number (numbered from 0) or the

actual name of the field in the DBF for which the index file references.

Typically each DBF file will have one or more index files each associated with one of the fields in the DBF file. Currently only Clipper NTX index files are supported.

Returns: Returns 0 upon success or a negative error code.

-o-

db_get

Synopsis:
rec_handle=db_get(db_handle,recno[,lock])

Arguments:
db_handle - A DBF file handle (as returned from db_open())
recno - The record number to read from the DBF file
[lock] - Optional locking (0=No lock, 1=Lock record)

Description: This function reads the record number defined by the *recno* argument from the DBF file specified by the database handle *db_handle.* If the optional *lock* argument is specified as a non-zero value then the record will be read with a record lock in place (so that another attempt to read the same record with a lock will fail).

Upon sucessfully reading the given record a record handle (*rec_handle)* is returned to an internal copy of the record. This *rec_handle* can then be used in calls to db_fget() to retrieve the values of individual fields in the record.

The record lock will be held until a call to db_rls() or db_close() or db_rlsall() or db_closeall() is made. Note that all record handles and database handles owned by a particular task will be released if the task is *killed* by a specific task_kill() or task_chain() command or a *stop* or *restart* statement.

Returns: Returns the record handle or a negative error code.

-o-

db_append

Synopsis:
rec_handle=db_append(db_handle)

Arguments:
db_handle - Database handle returned from db_open()

Description: This function appends a blank record to the given DBF file specified by *db_handle*. Note that the append byte for the DBF file will be locked and a blank entry will be added to any indexes associated with the DBF file and that have been opened using the db_ixopen() function.

After appending the record then a handle to the internal representation of the record will be returned. If there is already another task that has call db_append() on this DBF file but has not yet released the record with db_rls() then the function will fail with a locking error.

To release the record and write any changes made to the internal record representation made with db_fput() then and to remove the lock on the append byte then the db_rls() (or db_rlsall()) function should be called.

The record will also be released (and the lock removed) if the DBF file is explicitly closed through a call to db_close() or db_closeall() or if the task is killed through a call to task_kill() or task_chain() or if a *stop* or *restart* statement is encountered.

Returns: Returns the record handle or a negative error code.

-o-

db_fget

Synopsis:
value=db_fget(rec_handle,fieldnr/name[,pad])

Arguments:
rec_handle - The record handle
fieldnr/name - The field number or field name
[pad] - Flag to specify whether returned value should be padded with spaces to the full field width

Description: This function returns a field value from a record that has been copied to an internal record buffer by db_get() or db_append(). Either the field number (counted from 0) or the field name can be specified. If a non-zero value for *pad* is given then the returned value will be padded with spaces up to the full width of the field even if the contents of the field are less than this width.

Returns: Returns the value of the field or a negative error code.

-o-

db_fput

Synopsis:
ret_val=db_fput(rec_handle,fieldnr/name,data)

Arguments:
rec_handle - The record handle returned from db_get() or db_append()
fieldnr/name - The field number or field name
data - the value to write to the field

Description: This function writes the specified *data* to the internal copy of the record represented by *rec_handle.* Note that only the internal copy of the record is changed and that the actual DBF

file is not updated until a db_rls() call is made. However if the field being written is an index field with an associated open NTX file (opened by db_ixopen()), then this index will be locked, updated and unlocked. Therefore it is possible that for index fields this call can fail due to a locking conflict and the return value should be checked for this possiblility by the programmer (and the db_fput() call should be retried if this happens or a graceful recovery should be made).

Returns: Returns 0 upon success or a negative error code.

-o-

db_rls

Synopsis:
db_rls(rec_handle)

Arguments:
rec_handle - The record handle returned from db_get() or db_append()

Description: This function releases a record handle obtained by a call to db_get() or db_append() and writes any changes that have been made to the internal copy of the record back to the DBF file and releases and record (or append byte) locks.

Note that a record is automatically released if the task that obtained the record is killed (E.g. by an explicit call to task_kill() or task_chain()) or by encountering a *restart*, *stop* or *endmain* statement.

Returns: Returns 0 on success or a negative error code.

-o-

db_close

Synopsis:
db_close(db_handle)

Arguments:
db_handle - the database handle

Description: This function closes a database handle prevously opened by a call to db_open(). It will also close any indexes associated with the DBF file and that have been opened with a call to db_ixopen(). Also all currently aquired record handles (retrieved through db_get() or db_append() calls) will be released and any locks removed from the DBF file.

Returns: Returns 0 upon success or a negative error code.

-o-

db_nrecs

Synopsis:
num_recs=db_nrecs(db_handle)

Arguments:
db_handle - The database handle

Description: This function returns the number of records that are in the DBF table specified by the *db_handle* argument.

Returns: Returns the number of records in the table or a negative error code

-o-

db_nfields

Synopsis:
num_recs=db_nfields(db_handle)

Arguments:
db_handle - The database handle

Description: This function returns the number of fields in the records of the DBF table specified by the *db_handle* argument.

Returns: Returns the number of fields in the records belonging to the specified table else returns a negative error code if an invalid *db_handle* is given,

-o-

db_fwidth

Synopsis:
no_chars=db_fwidth(db_handle,field_nr/name)

Arguments:
db_handle - The database handle
fieldnr/name - The field number or field name

Description: This function returns the field width for the given *field_nr* or *field_name*, which is the number of characters required to hold the contents of the field.

Returns: Returns the field width or a negative error code.

-o-

db_fname

Synopsis:
field_name=db_fname(db_handle,field_nr)

Arguments:
db_handle - The database handle
field_nr - the field number

Description: This function returns the name of the field specified by the given *field_nr*. The *field_nr* defines the number of the field starting from 0.

Returns: Returns the field name or a negative error code.

-o-

db_rlsall

Synopsis:
db_rlsall()

Arguments:
NONE

Description: This function releases all record handles that have previously been obtained by the calling task (by calls to db_get() or db_append()) and then closes all database handles that have previously been opened by the task.

Note that when a record handle in released then all record locks are removed and any changes made to the records will be written to disk.

Note that db_rlsall() will automatically be called when a task is killed either through encountering a *stop* or *restart* or *endmain* or *endonsignal*statement or through an explict call to task_kill() or task_chain().

Returns: Returns 0

-o-

db_first

Synopsis:
rec_num=db_first(db_handle,fieldnr/name,search_term)

Arguments:
db_handle - The database handle
fieldnr/name - the field number or field name of the index field
search_term - The value to search for

Description: This function carries out an index search on the index associated with the given *fieldnr/name* (which must have previously been opened with the db_ixopen() call).

The db_first() function works in conjuction with the db_next() and db_prev() function calls to maintain an index pointer for a specific database handle that allows the calling task to search and step though an index in index key order.

If an exact match for *search_term* is found in the index then the index pointer will be positioned at that key and the record number of the corresponding record in the database table will be returned.

If an exact match for the *search_term* is not found then the index pointer will be positioned in between keys in the index and the function will return record number 0 (to indicate that no exact match was found). Calls to db_next() or db_prev() can then be used to step through the index keys in forward or reverse order from that point.

If a search is made for a index key value that is beyond then last key in the index then the index pointer will be positioned after the last key in the index. A call to db_next() will then return 0 to indicate that the index pointer is beyond the end of the index, whereas db_prev() will then return the last record in the index.

Similarly if a search is made for a key that is before the first entry in the index then the index pointer will be positioned prior to the first key in the index. A call to db_prev() will then return 0, whereas a call to db_next() will return the first key in the index.

Note that calls to db_first(), db_next() and db_prev() cannot be nested within a single task, since only one index pointer is maintainted per task.

Returns: If an exact match is found then the record number of that match is returned. If no exact match is found then the function will return 0 and the index pointer will be postitioned just before the next key that would match. If an invalide *db_handle* or non index *fieldnr/name* is specified then the function will return a negative error code.

-o-

db_next

Synopsis:
rec_num=db_next([exact_flag])

Arguments:
[exact_flag] **-** Optional argument whcih can be set to non-zero value if an exact match is required

Description: This function will move the index pointer to the next key in the index and return the record number of the corresponding record in the database table. A call to db_first() must have previously been made to start the search and set the initial position of the index pointer. If the call to db_next() causes the index pointer to go beyond the last key of the index then the function will return 0.

If the optional *exact_flag* argumment is set to a non-zero value then the function will only return a record number for a record that matches exactly the search_term speacified in the db_first() call. As soon as a key is encountered that does not match the search_term specified in the db_first() call then the function will return 0.

Returns: Returns the record number of the next key pointed to by the index pointer (set up previously be a call to db_first()) or returns 0 if the index pointer has moved beyond the end of the index .

if *exact_match* is set and the next key value does not match the *search_term* specified by db_first() the function will return also 0.

If a previous call to db_first() has not been made then the function will return a negative error code.

-o-

db_prev

Synopsis:
rec_num=db_prev([exact_flag])

Arguments:
[exact_flag] **-** Optional argument whcih can be set to non-zero value if an exact match is required

Description: This function will move the index pointer to the previous key in the index and return the record number of the corresponding record in the database table. A call to db_first() must have previously been made to start the search and set the initial position of the index pointer. If the call to db_next() causes the index pointer to move to before the first key of the index then the function will return 0.

If the optional *exact_flag* argumment is set to a non-zero value then the function will only return a record number for a record that matches exactly the search_term speacified in the db_first() call. As soon as a key is encountered that does not match the search_term specified in the db_first() call then the function will return 0.

Returns: Returns the record number of the previous key pointed to by the index pointer (set up previously be a call to db_first()) or returns 0 if the index pointer has moved prior to the start of the index .

if *exact_match* is set and the next key value does not match the *search_term* specified by db_first() the function will return also 0.

If a previous call to db_first() has not been made then the function will return a negative error code.

-o-

db_key

Synopsis:
key_value=db_key()

Arguments:
NONE

Description: This function returns key at current position in a search. A call to db_first() must

have previously been made to set the index pointer. This is useful for finding the current key when db_next() or db_prev() is used with a 0 (false) value for *exact_flag*. An empty string "" is returned if an error occurs, or if the current search is not positioned on a key value.

Returns: Returns the key value of the index key currently pointed to by the index pointer (set up through a call to db_first() (then possibly db_next() or db_prev()). If the index pointer does not point to a key entry in ythe index (either because db_first() hasn't been called or because the index pointer points between records), then the function will return an empty string.

-o-

db_recnum

Synopsis:

rec_num=db_recnum(rec_handle)

Arguments:

rec_handle - the record handle returned by db_get() or db_append()

Description: This function returns the record number (starting from 1) of the record specified by the given *rec_handle*.

Returns: Returns the record number of the speicifed record or a negative error code.

-o-

db_flock

Synopsis:

db_flock(db_handle,on_or_off)

Arguments:

db_handle - The database handle
on_or_off - set to non-zero to set the file lock, or 0 to release the file lock

Description: This function allows for an entire database table to be locked by setting the *on_or_off* argument to a non zero value. Once a lock has been set then any other task that attempts to call the db_flock() function will fail to obtain the lock until the lock is released.

Returns: Returns 0 or a negative error code. It will return -24 if another task has already obtained the lock.

-o-

Floating Point Library

Introduction

The Telecom Engine Language does not have built in support for floating point variable types. Instead the floating point operations are provided by the floating point library (CXFP.DLL).

-o-

Floating Point Library Quick Reference

fp_decs(places)
answer = fp_add(number1, number2[, decimals])
answer = fp_sub(number1, number2[, decimals])
answer = fp_mul(number1, number2[, decimals])
answer = fp_div(number1, number2[, decimals])
answer = fp_div(number1, number2[, decimals])
answer = fp_pow(number,power[, decimals])
answer = fp_rnd(number, type, digit[, decimals])

-o-

Floating Point Library Reference

fp_decs

Synopsis:
fp_decs(places)

Arguments:
places - The default number of decimal places for the library

Description: This function sets the default number of decimal places that will be returned by the library function calls. Note that this is a global setting and will effect all calls to the library. By default all functions will return results to two decimal places unless changed by this function (or if specified explicitly in the function call). The number of decimals can be set to any value between 0 and 9 inclusive.

Returns: Returns 0

-o-

fp_add

Synopsis:
answer = fp_add(number1, number2[, decimals])

Arguments:

number1 - The first floating point number

number2 - The second floating point number
[decimals] - Optional argument to specify the number of decimal places of the returned result

Description: This function adds the two floating pount numbers specified by *number1* and *number2* and returns the result. If the optional *decimals* argument is specified then the result will be returned with this number of decimal places, otherwise the global default number of decimal places will be used (as specified by fp_decs()).

Returns: Returns the result of adding *number1* to *number2*

-o-

fp_sub

Synopsis:
answer = fp_sub(number1, number2[, decimals])

Arguments:

number1 - The first floating point number
number2 - The second floating point number
[decimals] - Optional argument to specify the number of decimal places of the returned result

Description: This function subtracts *number2* from *number1* and returns the result. If the optional *decimals* argument is specified then the result will be returned with this number of decimal places, otherwise the global default number of decimal places will be used (as specified by fp_decs()).

Returns: Returns the result of subtracting *number2* from *number1*

-o-

fp_mul

Synopsis:
answer = fp_mul(number1, number2[, decimals])

Arguments:

number1 - The first floating point number
number2 - The second floating point number
[decimals] - Optional argument to specify the number of decimal places of the returned result

Description: This function multiplies the two floating pount numbers specified by *number1* and *number2* and returns the result. If the optional *decimals* argument is specified then the result will be returned with this number of decimal places, otherwise the global default number of decimal places will be used (as specified by fp_decs()).

Returns: Returns the result of multiplying *number1* and *number2*

-o-

fp_div

Synopsis:
answer = fp_div(number1, number2[, decimals])

Arguments:

number1 - The first floating point number
number2 - The second floating point number
[decimals] - Optional argument to specify the number of decimal places of the returned result

Description: This function divides *number1* by *number2* and returns the result. If the optional *decimals* argument is specified then the result will be returned with this number of decimal places, otherwise the global default number of decimal places will be used (as specified by fp_decs()).

If a divide by zero error is encountered then the function will return the string "ERROR" and an error message will be written to the error log.

Returns: Returns the result of dividing *number1* by *number2* or the string "ERROR" if a divide by zero error is encountered.

-o-

fp_pow

Synopsis:
answer = fp_pow(number,power[, decimals])

Arguments:
number - The number to raise to a power
power - The power by which to raise the number
[decimals] - Optional argument to specify the number of decimal places of the returned result

Description: Returns the *number* raised to the power specified by *power* (sometimes written x^y or x**y).

In the event of an error, e.g. fp_pow(-1, "0.5"), the function will return the string "ERROR" and an error message will be written to the error log file.

This function has an optional additional argument specifying the number of decimal places used for rounding (*decimals)*. The initial default number of decimal places used for rounding is 2, this default value may be changed by calling fp_decs(n) where n is 0 to 9 specifying the number of decimal places. The current default is over-ridden by specifying an explicit number of decimals.

Returns: Returns the result of raising *number* by the power specified by *power* or else the string "ERROR" if invalid values are specified (E.g. trying to take the square root of a negative number)

-o-

fp_rnd

Synopsis:
answer = fp_rnd(number, type, digit[, decimals])
Arguments:
number - The number to round
type - How to round (0=Truncate, 1=Round up, 2=Round nearest)
Digit - The digit position to round (+ for before decimal place, - for after decimal place)
[decimals] - Optional argument to specify the number of decimal places of the returned result

Description: This function allows for a floating point number to be rounded to a certain number of places. The number can be rounded up, down or truncated depending on the value specified in the *type* argument. *Type* can be specified as one of the following:

0 truncate (or round down)
1 round up
2 round up or down based on whether the least significant digit is less than or greater than 5.

The *digit* argument specifies the digit position to round to. If a positive value is given for *digit* then this specifies a digit position to the left of the decimal place. For example using the *type* truncate :

fp_rnd("1234.567",0,3) returns "1000.000"
fp_rnd("1234.567",0,2) returns "1200.000"
fp_rnd("1234.567",0,1) returns "1230.000"

If a negative value is given for *digit* then this specifies a digit position to the right of the decimal place:

fp_rnd("1234.567",0,-3) returns "1234.567"
fp_rnd("1234.567",0,-2) returns "1234.560"
fp_rnd("1234.567",0,-1) returns "1234.500"

This function has an optional additional argument specifying the number of decimal places used for rounding. The initial default number of decimal places used for rounding is 2, this default value may be changed by calling fp_decs(n) where n is 0 to 9 specifying the number of decimal places. The current default is over-ridden by specifying an explicit number of decimals.

Returns: Returns the *number* rounded as specified.

-o-

Sockets Library

Introduction

The Sockets Library (**CXSOCK.DLL**) provides the functionality to make TCP/IP socket connections and send and receive data over those sockets. It also provides connectionless data exchange through Datagram functionality.

-o-

socket=Sconnect(addr,port[,timeout_10ths])
Sclose(socket)
Srecv (socket, no_bytes [,timeout_10ths[,&pData_buf]]);
socket=Slisten(port)
socket=Saccept(lsock,[timeout_10ths,&pAddr,&pPort])
Ssend (socket, data, no_bytes, [, data1[,data2...]])
flag=Scheck(socket,rd_wr_err)
hostname=Shostname()
socket=SopenDGRAM(port)
SsendDGRAM(sock,addr,port,data,no_bytes,[data1[,data2...])
SrecvDGRAM (socket, no_bytes, pAddr, pPort ,timeout_10ths ,&pData_buf)
Strace(on_off)

-o-

Sockets Function Reference

Sconnect

Synopsis:
socket=Sconnect(addr,port[,timeout_10ths])

Arguments:
address - The IP address to connect to
port - The IP port to connect to
[timeout_10ths] - Optional timeout (in 10th seconds) to wait for the connection to complete

Description: This function will attempt to make a TCP connection to the IP *address* and *port* specified as arguments. The IP address can be specified either in dot notation (192.168.2.1) or as a named address. The function looks at the first character of the address and if this character is numeric then it will assume that the address is in dot notation, otherwise it will assume it is a named address.

If the optional argument *timeout_10ths* is specified then the function will block and will not return until the timeout expires or the connection completes and will return a handle to the connecting *socket* (or a negative error value).

If the *timeout_10ths* argument is not specified then the function will always return immediately with the connecting *socket* handle (or a negative error value).

Note that the connecting *socket* handle will not be fully useable for sending or receiving data until the socket connection has completed. If an attempt is made to send or receive on a socket that has not yet fully connected then the function will return -3 (EWOULDBLOCK).

The programmer can use Scheck() to check if the socket is writable or in error as a way of checking if the connection has completed or has failed to connect.

Example:

```
// Attempt to make a socket connection
socket=Sconnect("google.com",80);

// Loop waiting for connection or error
while(1)
      // Check if socket is ready for write (connection complete)
      x=Scheck(socket,1);
      if(x eq 1)
            break;
      endif

      // Check for error
      x=Scheck(socket,2);
      if(x eq 1)
            errlog("Error could not connect to google.com");
            stop;
      endif
      // prevent tight loop to allow windows to receive messages..
      sleep(1);
endwhile

// If we get here we are connected...
etc
```

Returns: The function will return the connecting socket handle (which is used in subsequent calls to socket functions) or else a negative error value.

-o-

Sclose

Synopsis:
Sclose(socket)

Arguments:
socket - The socket handle

Description: This function closes a sosket handle previously created by a call to Sconnect(), Slisten() or Saccept().

Returns: 0 if successful or a negative error value.

-o-

Srecv

Synopsis:

Srecv (socket, no_bytes [,timeout_10ths[,&pData_buf]]);

Arguments:

socket - The socket handle
no_bytes - The number of bytes to read from the socket
[timeout_10ths] - Optional timeout in 10ths seconds
[pData_buf] - Optional pointer to a variable to hold the returned data..

Description: This function is used to receive data on the given socket. The *no_bytes* argument specifies the maximum number of bytes to read from the socket. If there is less than *no_bytes* received waiting to be received on the socket then the function will return these bytes.

If the optional *timeout_10ths* argument is specified then the function will block until the speicifed number of bytes has been read or the timeout has expired (in which case the function will return -3). A timeout value of 0 will cause the function to return immediately (which is the default value for this optional argument). A negative timeout value will cause the function to

The data received on the socket is usually returned by the function, however this can sometime cause confusion if the data received is a string like "-3", which would be confused with the return value "-3" (EWOULDBLOCK). Therefore it is better to pass a pointer to a variable that will receive the data using the *pData_buf* argument. If this argument is given then all data received on the socket will be placed in the variable pointed to by this argument and the function itself will return 0 when some data has been received or a negative error value.

Below is an example where one character at a time is received from the socket and added to a string until a *newline* character is received.

```
// Keep reading from socket until we receive a newline character
while(1)
    var char:1;
    var msg:255;
    int x;

    x=Srecv(socket,1,0,&char);
    // -3 indicates no character is waiting
    if(x eq -3)
        sleep(1);
        continue;
    // Any other negative value is an error
    else if(x < 0)
        errlog("Error reading socket: err=",x);
        stop;
    endif endif

    // if we get here then we have received a character

    // append it to message variable (unless it is newline)
    if(char streq "`n")
        // newline indicates end of message so break loop
        break;
    endif
```

```
        // append character to message string
        msg=msg&char;
    endwhile
```

Returns: If pData_buf is given then the function will return 0 if successful or a negative error value (and the received data is place in the variable pointed to by pData_buf). If pData_buf is not given then the function will return the received data on the socket or a negative error code. In particular -3 (EWOULDBLOCK) is returned if there is less than the number of bytes specified by *no_bytes* waiting on the socket.

.

-o-

Slisten

Synopsis:

socket=Slisten(port)

Arguments:

port - The port to listen on

Description: This function returns a listening socket on the specified *port*. Only one socket can be listening on a particular port at any one time and if a second call to Slisten() is made on the same port then it will return an error. The socket returned by this function can be used in the Saccept() function to wait for inbound socket connections on the port.

For example:

```
    // Get a listening socket on port 5000
    Lsocket=Slisten(5000);

    // Loop waiting for inbound connections on this port
    while(1)
        Asocket=Saccept(Lsocket);
        // The above function will return -3 is there are no waiting inbound connections
        if(Asocket eq -3)
            sleep(1);
            continue;   //  keep polling
        // Also check for error
        else if(Asocket < 0)
            errlog("Error in Saccept():  err=", Asocket);
            stop;
        // else we must have an inbound connection request...
        else
            break;
        endif endif
    endwhile
```

Returns: A socket for the inbound connection or a negative error code (in particular -3 indicates that there is no inbound connection waiting yet).

-o-

Saccept

Synopsis:

socket=Saccept(lsock,[timeout_10ths,&pAddr,&pPort])

Arguments:

lsock - The listening socket to accept connections on
[timeout_10ths] - Optional timeout (in1/10 secs) to block waiting for connection
[pAddr] - Optional pointer to a variable that will hold the IP address of connecting client
[pPort] - Optional pointer to a variable that will hold the IP port of connecting client

Description: This function attempts to accept an incoming connection on a listening socket created by Slisten(). If there is are no inbound connections waiting then the function returns -3 (EWOULDBLOCK) which allows the function to be used to poll for incoming connections.

If the optional *timeout_10ths* argument is specified then the function will block until either a connection request is received or the timeout period expires (after which it will return -3).

The variables pointed to by the optional *pAddr* and *pPort* argument will hold the IP address and port of the connecting client if a connection is made.

The socket handle returned from this function (>=0) should then be used in subsequent calls to Srecv(), Ssend(), Scheck() etc.

An alternative way of polling for connections on a listening socket is to use Scheck() to check for read capabillity on the listening socket. As soon as the listening socket becomes readable then this indicates that the is an inbound connection request and a call to Saccept() is sure to succeed.

Returns: Returns the handle to an inbound socket connection (>= 0), or -3 (EWOULDBLOCK) if there were no inbound connections waiting, or any other negative value indicates and error.

-o-

Ssend

Synopsis:

Ssend (socket, data, no_bytes, [, data1[,data2...]])

Arguments:

socket - The socket handle
data - The data string to send
no_bytes - The number of bytes to send
[data1[,data2..]] - Optional additional data to send if data to send exceeds maximum Telecom Engine string length

Description: This function allows the specified number of bytes (*no_bytes*) of *data* to be sent over the specified *socket* handle. The data to send is passed in the *data* variable. However if

more data needs to be sent than can be fit into a single Telecom Engine string variable then addtional data can be sent by specifying the optional *[data1[,data2...]* arguments. These additional data arguments will be simple appended to the first *data* argument and sent out in a single send command. Note however that the same effect can be achieved by issuing multiple Ssend() commands.

If the socket is not currently ready to send data (for example if the send buffer is full) then the function will return -3 (EWOULDBLOCK). A call to Scheck() can be made prior to the Ssend() call to check whether the socket is ready for writing.

Returns: Returns 0 upon success or a negative error code.

-o-

Scheck

Synopsis:

flag=Scheck(socket,rd_wr_err)

Arguments:

socket - Socet handle
rd_wr_err - Specifies whether to check for read, write or error (0,1 or 2) on the socket

Description: This function allows a *socket* handle to be checked to see if it is ready to be read from, written to or has an error condition. The *d_wr_err* argument should be set to one of the values: 0, 1 or 2 to check for read, write or error repsectively. The Scheck() function will return 1 if the specified condition has been found, otherwise the function will return 0.

This function is usually used to poll a socket to check that the socket has reached a certain state before continuing to carry out an operation on the socket.

For example, after a call to Sconnect() the socket can be polled by checking for *write* capability which will indicate that the socket has successfully connected to the far end, or otherwise the socket can be checked for *error:*

```
socket=Sconnect("google.com",80);

// Loop waiting for socket connection or error
while(1)
    const CHK_WRITE=1;
    const CHK_ERROR=2;
    if(Scheck(socket,CHK_WRITE)
        applog("Socket connected..");
        break;
    else if(Scheck(socket,CHK_ERROR)
        errlog("Error connecting to google.com");
        stop;
    endif endif

    // Prevent tight loop to allow windows messages to be processed
```

```
        sleep(1);
    endwhile
```

Also after a listening socket has been created by Slisten() the socket can be polled for *read* capability which will indicate that an inbound connection request has been received.

Returns: Returns 1 if the specified condition (read, write or error) has been met on the given socket, otherwise returns 0. A negative error code is returned if an invalid socket is given.

-o-

Shostname

Synopsis:
hostname=Shostname()

Arguments:
NONE

Description: This function returns the name of the local host machine.

Returns: Returns the name of the machine upon which the program is running on.

-o-

SopenDGRAM

Synopsis:
socket=SopenDGRAM(port)

Arguments:
port - The specified port on which to receive datagrams

Description: This function opens a connectionless datagram socket on the specified port. After making this call the *socket* handle returned will be able to send and receive inbound datagram messages on that port.

Returns: Returns the socket handle or a negative error code.

-o-

SsendDGRAM

Synopsis:
SsendDGRAM(sock,addr,port,data,no_bytes,[data1[,data2...])

Arguments:

socket - The socket handle
addr - The IP address to send data to
port - The IP port to send data to
data - The data string to send
no_bytes - The number of bytes to send
[data1[,data2..]] - Optional additional data to send if data to send exceeds maximum Telecom Engine string length

Description: This function allows for a connectionless datagram message to be sent to the specified *addr* and *port.* The data to send is passed in the *data* variable and the number of bytes to send is specified by the argument *no_bytes*. However if more data needs to be sent than can be fit into a single Telecom Engine string variable then addtional data can be sent by specifying the optional *[data1[,data2...]* arguments. These additional data arguments will be simple appended to the first *data* argument and sent out in a single send command. Note however that the same effect can be achieved by issuing multiple SsendDGRAM() commands.

Note that datagrams do not guarantee that the data will be delivered to the far end so should only be used over a reliable network for non-essential message (or else an underlying protocol (for example using acknowledgements and timeouts) should be implemented).

Returns: Returns 0 on success or negative error code. Not that a 0 return value only indicates that the data was sent succesfully, it does not gaurantee that the data will get to the other end.

-o-

SrecvDGRAM

Synopsis:

SrecvDGRAM (socket, no_bytes, pAddr, pPort ,timeout_10ths ,&pData_buf)

Arguments:

socket - The socket handle
no_bytes - The maximum number of bytes of data to receive
pAddr - The IP address of the sending party
pPort - The IP port of the sending party
timeout_10ths - The timeout (in 1/10th seconds) to wait for a message to arrive
[*&pData_buf]* - Optional pointer to a variable that will hold the received datagram

Description: This function allows datagram messages to be received up to the number of bytes specified by the *no_bytes* argumnent. If there is less than *no_bytes* of data waiuting then the function will retrieve this data The *socket* handle specifies a datagram socket handle opened using SopenDGRAM(). If a timeout is specified in the *timeout_10ths* argument then the function will block until some data has been received or the timeout expires (After which it would return EWOULDBLOCk(-3)). If a zeero timeout is specified then the function will return immediately with EWOULDBLOCK if there is no data available. If a negative timeout is specified then the function will wait indefinitely for data to arrivet the socket.

If the optional argument *pData_buf* is specified then all received data will be returned in the

variable pointed to by this argument (and the function will return 0 to indicate that data has been received. If the *pData_buf* argument is not specified the function will return the data received as the return value.

Returns: If *pData_buf* is specified then the function will return 0 if data was received or -3 (EWOULDBLOCK) if there was no data waiting, or a negative error value.

If *pData_buf* is not specified then the function will return the actual data received or or -3 (EWOULDBLOCK) if there was no data waiting, or a negative error value.

-o-

Strace

Synopsis:
Strace(on_off)

Arguments:
on_off - Set to 0 to switch trace off or 1 to switch trace on

Description: This function is for diagnostic and debugging purposes and causes diagnostiv trace messages to be written to the Telecom Engine trace log

Returns: Always returns 0

-o-

Aculab E1/T1 Card Library

Introduction

There are two Telecom Engine libraries that provide access to the functionality of the Aculab API. CXACULAB.DLL provides the call control and switching capabilities and the CXACUDSP.DLL provides the functionality for the Prosody speech and digital signal processing capabilities.

This section describes the call control and switching library (CXACULAB.DLL).

-o-

The ACUCFG.CFG Configuration file

When the Telecom Engine loads the CXACULAB.DLL (and CXACUDSP.DLL) libraries upon start-up, it first looks for the presence of a configuration file called ACUCFG.CFG.

This file tells the CXACULAB.DLL (and CXACUDSP.DLL) which Aculab cards to open and which ports/modules to open on those cards.

The location of the ACUCFG.CFG file can be defined by the environment variable called ACUCFGDIR (set it to a directory path: e.g. SET ACUCFGDIR=C:\TelecomEngine\Node08\config"). If the ACUCFGDIR environment variable is not set then the library will look in the current directory for the ACUCFG.CFG file.

If the ACUCFG.CFG file is not found then the boards are opened in the order that they are found in the Aculab Configuration Tool (ACT), and which is the order returned by the *acu_get_system_snapshot*() function, and all ports or modules found on those boards will be opened in sequential order.

It should be noted that most of the call control functions in the CXACULAB.DLL library take a *port* number and a *channel* number as the first two arguments.. E.g. CCenablein(port,channel). The *port* number specified here is a *logical port* number where the first *logical port* in the system is port 0 and then increases sequentially through ports 1, 2, 3 .. etc for every other port opened in the system (defined by the order that they are opened).

For example if there were two Prosody X cards in a system, each with 8 E1 ports and two IP ports, then if all of these ports were opened the *logical port* numbers would range from 0 throught to 9 for the first card (including the VOIP ports 8 and 9), then *logical ports* 10 through to 19 for the second card.

Note: For Speech cards, the speech channels opened upon startup are numbered sequentially from 1 up to the number of speech channels in the system (E.g. if there were two Prosody Speech modules of 150 channels each then the logical channel numbers (as used by the SMxxxx(vox_chan,...) functions) would range from 1 through to 300).

The entries in the ACUCFG.CFG must always start with a ***board=<serial number>*** statement, then be followed by one or more of the following statements (comments are preceded by a # character):

The **ports** statement:

ports=<port no. 1[:no.chans]>[,<port no. 2[:no.chans]>[,port no. 3...]]

This specifies which of the physical ports on the card to open (and in which order). For example:

```
# Open four E1 ports of first aculab card
board=192821
ports=0,1,2,3

# Open four E1 ports of second aculab card
board=192822
ports=0,1,2,3
```

The above would result in eight *logical port* numbers being created at startup (0..7) for the eight ports opened across the two Aculab cards specified. It is also possible to specify the number of channels to open on each port, but if this is omitted then the port will be opened with the default number of channels (31 for E1 ports, 30 for IP ports), although for E1 channels the *valid_vector* field in the port info parameter will also be used to define which channels are bearer channels and which channels are signalling channels.

For E1 ports its usually best not to specify the number of channels directly unless you specifically want to allocate less channels that the full number. For example the following will open two

ports on the card, but will only allocate the first 15 channels on these ports:

```
# Open four E1 ports of first aculab card
board=192821
ports=0:15,1:15
```

For IP ports on Prosody X cards the default number of channels to allocate is 30 if it is not explicitly specified. In the following ACUCFG.CFG a single Prosody X card has 8 E1 ports and 2 IP ports. Only the first IP port is opened here and 150 IP channels are allocated to it:

```
board=192799
ports=0,1,2,3,4,5,6,7,8:150
```

It is possible to skip one or more physical ports or rearrange the order that they are opened if required:

```
# Open four E1 ports of first aculab card
board=192821
ports=0,1,3

# Open four E1 ports of second aculab card
board=192822
ports=0,2,3,1
```

In the above example only seven *logical ports* are created (0..6) and on the second board the order that the physical ports are to be opened has been changed (physical ports 0,2,3,1 on the second card would map to *logical ports* 3,4,5,6 in this example).

The **modules** statement:

modules=<module no. 1[:no. channels]>[,<module no. 2[:no channels]>[,...]]

This specifies which DSP modules to open on the card (and in which order), plus the number of channels on the module to allocate can optionally be specified (otherwise all the channels on the module will be allocated (150 for Prosody X modules, 60 for older Prosody modules).

For example the following will open a single board with 4 E1 ports and 150 channels of speech (numbered from 1 to 150):

```
# Open four E1 ports and DSP module 0 (which has 150 channels)
board=192823
ports=0,1,2,3
modules=0:150
```

On a Prosody X board the default number of channels to open on a module is 150 so this doesn't necessariry need to beed specified. Below is the equivalent ACUCFG.CFG file to the above:

```
# Open four E1 ports and DSP module 0 (which has 150 channels)
board=192823
ports=0,1,2,3
modules=0
```

If there are multiple modules on a board then these can be specified in the order that they are to be opened:

```
# Open four E1 ports and DSP module 0 (which has 150 channels)
```

```
board=192823
ports=0,1,2,3
modules=0:150,1:150

board=192824
ports=0,1,2,3
modules=0:60
```

The above will open modules 0 and 1 on the first board (allocating 300 channels number 1 to 300), then will open module 0 on the second card, allocating 60 channels (which will be numbered 301 to 360).

The **ipports** statement:

ipports=<port 1 no. IP channels[:port 1 type]>[port 2 no. IP channels[:port 2 type]>[,...]]]

For Prosody S it is necessary to open system-wide IP port(s) to handle VOIP calls and the above statement provides the means to specify these ports. The ***ipports*** statement must follow the definition for a Prosody S type board (with serial number HS_PROSODYS) as follows:

```
# open Prosody S board
board=HS_PROSODYS
modules=0:150
ipports=30:S,30:S
```

If you specify an ***ipports*** statement for a non-prosody S board then it will be ignored. The ***port type*** parameter specifies the number of IP channels to allocate on the IP port and may be followed by an optional port type specifier where **S** is for a SIP port and **H** is for a H.323 port. By default the IP port is opened as a SIP port.

So in the above example the Prosody S card is opened and two system-wide IP ports are opened (one for SIP, one for H323) with 30 IP channels each. Note that with Prosody S boards there is a single **module (**which is actually an on-host media DSP) and the number of channels that can be allocated will depend upon the processing power of the host CPU.

When they are opened, the IP ports will be given a *logical port* number which is then used in the call control functions just like for normal E1 ports. In the above example two *logical ports* are opened (0 and 1) for the two IP ports specified.

In the example below the IP ports will be allocated logical port numbers 4 and 5 since there are also 4 E1 ports specified on the first Aculab card which will be allocated ports 0,1,2 and 3:

```
# open 4 E1 ports on first card
board=192821
ports=0,1,2,3

# open Prosody S board
board=HS_PROSODYS
modules=0:150
ipports=30:S,30:S
```

Below are some examples of ACUCFG.CFG files for typical purposes:

The following could be used for the Prosody S evaluation license which provides 4 channels of media and IP processing for up to 45 days:

```
# Prosody S board
board=HS_PROSODYS
modules=0:4
ipports=4:S
```

The following could be used for on-host Prosody S with licence for 150 media and IP ports

```
# Prosody S board
board=HS_PROSODYS
modules=0:150
ipports=150:S
```

The following opens a single prosody X card with 8 E1 ports plus the H323 port with 150 IP channels

```
# Prosody X board
board=192876
modules=0:150
ports=0,1,2,3,4,5,6,7,9
```

The following opens a single E1 card with two E1 ports, plus an old style Prosody speech card with a single DSP module:

```
# E1 card
board=178912
ports=0,1

#Prosody DSP card
board=165444
modules=0:60
```

-o-

Run-time Initialisation and configuration

Upon start-up the CXACULAB.DLL library opens and initialises the Aculab call control boards ready to make and receive calls. The order that the boards are opened is specified by the ACUCFG.CFG configuration file, or if this doesn't exist then the boards are opened in the order returned by the *acu_get_system_snapshot*() function. See ACUCFG.CFG configuration for a full description of this file.

The format of ACUCFG.CFG file is a text file that contains a set of statements that provides the list of Aculab board serial numbers to open, plus information about which ports or modules to open on each board.

The order that the call control boards are opened is important since it defines the logical port number that is used in many of the calls to the CXACULAB.DLL functions. Port numbers relate to the E1 or T1 port on the boards and is numbered from zero to one less than the number of E1/T1 ports opened across all the boards in the system.

For example if the system contains two boards each with 4 E1s then there will be a total of 8 E1

ports (numbered 0 to 3 and 4 to 7). The order that the serial numbers appear in the ACUCFG.CFG file defines which of these boards has the ports numbered 0 through 3 and which will have the ports numbered 4 through 7.

Most of the call control functions take the port number and the channel number as the first two parameters to the function call (for example. *CCtrace(port,channel,tracelevel) , CCabort(port,channel) etc)..*

The channel represents the channel on the port ranging from 1 to 31 (depending on the protocol being used and the number of signalling channels etc).

Once the boards are opened then the board capabilities are examined and any boards that have switching capabilities will have their transmit channels 'nailed' to the external H.100 or SCBUS. This provides a consistent method for switching between channels and in the current version of the library even channels on the same board will be switched through the external H.100 or SCBUS.

H.100 timeslots are defined by both a stream number and a timeslot number where each stream can have up to 128 timeslots. For the SCBUS there is only one stream and the timeslots range from 0 up to 4096. To create a consistent way of referencing these timeslots whether the external bus is a H.100 bus or an SCBUS the CXACULAB.DLL generates a logical handle which is calculated from the stream and the timeslot as follows:

*handle = stream * 4096 + timeslot*

For the SCbus the *stream* is always 24 which is the internal fixed stream that is used by the ACULAB firmware when SCBUS is present.

This handle is used/returned by the switching functions listed below.

handle=CCgetslot(port,channel);
x= CClisten(port,channel,handle);
x= CCunlisten(port,channel);

By default the first channel on the first logical port will be 'nailed' to stream 0, timeslot 0 of the H100 bus (or just timeslot 0 of the SCBUS). If there is other non-Aculab hardware in the system that is using these stream/timeslot ranges, or there is some other reason why a different stream/timeslot range should be used for nailing the transmit channels to the external bus then the environment variable ACUCC_TSOFFS can be set to define the start stream and timeslot offset to nail to.

This variable should be specified in the same form as defined above for the stream/timeslot handle. For example if you want to start nailing the call control channel starting at stream 64, timeslot 0 then you would set the environment variable as follows:

*REM set offset to: 64 * 4096 + 0*
SET ACUCC_TSOFFS=262144

N.B. The current version of the library does yet transparently support multi-chassis switching for Prosody-X functionality (although the programmer has access to the RTP functions and can therefore implement their own multi-chassis switching capability). The next version of the

library will include a transparent method for multi-chassis switching consistent with above function calls and methodology.

-o-

Some Simple Examples

Probably the best way to show the basic library functions and the library calling conventions is to provide a simple example. The example below simply waits for an incoming call on the first channel of the first E1 port, plays a message and then hangs up. It is assumed that the reader is familiar with the Telecom Engine standard library set and the Aculab Speech module library (CXACUDSP.DLL).

```
$include "aculab.inc"

int port, chan, vox_chan, x, event;
var filename:64;

main
   port=0
   chan=1;
   vox_chan=1;
   filename="hello.vox";

   // Make fill duplex H.100 bus routing between voice channel and network port/channel
   CClisten(port,chan,SMgetslot(vox_chan));
   SMlisten(vox_chan, CCgetslot(port, chan));

   // Enable inbound calls on this port/channel
   CCenablein(port,chan);

   // loop waiting for incoming call
   while(1)
       x=CCwait(port,chan,CC_WAIT_FOREVER,&event);
      if(x > 0 and event eq CS_INCOMING_CALL_DETECTED)
         break;
      endif
   endwhile

   // Answer the call
   CCaccept(port,chan);

   // Play a vox file to caller
   SMplay(vox_chan,filename);

   // Hangup the call
   CCdisconnect(port,chan,CAUSE_NORMAL);

  // Wait for state to return to IDLE the release call
  while(1)
     x=CCwait(port,chan,CC_WAIT_FOREVER,&event);
    if(x eq CS_IDLE)
        break;
    endif
   endwhile

   // release the call
   CCrelease(port,chan);

   // restart the application to wait for another call..
   restart;

endmain
```

The program should be fairly self explanatory but I will describe the key parts of the program below.

The "aculab.inc" file is provided with the library and defines all the constants that are used with the library such as CC_WAIT_FOREVER, CAUSE_NORMAL, CS_INCOMING_CALL_DETECTED etc. These are be described in more detail in the call control library function library reference (CXACULAB).

The first call: CClisten() simply makes the receiving stream/channel of the call control channel ' Listen' to the transmit stream/channel of the Voice channel, so that anything that is output by the voice channel will be heard by the caller.

The second call: SMlisten() makes the receiving stream/channel voice channel 'listen' to the transmit stream/channel of the Call control channel, so that any DTMF digits or other audio transmitted by the caller will be heard by the voice channel.

As mentioned above all this is done by switching from and to the extern H.100 or SCBUS.

The *CCenablein(port,channel)* allows inbound calls to be received on the channel, and then the application goes into a loop waiting for calls.

The *CCwait(port,channel,timeout_100ms,&event)* function call will wait for the specified timeout (in 10ths of a second) for an event. If the timeout is defined as -1 (CC_WAIT_FOREVER) then the call will not return until an event is found or it is aborted by a *CCabort()* call. Really the only event that should be received here is CS_INCOMING_CALL_DETECTED but we do a specific check for it anyway in case the channel was in a unknown state when the program started (probably some error handling should be carried out if we found an unexpected event).

Once a CS_INCOMING_CALL_DETECTED event has been received then the call is answered immediately with *CCaccept(port,channel)* and the voice prompt is played to the caller using the *SMplay(vox_chan,filename)* function from the CXACUDSP.DLL library.

The call is then disconnected using the *CCdisconnect(port,channel)* call and the application goes into a loop waiting for the channel to return to the CS_IDLE state before releasing the call with *CCrelease(port,channel)* and restarting the program to wait for the next call.

There are obviously a number of improvements that can be made to this application to make it more useful. Currently it only waits for and accepts a call on one channel, whereas in a real life situation there would be 30 or more channels on an E1.

Usually, one would have a 'master' program which would 'spawn' a task to take control of a single channel on a port. In the application below there are 4 E1 ports and so we 'spawn' a channel control task for each channel on each E1, something like this:

```
int port, channel;
const MAX_PORTS=4
const MAX_CHANNELS=32;

main

   for(port=1;port <= MAX_PORTS)
      for(channel=1;channel <= MAX_CHANNELS;channel++)
       // Skip the signalling channel
       if(channel <> 16)
          // spawn the task called chantask.tex
          // and pass the port and channel as arguments
          task_spawn("chantask",port,channel);
       endif
     endfor
   endfor
```

© Zentel Telecom Ltd, 2009

```
endmain
```

The chantask.tex application would then be similar to the first example but instead of the *port* and *chan* variables being hard-coded, we would instead take these from the arguments passed to the task through the *task_spawn()* function call:

```
$include "aculab.inc"

int port, chan, vox_chan, x, event;
var filename:64;

main
   port=arg(1);
   chan=arg(2);

   // Use voice channels sequentially 1..x
   vox_chan=1+(port*32+chan);
   filename="hello.vox";

   // Make fill duplex H.100 bus routing between
   // voice channel and network port/channel
   CClisten(port,chan,SMgetslot(vox_chan));
   SMlisten(vox_chan, CCgetslot(port, chan));

   // Enable inbound calls on this port/channel
   CCenablein(port,chan);

   // loop waiting for incoming call
   while(1)
       x=CCwait(port,chan,CC_WAIT_FOREVER,&event);
      if(x > 0 and event eq CS_INCOMING_CALL_DETECTED)
         break;
      endif
   endwhile

  ... etc

endmain
```

Also it is likely that rather than playing a fixed prompt, as in the program above, a more usual way of handling incoming calls is to inspect the DID or ANI and make a decision based on these about how to handle the call (usually a table lookup). Typically this would result in the *chantask.tex* program ‘chaining’ on to another application which then takes control of playing messages and receiving DTMF etc. When the caller hangs up or the application disconnects the call then the application would then ‘chain’ back to the *chantask.tex* program to wait for another call. The *CCgetparm(port,channel,ParmID)* function is used to extract call specific parameters such as the DID and ANI.

```
$include "aculab.inc"

int port, chan, vox_chan, x, event;
var filename:64, did:64,ani:64, service_name:64;

main
   port=arg(1);
   chan=arg(2);

   // Use voice channels sequentially based on the port/channel
   vox_chan=1+(port*32+chan);
   filename="hello.vox";

   // Make fill duplex H.100 bus routing between
   // voice channel and network port/channel
   CClisten(port,chan,SMgetslot(vox_chan));
   SMlisten(vox_chan, CCgetslot(port, chan));

   // Enable inbound calls on this port/channel
   CCenablein(port,chan);

   // loop waiting for incoming call
   while(1)
       x=CCwait(port,chan,CC_WAIT_FOREVER,&event);
```

```
    if(x > 0 and event eq CS_INCOMING_CALL_DETECTED) break; endif
  endwhile

  // Get the CLID and DID
  did=CCgetparm(port,chan,CP_DESTINATION_ADDR);
  ani=CCgetparm(port,chan,CP_ORIGINATING_ADDR);

  // Use the DID to decide which application to chain
  // This usually will be a database lookup based on the DID
  service_name=DID_LOOKUP(did);

  //  A blank service_name indicates unknown DID number
  if(service_name strneq "")
    // Answer the call
    CCaccept(port,chan);

    // Chain on to the IVR service task
    task_chain(service_name,port,channel);

    // If we get here then the chain call failed (ie. Invalid TEX file)
  endif


  // Hangup the call
  CCdisconnect(port,chan,CAUSE_NORMAL);

  // Wait for state to return to IDLE the release call
  while(1)
   x=CCwait(port,chan,CC_WAIT_FOREVER,&event);
   if(x eq CS_IDLE) break; endif
  endwhile

  // release the call
  CCrelease(port,chan);

  // restart the application to wait for another call..
  restart;

endmain
```

-o-

Simple VOIP -> TDM example

To make or receive a VOIP call a Virtual Media Processing channel needs to be created on a DSP module connected to the board which has the VOIP capability (Using SMcreateVMP()). The supported codecs can then be specified for that VMP channel using the SMsetcodec() function.

The usual call control functions can then be used to receive an inbound VOIP call, but before accepting the call the call accept parameters must be setup with the VMP channel so that the media stream (codec) can be processed. This is done using the CCsetparm() functions.

In the example below, once the VOIP call has been received we then make an outbound call over the E1 port and when this answers we create a TDM endpoint so that the E1 stream can be connected to the VMP port in a full duplex connection so that the two parties can talk to each other. This is done through the CCcreateTDM() functions and the SMfeedlisten() function.

Here is a small program showing this functionality:

```
$include "aculab.inc"

int vox_chan, port, chan, vmp_chan, module_id;
int call_state;

main
```

```
 vox_chan=1;
 module_id=0;
 e1port=0;
 e1chan=1;

 ipport=8;
 ipchan=1;

 vmp_chan=SMcreateVMP(module_id);        // Create a VMP channel
 // specify a codec on the vmp channel
  SMsetcodec(vmp_chan,0,G711_ALAW);

 // Keep a track of where we are so that we know how to clear down the call in onsignal
 call_state=0;

 // Wait for an inbound call on IP channel(using the newly created VMP when we accept)
 CCenablein(ipport,ipchan);
 CCuse(ipport,ipchan);    // Hangup will cause jump to onsignal
while(1)
        x=CCwait(ipport,ipchan,WAIT_FOREVER,&state);
        if(state eq CS_INCOMING_CALL_DET)
          call_state=1;    // need to clear down ip call
            CCalerting(ipport,ipchan);   // send INCOMING_RINGING event
        else if(state eq CS_WAIT_FOR_ACCEPT)

           // specify a codec on the vmp channel
             SMsetcodec(vmp_chan,0,G711_ALAW);

            // Set the VMP and CODEC array into the call accept parameters..
             CCclrparms(ipport,ipchan,PARM_TYPE_ACCEPT);
                CCsetparm(ipport,ipchan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPRXID,vmp_chan);
                CCsetparm(ipport,ipchan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPTXID,vmp_chan);
                CCsetparm(ipport,ipchan,PARM_TYPE_ACCEPT,CP_IPTEL_CODECS,vmp_chan);
           CCaccept(ipport,ipchan);
        break;
     endif endif
 endwhile

 task_sigctl("(");    // Temporarily prevent hangup from causing jump to onsignal

 // Create the TDM endpoint for the E1 port
 tdm_chan=CCcreateTDM(e1port,e1chan);    // Get a TDM endpoint for an E1 channel

 // ** Now connect the feeds from the VMP and TDM endpoint to make
 // ** a full duplex connection to connect the conversations from the IP to E1 calls
 // ** we do this here so that VOIP call will hear call progress tones..

 // The vmp_chan will now listen to the tdm_chan datafeed
  SMfeedlisten(vmp_chan,TYPE_VMP,tdm_chan,TYPE_TDM);
 // The tdm_chan will now listen to the vpm_chan datafeed
  SMfeedlisten(tdm_chan,TYPE_TDM,vpm_chan,TYPE_VPM);

 call_state=2;    // need to clear down ip call and destroy TDM
 task_sigctl(")");    // Allow hangup to cause jump to onsignal again..

 // hangup on E1 port will cause jump to onsignal
 CCuse(e1port,e1chan);
 call_state=3;    // need to clear down both IP and E1 calls
 // Now make outbound call on E1 port/channel
 x=CCmkcall(e1port,e1chan,"123456"."987654");
while(1)
        x=CCwait(e1port,e1chan,WAIT_FOREVER,&state);
        if(state eq CS_OUTGOING_RINGING)
            applog("Outgoing ringing")   // send INCOMING_RINGING event
```

© Zentel Telecom Ltd, 2009

```
          else if(state eq CS_CALL_CONNECTED)
           break;
         endif endif
    endwhile

       // We just loop here waiting for a hangup from either side which will cause jump to onsignal
     while(1)
          sleep(40);   // four second sleep
      endwhile
endmain

onsignal

    applog("We are in ONSIGNAL!!!!");

    // Release the VMP channel
     SMdestroyVMP(vmp_chan,VMP_TYPE);

    // do we need to clear down the inbound IP call?
   if(call_state)
        CCdisconnect(ipport,ipchan,LC_NORMAL);
      if(call_state >=2)
             SMdestroyTDM(tdm_chan);
      endif

      last_state=-1;
      while(1)
         # wait for idle
         applog("CCIN: in onsig CCwaiting for CS_IDLE");
         x=CCwait(ipport,ipchan,10,&state,last_state);
         applog("CCIN: CCwait returned x=",x," state=",state);
          if(state eq 0)
               applog("ipport=",ipport," ipchan=",ipchan," Incoming went to IDLE");
              CCrelease(ipport,ipchan);
             break;
         endif
         task_sleep(1);
      endwhile
   endif

   // do we need to clear down the outbound E1 call?
  if(call_state >=3)
        CCdisconnect(e1port,e1chan,LC_NORMAL);
      last_state=-1;
      while(1)
         # wait for idle
         applog("CCIN: in onsig CCwaiting for CS_IDLE");
         x=CCwait(e1port,e1chan,10,&state,last_state);
         applog("CCIN: CCwait returned x=",x," state=",state);
          if(state eq 0)
               applog("e1port=",e1port," ipchan=",e1chan," Incoming went to IDLE");
              CCrelease(e1port,e1chan);
             break;
         endif
         task_sleep(1);
      endwhile
   endif

   restart;
endonsignal
```

-o-

Aculab Call Control Quick Reference

num_ports=CCnports();
CCsigtype(port);
CCsiginfo(port,&pBearerMap,&pProtocol);
CCtrunktype(port);
CCwatchdog(port,alarm,timer_ms);
CCalarm(port,alarm);
CCtrace(port, channel, tracelevel);
handle=CCgetslot(port,channel);
CClisten(port,channel,ts_handle);
CCunlisten(port,channel)
state=CCstate(port,chan);
CCuse(port,timeselot[,flag]);
CCwait(port,chan,timeout_100ms,&pState);
CCabort(port,channel);
CCenablein(port,channel[,cnf_parm1[,cnf_parm2...]);
CCaccept(port, channel);
CCmkcall(port,channel,DID,CID[,send_comp[,cnf_parm1[,cnf_parm2....]);
CCdisconnect(port, channel,cause[,std_or_raw]);
CCrelease(port, channel);
CCsetparm(port, channel, parmType, parmId, Value);
CCclrparms(port,channel,ParmType);
CCgetparm(port, channel, parmId, &pVar);
CCalerting(port, channel);
CCgetcause(port, channel[,std_or_raw]);
CCoverlap(port, channel,dest_addr,sending_complete);
CCsetupack(port, channel[,progress,Display]);
CCproceed(port, channel[,unique_hex]);
CCprogress(port, channel[,progress[,Display]]);
CCgetaddr(port, channel);
CCanscode(port, channel,code);
CCputcharge(port, channel,charge[,meter]);
CCgetcharge(port, timeslot,&pType,&pCharge,&pMeter);
CCnotify(port, channel,notify_indicator);
CCkeypad(port, channel,keypadinfo[,display]);
CChold(port, channel);
CCreconnect(port, channel);
CCenquiry(port, channel,DID,CID,sending_complete[,cnf_parm1,cnf_parm2]....);
CCsetparty(port, channel,party);
CCtransfer(Aport, Achannel,Cport,Cchannel);
CCgetxparm(port, channel, parmId, &pVar[,connectionless_flag]);
CCsetxparm(port, channel, parmId, Value[,connectionless_flag]);
CCclrxparms(port,channel[,connectionless=1]);
CCgetcnctless(port);
SMcreatevmp(vox_chan,[local_addr])(port, channel, DID, CID,
sending_complete[,parm1,parm2....]);
CCsendfeat(port, channel);
CCsndcnctless(port);
hexstr=CCstrtohex(string);
hexstr=CCinttohex(unsigned_val,num_bytes);

hexstr=CCunstohex(int_val,num_bytes);

-o-

Aculab Call Control Function Reference

CCnports

Synopsis:
num_ports=CCnports()

Description: This function returned the number of logical E1/T1 ports that have been opened (either by reading the ACUCFG.CFG file or by opening all the boards returned from the *acu_get_system_snapshot*() function.

Returns: This function returns the number of logical ports in the system.

-o-

CCsigtype

Synopsis:
CCsigtype(port)

Arguments:
port – The logical E1/T1 port number.

Description: This function maps to the following Aculab function:

ACU_ERR call_type(ACU_PORT_ID *portnum*);

It will return the signalling type on the specidied port and will be one of the following as defined in the ACULAB.INC file:

```
const S_UNKNOWN = 0;

#- user end definitions-----------------
const S_1TR6 = 1;
const S_DASS = 2;
const S_DPNSS = 3;
const S_CAS = 4;
const S_AUSTEL = 5;
const S_ETS300 = 6;
const S_VN3 = 7;
const S_ATT = 8;
const S_CASTONE = 9;
const S_TNA_NZ = 10;
const S_FETEX_150= 11;
const S_SWETS300 = 12;
const S_IDAP = 13;
const S_T1CAS = 14;
const S_T1CAS_TONE= 15;
const S_NI2 = 16;
const S_DPNSS_EN = 17;
const S_ATT_T1 = 18;
```

```
const  S_QSIG      =  19;
# - user end definitions ------------
const  S_1TR6NET    =  20;
const  S_VN3NET     =  21;
const  S_ETSNET     =  22;
const  S_AUSTNET    =  23;
const  S_ATTNET     =  24;
const  S_DASSNET    =  25;
const  S_TNANET     =  26;
const  S_FETEXNET   = 27;
const  S_SWETSNET   =  28;
const  S_IDAPNET    =  29;
const  S_NI2NET     =  30;
const  S_ATTNET_T1 =  31;
const  S_DPNSS_T1 =  32;
const  S_FETEX_150_T1=33;
const  S_FETEXNET_T1 =34;
const  S_INS_T1      =35;
const  S_INSNET_T1   =36;
const  S_INS         =37;
const  S_INSNET      =38;

const  S_ISUP        =39;
const  S_GLOBAND     =40;
const  S_GLOBNET     =41;
const  S_MON         =42;
const  S_MON_T1      =43;
const  S_QSIG_T1     =44;
const  S_DPNSS_EN_T1 =45;
const  S_ETS300_T1   =46;
const  S_ETSNET_T1   =47;
const  S_H323        =48;
const  S_SIP         =49;
const  S_BR_ETS300   =50;
const  S_BR_NI1      =51;
const  S_BR_ATT      =52;
const  S_BR_INS      =53;

const  S_DMS100   =60;
const  S_DMS1NET     =61;

const  S_BR_ETSNET   =70;
const  S_BR_NI1NET  =71;
const  S_BR_ATTNET   =72;
const  S_BR_INSNET   =73;

const  S_SS5_TONE   =90;

const  S_BASE        =99;
```

Returns: Returns the signalling type or the negative error code returned from the Aculab *call_type*() function.

-o-

CCsiginfo

Synopsis:

CCsiginfo(port,&pBearerMap,&pProtocol);

Arguments:

port – The logical E1/T1 port number.

pBearerMap – Pointer to a variable that will hold the returned bearer channel mask as a 32 character string of 1s and 0s

pProtocol – Pointer to a variable that will hold the returned protocol name

Description: This function returns information about the signalling and bearer channels on the specified *port* as well as the name of the protocol running on the *port*.

The *pBearerMap* is a pointer to a variable that will hold the returned channel/timeslot bit mask which will be returned as a string of 0s and 1s where 1s represent the bearer channels and 0 represent the signalling/timing channels of the port. The rightmost character of the returned string represents bit 0 of the 32 bit vector.

For example, for a port running the Q931 protocol where timeslot 0 is used for clocking and timeslot 16 is used for signalling the returned value for *pBearerMap* would be as follows:

11111111111111101111111111111110

The values returned into the variable pointed to by *pProtocol* will be one of the following:

• ***E1 - ISDN signaling systems***
ETS300 ETSNET FETX150 FETXNET
DASS2 DASSNE DPNSS QSIG
• ***E1 - CAS signaling systems***
R2B2P CAS BTCU BTCN PTVU PTVN PD1D
PD1U PD1N R2L P8 EM BEZEQ
• ***E1 - CAS tone signaling systems - requires DSP***
R2T R2T1 ALSU ALSN BELGU BELGN EFRAT
EEMA PD1 PD1DD PD1UD PD1ND BTMC OTE2
FMFS SMFS I701 SS5
• ***T1 - ISDN signaling systems***
NI2 NI2NET INS_T1 INT1NET ATT1 ATT1NET ETST1U ETST1N
DMS1 DMS1NET
• ***T1 - CAS tone signaling systems - requires DSP***
F12 T1RB
• ***SS7***
ISUP
• ***Passive Monitor***
E1 - MONE
T1 — MONT
• ***IP Telephony***
H323 SIP

Returns: This function returns 0 upon success or a negative error code.

-o-

CCtrunktype

Synopsis:
CCtrunktype(port)

Arguments:
port – The logical E1/T1 port number.

Description: This function maps to the following Aculab function:

ACU_ERR call_line (ACU_PORT_ID *portnum*);

It returns the type of trunk supported by the port and will be one of the following defined in ACULAB.INC:

```
const  L_E1          = 1;
const  L_T1_CAS       = 2;
const  L_T1_ISDN      = 3;
const  L_BASIC_RATE  = 4;
const  L_PSN         = 5;    # packet stream
```

Returns: Returns the trunk type or a negative error code.

-o-

CCwatchdog

Synopsis:
CCwatchdog(port,alarm,timer_ms)

Arguments:
port – The logical E1/T1 port number.
alarm – The ID of the alarm to present.
timer_ms – The timeout in ms before the layer 1 alarm is presented

Description: This function maps to the following Aculab function:

ACU_ERR call_watchdog(WATCHDOG_XPARMS **watchp*);

Once CCwatchdog() has been called with a timer (in milliseconds) and alarm code set, then the application must continue to call CCwatchdog() repeatedly thereafter to keep refreshing the watchdog timer. If CCwatchdog() is not called before the timer expires then the firmware will present the specified layer 1 alarm to the network.

To disable the watchdog timer simply set the timeout to 0.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCalarm

Synopsis:
CCalarm(port,alarm)

Arguments:
port – The logical E1/T1 port number.
alarm – The ID of the alarm to present.

Description: This function maps to the following Aculab function:

ACU_ERR call_send_alarm(ALARM_XPARMS **alarmp*);

It causes a layer 1 alarms to be sent to the network on the specified port. The alarm ID can be one of the following defined in ACULAB.INC:

```
const ALARM_NONE  =0;
const ALARM_AIS   =8192;
const ALARM_RRA   =2048;
const ALARM_CML   =64;
```

Returns: Returns upon success or a negative error code.

-o-

CCtrace

Synopsis:
CCtrace(port, channel, tracelevel)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.
tracelevel – 0 turns trace off, 1 turns trace on.

Description: This function switches on or off the tracing of Call Control events. Trace will be written to the Telecom Engine trace log.

Returns: 0 upon success or -1 if a bad port or channel was provided.

-o-

CCgetslot

Synopsis:
handle=CCgetslot(port,channel)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.

Description: This function returns a handle to the external H.100 or SCBUS transmit timeslot for the *port* and *channel*. The timeslot handle references the *transmit* timeslot of the given *port* and *channel* which has been '*nailed*' to a H.100 or SCBUS timeslot by the library at start-up.
This only applies to boards that have switching capabilities and not to the PROSODY_S type virtual boards which do not have any external switching capability.

The returned handle is actually obtained from the physical H.100 or SCBUS stream and timeslot from the following formula:

*handle = stream * 4096 + timeslot*

For the SCbus the *stream* is always 24 which is the internal fixed stream that is used by the ACULAB firmware when SCBUS is present.

For example, if a board has four E1 ports and is fitted with a H.100 bus then upon startup the CACULAB.DLL library will automatically 'nail' the transmit timeslots from the E1 ports to the H.100 bus starting at stream 0, timeslot 0 (or as defined by the ACUCC_TSOFFS environment variable) . Since each H.100 stream can have 128 timeslots then all four E1 ports will be nailed to the timeslots of stream 0 on the H.100 bus (with appropriate gaps wherever there are signalling channels on the ports).

From the formula shown above, the handles returned by the CCgetslot() function would range from 0 though to 127 for the channels on these four ports.

If there was a second board present in the system with another four E1 ports then these channels would be nailed to stream 1 of the H.100 bus and the handles returned by CCgetslot() for the channels on these four ports would range from 4096 to 4223.

The external bus *handle* returned by CCgetslot() can be used in the CClisten() function to allow the 'receive' timeslot of one channel to '*listen'* to the 'transmit' timeslot from another channel via the external H.100 or SCBUS.

For example, if there was an inbound call on E1 port 0, channel 1 and an outbound call on E1 port 1, channel 12 then these two conversations could be connected together with the following calls:

```
CClisten(0,1,CCgetslot(1,12));
CClisten(1,12,CCgetslot(0,1));
```

Returns: This function returns the logical timeslot handle for the given port and *channel* or a negative error code.

-o-

CClisten

Synopsis:

CClisten(port,channel,ts_handle)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
ts_handle – The logical timeslot handle returned from CCgetslot()

Description: This function causes the receive timeslot of the given *port* and *channel* to 'listen' to the transmit timeslot that has been nailed to the external H.100 or SCBUS. The *ts_handle* is a logical handle that references an external H.100 or SCBUS stream/timeslot as returned by CCgetslot() or SMgetslot() or which can be obtained by using the formula:

*handle = stream * 4096 + timeslot*

Where *stream* and *timeslot* are the stream and timeslot on the external H.100 or SCBUS. For the SCBUS the *stream* is hardcoded to 24.

For example the following code makes channel 1 on port 0 listen to a voice channel so that any

voice prompts played on the voice channel will be heard by the caller.

x=CClisten(0,1,SMgetslot(1));

(The SMgetslot() function is part of the Aculab speech module library CXACUDSP.DLL and is similar to the CCgetslot() function).

Returns: This function returns 0 upon success or a negative error code.

-o-

CCunlisten

Synopsis:
CCunlisten(port,channel)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.

Description: This function stops the receive timeslot of the given *port* and *channel* from listening to any H.100 or SCBUS transmit timeslot:

Returns: This function returns 0 upon success or a negative error code.

-o-

CCstate

Synopsis:
CCstate(port,chan)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.

Description: This function returns current state of the channel. The state returned will be one of the following (as defined in the ACULAB.INC file provided with the library):

```
const CS_IDLE                =0x0;
const CS_WAIT_FOR_INCOMING   =0x1;
const CS_INCOMING_CALL_DET   =0x2;
const CS_CALL_CONNECTED      =0x4;
const CS_WAIT_FOR_OUTGOING   =0x8;
const CS_OUTGOING_RINGING    =0x10;
const CS_INCOMING_DETAILS    =0x20;
const CS_CALL_CHARGE         =0x21;
const CS_EMERGENCY_CONNECT   =0x80;
const CS_TEST_CONNECT        =0x100;
const CS_REMOTE_DISCONNECT   =0x400;
const CS_WAIT_FOR_ACCEPT     =0x800;
const CS_PROGRESS            =0x1000;
const CS_OUTGOING_PROCEEDING=0x2000;
const CS_NOTIFY              =0x4000;
const CS_INFO                =0x8000;
const CS_HOLD                =0x10000;
```

```
const CS_HOLD_REJECT          =0x20000;
const CS_TRANSFER_REJECT      =0x40000;
const CS_RECONNECT_REJECT     =0x80000;
const CS_CHARGE_INT           =0x100000;
const CS_EXTENDED             =0x200000;
const CS_DPNS_TRANSIT         =0x1000000;     # DPNSS Enhanced Only
const CS_DPNS_IN_TRANSIT      =0x2000000;     # DPNSS Enhanced Only
const CS_DPNS_HOLDING         =0x4000000;     # DPNSS Enhanced Only
const CS_DPNS_HELD            =0x8000000;     # DPNSS Enhanced Only
const CS_DPNS_CONFERENCE      =0x10000000;    # DPNSS Enhanced Only
const CS_DPNS_INTRUDING       =0x20000000;    # DPNSS Enhanced Only
const CS_DPNS_IN_TRANSIT      =0x40000000;     # DPNSS Enhanced Only
const CS_MEDIA           =0x00000101;     # IP only
const CS_MEDIA_PROPOSE        =0x00000102;     #IP only
const CS_MEDIA_REJECT_PROPOSAL          =0x00000103;   # IP only
const CS_MEDIA_REQUEST_PROPOSAL         =0x00000104;   # IP only
const CS_MEDIA_REJECT_REQUEST_PROPOSAL =0x00000105;   # IP only
const CS_INSUFFICIENT_MEDIA_RESOURCE_FOR_CALL =0x00000106;  # IP only
```

Returns: Returns the current state of the channel or -1 if a bad port or channel is given.

-o-

CCuse

Synopsis:

CCuse(port,timeselot[,flag])

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
[flag] – Set to 1 (default) to cause task to jump to *onsignal* if a CS_REMOTE_DISCONNECT is received. Set to 0 to stop task jumping to *onsignal*.

Description: This function allows the current Telecom Engine task to be associated with a *port* and *channel* in such a way that if a call on the specified *port* and *channel* receives a CS_REMOTE_DISCONNECT event then the task will be forced to jump to its *onsignal* function.

The default value of *flag* if it is not specified is 1. To clear the association between the task and a *port* and *channel* so that it will no longer jump to the *onsignal* function upon receiving a CS_REMOTE_DISCONNECT event then the *flag* should be set to 0.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCwait

Synopsis:

CCwait(port,chan,timeout_100ms,&pState)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
pState – Pointer to a variable to receive the ID of the event that terminated the CCwait() call.

Description: This function will wait for an event to be received on a particular port and channel it will return either when an event is detected on the channel or the timeout has expired (or if it is

aborted by CCabort()). The timeout is specified in 100ms units (tenths of a second) after which the function will return if no event has been received. If -1 (CC_WAIT_FOREVER) is specified for the timeout then the call will wait forever for an event.

The function takes a pointer to a variable which will hold the ID of the event received.

Note that the function keeps an internal track of events on a channel and if the state of a channel has changed since the last time it was called then it will return immediately with the current state of the channel. This is to prevent events from being missed in between calls to CCwait() but it means that the programmer should always check the returned state in a loop to ensure that the expected event is received. For example:

```
// loop waiting for incoming call
while(1)
    x=CCwait(port,chan,CC_WAIT_FOREVER,&event);
    // Make sure the event we want is received..
    if(x > 0 and event eq CS_INCOMING_CALL_DETECTED)
       break;
    endif
endwhile

// Answer the call
CCaccept(port,chan);

// Play a vox file to caller
SMplay(vox_chan,filename);

// Hangup the call
CCdisconnect(port,chan,CAUSE_NORMAL);

// Wait for state to return to IDLE the release call
while(1)
     // THIS WILL RETURN IMMEDIATEDELY THE FIRST TIME IT IS      x=CCwait(port,chan,CC_WAIT_FOREVER,&event);
  if(x eq CS_IDLE)
     break;
  endif
endwhile
```

In the second loop above where the application calls CCwait() to check for a return to the state CS_IDLE, the CCwait() call will certainly return immediately the first time through the loop since the state will have changed since it was last called to check for CS_INCOMING_CALL_DETECTED.

Returns: The function will return 0 if the timeout has expired without receiving an event (or if the function was aborted by a CCabort() call). It will return 1 if the function terminated because an event was detected (and the event ID will be set in the variable pointed to by pState). It will return -1 is a bad port or channel was given.

-o-

CCabort

Synopsis:

CCabort(port,channel)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.

Description: This function aborts a CCwait() call on a particular port and channel.

Returns: Return 0 on success or -1 if a bad port or channel is given.

-o-

CCenablein

Synopsis:
CCenablein(port,channel[,cnf_parm1[,cnf_parm2...])

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.
[cnf_parm1,cnf_parm2].. – optional additional CNF parameters..

Description: This function maps to the following Aculab function:

ACU_ERR call_openin(IN_XPARMS **indetailsp*);

It opens the specified port/channel to allow incoming calls to be received on that channel.

The optional *cnf_parms* allow for the IN_XPARMS.cnf field to be set. If one or more of these optional *cnf_parms* are specified then they are each ORed in turn with IN_XPARMS.cnf field. If no *cnf_parms* are specifed then by default the IN_XPARMS.cnf field is set to CNF_REM_DISC which stops the channel automatically returning to the idle state when a remote end disconnect occurs (instead the CCrelease() call must be used to return the channel to CS_IDLE state).

See the Aculab documentation for the call_openin() function for a more detailed description of the cnf field values.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCaccept

Synopsis:
CCaccept(port, channel)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.

Description: This function maps to the following Aculab functions:

ACU_ERR call_accept(int *handle*);
or ACU_ERR xcall_accept(ACCEPT_XPARMS **acceptp*);

It is used to accept (answer) an incoming call after a CS_INCOMING_CALL_DET event has been indicated. If any of the extended parameters have been set using the CCsetparm() function with a *ParmType* of PARM_TYPE_ACCEPT then the extended version of the function will be called.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCmkcall

Synopsis:

CCmkcall(port,channel,DID,CID[,send_comp[,cnf_parm1[,cnf_parm2....])

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
DID – The destination address
CID – The originating address
[send_comp] – Optional Sending complete flag
[cnf_parm1,cnf_parm2..] – Optional cnf_parms

Description: This function maps to the following Aculab function:

ACU_ERR call_openout(OUT_XPARMS **outdetailsp*);

It attempts to make an outgoing call on the specified port and channel.

The DID and CID arguments specify the destination and originating addresses respectively. The option *send_comp* argument allows the OUT_XPARMS.sending_complete flag to be set and should be set to 0 for overlap sending (more digits to come) or 1 for en-bloc sending. The default is en-bloc sending if this aragument is not given.

The optional *cnf_parms* allow for the OUT_XPARMS.cnf field to be set. If one or more of these optional *cnf_parms* are specified then they are each ORed in turn with OUT_XPARMS.cnf field. If no *cnf_parms* are specifed then by default the IN_XPARMS.cnf field is set to CNF_REM_DISC which stops the channel automatically returning to the idle state when a remote end disconnect occurs (instead the CCrelease() call must be used to return the channel to CS_IDLE state).

See the Aculab documentation for the call_openout() function for a more detailed description of the cnf field values.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCdisconnect

Synopsis:

CCdisconnect(port, channel,cause[,std_or_raw])

Arguments:

© Zentel Telecom Ltd, 2009

port – The logical E1/T1 port number.
channel – The channel number.
cause – The cause code
[std_or_raw] – Optional flag to define whether to use the generic (standardised cause code or protocol specific raw cause code)

Description: This function maps to the following Aculab functions:

```
     ACU_ERR call_disconnect(CAUSE_XPARMS *causep);
or   ACU_ERR xcall_disconnect(DISCONNECT_XPARMS *causep);
```

It is used to disconnect an incoming or outgoing call on the specified logical port and channel. The *cause* parameter can either be one of the Aculab defined generic cause codes (which will be translated to the appropriate protocol specific cause code where possible) or else a raw protocol specific code can be used (by setting the optional *std_or_raw* flag to 1). The standard cause codes are defined in the ACULAB.INC file as follows:

```
const LC_NORMAL               = 0;
const LC_NUMBER_BUSY          = 1;
const LC_NO_ANSWER            = 2;
const LC_NUMBER_UNOBTAINABLE   = 3;
const LC_NUMBER_CHANGED        = 4;
const LC_OUT_OF_ORDER         = 5;
const LC_INCOMING_CALLS_BARRED = 6;
const LC_CALL_REJECTED        = 7;
const LC_CALL_FAILED          = 8;
const LC_CHANNEL_BUSY         = 9;
const LC_NO_CHANNELS          =10;
const LC_CONGESTION           =11;
```

If any of the extended parameters have been set using the CCsetparm() function with a *ParmType* of PARM_TYPE_DISCONNECT then the extended version of the function will be called.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCrelease

Synopsis:
CCrelease(port, channel)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.

Description: This function maps to the following Aculab functions:

```
     ACU_ERR call_release(CAUSE_XPARMS *causep);
or   ACU_ERR xcall_release(DISCONNECT_XPARMS *causep);
```

This function is used to release the call handle associated with an inbound or outbound call in response to the channel returning to the CS_IDLE state.
If any of the extended parameters have been set using the CCsetparm() function with a *ParmType* of PARM_TYPE_DISCONNECT then the extended version of the function will be called.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCsetparm

Synopsis:

CCsetparm(port, channel, parmType, parmId, Value)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
parmType – The type of parameter being set
parmID - The ID of the parameter being set
Value - The value to set the parameter to..

Description: This function is a very important function in the CXACULAB.DLL library as it allows individual fields to be set in the various Aculab call control structures.

For many Aculab call control functions there are two versions of the function: a plain vanilla function call and an extended function call which allows additional parameters to be specified in an Aculab extended structure.

Also for the call_openin() and call_openout() functions there are some unique extended parameters that can be set depending upon the network protocol being used.

The Aculab functions where there are both plain and extended versions of the function call are shown below:

ACU_ERR call_incoming_ringing(int *handle*);
ACU_ERR xcall_incoming_ringing(INCOMING_RINGING_XPARMS **ringingp*);

ACU_ERR call_accept(int *handle*);
ACU_ERR xcall_accept(ACCEPT_XPARMS **acceptp*);

ACU_ERR call_disconnect(CAUSE_XPARMS **causep*);
ACU_ERR xcall_disconnect(DISCONNECT_XPARMS **causep*);

ACU_ERR call_release(CAUSE_XPARMS **causep*);
ACU_ERR xcall_release(DISCONNECT_XPARMS **causep*);

ACU_ERR call_getcause(CAUSE_XPARMS **causep*);
ACU_ERR xcall_getcause(DISCONNECT_XPARMS **causep*);

ACU_ERR call_get_originating_addr(int *handle*);
ACU_ERR xcall_get_originating_addr(GET_ORIGINATING_ADDR_XPARMS* *originating_parms*);

ACU_ERR call_hold(int *handle*);
ACU_ERR xcall_hold(HOLD_XPARMS **holdp*);

ACU_ERR call_reconnect(int *handle*);
ACU_ERR xcall_reconnect(HOLD_XPARMS **holdp*);

Each channel opened by the library has its own individual copy of the structures INCOMING_RINGING_XPARMS, ACCEPT_XPARMS, DISCONNECT_XPARMS, GET_ORIGINATING_ADDR_XPARMS and HOLD_XPARMS, as well as individual copies of

the OUT_XPARMS and IN_XPARMS used by the call_openin() and call_openout() functions (CCenablein(), CCmkcall()).

The CCsetparm() function allows for the fields of these structures to be set as required. The *parmType* argument defines which of the extended structures the parameter being set is part of, and can be one of the following values (defined in the ACULAB.INC file):

```
const  PARM_TYPE_OUT         =0;   # For CCmkcall()
const  PARM_TYPE_IN          =1;   # For CCenablein()
const  PARM_TYPE_ALERTING    =2;   # For CCalerting()
const  PARM_TYPE_ACCEPT      =3;   # For CCaccept()
const  PARM_TYPE_DISCON      =4;   # For CCdisconnect()/CCrelease()
const  PARM_TYPE_HOLD        =5;   # COMING SOON
const  PARM_TYPE_GETADDR     =6;   # COMING SOON
```

Then the *parmID* argument specifies which field of the specified structure is to be set. Each field in each structure has been given a unique identifier which maps directly to one of the fields of the Aculab extended structures.

For example lets say for an incoming IP call we wanted to set the accept_xparms.unique_xparms.sig_iptel.destination_display_name when we accepted the call then we would have something like:

```
x=CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_DEST_DISPLAY,"Joe Bloggs");
x=CCaccept(port,chan);
```

Note that once the CCsetparm() function has been called to set one of the extended structure fields then it will be the extended version of the function (xcall_accept() in the case above) rather than the standard version that will be called thereafter or until a call to CCclrparms() is made.

The *ParmID* The *parmID* values and the field and structure they map to are shown below:

PARM_TYPE_ACCEPT:

parmID	Structure and Field it maps to:	Field type
CP_Q931_PROGRESS_INDICATOR	accept_xparms.unique_xparms.sig_q931.progress_indicator.ie	PT_HEXSTR
CP_Q931_PROGRESS_LASTMSG	accept_xparms.unique_xparms.sig_q931.progress_indicator.last_msg	PT_UCHAR
CP_Q931_LOLAYER	accept_xparms.unique_xparms.sig_q931.lolayer.ie	PT_HEXSTR
CP_Q931_LOLAYER_LASTMSG	accept_xparms.unique_xparms.sig_q931.lolayer.last_msg	PT_UCHAR
CP_Q931_DISPLAY	accept_xparms.unique_xparms.sig_q931.display.ie	PT_HEXSTR
CP_Q931_DISPLAY_LASTMSG	accept_xparms.unique_xparms.sig_q931.display.last_msg	PT_UCHAR
CP_Q931_CONN_ADDR	accept_xparms.unique_xparms.sig_q931.connected_addr	PT_STRING
CP_Q931_CONN_NUMBERING_TYPE	accept_xparms.unique_xparms.sig_q931.conn_numbering_type	PT_UCHAR
CP_Q931_CONN_NUMBERING_PLAN	accept_xparms.unique_xparms.sig_q931.conn_numbering_plan	PT_UCHAR
CP_Q931_CONN_NUMBERING_PRESENTATION	accept_xparms.unique_xparms.sig_q931.conn_numbering_presentation	PT_UCHAR
CP_Q931_CONN_NUMBERING_SCREENING	accept_xparms.unique_xparms.sig_q931.conn_numbering_screening	PT_UCHAR
CP_ISUP_PROGRESS_INDICATOR	accept_xparms.unique_xparms.sig_isup.progress_indicator.ie	PT_HEXSTR
CP_ISUP_PROGRESS_LASTMSG	accept_xparms.unique_xparms.sig_isup.progress_indicator.last_msg	PT_UCHAR
CP_ISUP_LOLAYER	accept_xparms.unique_xparms.sig_isup.lolayer.ie	PT_HEXST

		R
CP_ISUP_LOLAYER_LASTMSG	accept_xparms.unique_xparms.sig_isup.lolayer.last_msg	PT_UCHAR
CP_ISUP_CONN_ADDR	accept_xparms.unique_xparms.sig_isup.connected_addr	PT_STRING
CP_ISUP_CONN_NATUREOF_ADDR	accept_xparms.unique_xparms.sig_isup.conn_natureof_addr	PT_UCHAR
CP_ISUP_CONN_NUMBERING_PLAN	accept_xparms.unique_xparms.sig_isup.conn_numbering_plan	PT_UCHAR
CP_ISUP_CONN_NUMBERING_PRESENTATION	accept_xparms.unique_xparms.sig_isup.conn_numbering_presentation	PT_UCHAR
CP_ISUP_CONN_NUMBERING_SCREENING	accept_xparms.unique_xparms.sig_isup.conn_numbering_screening	PT_UCHAR
CP_ISUP_CHARGE_IND	accept_xparms.unique_xparms.sig_isup.charge_ind	PT_UCHAR
CP_ISUP_DEST_CATEGORY	accept_xparms.unique_xparms.sig_isup.dest_category	PT_UCHAR
CP_ISUP_ACC_IND_VALID	accept_xparms.unique_xparms.sig_isup.isdn_access_ind.valid	PT_UCHAR
CP_ISUP_ACC_IND_VALUE	accept_xparms.unique_xparms.sig_isup.isdn_access_ind.value	PT_UCHAR
CP_ISUP_USERP_IND_VALID	accept_xparms.unique_xparms.sig_isup.isdn_userpart_ind.valid	PT_UCHAR
CP_ISUP_USERP_IND_VALUE	accept_xparms.unique_xparms.sig_isup.isdn_userpart_ind.value	PT_UCHAR
CP_ISUP_INTERW_IND_VALID	accept_xparms.unique_xparms.sig_isup.interworking_ind.valid	PT_UCHAR
CP_ISUP_INTERW_IND_VALUE	accept_xparms.unique_xparms.sig_isup.interworking_ind.value	PT_UCHAR
CP_IPTEL_DEST_DISPLAY	accept_xparms.unique_xparms.sig_iptel.destination_display_name	PT_STRING
CP_IPTEL_CODECS	accept_xparms.unique_xparms.sig_iptel.codecs	PT_HEXSTR
CP_IPTEL_MEDIA_TDM_ENC	accept_xparms.unique_xparms.sig_iptel.media_settings.tdm_encoding	PT_INT
CP_IPTEL_MEDIA_ENC_GAIN	accept_xparms.unique_xparms.sig_iptel.media_settings.encode_gain	PT_INT
CP_IPTEL_MEDIA_DEC_GAIN	accept_xparms.unique_xparms.sig_iptel.media_settings.decode_gain	PT_INT
CP_IPTEL_MEDIA_ECHO_CANC	accept_xparms.unique_xparms.sig_iptel.media_settings.echo_cancellation	PT_INT
CP_IPTEL_MEDIA_ECHO_SUPP	accept_xparms.unique_xparms.sig_iptel.media_settings.echo_suppression	PT_INT
CP_IPTEL_MEDIA_ECHO_SPAN	accept_xparms.unique_xparms.sig_iptel.media_settings.echo_span	PT_INT
CP_IPTEL_MEDIA_RTP_TOS	accept_xparms.unique_xparms.sig_iptel.media_settings.rtp_tos	PT_INT
CP_IPTEL_MEDIA_RTCP_TOS	accept_xparms.unique_xparms.sig_iptel.media_settings.rtcp_tos	PT_INT
CP_IPTEL_MEDIA_DTMF_DET	accept_xparms.unique_xparms.sig_iptel.media_settings.dtmf_detector	PT_INT
CP_IPTEL_VMPRXID	accept_xparms.unique_xparms.sig_iptel.vmprxid	PT_HEXSTR
CP_IPTEL_VMPTXID	accept_xparms.unique_xparms.sig_iptel.vmptxid	PT_HEXSTR
CP_H323_DEST_ALIAS	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.destination_alias	PT_STRING
CP_H323_ORIG_ALIAS	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.originating_alias	PT_STRING
CP_H323_H245_TUNNELING	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.h245_tunneling	PT_INT
CP_H323_FASTSTART	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.faststart	PT_INT
CP_H323_EARLY_H245	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.early_h245	PT_INT
CP_H323_DTMF	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.dtmf	PT_STRING
CP_H323_PROGRESS_LOC	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.progress_location	PT_INT
CP_H323_PROGRESS_DESC	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.progress_description	PT_INT
CP_SIP_CONTACT_ADDR	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.contact_address	PT_STRING
CP_SIP_ZERO_CONN_ADDR_HOLD	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.zero_connection_address_hold	PT_INT
CP_SIP_DISABLE_REL_PROV	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.disable_reliable_provisional_response	PT_INT
CP_SIP_DISABLE_EARLY_MED	accept_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.disable_early_media	PT_INT

PARM_TYPE_INRINGING:

parmID	Structure and Field it maps to:	Field type

CP_Q931_PROGRESS_INDICATOR	incoming_ringing_xparms.unique_xparms.sig_q931.progress_indicator.ie	PT_HEXSTR
CP_Q931_PROGRESS_LASTMSG	incoming_ringing_xparms.unique_xparms.sig_q931.progress_indicator.last_msg	PT_UCHAR
CP_Q931_DISPLAY	incoming_ringing_xparms.unique_xparms.sig_q931.display.ie	PT_HEXSTR
CP_Q931_DISPLAY_LASTMSG	incoming_ringing_xparms.unique_xparms.sig_q931.display.last_msg	PT_UCHAR
CP_ISUP_PROGRESS_INDICATOR	incoming_ringing_xparms.unique_xparms.sig_isup.progress_indicator.ie	PT_HEXSTR
CP_ISUP_PROGRESS_LASTMSG	incoming_ringing_xparms.unique_xparms.sig_isup.progress_indicator.last_msg	PT_UCHAR
CP_ISUP_CHARGE_IND	incoming_ringing_xparms.unique_xparms.sig_isup.charge_ind	PT_UCHAR
CP_ISUP_IN_BAND	incoming_ringing_xparms.unique_xparms.sig_isup.in_band	PT_UCHAR
CP_ISUP_DEST_CATEGORY	incoming_ringing_xparms.unique_xparms.sig_isup.dest_category	PT_UCHAR
CP_ISUP_ACC_IND_VALID	incoming_ringing_xparms.unique_xparms.sig_isup.isdn_access_ind.valid	PT_UCHAR
CP_ISUP_ACC_IND_VALUE	incoming_ringing_xparms.unique_xparms.sig_isup.isdn_access_ind.value	PT_UCHAR
CP_ISUP_USERP_IND_VALID	incoming_ringing_xparms.unique_xparms.sig_isup.isdn_userpart_ind.valid	PT_UCHAR
CP_ISUP_USERP_IND_VALUE	incoming_ringing_xparms.unique_xparms.sig_isup.isdn_userpart_ind.value	PT_UCHAR
CP_ISUP_INTERW_IND_VALID	incoming_ringing_xparms.unique_xparms.sig_isup.interworking_ind.valid	PT_UCHAR
CP_ISUP_INTERW_IND_VALUE	incoming_ringing_xparms.unique_xparms.sig_isup.interworking_ind.value	PT_UCHAR
CP_IPTEL_DEST_DISPLAY	incoming_ringing_xparms.unique_xparms.sig_iptel.destination_display_name	PT_STRING
CP_IPTEL_CODECS	incoming_ringing_xparms.unique_xparms.sig_iptel.codecs	PT_HEXSTR
CP_IPTEL_MEDIA_TDM_ENC	incoming_ringing_xparms.unique_xparms.sig_iptel.media_settings.tdm_encoding	PT_INT
CP_IPTEL_MEDIA_ENC_GAIN	incoming_ringing_xparms.unique_xparms.sig_iptel.media_settings.encode_gain	PT_INT
CP_IPTEL_MEDIA_DEC_GAIN	incoming_ringing_xparms.unique_xparms.sig_iptel.media_settings.decode_gain	PT_INT
CP_IPTEL_MEDIA_ECHO_CANC	incoming_ringing_xparms.unique_xparms.sig_iptel.media_settings.echo_cancellation	PT_INT
CP_IPTEL_MEDIA_ECHO_SUPP	incoming_ringing_xparms.unique_xparms.sig_iptel.media_settings.echo_suppression	PT_INT
CP_IPTEL_MEDIA_ECHO_SPAN	incoming_ringing_xparms.unique_xparms.sig_iptel.media_settings.echo_span	PT_INT
CP_IPTEL_MEDIA_RTP_TOS	incoming_ringing_xparms.unique_xparms.sig_iptel.media_settings.rtp_tos	PT_INT
CP_IPTEL_MEDIA_RTCP_TOS	incoming_ringing_xparms.unique_xparms.sig_iptel.media_settings.rtcp_tos	PT_INT
CP_IPTEL_MEDIA_DTMF_DET	incoming_ringing_xparms.unique_xparms.sig_iptel.media_settings.dtmf_detector	PT_INT
CP_IPTEL_VMPRXID	incoming_ringing_xparms.unique_xparms.sig_iptel.vmprxid	PT_HEXSTR
CP_IPTEL_VMPTXID	incoming_ringing_xparms.unique_xparms.sig_iptel.vmptxid	PT_HEXSTR
CP_H323_H245_TUNNELING	incoming_ringing_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.h245_tunneling	PT_INT
CP_H323_FASTSTART	incoming_ringing_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.faststart	PT_INT
CP_H323_EARLY_H245	incoming_ringing_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.early_h245	PT_INT
CP_H323_PROGRESS_LOC	incoming_ringing_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.progress_location	PT_INT
CP_H323_PROGRESS_DESC	incoming_ringing_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.progress_description	PT_INT
CP_SIP_CONTACT_ADDR	incoming_ringing_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.contact_address	PT_STRING
CP_SIP_SEND_EARLY_MED	incoming_ringing_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.send_early_media	PT_INT
CP_SIP_USE_183	incoming_ringing_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.use_183_response_for_early_media	PT_INT
CP_SIP_SEND_REL_PROV	incoming_ringing_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.send_reliable_provisional_response	PT_INT

PARM_TYPE_DISCON:

parmID	Structure and Field it maps to:	Field type

CP_Q931_CAUSE_RAW	discon_xparms.unique_xparms.sig_q931.raw	PT_INT
CP_Q931_PROGRESS_INDICATOR	discon_xparms.unique_xparms.sig_q931.progress_indicator.ie	PT_HEXSTR
CP_Q931_PROGRESS_LASTMSG	discon_xparms.unique_xparms.sig_q931.progress_indicator.last_msg	PT_UCHAR
CP_Q931_DISPLAY	discon_xparms.unique_xparms.sig_q931.display.ie	PT_HEXSTR
CP_Q931_DISPLAY_LASTMSG	discon_xparms.unique_xparms.sig_q931.display.last_msg	PT_UCHAR
CP_Q931_NOTIFY_INDICATOR	discon_xparms.unique_xparms.sig_q931.notify_indicator.ie	PT_HEXSTR
CP_Q931_NOTIFY_LASTMSG	discon_xparms.unique_xparms.sig_q931.notify_indicator.last_msg	PT_UCHAR
CP_Q931_CAUSE_LOC	discon_xparms.unique_xparms.sig_q931.location	PT_INT
CP_ISUP_CAUSE_RAW	discon_xparms.unique_xparms.sig_isup.raw	PT_INT
CP_ISUP_PROGRESS_INDICATOR	discon_xparms.unique_xparms.sig_isup.progress_indicator.ie	PT_HEXSTR
CP_ISUP_PROGRESS_LASTMSG	discon_xparms.unique_xparms.sig_isup.progress_indicator.last_msg	PT_UCHAR
CP_ISUP_CAUSE_LOC	discon_xparms.unique_xparms.sig_isup.location	PT_INT
CP_ISUP_REATTEMPT	discon_xparms.unique_xparms.sig_isup.reattempt	PT_INT
CP_ITR6_CAUSE_RAW	discon_xparms.unique_xparms.sig_1tr6.raw	PT_INT
CP_DASS_CAUSE_RAW	discon_xparms.unique_xparms.sig_dass.raw	PT_INT
CP_DPNSS_CAUSE_RAW	discon_xparms.unique_xparms.sig_dpnss.raw	PT_INT
CP_CAS_CAUSE_RAW	discon_xparms.unique_xparms.sig_cas.raw	PT_INT

PARM_TYPE_OUT:

parmID	Structure and Field it maps to:	Field type
CP_Q931_SERVICE_OCTET	out_xparms.unique_xparms.sig_q931.service_octet	PT_UCHAR
CP_Q931_ADD_INFO_OCTET	out_xparms.unique_xparms.sig_q931.add_info_octet	PT_UCHAR
CP_Q931_DEST_NUMBERING_TYPE	out_xparms.unique_xparms.sig_q931.dest_numbering_type	PT_UCHAR
CP_Q931_DEST_NUMBERING_PLAN	out_xparms.unique_xparms.sig_q931.dest_numbering_plan	PT_UCHAR
CP_Q931_BEARER	out_xparms.unique_xparms.sig_q931.bearer.ie	PT_HEXSTR
CP_Q931_BEARER_LASTMSG	out_xparms.unique_xparms.sig_q931.bearer.last_msg	PT_UCHAR
CP_Q931_ORIG_NUMBERING_TYPE	out_xparms.unique_xparms.sig_q931.orig_numbering_type	PT_UCHAR
CP_Q931_ORIG_NUMBERING_PLAN	out_xparms.unique_xparms.sig_q931.orig_numbering_plan	PT_UCHAR
CP_Q931_ORIG_NUMBERING_PRESENTATION	out_xparms.unique_xparms.sig_q931.orig_numbering_presentation	PT_UCHAR
CP_Q931_ORIG_NUMBERING_SCREENING	out_xparms.unique_xparms.sig_q931.orig_numbering_screening	PT_UCHAR
CP_Q931_CONN_NUMBERING_TYPE	out_xparms.unique_xparms.sig_q931.conn_numbering_type	PT_UCHAR
CP_Q931_CONN_NUMBERING_PLAN	out_xparms.unique_xparms.sig_q931.conn_numbering_plan	PT_UCHAR
CP_Q931_CONN_NUMBERING_PRESENTATION	out_xparms.unique_xparms.sig_q931.conn_numbering_presentation	PT_UCHAR
CP_Q931_CONN_NUMBERING_SCREENING	out_xparms.unique_xparms.sig_q931.conn_numbering_screening	PT_UCHAR
CP_Q931_DEST_SUBADDR	out_xparms.unique_xparms.sig_q931.dest_subaddr	PT_HEXSTR
CP_Q931_ORIG_SUBADDR	out_xparms.unique_xparms.sig_q931.orig_subaddr	PT_HEXSTR
CP_Q931_HILAYER	out_xparms.unique_xparms.sig_q931.hilayer.ie	PT_HEXSTR
CP_Q931_HILAYER_LASTMSG	out_xparms.unique_xparms.sig_q931.hilayer.last_msg	PT_UCHAR
CP_Q931_LOLAYER	out_xparms.unique_xparms.sig_q931.lolayer.ie	PT_HEXSTR
CP_Q931_LOLAYER_LASTMSG	out_xparms.unique_xparms.sig_q931.lolayer.last_msg	PT_UCHAR
CP_Q931_PROGRESS_INDICATOR	out_xparms.unique_xparms.sig_q931.progress_indicator.ie	PT_HEXST

		R
CP_Q931_PROGRESS_LASTMSG	out_xparms.unique_xparms.sig_q931.progress_indicator.last_msg	PT_UCHAR
CP_Q931_NOTIFY_INDICATOR	out_xparms.unique_xparms.sig_q931.notify_indicator.ie	PT_HEXSTR
CP_Q931_NOTIFY_LASTMSG	out_xparms.unique_xparms.sig_q931.notify_indicator.last_msg	PT_UCHAR
CP_Q931_KEYPAD	out_xparms.unique_xparms.sig_q931.keypad.ie	PT_HEXSTR
CP_Q931_KEYPAD_LASTMSG	out_xparms.unique_xparms.sig_q931.keypad.last_msg	PT_UCHAR
CP_Q931_DISPLAY	out_xparms.unique_xparms.sig_q931.display.ie	PT_HEXSTR
CP_Q931_DISPLAY_LASTMSG	out_xparms.unique_xparms.sig_q931.display.last_msg	PT_UCHAR
CP_Q931_SLOTMAP	out_xparms.unique_xparms.sig_q931.slotmap	PT_LONG
CP_Q931_EP_USID	out_xparms.unique_xparms.sig_q931.endpoint_id.usid	PT_UCHAR
CP_Q931_EP_TID	out_xparms.unique_xparms.sig_q931.endpoint_id.tid	PT_UCHAR
CP_Q931_EP_INTERPRETER	out_xparms.unique_xparms.sig_q931.endpoint_id.interpreter	PT_UCHAR
CP_Q931_CAUSE	out_xparms.unique_xparms.sig_q931.cause.ie	PT_HEXSTR
CP_Q931_CAUSE_LASTMSG	out_xparms.unique_xparms.sig_q931.cause.last_msg	PT_UCHAR
CP_Q931_ADD_ORIG_ADDR	out_xparms.unique_xparms.sig_q931.additional_orig_addr	PT_HEXSTR
CP_Q931_ADD_ORIG_NUMBERING_TYPE	out_xparms.unique_xparms.sig_q931.add_orig_numbering_type	PT_UCHAR
CP_Q931_ADD_ORIG_NUMBERING_PLAN	out_xparms.unique_xparms.sig_q931.add_orig_numbering_plan	PT_UCHAR
CP_Q931_ADD_ORIG_NUMBERING_PRESENTATION	out_xparms.unique_xparms.sig_q931.add_orig_numbering_presentation	PT_UCHAR
CP_Q931_ADD_ORIG_NUMBERING_SCREENING	out_xparms.unique_xparms.sig_q931.add_orig_numbering_screening	PT_UCHAR
CP_Q931_OMIT_CALLING_PARTY_IE	out_xparms.unique_xparms.sig_q931.omit_calling_party_ie	PT_UCHAR
CP_Q931_CALL_REF	out_xparms.unique_xparms.sig_q931.call_ref_value	PT_ULONG
CP_DASS_SIC1	out_xparms.unique_xparms.sig_dass.sic1	PT_UCHAR
CP_DASS_SIC2	out_xparms.unique_xparms.sig_dass.sic2	PT_UCHAR
CP_DPNSS_SIC1	out_xparms.unique_xparms.sig_dpnss.sic1	PT_UCHAR
CP_DPNSS_SIC2	out_xparms.unique_xparms.sig_dpnss.sic2	PT_UCHAR
CP_DPNSS_CLC	out_xparms.unique_xparms.sig_dpnss.clc	PT_STRING
CP_CAS_CATEGORY	out_xparms.unique_xparms.sig_cas.category	PT_UCHAR
CP_ISUP_SERVICE_OCTET	out_xparms.unique_xparms.sig_isup.service_octet	PT_UCHAR
CP_ISUP_ADD_INFO_OCTET	out_xparms.unique_xparms.sig_isup.add_info_octet	PT_UCHAR
CP_ISUP_DEST_NATUREOF_ADDR	out_xparms.unique_xparms.sig_isup.dest_natureof_addr	PT_UCHAR
CP_ISUP_DEST_NUMBERING_PLAN	out_xparms.unique_xparms.sig_isup.dest_numbering_plan	PT_UCHAR
CP_ISUP_BEARER	out_xparms.unique_xparms.sig_isup.bearer.ie	PT_HEXSTR
CP_ISUP_BEARER_LASTMSG	out_xparms.unique_xparms.sig_isup.bearer.last_msg	PT_UCHAR
CP_ISUP_ORIG_NATUREOF_ADDR	out_xparms.unique_xparms.sig_isup.orig_natureof_addr	PT_UCHAR
CP_ISUP_ORIG_NUMBERING_PLAN	out_xparms.unique_xparms.sig_isup.orig_numbering_plan	PT_UCHAR
CP_ISUP_ORIG_NUMBERING_PRESENTATION	out_xparms.unique_xparms.sig_isup.orig_numbering_presentation	PT_UCHAR
CP_ISUP_ORIG_NUMBERING_SCREENING	out_xparms.unique_xparms.sig_isup.orig_numbering_screening	PT_UCHAR
CP_ISUP_CONN_NATUREOF_ADDR	out_xparms.unique_xparms.sig_isup.conn_natureof_addr	PT_UCHAR
CP_ISUP_CONN_NUMBERING_PLAN	out_xparms.unique_xparms.sig_isup.conn_numbering_plan	PT_UCHAR
CP_ISUP_CONN_NUMBERING_PRESENTATION	out_xparms.unique_xparms.sig_isup.conn_numbering_presentation	PT_UCHAR
CP_ISUP_CONN_NUMBERING_SCREENING	out_xparms.unique_xparms.sig_isup.conn_numbering_screening	PT_UCHAR
CP_ISUP_CONN_NUMBER_REQ	out_xparms.unique_xparms.sig_isup.conn_number_req	PT_UCHAR
CP_ISUP_ORIG_CATEGORY	out_xparms.unique_xparms.sig_isup.orig_category	PT_UCHAR

CP_ISUP_ORIG_NUMBER_INCOMPLETE	out_xparms.unique_xparms.sig_isup.orig_number_incomplete	PT_UCHAR
CP_ISUP_DEST_SUBADDR	out_xparms.unique_xparms.sig_isup.dest_subaddr	PT_HEXSTR
CP_ISUP_ORIG_SUBADDR	out_xparms.unique_xparms.sig_isup.orig_subaddr	PT_HEXSTR
CP_ISUP_HILAYER	out_xparms.unique_xparms.sig_isup.hilayer.ie	PT_HEXSTR
CP_ISUP_HILAYER_LASTMSG	out_xparms.unique_xparms.sig_isup.hilayer.last_msg	PT_UCHAR
CP_ISUP_LOLAYER	out_xparms.unique_xparms.sig_isup.lolayer.ie	PT_HEXSTR
CP_ISUP_LOLAYER_LASTMSG	out_xparms.unique_xparms.sig_isup.lolayer.last_msg	PT_UCHAR
CP_ISUP_PROGRESS_INDICATOR	out_xparms.unique_xparms.sig_isup.progress_indicator.ie	PT_HEXSTR
CP_ISUP_PROGRESS_LASTMSG	out_xparms.unique_xparms.sig_isup.progress_indicator.last_msg	PT_UCHAR
CP_ISUP_IN_BAND	out_xparms.unique_xparms.sig_isup.in_band	PT_UCHAR
CP_ISUP_NAT_INTER_CALL_IND	out_xparms.unique_xparms.sig_isup.nat_inter_call_ind	PT_UCHAR
CP_ISUP_INTERWORKING_IND	out_xparms.unique_xparms.sig_isup.interworking_ind	PT_UCHAR
CP_ISUP_ISDN_USERPART_IND	out_xparms.unique_xparms.sig_isup.isdn_userpart_ind	PT_UCHAR
CP_ISUP_ISDN_USERPART_PREF_IND	out_xparms.unique_xparms.sig_isup.isdn_userpart_pref_ind	PT_UCHAR
CP_ISUP_ISDN_ACCESS_IND	out_xparms.unique_xparms.sig_isup.isdn_access_ind	PT_UCHAR
CP_ISUP_DEST_INT_NW_IND	out_xparms.unique_xparms.sig_isup.dest_int_nw_ind	PT_UCHAR
CP_ISUP_CONTINUITY_CHECK_IND	out_xparms.unique_xparms.sig_isup.continuity_check_ind	PT_UCHAR
CP_ISUP_SATELLITE_IND	out_xparms.unique_xparms.sig_isup.satellite_ind	PT_UCHAR
CP_ISUP_CHARGE_IND	out_xparms.unique_xparms.sig_isup.charge_ind	PT_UCHAR
CP_ISUP_DEST_CATEGORY	out_xparms.unique_xparms.sig_isup.dest_category	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_QUAL	out_xparms.unique_xparms.sig_isup.add_calling_num_qualifier_ind	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_NOAI	out_xparms.unique_xparms.sig_isup.add_calling_num_natureof_addr	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_PLAN	out_xparms.unique_xparms.sig_isup.add_calling_num_plan	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_PRESENT	out_xparms.unique_xparms.sig_isup.add_calling_num_presentation	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_SCREEN	out_xparms.unique_xparms.sig_isup.add_calling_num_screening	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_INCOMP	out_xparms.unique_xparms.sig_isup.add_calling_num_incomplete	PT_UCHAR
CP_ISUP_ADD_CALL_NUM	out_xparms.unique_xparms.sig_isup.add_calling_num	PT_HEXSTR
CP_ISUP_EXCHANGE_TYPE	out_xparms.unique_xparms.sig_isup.exchange_type	PT_UCHAR
CP_ISUP_COLLECT_CALL	out_xparms.unique_xparms.sig_isup.collect_call_request_ind	PT_UCHAR
CP_IPTEL_DEST_DISPLAY	out_xparms.unique_xparms.sig_iptel.destination_display_name	PT_STRING
CP_IPTEL_ORIG_DISPLAY	out_xparms.unique_xparms.sig_iptel.originating_display_name	PT_STRING
CP_IPTEL_CODECS	out_xparms.unique_xparms.sig_iptel.codecs	PT_HEXSTR
CP_IPTEL_MEDIA_TDM_ENC	out_xparms.unique_xparms.sig_iptel.media_settings.tdm_encoding	PT_INT
CP_IPTEL_MEDIA_ENC_GAIN	out_xparms.unique_xparms.sig_iptel.media_settings.encode_gain	PT_INT
CP_IPTEL_MEDIA_DEC_GAIN	out_xparms.unique_xparms.sig_iptel.media_settings.decode_gain	PT_INT
CP_IPTEL_MEDIA_ECHO_CANC	out_xparms.unique_xparms.sig_iptel.media_settings.echo_cancellation	PT_INT
CP_IPTEL_MEDIA_ECHO_SUPP	out_xparms.unique_xparms.sig_iptel.media_settings.echo_suppression	PT_INT
CP_IPTEL_MEDIA_ECHO_SPAN	out_xparms.unique_xparms.sig_iptel.media_settings.echo_span	PT_INT
CP_IPTEL_MEDIA_RTP_TOS	out_xparms.unique_xparms.sig_iptel.media_settings.rtp_tos	PT_INT
CP_IPTEL_MEDIA_RTCP_TOS	out_xparms.unique_xparms.sig_iptel.media_settings.rtcp_tos	PT_INT
CP_IPTEL_MEDIA_DTMF_DET	out_xparms.unique_xparms.sig_iptel.media_settings.dtmf_detector	PT_INT
CP_IPTEL_VMPRXID	out_xparms.unique_xparms.sig_iptel.vmprxid	PT_HEXSTR
CP_IPTEL_VMPTXID	out_xparms.unique_xparms.sig_iptel.vmptxid	PT_HEXSTR

CP_IPTEL_MEDIA_CALL_TYPE	out_xparms.unique_xparms.sig_iptel.media_call_type	PT_STRING
CP_H323_DEST_ALIAS	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.destination_alias	PT_STRING
CP_H323_ORIG_ALIAS	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.originating_alias	PT_STRING
CP_H323_H245_TUNNELING	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.h245_tunneling	PT_INT
CP_H323_FASTSTART	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.faststart	PT_INT
CP_H323_EARLY_H245	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.early_h245	PT_INT
CP_H323_DTMF	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.dtmf	PT_STRING
CP_H323_PROGRESS_LOC	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.progress_location	PT_INT
CP_H323_PROGRESS_DESC	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.progress_description	PT_INT
CP_SIP_CONTACT_ADDR	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.contact_address	PT_STRING
CP_SIP_ZERO_CONN_ADDR_HOLD	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.zero_connection_address_hold	PT_INT
CP_SIP_DISABLE_REL_PROV	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.disable_reliable_provisional_response	PT_INT
CP_SIP_DISABLE_EARLY_MED	out_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.disable_early_media	PT_INT

PARM_TYPE_IN:

parmID	Structure and Field it maps to:	Field type
CP_Q931_SERVICE_OCTET	in_xparms.unique_xparms.sig_q931.service_octet	PT_UCHAR
CP_Q931_ADD_INFO_OCTET	in_xparms.unique_xparms.sig_q931.add_info_octet	PT_UCHAR
CP_Q931_DEST_NUMBERING_TYPE	in_xparms.unique_xparms.sig_q931.dest_numbering_type	PT_UCHAR
CP_Q931_DEST_NUMBERING_PLAN	in_xparms.unique_xparms.sig_q931.dest_numbering_plan	PT_UCHAR
CP_Q931_BEARER	in_xparms.unique_xparms.sig_q931.bearer.ie	PT_HEXSTR
CP_Q931_BEARER_LASTMSG	in_xparms.unique_xparms.sig_q931.bearer.last_msg	PT_UCHAR
CP_Q931_ORIG_NUMBERING_TYPE	in_xparms.unique_xparms.sig_q931.orig_numbering_type	PT_UCHAR
CP_Q931_ORIG_NUMBERING_PLAN	in_xparms.unique_xparms.sig_q931.orig_numbering_plan	PT_UCHAR
CP_Q931_ORIG_NUMBERING_PRESENTATION	in_xparms.unique_xparms.sig_q931.orig_numbering_presentation	PT_UCHAR
CP_Q931_ORIG_NUMBERING_SCREENING	in_xparms.unique_xparms.sig_q931.orig_numbering_screening	PT_UCHAR
CP_Q931_CONN_NUMBERING_TYPE	in_xparms.unique_xparms.sig_q931.conn_numbering_type	PT_UCHAR
CP_Q931_CONN_NUMBERING_PLAN	in_xparms.unique_xparms.sig_q931.conn_numbering_plan	PT_UCHAR
CP_Q931_CONN_NUMBERING_PRESENTATION	in_xparms.unique_xparms.sig_q931.conn_numbering_presentation	PT_UCHAR
CP_Q931_CONN_NUMBERING_SCREENING	in_xparms.unique_xparms.sig_q931.conn_numbering_screening	PT_UCHAR
CP_Q931_DEST_SUBADDR	in_xparms.unique_xparms.sig_q931.dest_subaddr	PT_HEXSTR
CP_Q931_ORIG_SUBADDR	in_xparms.unique_xparms.sig_q931.orig_subaddr	PT_HEXSTR
CP_Q931_HILAYER	in_xparms.unique_xparms.sig_q931.hilayer.ie	PT_HEXSTR
CP_Q931_HILAYER_LASTMSG	in_xparms.unique_xparms.sig_q931.hilayer.last_msg	PT_UCHAR
CP_Q931_LOLAYER	in_xparms.unique_xparms.sig_q931.lolayer.ie	PT_HEXSTR
CP_Q931_LOLAYER_LASTMSG	in_xparms.unique_xparms.sig_q931.lolayer.last_msg	PT_UCHAR

CP_Q931_PROGRESS_INDICATOR	in_xparms.unique_xparms.sig_q931.progress_indicator.ie	PT_HEXSTR
CP_Q931_PROGRESS_LASTMSG	in_xparms.unique_xparms.sig_q931.progress_indicator.last_msg	PT_UCHAR
CP_Q931_NOTIFY_INDICATOR	in_xparms.unique_xparms.sig_q931.notify_indicator.ie	PT_HEXSTR
CP_Q931_NOTIFY_LASTMSG	in_xparms.unique_xparms.sig_q931.notify_indicator.last_msg	PT_UCHAR
CP_Q931_KEYPAD	in_xparms.unique_xparms.sig_q931.keypad.ie	PT_HEXSTR
CP_Q931_KEYPAD_LASTMSG	in_xparms.unique_xparms.sig_q931.keypad.last_msg	PT_UCHAR
CP_Q931_DISPLAY	in_xparms.unique_xparms.sig_q931.display.ie	PT_HEXSTR
CP_Q931_DISPLAY_LASTMSG	in_xparms.unique_xparms.sig_q931.display.last_msg	PT_UCHAR
CP_Q931_SLOTMAP	in_xparms.unique_xparms.sig_q931.slotmap	PT_LONG
CP_Q931_EP_USID	in_xparms.unique_xparms.sig_q931.endpoint_id.usid	PT_UCHAR
CP_Q931_EP_TID	in_xparms.unique_xparms.sig_q931.endpoint_id.tid	PT_UCHAR
CP_Q931_EP_INTERPRETER	in_xparms.unique_xparms.sig_q931.endpoint_id.interpreter	PT_UCHAR
CP_Q931_CAUSE	in_xparms.unique_xparms.sig_q931.cause.ie	PT_HEXSTR
CP_Q931_CAUSE_LASTMSG	in_xparms.unique_xparms.sig_q931.cause.last_msg	PT_UCHAR
CP_Q931_ADD_ORIG_ADDR	in_xparms.unique_xparms.sig_q931.additional_orig_addr	PT_HEXSTR
CP_Q931_ADD_ORIG_NUMBERING_TYPE	in_xparms.unique_xparms.sig_q931.add_orig_numbering_type	PT_UCHAR
CP_Q931_ADD_ORIG_NUMBERING_PLAN	in_xparms.unique_xparms.sig_q931.add_orig_numbering_plan	PT_UCHAR
CP_Q931_ADD_ORIG_NUMBERING_PRESENTATION	in_xparms.unique_xparms.sig_q931.add_orig_numbering_presentation	PT_UCHAR
CP_Q931_ADD_ORIG_NUMBERING_SCREENING	in_xparms.unique_xparms.sig_q931.add_orig_numbering_screening	PT_UCHAR
CP_Q931_OMIT_CALLING_PARTY_IE	in_xparms.unique_xparms.sig_q931.omit_calling_party_ie	PT_UCHAR
CP_Q931_CALL_REF	in_xparms.unique_xparms.sig_q931.call_ref_value	PT_ULONG
CP_DASS_SIC1	in_xparms.unique_xparms.sig_dass.sic1	PT_UCHAR
CP_DASS_SIC2	in_xparms.unique_xparms.sig_dass.sic2	PT_UCHAR
CP_DPNSS_SIC1	in_xparms.unique_xparms.sig_dpnss.sic1	PT_UCHAR
CP_DPNSS_SIC2	in_xparms.unique_xparms.sig_dpnss.sic2	PT_UCHAR
CP_DPNSS_CLC	in_xparms.unique_xparms.sig_dpnss.clc	PT_STRING
CP_CAS_CATEGORY	in_xparms.unique_xparms.sig_cas.category	PT_UCHAR
CP_ISUP_SERVICE_OCTET	in_xparms.unique_xparms.sig_isup.service_octet	PT_UCHAR
CP_ISUP_ADD_INFO_OCTET	in_xparms.unique_xparms.sig_isup.add_info_octet	PT_UCHAR
CP_ISUP_DEST_NATUREOF_ADDR	in_xparms.unique_xparms.sig_isup.dest_natureof_addr	PT_UCHAR
CP_ISUP_DEST_NUMBERING_PLAN	in_xparms.unique_xparms.sig_isup.dest_numbering_plan	PT_UCHAR
CP_ISUP_BEARER	in_xparms.unique_xparms.sig_isup.bearer.ie	PT_HEXSTR
CP_ISUP_BEARER_LASTMSG	in_xparms.unique_xparms.sig_isup.bearer.last_msg	PT_UCHAR
CP_ISUP_ORIG_NATUREOF_ADDR	in_xparms.unique_xparms.sig_isup.orig_natureof_addr	PT_UCHAR
CP_ISUP_ORIG_NUMBERING_PLAN	in_xparms.unique_xparms.sig_isup.orig_numbering_plan	PT_UCHAR
CP_ISUP_ORIG_NUMBERING_PRESENTATION	in_xparms.unique_xparms.sig_isup.orig_numbering_presentation	PT_UCHAR
CP_ISUP_ORIG_NUMBERING_SCREENING	in_xparms.unique_xparms.sig_isup.orig_numbering_screening	PT_UCHAR
CP_ISUP_CONN_NATUREOF_ADDR	in_xparms.unique_xparms.sig_isup.conn_natureof_addr	PT_UCHAR
CP_ISUP_CONN_NUMBERING_PLAN	in_xparms.unique_xparms.sig_isup.conn_numbering_plan	PT_UCHAR
CP_ISUP_CONN_NUMBERING_PRESENTATION	in_xparms.unique_xparms.sig_isup.conn_numbering_presentation	PT_UCHAR
CP_ISUP_CONN_NUMBERING_SCREENING	in_xparms.unique_xparms.sig_isup.conn_numbering_screening	PT_UCHAR
CP_ISUP_CONN_NUMBER_REQ	in_xparms.unique_xparms.sig_isup.conn_number_req	PT_UCHAR
CP_ISUP_ORIG_CATEGORY	in_xparms.unique_xparms.sig_isup.orig_category	PT_UCHAR

CP_ISUP_ORIG_NUMBER_INCOMPLETE	in_xparms.unique_xparms.sig_isup.orig_number_incomplete	PT_UCHAR
CP_ISUP_DEST_SUBADDR	in_xparms.unique_xparms.sig_isup.dest_subaddr	PT_HEXSTR
CP_ISUP_ORIG_SUBADDR	in_xparms.unique_xparms.sig_isup.orig_subaddr	PT_HEXSTR
CP_ISUP_HILAYER	in_xparms.unique_xparms.sig_isup.hilayer.ie	PT_HEXSTR
CP_ISUP_HILAYER_LASTMSG	in_xparms.unique_xparms.sig_isup.hilayer.last_msg	PT_UCHAR
CP_ISUP_LOLAYER	in_xparms.unique_xparms.sig_isup.lolayer.ie	PT_HEXSTR
CP_ISUP_LOLAYER_LASTMSG	in_xparms.unique_xparms.sig_isup.lolayer.last_msg	PT_UCHAR
CP_ISUP_PROGRESS_INDICATOR	in_xparms.unique_xparms.sig_isup.progress_indicator.ie	PT_HEXSTR
CP_ISUP_PROGRESS_LASTMSG	in_xparms.unique_xparms.sig_isup.progress_indicator.last_msg	PT_UCHAR
CP_ISUP_IN_BAND	in_xparms.unique_xparms.sig_isup.in_band	PT_UCHAR
CP_ISUP_NAT_INTER_CALL_IND	in_xparms.unique_xparms.sig_isup.nat_inter_call_ind	PT_UCHAR
CP_ISUP_INTERWORKING_IND	in_xparms.unique_xparms.sig_isup.interworking_ind	PT_UCHAR
CP_ISUP_ISDN_USERPART_IND	in_xparms.unique_xparms.sig_isup.isdn_userpart_ind	PT_UCHAR
CP_ISUP_ISDN_USERPART_PREF_IND	in_xparms.unique_xparms.sig_isup.isdn_userpart_pref_ind	PT_UCHAR
CP_ISUP_ISDN_ACCESS_IND	in_xparms.unique_xparms.sig_isup.isdn_access_ind	PT_UCHAR
CP_ISUP_DEST_INT_NW_IND	in_xparms.unique_xparms.sig_isup.dest_int_nw_ind	PT_UCHAR
CP_ISUP_CONTINUITY_CHECK_IND	in_xparms.unique_xparms.sig_isup.continuity_check_ind	PT_UCHAR
CP_ISUP_SATELLITE_IND	in_xparms.unique_xparms.sig_isup.satellite_ind	PT_UCHAR
CP_ISUP_CHARGE_IND	in_xparms.unique_xparms.sig_isup.charge_ind	PT_UCHAR
CP_ISUP_DEST_CATEGORY	in_xparms.unique_xparms.sig_isup.dest_category	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_QUAL	in_xparms.unique_xparms.sig_isup.add_calling_num_qualifier_ind	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_NOAI	in_xparms.unique_xparms.sig_isup.add_calling_num_natureof_addr	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_PLAN	in_xparms.unique_xparms.sig_isup.add_calling_num_plan	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_PRESENT	in_xparms.unique_xparms.sig_isup.add_calling_num_presentation	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_SCREEN	in_xparms.unique_xparms.sig_isup.add_calling_num_screening	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_INCOMP	in_xparms.unique_xparms.sig_isup.add_calling_num_incomplete	PT_UCHAR
CP_ISUP_ADD_CALL_NUM	in_xparms.unique_xparms.sig_isup.add_calling_num	PT_HEXSTR
CP_ISUP_EXCHANGE_TYPE	in_xparms.unique_xparms.sig_isup.exchange_type	PT_UCHAR
CP_ISUP_COLLECT_CALL	in_xparms.unique_xparms.sig_isup.collect_call_request_ind	PT_UCHAR
CP_IPTEL_DEST_DISPLAY	in_xparms.unique_xparms.sig_iptel.destination_display_name	PT_STRING
CP_IPTEL_ORIG_DISPLAY	in_xparms.unique_xparms.sig_iptel.originating_display_name	PT_STRING
CP_IPTEL_CODECS	in_xparms.unique_xparms.sig_iptel.codecs	PT_HEXSTR
CP_IPTEL_MEDIA_TDM_ENC	in_xparms.unique_xparms.sig_iptel.media_settings.tdm_encoding	PT_INT
CP_IPTEL_MEDIA_ENC_GAIN	in_xparms.unique_xparms.sig_iptel.media_settings.encode_gain	PT_INT
CP_IPTEL_MEDIA_DEC_GAIN	in_xparms.unique_xparms.sig_iptel.media_settings.decode_gain	PT_INT
CP_IPTEL_MEDIA_ECHO_CANC	in_xparms.unique_xparms.sig_iptel.media_settings.echo_cancellation	PT_INT
CP_IPTEL_MEDIA_ECHO_SUPP	in_xparms.unique_xparms.sig_iptel.media_settings.echo_suppression	PT_INT
CP_IPTEL_MEDIA_ECHO_SPAN	in_xparms.unique_xparms.sig_iptel.media_settings.echo_span	PT_INT
CP_IPTEL_MEDIA_RTP_TOS	in_xparms.unique_xparms.sig_iptel.media_settings.rtp_tos	PT_INT
CP_IPTEL_MEDIA_RTCP_TOS	in_xparms.unique_xparms.sig_iptel.media_settings.rtcp_tos	PT_INT
CP_IPTEL_MEDIA_DTMF_DET	in_xparms.unique_xparms.sig_iptel.media_settings.dtmf_detector	PT_INT
CP_IPTEL_VMPRXID	in_xparms.unique_xparms.sig_iptel.vmprxid	PT_HEXSTR
CP_IPTEL_VMPTXID	in_xparms.unique_xparms.sig_iptel.vmptxid	PT_HEXSTR
CP_IPTEL_MEDIA_CALL_TYPE	in_xparms.unique_xparms.sig_iptel.media_call_type	PT_STRING

CP_H323_DEST_ALIAS	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.destination_alias	PT_STRING
CP_H323_ORIG_ALIAS	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.originating_alias	PT_STRING
CP_H323_H245_TUNNELING	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.h245_tunneling	PT_INT
CP_H323_FASTSTART	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.faststart	PT_INT
CP_H323_EARLY_H245	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.early_h245	PT_INT
CP_H323_DTMF	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.dtmf	PT_STRING
CP_H323_PROGRESS_LOC	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.progress_location	PT_INT
CP_H323_PROGRESS_DESC	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.progress_description	PT_INT
CP_SIP_CONTACT_ADDR	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.contact_address	PT_STRING
CP_SIP_ZERO_CONN_ADDR_HOLD	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.zero_connection_address_hold	PT_INT
CP_SIP_DISABLE_REL_PROV	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.disable_reliable_provisional_response	PT_INT
CP_SIP_DISABLE_EARLY_MED	in_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.disable_early_media	PT_INT

In the previous tables the third column defines the type of value that the field can be set to and thus defines how the *value* argument of the CCsetparm() function call is treated. A description of each of these types is given below:

PT_CHAR	Single byte signed integer (range -127 to +127)
PT_UCHAR	Single byte unsigned integer (range (0 to 255)
PT_INT	Two byte unsigned integer (range -32767 to +32767)
PT_UINT	Two byte unsigned integer (range 0 to 65535)
PT_LONG	Four byte signed integer (range -2147483647 to +2147483647)
PT_ULONG	Four byte signed integer (range 0 to 4294967295)
PT_STRING	Literal string value
PT_HEXSTR	This is a special type where the data stored in the field is not one of the above types and is instead a multi-byte non-string field with arbitrary structure. For fields of this type the *value* passed to the CCsetparm() function should be in the form of a Hexadecimal string where each byte value is represented in the string by a two hexadecimal characters.

The PT_HEXSTR type above probably needs more clarification and the example shown below should make the use of this type more clear.

In the PARM_TYPE_IN and PARM_TYPE_OUT parameters the CP_IPCODECS field allows the list of supported IP codecs to be specified in order of priority. This parameter is in fact an array of ACU_CODEC structures which have the structure shown below:

```
struct acu_codec {
```

```
                              ACU_INT     codec_type;
                              ACU_INT     vad;
                              ACU_INT     fpp;
                               ACU_ULONG     options;
}
```

So each element in the array consists of a 10 byte structure which can be represented as a 20 digit HEXSTR. For example to set the CP_IPCODECS array to contain only one element with a codec_type of 1 (G711_ALAW), Voice activation off (vad=0) and frames per packet (fpp) to 2 then the following call would be made:

x=CCsetparm(port,chan,PARM_TYPE_IN,CP_IPCODECS,"01000000020000000000");

There are a number of helper functions that provide the means to build up these Hex strings bit by bit which will be described later.

NOTE: Once a parameter has been set in one of the extended structures then the extended version of the Aculab function call will be used from then on, until a call is made to CCclrparms() which clears all the values from the extended structure (see below).

The full set of *ParmID* values that are used in this function and which are specified in the above table are defined in the ACULAB.INC file as follows:

```
const CP_VALID                        =1;
const CP_STREAM                       =2;
const CP_TS                           =3;
const CP_CALLTYPE                     =4;
const CP_SENDING_COMPLETE             =5;
const CP_DESTINATION_ADDR             =6;
const CP_ORIGINATING_ADDR             =7;
const CP_CONNECTED_ADDR               =8;
const CP_FEATURE_INFORMATION          =9;
const CP_Q931_SERVICE_OCTET           =101;
const CP_Q931_ADD_INFO_OCTET          =102;
const CP_Q931_DEST_NUMBERING_TYPE     =103;
const CP_Q931_DEST_NUMBERING_PLAN     =104;
const CP_Q931_BEARER                  =105;
const CP_Q931_ORIG_NUMBERING_TYPE     =106;
const CP_Q931_ORIG_NUMBERING_PLAN     =107;
const CP_Q931_ORIG_NUMBERING_PRESENTATION =108;
const CP_Q931_ORIG_NUMBERING_SCREENING    =109;
const CP_Q931_CONN_NUMBERING_TYPE     =110;
const CP_Q931_CONN_NUMBERING_PLAN     =111;
const CP_Q931_CONN_NUMBERING_PRESENTATION =112;
const CP_Q931_CONN_NUMBERING_SCREENING    =113;
const CP_Q931_DEST_SUBADDR            =114;
const CP_Q931_ORIG_SUBADDR            =115;
const CP_Q931_HILAYER                 =116;
const CP_Q931_LOLAYER                 =117;
const CP_Q931_PROGRESS_INDICATOR      =118;
const CP_Q931_NOTIFY_INDICATOR        =119;
const CP_Q931_KEYPAD                  =120;
const CP_Q931_DISPLAY                 =121;
const CP_Q931_SLOTMAP                 =122;
const CP_Q931_EP_USID                 =123;
const CP_Q931_EP_TID                  =124;
const CP_Q931_EP_INTERPRETER          =125;
const CP_Q931_BEARER_LASTMSG          =126;
```

```
const CP_Q931_HILAYER_LASTMSG               =127;
const CP_Q931_LOLAYER_LASTMSG               =128;
const CP_Q931_PROGRESS_LASTMSG              =129;
const CP_Q931_NOTIFY_LASTMSG                =130;
const CP_Q931_KEYPAD_LASTMSG                =131;
const CP_Q931_DISPLAY_LASTMSG               =132;
const CP_Q931_CAUSE                         =133;
const CP_Q931_CAUSE_LASTMSG                 =134;
const CP_Q931_ADD_ORIG_ADDR                 =135;
const CP_Q931_ADD_ORIG_NUMBERING_TYPE         =136;
const CP_Q931_ADD_ORIG_NUMBERING_PLAN         =137;
const CP_Q931_ADD_ORIG_NUMBERING_PRESENTATION =138;
const CP_Q931_ADD_ORIG_NUMBERING_SCREENING    =139;
const CP_Q931_OMIT_CALLING_PARTY_IE           =140;
const CP_Q931_CALL_REF                      =141;
const CP_Q931_CONN_ADDR                     =142;
const CP_Q931_CAUSE_RAW                     =143;
const CP_Q931_CAUSE_LOC                     =144;
const CP_1TR6_SERVICE_OCTET                 =201;
const CP_1TR6_ADD_INFO_OCTET                =202;
const CP_1TR6_NUMBERING_TYPE                =203;
const CP_1TR6_NUMBERING_PLAN                =204;
const CP_1TR6_CAUSE_RAW                     =205;
const CP_DASS_SIC1                          =301;
const CP_DASS_SIC2                          =302;
const CP_DASS_CAUSE_RAW                     =303;
const CP_DPNSS_SIC1                         =401;
const CP_DPNSS_SIC2                         =402;
const CP_DPNSS_CLC                          =403;
const CP_DPNSS_CAUSE_RAW                    =404;
const CP_CAS_CATEGORY                       =501;
const CP_CAS_CAUSE_RAW                      =502;
const CP_ISUP_SERVICE_OCTET                 =601;
const CP_ISUP_ADD_INFO_OCTET                =602;
const CP_ISUP_DEST_NATUREOF_ADDR            =603;
const CP_ISUP_DEST_NUMBERING_PLAN           =604;
const CP_ISUP_BEARER                        =605;
const CP_ISUP_ORIG_NATUREOF_ADDR            =606;
const CP_ISUP_ORIG_NUMBERING_PLAN           =607;
const CP_ISUP_ORIG_NUMBERING_PRESENTATION   =608;
const CP_ISUP_ORIG_NUMBERING_SCREENING      =609;
const CP_ISUP_CONN_NATUREOF_ADDR            =610;
const CP_ISUP_CONN_NUMBERING_PLAN           =611;
const CP_ISUP_CONN_NUMBERING_PRESENTATION   =612;
const CP_ISUP_CONN_NUMBERING_SCREENING      =613;
const CP_ISUP_CONN_NUMBER_REQ               =614;
const CP_ISUP_ORIG_CATEGORY                 =615;
const CP_ISUP_ORIG_NUMBER_INCOMPLETE        =616;
const CP_ISUP_DEST_SUBADDR                  =617;
const CP_ISUP_ORIG_SUBADDR                  =618;
const CP_ISUP_HILAYER                       =619;
const CP_ISUP_LOLAYER                       =620;
const CP_ISUP_PROGRESS_INDICATOR            =621;
const CP_ISUP_IN_BAND                       =622;
const CP_ISUP_NAT_INTER_CALL_IND            = 623;
const CP_ISUP_INTERWORKING_IND              = 624;
const CP_ISUP_ISDN_USERPART_IND             = 625;
const CP_ISUP_ISDN_USERPART_PREF_IND        = 626;
const CP_ISUP_ISDN_ACCESS_IND               = 627;
const CP_ISUP_DEST_INT_NW_IND               = 628;
const CP_ISUP_CONTINUITY_CHECK_IND          = 629;
const CP_ISUP_SATELLITE_IND                 = 630;
const CP_ISUP_CHARGE_IND                    = 631;
const CP_ISUP_BEARER_LASTMSG                = 632;
const CP_ISUP_HILAYER_LASTMSG               = 633;
```

```
const CP_ISUP_LOLAYER_LASTMSG          = 634;
const CP_ISUP_PROGRESS_LASTMSG         = 635;
const CP_ISUP_DEST_CATEGORY            = 636;
const CP_ISUP_ADDCALL_NUM_QUAL         = 637;
const CP_ISUP_ADD_CALL_NUM_NOAI        = 638;
const CP_ISUP_ADD_CALL_NUM_PLAN        = 639;
const CP_ISUP_ADD_CALL_NUM_PRESENT     = 640;
const CP_ISUP_ADD_CALL_NUM_SCREEN      = 641;
const CP_ISUP_ADD_CALL_NUM_INCOMP      = 642;
const CP_ISUP_ADD_CALL_NUM             = 642;
const CP_ISUP_EXCHANGE_TYPE            = 643;
const CP_ISUP_COLLECT_CALL             = 644;
const CP_ISUP_ACC_IND_VALID            = 645;
const CP_ISUP_ACC_IND_VALUE            = 646;
const CP_ISUP_USERP_IND_VALID          = 647;
const CP_ISUP_USERP_IND_VALUE          = 648;
const CP_ISUP_INTERW_IND_VALID         = 649;
const CP_ISUP_INTERW_IND_VALUE         = 650;
const CP_ISUP_CONN_ADDR                = 651;
const CP_ISUP_CAUSE_RAW                = 652;
const CP_ISUP_CAUSE_LOC                = 653;
const CP_ISUP_REATTEMPT                = 654;
const CP_IPTEL_DEST_DISPLAY            = 700;
const CP_IPTEL_ORIG_DISPLAY            = 701;
const CP_IPTEL_CODECS                  = 702;
const CP_IPTEL_MEDIA_TDM_ENC           = 703;
const CP_IPTEL_MEDIA_ENC_GAIN          = 704;
const CP_IPTEL_MEDIA_DEC_GAIN          = 705;
const CP_IPTEL_MEDIA_ECHO_CANC         = 706;
const CP_IPTEL_MEDIA_ECHO_SUPP         = 707;
const CP_IPTEL_MEDIA_ECHO_SPAN         = 708;
const CP_IPTEL_MEDIA_RTP_TOS           = 709;
const CP_IPTEL_MEDIA_RTCP_TOS          = 710;
const CP_IPTEL_MEDIA_DTMF_DET          = 711;
const CP_IPTEL_VMPRXID                 = 712;
const CP_IPTEL_VMPTXID                 = 713;
const CP_IPTEL_MEDIA_CALL_TYPE         = 714;
const CP_H323_DEST_ALIAS               = 715;
const CP_H323_ORIG_ALIAS               = 716;
const CP_H323_H245_TUNNELING           = 717;
const CP_H323_FASTSTART                = 718;
const CP_H323_EARLY_H245               = 719;
const CP_H323_DTMF                     = 710;
const CP_H323_PROGRESS_LOC             = 711;
const CP_H323_PROGRESS_DESC            = 712;
const CP_SIP_CONTACT_ADDR              = 713;
const CP_SIP_ZERO_CONN_ADDR_HOLD       = 714;
const CP_SIP_DISABLE_REL_PROV          = 715;
const CP_SIP_DISABLE_EARLY_MED         = 716;
const CP_SIP_SEND_EARLY_MED            = 717;
const CP_SIP_USE_183                   = 718;
const CP_SIP_SEND_REL_PROV             = 719;
```

Returns: This function returns 0 if successful or a negative error code.

-o-

CCclrparms

Synopsis:

CCclrparms(port,channel,ParmType)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
ParmType – The parameter type to clear

Description: This function clears all the fields from the Aculab extended structure specified by the *ParmType* argument so that the non-extended version of the relevant Aculab call with be used next time a call is made.

The *ParmType* can be one of the following as defined in the ACULAB.INC:

```
const  PARM_TYPE_OUT         =0;   # For CCmkcall()
const  PARM_TYPE_IN          =1;   # For CCenablein()
const  PARM_TYPE_ALERTING    =2;   # For CCalerting()
const  PARM_TYPE_ACCEPT      =3;   # For CCaccept()
const  PARM_TYPE_DISCON      =4;   # For CCdisconnect()/CCrelease()
const  PARM_TYPE_HOLD        =5;   # COMING SOON
const  PARM_TYPE_GETADDR     =6;   # COMING SOON
```

Returns: This function returns 0 upon success or a negative error code.

-o-

CCgetparm

Synopsis:

CCgetparm(port, channel, parmId, &pVar)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
ParmID - The ID of the parameter to get
pVar - Pointer to a variable to hold the returned parameter value.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_details(DETAIL_XPARMS *detailsp);
```

It allows for various parameters to be obtained relating to the current call. The *ParmID* is a unique identifier that maps to one of the fields from the DETAIL_XPARMS structure or the DETAIL_XPARMS.unique_xparms structure returned by the *call_details()* function call.

For example to obtain the *destination_addr* and *originating_addr* fields from DETAIL_XPARMS structure, one would have som,ething like the following:

```
 var  DID:64,CLID:64;
CCgetparm(port,chan,CP_DESTINATION_ADDR,&DID);
CCgetparm(port,chan,CP_ORIGINATING_ADDR,&CLID);
```

The list of *ParmID* constants (as defined in ACULAB.INC) and their mapping to the DETAIL_XPARMS structure is shown in the table below:

parmID	Structure and Field it maps to:	Field type

CP_VALID	detail_xparms.valid	PT_INT
CP_STREAM	detail_xparms.stream	PT_INT
CP_TS	detail_xparms.ts	PT_INT
CP_CALLTYPE	detail_xparms.calltype	PT_INT
CP_SENDING_COMPLETE	detail_xparms.sending_complete	PT_INT
CP_DESTINATION_ADDR	detail_xparms.destination_addr	PT_STRING
CP_ORIGINATING_ADDR	detail_xparms.originating_addr	PT_STRING
CP_CONNECTED_ADDR	detail_xparms.connected_addr	PT_STRING
CP_FEATURE_INFORMATION	detail_xparms.feature_information	PT_ULONG
CP_Q931_SERVICE_OCTET	detail_xparms.unique_xparms.sig_q931.service_octet	PT_UCHAR
CP_Q931_ADD_INFO_OCTET	detail_xparms.unique_xparms.sig_q931.add_info_octet	PT_UCHAR
CP_Q931_DEST_NUMBERING_TYPE	detail_xparms.unique_xparms.sig_q931.dest_numbering_type	PT_UCHAR
CP_Q931_DEST_NUMBERING_PLAN	detail_xparms.unique_xparms.sig_q931.dest_numbering_plan	PT_UCHAR
CP_Q931_BEARER	detail_xparms.unique_xparms.sig_q931.bearer.ie	PT_HEXSTR
CP_Q931_BEARER_LASTMSG	detail_xparms.unique_xparms.sig_q931.bearer.last_msg	PT_UCHAR
CP_Q931_ORIG_NUMBERING_TYPE	detail_xparms.unique_xparms.sig_q931.orig_numbering_type	PT_UCHAR
CP_Q931_ORIG_NUMBERING_PLAN	detail_xparms.unique_xparms.sig_q931.orig_numbering_plan	PT_UCHAR
CP_Q931_ORIG_NUMBERING_PRESENTATION	detail_xparms.unique_xparms.sig_q931.orig_numbering_presentation	PT_UCHAR
CP_Q931_ORIG_NUMBERING_SCREENING	detail_xparms.unique_xparms.sig_q931.orig_numbering_screening	PT_UCHAR
CP_Q931_CONN_NUMBERING_TYPE	detail_xparms.unique_xparms.sig_q931.conn_numbering_type	PT_UCHAR
CP_Q931_CONN_NUMBERING_PLAN	detail_xparms.unique_xparms.sig_q931.conn_numbering_plan	PT_UCHAR
CP_Q931_CONN_NUMBERING_PRESENTATIO N	detail_xparms.unique_xparms.sig_q931.conn_numbering_presentation	PT_UCHAR
CP_Q931_CONN_NUMBERING_SCREENING	detail_xparms.unique_xparms.sig_q931.conn_numbering_screening	PT_UCHAR
CP_Q931_DEST_SUBADDR	detail_xparms.unique_xparms.sig_q931.dest_subaddr	PT_HEXSTR
CP_Q931_ORIG_SUBADDR	detail_xparms.unique_xparms.sig_q931.orig_subaddr	PT_HEXSTR
CP_Q931_HILAYER	detail_xparms.unique_xparms.sig_q931.hilayer.ie	PT_HEXSTR
CP_Q931_HILAYER_LASTMSG	detail_xparms.unique_xparms.sig_q931.hilayer.last_msg	PT_UCHAR
CP_Q931_LOLAYER	detail_xparms.unique_xparms.sig_q931.lolayer.ie	PT_HEXSTR
CP_Q931_LOLAYER_LASTMSG	detail_xparms.unique_xparms.sig_q931.lolayer.last_msg	PT_UCHAR
CP_Q931_PROGRESS_INDICATOR	detail_xparms.unique_xparms.sig_q931.progress_indicator.ie	PT_HEXSTR
CP_Q931_PROGRESS_LASTMSG	detail_xparms.unique_xparms.sig_q931.progress_indicator.last_msg	PT_UCHAR
CP_Q931_NOTIFY_INDICATOR	detail_xparms.unique_xparms.sig_q931.notify_indicator.ie	PT_HEXSTR
CP_Q931_NOTIFY_LASTMSG	detail_xparms.unique_xparms.sig_q931.notify_indicator.last_msg	PT_UCHAR
CP_Q931_KEYPAD	detail_xparms.unique_xparms.sig_q931.keypad.ie	PT_HEXSTR
CP_Q931_KEYPAD_LASTMSG	detail_xparms.unique_xparms.sig_q931.keypad.last_msg	PT_UCHAR
CP_Q931_DISPLAY	detail_xparms.unique_xparms.sig_q931.display.ie	PT_HEXSTR
CP_Q931_DISPLAY_LASTMSG	detail_xparms.unique_xparms.sig_q931.display.last_msg	PT_UCHAR
CP_Q931_SLOTMAP	detail_xparms.unique_xparms.sig_q931.slotmap	PT_LONG
CP_Q931_EP_USID	detail_xparms.unique_xparms.sig_q931.endpoint_id.usid	PT_UCHAR
CP_Q931_EP_TID	detail_xparms.unique_xparms.sig_q931.endpoint_id.tid	PT_UCHAR
CP_Q931_EP_INTERPRETER	detail_xparms.unique_xparms.sig_q931.endpoint_id.interpreter	PT_UCHAR
CP_Q931_CAUSE	detail_xparms.unique_xparms.sig_q931.cause.ie	PT_HEXSTR
CP_Q931_CAUSE_LASTMSG	detail_xparms.unique_xparms.sig_q931.cause.last_msg	PT_UCHAR
CP_Q931_ADD_ORIG_ADDR	detail_xparms.unique_xparms.sig_q931.additional_orig_addr	PT_HEXSTR
CP_Q931_ADD_ORIG_NUMBERING_TYPE	detail_xparms.unique_xparms.sig_q931.add_orig_numbering_type	PT_UCHAR
CP_Q931_ADD_ORIG_NUMBERING_PLAN	detail_xparms.unique_xparms.sig_q931.add_orig_numbering_plan	PT_UCHAR
CP_Q931_ADD_ORIG_NUMBERING_PRESENT ATION	detail_xparms.unique_xparms.sig_q931.add_orig_numbering_presentation	PT_UCHAR
CP_Q931_ADD_ORIG_NUMBERING_SCREENI NG	detail_xparms.unique_xparms.sig_q931.add_orig_numbering_screening	PT_UCHAR

CP_Q931_OMIT_CALLING_PARTY_IE	detail_xparms.unique_xparms.sig_q931.omit_calling_party_ie	PT_UCHAR
CP_Q931_CALL_REF	detail_xparms.unique_xparms.sig_q931.call_ref_value	PT_ULONG
CP_DASS_SIC1	detail_xparms.unique_xparms.sig_dass.sic1	PT_UCHAR
CP_DASS_SIC2	detail_xparms.unique_xparms.sig_dass.sic2	PT_UCHAR
CP_DPNSS_SIC1	detail_xparms.unique_xparms.sig_dpnss.sic1	PT_UCHAR
CP_DPNSS_SIC2	detail_xparms.unique_xparms.sig_dpnss.sic2	PT_UCHAR
CP_DPNSS_CLC	detail_xparms.unique_xparms.sig_dpnss.clc	PT_STRING
CP_CAS_CATEGORY	detail_xparms.unique_xparms.sig_cas.category	PT_UCHAR
CP_ISUP_SERVICE_OCTET	detail_xparms.unique_xparms.sig_isup.service_octet	PT_UCHAR
CP_ISUP_ADD_INFO_OCTET	detail_xparms.unique_xparms.sig_isup.add_info_octet	PT_UCHAR
CP_ISUP_DEST_NATUREOF_ADDR	detail_xparms.unique_xparms.sig_isup.dest_natureof_addr	PT_UCHAR
CP_ISUP_DEST_NUMBERING_PLAN	detail_xparms.unique_xparms.sig_isup.dest_numbering_plan	PT_UCHAR
CP_ISUP_BEARER	detail_xparms.unique_xparms.sig_isup.bearer.ie	PT_HEXSTR
CP_ISUP_BEARER_LASTMSG	detail_xparms.unique_xparms.sig_isup.bearer.last_msg	PT_UCHAR
CP_ISUP_ORIG_NATUREOF_ADDR	detail_xparms.unique_xparms.sig_isup.orig_natureof_addr	PT_UCHAR
CP_ISUP_ORIG_NUMBERING_PLAN	detail_xparms.unique_xparms.sig_isup.orig_numbering_plan	PT_UCHAR
CP_ISUP_ORIG_NUMBERING_PRESENTATION	detail_xparms.unique_xparms.sig_isup.orig_numbering_presentation	PT_UCHAR
CP_ISUP_ORIG_NUMBERING_SCREENING	detail_xparms.unique_xparms.sig_isup.orig_numbering_screening	PT_UCHAR
CP_ISUP_CONN_NATUREOF_ADDR	detail_xparms.unique_xparms.sig_isup.conn_natureof_addr	PT_UCHAR
CP_ISUP_CONN_NUMBERING_PLAN	detail_xparms.unique_xparms.sig_isup.conn_numbering_plan	PT_UCHAR
CP_ISUP_CONN_NUMBERING_PRESENTATIO N	detail_xparms.unique_xparms.sig_isup.conn_numbering_presentation	PT_UCHAR
CP_ISUP_CONN_NUMBERING_SCREENING	detail_xparms.unique_xparms.sig_isup.conn_numbering_screening	PT_UCHAR
CP_ISUP_CONN_NUMBER_REQ	detail_xparms.unique_xparms.sig_isup.conn_number_req	PT_UCHAR
CP_ISUP_ORIG_CATEGORY	detail_xparms.unique_xparms.sig_isup.orig_category	PT_UCHAR
CP_ISUP_ORIG_NUMBER_INCOMPLETE	detail_xparms.unique_xparms.sig_isup.orig_number_incomplete	PT_UCHAR
CP_ISUP_DEST_SUBADDR	detail_xparms.unique_xparms.sig_isup.dest_subaddr	PT_HEXSTR
CP_ISUP_ORIG_SUBADDR	detail_xparms.unique_xparms.sig_isup.orig_subaddr	PT_HEXSTR
CP_ISUP_HILAYER	detail_xparms.unique_xparms.sig_isup.hilayer.ie	PT_HEXSTR
CP_ISUP_HILAYER_LASTMSG	detail_xparms.unique_xparms.sig_isup.hilayer.last_msg	PT_UCHAR
CP_ISUP_LOLAYER	detail_xparms.unique_xparms.sig_isup.lolayer.ie	PT_HEXSTR
CP_ISUP_LOLAYER_LASTMSG	detail_xparms.unique_xparms.sig_isup.lolayer.last_msg	PT_UCHAR
CP_ISUP_PROGRESS_INDICATOR	detail_xparms.unique_xparms.sig_isup.progress_indicator.ie	PT_HEXSTR
CP_ISUP_PROGRESS_LASTMSG	detail_xparms.unique_xparms.sig_isup.progress_indicator.last_msg	PT_UCHAR
CP_ISUP_IN_BAND	detail_xparms.unique_xparms.sig_isup.in_band	PT_UCHAR
CP_ISUP_NAT_INTER_CALL_IND	detail_xparms.unique_xparms.sig_isup.nat_inter_call_ind	PT_UCHAR
CP_ISUP_INTERWORKING_IND	detail_xparms.unique_xparms.sig_isup.interworking_ind	PT_UCHAR
CP_ISUP_ISDN_USERPART_IND	detail_xparms.unique_xparms.sig_isup.isdn_userpart_ind	PT_UCHAR
CP_ISUP_ISDN_USERPART_PREF_IND	detail_xparms.unique_xparms.sig_isup.isdn_userpart_pref_ind	PT_UCHAR
CP_ISUP_ISDN_ACCESS_IND	detail_xparms.unique_xparms.sig_isup.isdn_access_ind	PT_UCHAR
CP_ISUP_DEST_INT_NW_IND	detail_xparms.unique_xparms.sig_isup.dest_int_nw_ind	PT_UCHAR
CP_ISUP_CONTINUITY_CHECK_IND	detail_xparms.unique_xparms.sig_isup.continuity_check_ind	PT_UCHAR
CP_ISUP_SATELLITE_IND	detail_xparms.unique_xparms.sig_isup.satellite_ind	PT_UCHAR
CP_ISUP_CHARGE_IND	detail_xparms.unique_xparms.sig_isup.charge_ind	PT_UCHAR
CP_ISUP_DEST_CATEGORY	detail_xparms.unique_xparms.sig_isup.dest_category	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_QUAL	detail_xparms.unique_xparms.sig_isup.add_calling_num_qualifier_ind	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_NOAI	detail_xparms.unique_xparms.sig_isup.add_calling_num_natureof_addr	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_PLAN	detail_xparms.unique_xparms.sig_isup.add_calling_num_plan	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_PRESENT	detail_xparms.unique_xparms.sig_isup.add_calling_num_presentation	PT_UCHAR
CP_ISUP_ADD_CALL_NUM_SCREEN	detail_xparms.unique_xparms.sig_isup.add_calling_num_screening	PT_UCHAR

CP_ISUP_ADD_CALL_NUM_INCOMP	detail_xparms.unique_xparms.sig_isup.add_calling_num_incomplete	PT_UCHAR
CP_ISUP_ADD_CALL_NUM	detail_xparms.unique_xparms.sig_isup.add_calling_num	PT_HEXSTR
CP_ISUP_EXCHANGE_TYPE	detail_xparms.unique_xparms.sig_isup.exchange_type	PT_UCHAR
CP_ISUP_COLLECT_CALL	detail_xparms.unique_xparms.sig_isup.collect_call_request_ind	PT_UCHAR
CP_IPTEL_DEST_DISPLAY	detail_xparms.unique_xparms.sig_iptel.destination_display_name	PT_STRING
CP_IPTEL_ORIG_DISPLAY	detail_xparms.unique_xparms.sig_iptel.originating_display_name	PT_STRING
CP_IPTEL_CODECS	detail_xparms.unique_xparms.sig_iptel.codecs	PT_HEXSTR
CP_IPTEL_MEDIA_TDM_ENC	detail_xparms.unique_xparms.sig_iptel.media_settings.tdm_encoding	PT_INT
CP_IPTEL_MEDIA_ENC_GAIN	detail_xparms.unique_xparms.sig_iptel.media_settings.encode_gain	PT_INT
CP_IPTEL_MEDIA_DEC_GAIN	detail_xparms.unique_xparms.sig_iptel.media_settings.decode_gain	PT_INT
CP_IPTEL_MEDIA_ECHO_CANC	detail_xparms.unique_xparms.sig_iptel.media_settings.echo_cancellation	PT_INT
CP_IPTEL_MEDIA_ECHO_SUPP	detail_xparms.unique_xparms.sig_iptel.media_settings.echo_suppression	PT_INT
CP_IPTEL_MEDIA_ECHO_SPAN	detail_xparms.unique_xparms.sig_iptel.media_settings.echo_span	PT_INT
CP_IPTEL_MEDIA_RTP_TOS	detail_xparms.unique_xparms.sig_iptel.media_settings.rtp_tos	PT_INT
CP_IPTEL_MEDIA_RTCP_TOS	detail_xparms.unique_xparms.sig_iptel.media_settings.rtcp_tos	PT_INT
CP_IPTEL_MEDIA_DTMF_DET	detail_xparms.unique_xparms.sig_iptel.media_settings.dtmf_detector	PT_INT
CP_IPTEL_VMPRXID	detail_xparms.unique_xparms.sig_iptel.vmprxid	PT_HEXSTR
CP_IPTEL_VMPTXID	detail_xparms.unique_xparms.sig_iptel.vmptxid	PT_HEXSTR
CP_IPTEL_MEDIA_CALL_TYPE	detail_xparms.unique_xparms.sig_iptel.media_call_type	PT_STRING
CP_H323_DEST_ALIAS	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.destination_alias	PT_STRING
CP_H323_ORIG_ALIAS	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.originating_alias	PT_STRING
CP_H323_H245_TUNNELING	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.h245_tunneling	PT_INT
CP_H323_FASTSTART	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.faststart	PT_INT
CP_H323_EARLY_H245	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.early_h245	PT_INT
CP_H323_DTMF	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.dtmf	PT_STRING
CP_H323_PROGRESS_LOC	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.progress_location	PT_INT
CP_H323_PROGRESS_DESC	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_h323.progress_description	PT_INT
CP_SIP_CONTACT_ADDR	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.contact_address	PT_STRING
CP_SIP_ZERO_CONN_ADDR_HOLD	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.zero_connection_address_hold	PT_INT
CP_SIP_DISABLE_REL_PROV	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.disable_reliable_provisional_response	PT_INT
CP_SIP_DISABLE_EARLY_MED	detail_xparms.unique_xparms.sig_iptel.protocol_specific.sig_sip.disable_early_media	PT_INT

In the previous tables the third column defines the type of value that the field will be returned as. A description of each of these types is given below:

PT_CHAR	Single byte signed integer (range -127 to +127)
PT_UCHAR	Single byte unsigned integer (range (0 to 255)
PT_INT	Two byte unsigned integer (range -32767 to +32767)
PT_UINT	Two byte unsigned integer (range 0 to 65535)
PT_LONG	Four byte signed integer (range -2147483647 to +2147483647)

PT_ULONG	Four byte signed integer (range 0 to 4294967295)
PT_STRING	Literal string value
PT_HEXSTR	This is a special type where the data stored in the field is not one of the above types and is instead a multi-byte non-string field with arbitrary structure. For fields of this type the value returned *by* the CCgetparm() function will be in the form of a Hexadecimal string where each byte value is represented in the string by a two hexadecimal characters. See CCsetparm() for more information about HEXSTR types.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCalerting

Synopsis:
CCalerting(port, channel)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.

Description: This function maps to the following Aculab functions:

ACU_ERR call_incoming_ringing(int *handle*);
ACU_ERR xcall_incoming_ringing(INCOMING_RINGING_XPARMS **ringingp*);

Used to send and alerting (or ringing) message to the network causing the caller to hear the ring tone. This function is used after an incoming call has been detected but before the call has been accepted. Use of the function will stop further call details, such as DDI digits, from being received.
If any parameters have been set using the CCsetparm() function with a *ParmType* of PARM_TYPE_ALERTING then the extended version of the function will be called.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCgetcause

Synopsis:
CCgetcause(port, channel[,std_or_raw])

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.
[std_or_raw] – Optional parameter to define whether to return the standard (0) or raw (1) cause value.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_getcause(CAUSE_XPARMS *causep);
```

This function is used to return the clearing cause when an incoming or outgoing call clears. The returned clearing cause will only be valid when the call state is either EV_IDLE or EV_REMOTE_DISCONNECT.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCoverlap

Synopsis:
CCoverlap(port, channel,dest_addr,sending_complete)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.
dest_addr – The digits to send.
sending_complete – Set to 0 if there may be more digits to follow, or 1 if all digits have been sent.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_send_overlap(OVERLAP_XPARMS *overlapp);
```

This function may be used to send the destination address of an outgoing call one or more digits at a time. The function may also be used any time that a valid outgoing call handle is available and the state of the call is CS_WAIT_FOR_OUTGOING.

The *dest_addr* holds the digits to send (one of more) of the destination address.

The *sending_complete* should be set to 1 when all digits of the destination address have been sent.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCgetcharge

Synopsis:
CCgetcharge(port, timeslot,&pType,&pCharge,&pMeter)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.
pType - Pointer to a variable that wiull hold the charging type
pCharge - Pointer to a variable holding the returned charging information (returned as a 34byte (68 char) hexi-string)

pMeter - Pointer to a variable holding the number of metering pulse received.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_get_charge(GET_CHARGE_XPARMS *chargep);
```

This function obtains information regarding the cost of a call. The function may be used any time that a valid call handle is available, however, the call charge information may not be valid until the call has cleared and the call has gone to the EV_IDLE state. The function provides for the receipt of call charging information and/or the accumulation of meter pulses.

The *pType* argument is a pointer to a varibale that will hold the received charging type which will be one of the following values (As defined in ACULAB.INC):

```
const CHARGE_NONE       =    0;
const CHARGE_INFO       =    1;
const CHARGE_METER  =    2;
```

The meaning of these types is described below:

• CHARGE_NONE - There is no valid charging information available in either the charge or meter fields.
• CHARGE_INFO - The information contained in the element charge is valid and may be used.
• CHARGE_METER -The information contained within the meter element is valid and may be used.
• CHARGE_INFO + CHARGE_METER - The information contained in both the charge and meter elements is valid and may be used.

The pC*harge* argument is a pointer to a variable that will contain the charging information, returned as as 34byte (68 char) hexstring.

The pMeter argument is a pointer to a variable that will hold the number of metering pulses.

Returns: Returns 0 if successful or a negative error code

-o-

CCsetupack

Synopsis:
CCsetupack(port, channel[,progress,Display])

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.
[progress] – option progress indicators supplied as a hexadecimal string
[Display] - option display indicators supplied as a hexadecimal string

Description: This function maps to the following Aculab function:

```
ACU_ERR call_setup_ack(SETUP_ACK_XPARMS *setup_ackp);
```

This function may be used on an incoming call to send a Q.931 SETUP_ACKNOWLEDGE message to the calling party.

The optional *progress* or *display* parameters can be used to set the progress indicator or display fields in the Aculab sig_q931 structure. These must be passed as hexadecimal strings where each byte is represented by two hexadecimal string characters.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCproceed

Synopsis:

CCproceed(port, channel[,unique_hex])

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
[unique_hex] – Optional: Hexidecimal string representation of the Aculab PROCEEDING_XPARMS.unique_xparms structure.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_proceeding(PROCEEDING_XPARMS *proceedingp);
```

This function may be used on an incoming call to send a message to the calling party to indicate that sufficient information has been obtained to proceed with the call.

If any of the elements of the PROCEEDING_XPARMS.unique_xparms structure need to be set then the *unique_hex* parameter allows a hexadecimal string representation of this structure to be passed.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCprogress

Synopsis:

CCprogress(port, channel[,progress[,Display]])

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
[progress] – option progress indicators supplied as a hexadecimal string
[Display] - option display indicators supplied as a hexadecimal string

Description: This function maps to the following Aculab function:

```
ACU_ERR call_progress(PROGRESS_XPARMS *progressp);
```

This function may be used to send call progress information to the network. This function may be used on an incoming call in the event of interworking or to indicate that in-band information is now available.

The optional *progress* or *display* parameters can be used to set the progress indicator or display fields in the Aculab sig_q931 structure. These must be passed as hexadecimal strings where each byte is represented by two hexadecimal string characters.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCgetaddr

Synopsis:
CCgetaddr(port, channel)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_get_originating_addr(int handle);
```

This function may be used to obtain the originating address of an incoming call in some Channel Associated Signalling (CAS) systems and ISUP variants where the application must explicitly request the originating address from the network. After calling this function the call state will change to CS_DETAILS once the originating address is available after which a call to *CCgetparm ()* can be used to obtain the originating address.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCanscode

Synopsis:
CCanscode(port, channel,code)
Arguments:
port – The logical E1/T1 port number.
channel – The channel number.
code – The answer code to send
Description: This function maps to the following Aculab function:

```
ACU_ERR call_answercode(CAUSE_XPARMS *answerp);
```

This function allows for an answer code to be sent during the CS_CALL_CONNECTED state which provides information about how the call is to be handled. This is primarily used for CAS protocols and will be ignored for protocols where this is not relevant.

© Zentel Telecom Ltd, 2009

Below are the list of supported answer codes as defined in ACULAB.INC:

```
const AC_NORMAL          =0; # default acceptance code
const AC_CHARGE          =100; # answer call with charging
const AC_NOCHARGE        =101; # answer call without charging
const AC_LAST_RELEASE    =102; # last party release
const AC_SPARE1          =103; # spare
const AC_SPARE2          =104; # spare
```

Returns: This function returns 0 upon success or a negative error code.

-o-

CCputcharge

Synopsis:

CCputcharge(port, channel,charge[,meter])

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
charge – The charging information passed as a 34byte (68 char) hexi-string
[meter] – optional parmater specifiying the number of metering pulses.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_put_charge(PUT_CHARGE_XPARMS *chargep);
```

This function may be used to send call charging information on the network and may be used any time that a valid call handle is available and the call is in the CS_CALL_CONNECTED state. It should be noted that it is normally only possible to send this information from a Network end protocol. The function provides for sending of call charge information and/or meter pulses. The choice of information is dependent upon the type of signalling system supported by the device driver.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCnotify

Synopsis:

CCnotify(port, channel,notify_indicator)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
notify_indicator – The notify indicator passed as a hexadecimal string.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_notify(NOTIFY_XPARMS *notifyp);
```

This function may be used on a call to send a message to the network to indicate an appropriate call related event during the active state of a call (such as user suspended). This function is supported in some Q.931 protocols. This function is dependent on the signalling system and reference should be made to the appropriate specification for the protocol.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCkeypad

Synopsis:

CCkeypad(port, channel,keypadinfo[,display])

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
keypadinfo –The keypadinfo specified as a hexadecimal string
[display] – Optional display information for Q931 procotol (As hexadecimal string)

Description: This function maps to the following Aculab function:

```
ACU_ERR call_send_keypad_info(KEYPAD_XPARMS *keypadp);
```

This function may be used to send keypad information during a call. This function is only supported in Q.931 and H.323 protocols. The *keypadinfo* and optional *display* parameter must be passed as hexadecimal strings.

Returns: This function returns 0 upon success or a negative error code.

-o-

CChold

Synopsis:

CChold(port, channel)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_hold(int handle);
```

This function allows an incoming or outgoing call to be put on hold. If the call is successful then a second call handle is returned on this *port* and *channel*. In order to distinguish between the original call handle and the new call handle on a *port* and *channel* then a call to the CCsetparty() must be made.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCreconnect

Synopsis:

CCreconnect(port, channel)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_reconnect(int handle);
```

This function causes a call that is in the held state to be reconnected.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCenquiry

Synopsis:

CCenquiry(port, channel,DID,CID,sending_complete[,cnf_parm1,cnf_parm2]....)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
DID – The destination address
CID – The originating address
[send_comp] – Optional Sending complete flag
[cnf_parm1,cnf_parm2..] – Optional cnf_parms

Description: This function maps to the following Aculab function:

```
ACU_ERR call_enquiry (struct out_xparms *enquiryp);
```

During the process of call transfer, this function allows an application to make an enquiry call, that is, an outgoing call to a third party. The function is essentially the same as CCmkcall (), having all of the same call states and events.

The DID and CID arguments specify the destination and originating addresses respectively. The option *send_comp* argument allows the OUT_XPARMS.sending_complete flag to be set and should be set to 0 for overlap sending (more digits to come) or 1 for en-bloc sending. The default is en-bloc sending if this aragument is not given.

The optional *cnf_parms* allow for the OUT_XPARMS.cnf field to be set. If one or more of these optional *cnf_parms* are specified then they are each ORed in turn with OUT_XPARMS.cnf field. If no *cnf_parms* are specifed then by default the IN_XPARMS.cnf field is set to CNF_REM_DISC which stops the channel automatically returning to the idle state when a remote end disconnect occurs (instead the CCrelease() call must be used to return the channel to CS_IDLE

state).

See the Aculab documentation for the call_enquiry() function for a more detailed description of the cnf field values.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCsetparty

Synopsis:

CCsetparty(port, channel,party)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
party – The call party (0 or 1)

Description: After a successful call to CChold() a second call handle is assigned to the specified *port* and *channel*. To allow both calls to be managed then the call party can be specified to define which call handle to use prior to making other calls to CXACULAB.DLL function calls. The original call handle on the *port* and *channel* is defined as party 0 and the call handle for the call created by the CChold() call is defined as party 1.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCtransfer

Synopsis:

CCtransfer(Aport, Achannel,Cport,Cchannel)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_transfer(TRANSFER_XPARMS transferp);
```

This function allows a call to be transferred after an successful call to CCenquiry(). The call to CCenquiry() is deemed to be successful if the state of the enquiry call becomes CS_CONNECTED or CS_OUTGOING_RINGING (for 'blind' transfer).

Returns: This function returns 0 upon success or a negative error code.

-o-

CCgetxparm

Synopsis:

CCgetxparm(port, channel, parmId, &pVar[,connectionless_flag])

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
parmId - The Feature detail parameter ID
pVar – Pointer to a variable to hold the return value
[connectionless_flag] – If this flag is set then the details are retrieved from the port specific FEATURE_DETAIL_XPAMRS structure and the *channel* is ignored.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_feature_details (FEATURE_DETAIL_XPARMS *feature_detailsp);
```

This function allows the retrieval of supplementary service information which may arrive at different stages during the lifetime of a call. After a call to CCgetparm() if the CP_FEATURE_INFORMATION field is set then this indicates that there are some feature details that can be retrieved through a call to CCgetxparm().

Note that this function is also used to retrieve connectionless feature information which is retrieved through a call to *CCgetcnctless(port).* To retrieve the connectionless feature details then the *[connectionless_flag]* should be set to 1 and the *channel* set to 0.

The list of *ParmID* constants (as defined in ACULAB.INC) and their mapping to the FEATURE_DETAIL_XPARMS structure is shown in the table below:

parmID	Structure and Field it maps to:	Field type
FP_MSG_CONTROL	get_feature_parm.message_control	PT_INT
FP_UUI_COMMAND	get_feature_parm.feature.uui.command	PT_INT
FP_UUI_REQUEST	get_feature_parm.feature.uui.request	PT_UINT
FP_UUI_TX_RESPONSE	get_feature_parm.feature.uui.tx_response	PT_INT
FP_UUI_RX_RESPONSE	get_feature_parm.feature.uui.rx_response	PT_INT
FP_UUI_CONTROL	get_feature_parm.feature.uui.control	PT_CHAR
FP_UUI_FLOW_CONTROL	get_feature_parm.feature.uui.flow_control	PT_CHAR
FP_UUI_PROTOCOL	get_feature_parm.feature.uui.protocol	PT_UCHAR
FP_UUI_MORE	get_feature_parm.feature.uui.more	PT_CHAR
FP_UUI_LENGTH	get_feature_parm.feature.uui.length	PT_UCHAR
FP_UUI_DATA	get_feature_parm.feature.uui.data	PT_STRING
FP_FAC_COMMAND	get_feature_parm.feature.facility.command	PT_INT
FP_FAC_CONTROL	get_feature_parm.feature.facility.control	PT_UCHAR
FP_FAC_LENGTH	get_feature_parm.feature.facility.length	PT_UCHAR
FP_FAC_DATA	get_feature_parm.feature.facility.data	PT_STRING
FP_FAC_DESTINATION_ADDR	get_feature_parm.feature.facility.destination_addr	PT_STRING
FP_FAC_ORIGINATING_ADDR	get_feature_parm.feature.facility.originating_addr	PT_STRING
FP_FAC_DEST_SUBADDR	get_feature_parm.feature.facility.dest_subaddr	PT_STRING
FP_FAC_DEST_NUMBERING_TYPE	get_feature_parm.feature.facility.dest_numbering_type	PT_UCHAR
FP_FAC_DEST_NUMBERING_PLAN	get_feature_parm.feature.facility.dest_numbering_plan	PT_UCHAR

FP_FAC_ORIG_NUMBERING_TYPE	get_feature_parm.feature.facility.orig_numbering_type	PT_UCHAR
FP_FAC_ORIG_NUMBERING_PLAN	get_feature_parm.feature.facility.orig_numbering_plan	PT_UCHAR
FP_FAC_ORIG_NUMBERING_PRESENTATION	get_feature_parm.feature.facility.orig_numbering_presentation	PT_UCHAR
FP_FAC_ORIG_NUMBERING_SCREENING	get_feature_parm.feature.facility.orig_numbering_screening	PT_UCHAR
FP_FAC_MORE	get_feature_parm.feature.facility.more	PT_UCHAR
FP_DIV_DIVERTING_REASON	get_feature_parm.feature.diversion.diverting_reason	PT_UCHAR
FP_DIV_DIVERTING_COUNTER	get_feature_parm.feature.diversion.diverting_counter	PT_UCHAR
FP_DIV_DIVERTING_TO_ADDR	get_feature_parm.feature.diversion.diverting_to_addr	PT_STRING
FP_DIV_DIVERTING_FROM_ADDR	get_feature_parm.feature.diversion.diverting_from_addr	PT_STRING
FP_DIV_ORIGINAL_CALLED_ADDR	get_feature_parm.feature.diversion.original_called_addr	PT_STRING
FP_DIV_DIVERTING_FROM_TYPE	get_feature_parm.feature.diversion.diverting_from_type	PT_UCHAR
FP_DIV_DIVERTING_FROM_PLAN	get_feature_parm.feature.diversion.diverting_from_plan	PT_UCHAR
FP_DIV_DIVERTING_FROM_PRESENTATION	get_feature_parm.feature.diversion.diverting_from_presentation	PT_UCHAR
FP_DIV_DIVERTING_FROM_SCREENING	get_feature_parm.feature.diversion.diverting_from_screening	PT_UCHAR
FP_DIV_DIVERTING_INDICATOR	get_feature_parm.feature.diversion.diverting_indicator	PT_UCHAR
FP_DIV_ORIGINAL_DIVERTING_REASON	get_feature_parm.feature.diversion.original_diverting_reason	PT_UCHAR
FP_DIV_DIVERTING_TO_TYPE	get_feature_parm.feature.diversion.diverting_to_type	PT_UCHAR
FP_DIV_DIVERTING_TO_PLAN	get_feature_parm.feature.diversion.diverting_to_plan	PT_UCHAR
FP_DIV_DIVERTING_TO_INT_NW_INDICATOR	get_feature_parm.feature.diversion.diverting_to_int_nw_indicator	PT_UCHAR
FP_DIV_ORIGINAL_CALLED_TYPE	get_feature_parm.feature.diversion.original_called_type	PT_UCHAR
FP_DIV_ORIGINAL_CALLED_PLAN	get_feature_parm.feature.diversion.original_called_plan	PT_UCHAR
FP_DIV_ORIGINAL_CALLED_PRESENTATION	get_feature_parm.feature.diversion.original_called_presentation	PT_UCHAR
FP_DIV_OPERATION	get_feature_parm.feature.diversion.operation	PT_INT
FP_DIV_OPERATION_TYPE	get_feature_parm.feature.diversion.operation_type	PT_INT
FP_DIV_ERROR	get_feature_parm.feature.diversion.error	PT_INT
FP_HOLD_COMMAND	get_feature_parm.feature.hold.command	PT_INT
FP_HOLD_CAUSE	get_feature_parm.feature.hold.cause	PT_INT
FP_HOLD_Q931_TS	get_feature_parm.feature.hold.unique_xparms.sig_q931.ts	PT_INT
FP_HOLD_Q931_RAW	get_feature_parm.feature.hold.unique_xparms.sig_q931.raw	PT_INT
FP_HOLD_Q931_DISPLAY	get_feature_parm.feature.hold.unique_xparms.sig_q931.display	PT_HEXSTR
FP_XFER_CONTROL	get_feature_parm.feature.transfer.control	PT_CHAR
FP_XFER_Q931_OPERATION	get_feature_parm.feature.transfer.unique_xparms.sig_q931.operation	PT_INT
FP_XFER_Q931_OPERATION_TYPE	get_feature_parm.feature.transfer.unique_xparms.sig_q931.operation_type	PT_INT
FP_XFER_Q931_ERROR	get_feature_parm.feature.transfer.unique_xparms.sig_q931.error	PT_INT
FP_XFER_Q931_ETS_LINKID	get_feature_parm.feature.transfer.unique_xparms.sig_q931.specific.ets.LinkID	PT_INT
FP_NSTD_ID_TYPE	get_feature_parm.feature.non_standard.id_type	PT_INT
FP_NSTD_LENGTH	get_feature_parm.feature.non_standard.length	PT_INT
FP_NSTD_DATA	get_feature_parm.feature.non_standard.data	PT_STRING
FP_NSTD_H221_ID_CC	get_feature_parm.feature.non_standard.id.h221_id.cc	PT_UINT
FP_NSTD_H221_ID_EXT	get_feature_parm.feature.non_standard.id.h221_id.ext	PT_UINT
FP_NSTD_H221_ID_CODE	get_feature_parm.feature.non_standard.id.h221_id.code	PT_UINT
FP_NSTD_OBJECT_ID_LENGTH	get_feature_parm.feature.non_standard.id.object_id.length	PT_INT
FP_NSTD_OBJECT_ID_ID	get_feature_parm.feature.non_standard.id.object_id.id	PT_STRING
FP_RAW_LENGTH	get_feature_parm.feature.raw_data.length	PT_INT
FP_RAW_DATA	get_feature_parm.feature.raw_data.data	PT_STRING
FP_RAW_MORE	get_feature_parm.feature.raw_data.more	PT_UCHAR

Returns: This function returns 0 upon success or a negative error code.

-o-

CCsetxparm

Synopsis:
CCsetxparm(port, channel, parmId, Value[,connectionless_flag])

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.
parmID – The Id of the parameter to set
Value - The value to set the parameter to..
[connectionless_flag] – If this flag is set then the parameter will be set in the port specific FEATURE_DETAIL_XPARMS structure rather then the channel specific FEATURE_DETAIL_XPARMS structure. If the *connectionless_flag* is set then the *channel* is ignored.

Description: This function allows for a channel specific FEATURE_DETAIL_XPARMS structure to be set (or port specific structure if *connectionless_flag* is set). After setting any of the channel specific parameters in the FEATURE_DETAIL_XPARMS using the ccsetxparm() function, then any future calls to CCmkxcall() or CCsendfeat() will use the channel specific FEATURE_DETAIL_XPARMS structure.

If the *connectionless_flag* is set then the *port* specific copy of the FEATURE_DETAIL_XPARMS structure will be used in any subsequent calls to CCsndcnctless() function calls.

To clear any values set in the channel or port specific FEATURE_DETAIL_XPARMS structure then use the CCclrxparms().

The full list of parameters that can be set using the CCsetxparms() function is described in the CCgetxparm() function description.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCclrxparms

Synopsis:
CCclrxparms(port,channel[,connectionless=1])

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.
connectionless_flag – IF this is set to a non zero value then the port specific copy of the FEATURE_DETAIL_XPARMS structure will be used instead of the channel specific copy.

Description: This function clears the channel or port specific (if *connectionless_flag* is set to 1) FEATURE_DETAIL_XPARMS structure, setting all values to 0:

Returns: This function returns 0 upon success or a negative error code.

-o-

CCgetcnctless

Synopsis:

CCgetcnctless(port)

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_get_connectionless (FEATURE_DETAIL_XPARMS *feature_detailsp);
```

This function allows port specific (connectionless) FACILITY messages to be retrieved. After calling this function then the individual fields from the FEATURE_DETAIL_XPARMS structure can be retrived using the CCgetparm() function with the *connectionless_flag* set to 1.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCmkxcall

Synopsis:

CCmkxcall(port, channel, DID, CID, sending_complete[,parm1,parm2....])

Arguments:

port – The logical E1/T1 port number.
channel – The channel number.
DID – The destination address
CID – The originating address
[send_comp] – Optional Sending complete flag
[cnf_parm1,cnf_parm2..] – Optional cnf_parms

Description: This function maps to the following Aculab function:

```
ACU_ERR call_feature_openout (FEATURE_OUT_XPARMS *feature_out);
```

It attempts to make an outgoing call on the specified port and channel whilst also sending feature information from the FEATURE_OUT_XPARMS structure as set by CCsetxparm().

The DID and CID arguments specify the destination and originating addresses respectively. The option *send_comp* argument allows the OUT_XPARMS.sending_complete flag to be set and should be set to 0 for overlap sending (more digits to come) or 1 for en-bloc sending. The default is en-bloc sending if this aragument is not given.

The optional *cnf_parms* allow for the OUT_XPARMS.cnf field to be set. If one or more of these optional *cnf_parms* are specified then they are each ORed in turn with OUT_XPARMS.cnf field. If no *cnf_parms* are specifed then by default the IN_XPARMS.cnf field is set to CNF_REM_DISC which stops the channel automatically returning to the idle state when a remote

end disconnect occurs (instead the CCrelease() call must be used to return the channel to CS_IDLE state).

See the Aculab documentation for the call_openout() function for a more detailed description of the cnf field values.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCsendfeat

Synopsis:
CCsendfeat(port, channel)

Arguments:
port – The logical E1/T1 port number.
channel – The channel number.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_feature_send(FEATURE_DETAIL_XPARMS *feature_detailsp);
```

It allows for feature information to be transmitted during the lifetime of a call on a particular *port* and *channel*. Use CCsetxparm() to set the specific fields of the FEATURE_DETAIL_XPARM structure prior to calling this function.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCsndcnctless

Synopsis:
CCsndcnctless(port)

Arguments:
port – The logical E1/T1 port number.

Description: This function maps to the following Aculab function:

```
ACU_ERR call_send_connectionless(FEATURE_DETAIL_XPARMS *feature_detailsp);
```

It allows FACILITY messages to be sent to the network that are not specific to a particular call on a particular channel. Prior to calling this function the specific fields of the port specific FEATURE_DETAIL_XPARMS structure should be set with calls to CCsetxparm() with the *connectionless_flag* set to 1.

Returns: This function returns 0 upon success or a negative error code.

-o-

CCstrtohex

Synopsis:
hexstr=CCstrtohex(string);

Arguments:
string - The string to convert to a hexadecimal string

Description: This function converts the *string* argument into the corresponding hexadecimal string. Each character of the *string* will be converted to exactly two hexadecimal characters in the returned string. This function is useful in the CCsetparm() function where a hexadecimal string is required for a parameter value.

Examples:

```
    // This will return hex_str="3434323038323232"
hex_str=CCstrtohex("442082222");

//This will return hex_str="010fffde"
hex_str=CCstrtohex("`01`0f`ff`de");
```

-o-

CCinttohex

Synopsis:
hexstr=CCinttohex(unsigned_val,num_bytes);

Arguments:
int_val - The integer value to convert to a hexadecimal string
num_bytes - Set to 1, 2 or 4 for byte, short integer or long integer value

Description: This function converts the integer value *int_val* argument into the corresponding hexadecimal string. The *num_bytes* argument defines whether the integer should be treated as a 1 byte (char), 2 byte (short) or 4 byte (long) integer and will thus be converted into a 2,4 or 8 character hexidecimal string respectively.

Each byte of the integer will be converted to exactly two hexadecimal characters in the returned string in *little endian byte order* (i.e byte order will be low to high). This function is useful in the CCsetparm() function where a hexadecimal string is required for a parameter value.

Examples:

```
// This will return hex_str="ff"
 hex_str=CCinttohex(255,1);

// This will return hex_str="ff00"
 hex_str=CCinttohex(255,2);

// This will return hex_str="ff000000"
 hex_str=CCinttohex(255,4);
```

```
// This will return hex_str="80000000"
hex_str=CCinttohex(-128,4);
```

Returns: Returns the hexidecimal string representation of the given integer

-o-

CCunstohex

Synopsis:

hexstr=CCunstohex(int_val,num_bytes);

Arguments:

unsigned_val - The unsigned integer value to convert to a hexadecimal string
num_bytes - Set to 1, 2 or 4 for byte, short integer or long integer value

Description: This function converts the unsigned integer value *unsigned_val* argument into the corresponding hexadecimal string. The *num_bytes* argument defines whether the integer should be treated as a 1 byte (char), 2 byte (short) or 4 byte (long) integer and will thus be converted into a 2,4 or 8 character hexidecimal string respectively. Note that if a negative number is passed to the function then this will be converted to an unsigned integer first which will result in an interger overflow.

Each byte of the integer will be converted to exactly two hexadecimal characters in the returned string in *little endian byte order* (i.e byte order will be low to high). This function is useful in the CCsetparm() function where a hexadecimal string is required for a parameter value.

Examples:

```
// This will return hex_str="ff"
hex_str=CCunstohex(255,1);

// This will return hex_str="ff00"
hex_str=CCunstohex(255,2);

// This will return hex_str="ff000000"
hex_str=CCunstohex(255,4);

// This will return hex_str="ffffffff" since a -ve integer will overflow when converted to unsigned
hex_str=CCunstohex(-1,4);
```

Returns: Returns the hexidecimal string representation of the given unsigned integer

-o-

SWmode

-o-

SWquery

-o-

SWset

-o-

CCcreateTDM

Synopsis:
tdm_chan=CCcreateTDM(port,chan)
Arguments:
port – The logical E1/T1 port number.
channel – The channel number.

Description: This function creates a TDM endpoint for the port and channel specified by the *port* and *chan* arguments. The *port* and *chan* must reside on a board the supports TDM end-points (e.g Prosody X) and upon success it returns a TDM channel handle which should be used in all future function calls that reference this TDM end-point. The internal Prosody stream and time slot and module id are obtained from the internal data associated with the *port* and *chan* and are used when creating the TDM endpoint.

The function creates **tdmprx** and **tdmtx** end-points to allow RTP data to be transmitted to and received from a data feed and transmittted onto the internal TDM stream and timeslot. This function maps to the Aculab **sm_tdmrx_create()** and **sm_tdmtx_create()** functions.

To create a TDM endpoint for an VOX channel you should use the SMcreateTDM(vox_chan) function.

Once created then the functions SMtraceTDM() and SMdestroyTDM() can be used to trace and release the created TDM channel (i.e there is no CCtraceTDM() or CCdestroyTDM() as these would be identical to the above functions).

Returns: Upon success this function returns a TDM channel handle, otherwise it returns a negative error code..

-o-

Aculab Prosody Card Library

Introduction

There are two Telecom Engine libraries that provide access to the functionality of the Aculab API. CXACULAB.DLL provides the call control and switching capabilities and the CXACUDSP.DLL

provides the functionality for the Prosody speech and digital signal processing capabilities.

This section describes the speech and digital signal processing library (CXACUDSP.DLL).

-o-

Some Simple Examples

Probably the best way to show the basic library functions and the library calling conventions is to provide a simple example. The example below simply waits for an incoming call on the first channel of the first E1 port, then plays a message and receives some DTMF. It is assumed that the reader is familiar with the Telecom Engine standard library set and the Aculab Call Control library (CXACULAB.DLL).

```
$include "aculab.inc"

int port, chan, vox_chan, x, event;
var filename:64;
var tone:1;

main
   port=0
   chan=1;
   vox_chan=1;
   filename="hello.vox";

   // Make fill duplex H.100 bus routing between voice channel and network port/channel
   CClisten(port,chan,SMgetslot(vox_chan));
   SMlisten(vox_chan, CCgetslot(port, chan));

   // Enable inbound calls on this port/channel
   CCenablein(port,chan);

   // loop waiting for incoming call
   while(1)
       x=CCwait(port,chan,CC_WAIT_FOREVER,&event);
      if(x > 0 and event eq CS_INCOMING_CALL_DETECTED)
         break;
      endif
   endwhile

   // Cause jump to onsignal if remote disconnect is received
   CCuse(port,chan);

   // Answer the call
   CCaccept(port,chan);


   // Play a vox file to caller
   SMplay(vox_chan,filename);

   // Wait for some digits.. parmaters are:
    //SMwaittones(voice_chan,max_tones,first_delay10ths,inter_delay10ths),term_digits,&num_digits]
   SMwaittones(vox_chan,1,40,40);

   // Get any received digits
   tone=Smgettones(vox_chan);
   if(tone strneq "")
     // make the file name up from the tone received
     filename="PROMPT"& tone & ".vox";
     SMplay(vox_chan,filename);
   endif

   // Cause jump top onsignal
   task_hangup(task_getpid());
endmain

onsignal
```

```
  // Hangup the call
  CCdisconnect(port,chan,CAUSE_NORMAL);

  // Wait for state to return to IDLE the release call
  while(1)
    x=CCwait(port,chan,CC_WAIT_FOREVER,&event);
    if(x eq CS_IDLE)
       break;
    endif
  endwhile

  // release the call
  CCrelease(port,chan);

  // restart the application to wait for another call..
  restart;

endonsignal
```

The program should be fairly self explanatory but I will describe the key parts of the program below.

The "aculab.inc" file is provided with the library and defines all the constants that are used with the library such as CC_WAIT_FOREVER, CAUSE_NORMAL, CS_INCOMING_CALL_DETECTED etc. These are be described in more detail in the call control library function library reference (CXACULAB).

The first call: CClisten() simply makes the receiving stream/channel of the call control channel ' Listen' to the transmit stream/channel of the Voice channel, so that anything that is output by the voice channel will be heard by the caller.

The second call: SMlisten() makes the receiving stream/channel voice channel 'listen' to the transmit stream/channel of the Call control channel, so that any DTMF digits or other audio transmitted by the caller will be heard by the voice channel.

As mentioned above all this is done by switching from and to the extern H.100 or SCBUS.

The *CCenablein(port,channel)* allows inbound calls to be received on the channel, and then the application goes into a loop waiting for calls.

The *CCwait(port,channel,timeout_100ms,&event)* function call will wait for the specified timeout (in 10ths of a second) for an event. If the timeout is defined as -1 (CC_WAIT_FOREVER) then the call will not return until an event is found or it is aborted by a *CCabort()* call. Really the only event that should be received here is CS_INCOMING_CALL_DETECTED but we do a specific check for it anyway in case the channel was in a unknown state when the program started (probably some error handling should be carried out if we found an unexpected event).

Once a CS_INCOMING_CALL_DETECTED event has been received then the call is answered immediately with *CCaccept(port,channel)* and *CCuse(port,channel)* forces the program to jump to the *onsignal* function if the CS_REMOTE_DISCONNECT event is received from this point on.

After this there are some calls to the Speech Module library fuctions.

First a voice prompt is played to the caller using the *SMplay(vox_chan,filename)* function after which the application waits for some DTMF input using the *SMwaittones(vox_chan, num_dig,first_delay10ths, inter_delay10ths)* function.

The *SMwaitones()* function puts the task into a blocking state until one of the terminating conditions is met. The terminating condition could be that the requested number of digits has been received (*num_dig)* or the timeout waiting for the first digit (*first_delay10ths)* was exceeded, or the inter-digit timeout was exceeded (*inter_delay10ths)*. *SMwaittones()* will then copy any digits received into the internal digit buffer for the voice channel. The next call *SMgettones(vox_chan)* returns any digits that have been copied to the internal digit buffer for the voice channel.

The application then checks if a tone was received and if so will use the received DTMF to make the name of a prompt file which is then played using the SMplay() function.

The application then forces itself to the *onsignal* function by calling task_hagup() with it's own process id where the call is disconnected using the *CCdisconnect(port,channel)* call and the application goes into a loop waiting for the channel to return to the CS_IDLE state before releasing the call with *CCrelease(port,channel)* and restarting the program to wait for the next call.

-o-

Simple VOIP example

The following example shows how to receive a VOIP call using Virtual Media Processing channels. To receive a VOIP call one needs to create a VMP channel and configure some codecs on that VMP channel. The VMP channel is then specified in the CCaccept() parameters when accepting the inbound call. The datafeeds from the VOX and VMP channels can then be connected in full duplex to allow voice prompts to be played or recorded over the VOIP call..

```
$include "aculab.inc"

main

    int port,chan,vmp_chan,vox_chan,module_id;
    int state,x,last_state;
    var DNIS:50;

    // Hard code these for this simple test..
    port=0;
    chan=1;
    vox_chan=1;
    module_id=0;

    // Create a VMP port on DSP module 0
    vmp_chan=SMcreateVMP(module_id);
    if(vmp_chan < 0)
        errlog("Could not open VMP channel: err=",vmp_chan);
        stop;
    endif

    // Set G711 ALAW codec at element 0 in array of accepted codecs.
    SMsetcodec(vmp_chan,0,G711_ALAW);

    // Create full duplex connection between the VMP and the VOX datafeeds in advance..
    SMfeedlisten(vox_chan,TYPE_VOX,vmp_chan,TYPE_VMP);
    SMfeedlisten(vmp_chan,TYPE_VMP,vox_chan,TYPE_VOX);

    // Now go wait for inbound IP call..
```

```
    CCenablein(port,chan);
    CCuse(port,chan);     // cause jump to onsignal on disconnect

    last_state=-1;
    while(1)
        // wait forever for a change in state
         x=CCwait(port,chan,WAIT_FOREVER,&state,last_state);
        last_state=state;
          if(state eq CS_INCOMING_CALL_DET)
             CCalerting(port,chan);    // Send INCOMING RINGING.
              CCgetparm(port,chan,CP_DESTINATION_ADDR,&DNIS);
             applog("INCOMING CALL: DNIS=" & DNIS);

          else if(state eq CS_WAIT_FOR_ACCEPT)
             // We must set the VMP channel in the accept parameters for VOIP
              CCclrparms(port,chan,PARM_TYPE_ACCEPT);
                CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPRXID,vmp_chan);
                CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPTXID,vmp_chan);
                CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_CODECS,vmp_chan);
            CCaccept(port,chan);
          break;
        endif endif
    endwhile

    //Now loop playing file /recording file/playing recording/getting DTMF...
    //only hangup signal will interrupt this..
    while(1)
         SMplay(vox_line,"demo.vox");
        x=SMplaytone(vox_line,19,100);    // play a short beep
        SMrecord(vox_line,"test.vox",10,5);          // Record a short message
        SMplay(vox_line,"test.vox");         // playback the recording
         SMwaittones(vox_line,4,50,50);      // wait for some DTMF
        input=SMgettones(vox_line);          // Retrieve the digits..
       applog("Got input=",input);           // display the digits..
    endwhile

endmain

onsignal
     applog("We are in ONSIGNAL!!!! Hangup signal was received..");
      CCdisconnect(port,chan,LC_NORMAL);
    # Now go wait for call to go idle
    if(last_state <> CS_IDLE)
      while(1)
          # wait for idle
          applog("In onsig CCwaiting for CS_IDLE");
          x=CCwait(port,chan,10,&state,last_state);
          applog("CCwait returned x=",x," state=",state);
         if(state eq 0)
             applog("port=" & port & " chan=" & chan & " Incoming went to IDLE");
            break;
          endif
          sleep(1);
       endwhile
    endif
     CCrelease(port,chan);
     // Restart the program to receive another call..
    restart;
endonsignal
```

-o-

Board Opening Order

Upon start-up the CXACUDSP.DLL library opens and initialises the Aculab call control boards ready to make and receive calls. The order that the boards are opened is specified by a configuration file called ACUCFG.CFG whose path can be defined by the environment variable called ACUCFGDIR. If the ACUCFGDIR environment variable is not set then the library will look in the current directory for the ACUCFG.CFG.

If the ACUCFG.CFG file is not found then the boards are opened in the order that they are found in the Aculab Configuration Tool (ACT), and which is the order returned by the *acu_get_system_snapshot*() function.

The format of ACUCFG.CFG file is described here.

The order that the Prosody speech boards are opened is important since it defines the logical channel number that is used in many of the calls to the CXACUDSP.DLL functions. Voice channels are numbered starting from 1 up to the maximum number of voice channels in the system. For example in the following ACUCFG.CFG there are two modules each with 150 channels. These will be assigned voice channel numbers 1 through to 300:

```
# One Prosody X card with 4 E1 ports and 2 DSP modules
board=189747
ports=0,1,2,3
modules=0:150,1:150
```

Note: If the number of voice channels is not specified in the ACUCFG.CFG file, or no ACUCFG.CFG file exists then the current version of the library will assume that there are 150 voice channels for every DSP fitted onto a Prosody X speech card and 60 voice channels for every DSP fitted onto the old Prosody cards.

-o-

Nailing transmit timeslots to H.100 or SCBUS

Once the boards are opened then the board capabilities are examined and any boards that have switching capabilities will have their transmit channels ‘nailed’ to the external H.100 or SCBUS. This provides a consistent method for switching between channels and in the current version of the library even channels on the same board will be switched through the external H.100 or SCBUS.

H.100 channels are defined by both a stream number and a channel number where each stream can have up to 128 timeslots. For the SCBUS there is only one stream and the timeslots range from 0 up to 4096. To create a consistent way of referencing these channels whether the external bus is a H.100 bus or an SCBUS the CXACUDSP.DLL generates a stream/timeslot handle which is calculated from the stream and the timeslot as follows:

*handle = stream * 4096 + timeslot*

For the SCbus the *stream* is always 24 which is the internal fixed stream that is used by the ACULAB firmware when SCBUS is present.

This handle is used/returned by the switching functions listed below.

handle=SMgetslot(port,channel);
x= SMlisten(channel,handle);
x= SMunlisten(channel);

By default the first channel on the first logical port will be 'nailed' to stream 8, timeslot 0 of the H.100 bus (or just timeslot 512 of the SCBUS). If there is other non-Aculab hardware in the system that is using these stream/timeslot ranges, or there is some other reason why a different stream/timeslot range should be used for nailing the transmit timeslots to the external bus then the environment variable PROSODY_TSOFFS can be set to define the start stream and timeslot offset to nail to.

This variable should be specified in the same form as defined above for the stream/timeslot handle. For example if you want to start nailing the call control channel starting at stream 64, channel 0 then you would set the environment variable as follows:

*REM set offset to: 64 * 4096 + 0*
SET PROSODY_TSOFFS=262144

or for the SCBUS where the stream is hardcoded to 24, if for example you wanted to start 'nailing' the voice channels from SCbus timeslot 1024 the you would have:

*REM set offset to: 24 * 4096 + 1024*
SET PROSODY_TSOFFS=99328

N.B. The current version of the library does yet transparently support multi-chassis switching for Prosody-X functionality (although the programmer has access to the RTS functions and can therefore implement their own multi-chassis switching capability). The next version of the library will include a transparent method for multi-chassis switching consistent with above function calls and methodology.

-o-

Indexed Prompt Files (IPFs)

Indexed prompt files (IPF) are single files that contain a number of different voice prompts. The top of the files contains an index giving byte offset and length of each prompt in the file. The purpose of index prompt files is to provide a more efficient way of building up voice messages from multiple prompts and is used primarily for speaking dates, times, numbers etc. Since the index prompt file is a single file then it only needs to be opened one at startup, thereafter it can be referred to by an *Ipf_id* number in the SMword(), SMplayph() and SMplaypr() functions.

The list of IPF files to open at startup is found in a file called PR.PAR which is created from a text file called PR.DEF which is converted to the PR.PAR file using the MKPR.EXE utlity. The PR.DEF has the following format:

<IPF_Id> <IPF filename>

The <IPF_Id> defines the Index Prompt File ID number and must range from 1 upwards. For example the following is a valid PR.DEF file

```
1 ENGLISH.IPF
2 ARABIC.IPF
3 JAPANESE.IPF
4 MANDARIN.IPF
5 CANTONESE.IPF
```

The above file defines five Indexed prompt files and the IPF_Id for each. Presumably the above PR.DEF defines IPF files that hold the prompts needed to make up dates, times, numbers etc in various languages. To turn this in to the PR.PAR file then the utility MKPR.EXE should be run from the command line in the directory where the PR.DEF file exists.

Upon startup the CXACUDSP.DLL library will use the PRDIR environment variable to find the path of the PR.PAR file (or will look in the current directory if PRDIR is not set), and if it exists it will open the IPF files listed. If any of the IPF files listed in the PR.PAR file do not exist then the CXACUDSP.DLL will cause the Telecom Engine to quit with an error message.

-o-

Teminating Events

Many of the speech functions such as SMplay() and SMrecord() will cause the calling task to block until the function completes with a terminating event (unless SMmode() is called to allow non-blocking functionality).

The list of blocking functions for which terminating events apply are listed below:

SMplay(vox_chan,filename[,mode,sample_rate])
SMplayh(vox_chan,filehandle[bytes,mode,sample_rate])
SMrecord(vox_chan,filename,[seconds,silence,mode,sample_rate,beep])
SMwaittones(vox_chan,max_tones,first_delay10ths,inter_delay10ths[,term_digits,&num_digits])
SMplaytone(vox_chan,toneid,duration_ms)
SMplaydigits(vox_chan,digit_str,[inter_delay_ms,dig_dur_ms])
SMplaycptone(vox_chan,duration_ms,type,tone_id1,on_cad1,off_cad1[,tone_id2,on_cad2,off_cad2.....])
SMgetrecognised(vox_chan,timeout,&type,¶m0,¶m1)
SMplayph(vox_chan[,slot, dataformat, samplerate])
SMplaypr(vox_chan,slot, prompt_no [,dataformat, samplerate])

There are a number of different reasons why a function might terminate such as reaching the end of the file or a DTMF digit being received etc. The full list of possible terminating events for all blocking speech functions are shown below (as defined in ACULAB.INC):

```
# Terminating events
const TERM_ERROR      = -1;
const TERM_TONE       = 1;
const TERM_MAXDTMF    = 2;
const TERM_TIMEOUT    = 3;
const TERM_INTERDELAY = 4;
const TERM_SILENCE    = 5;
const TERM_ABORT      = 6;
const TERM_EODATA     = 7;
const TERM_PLAYTONE   = 8;
```

© Zentel Telecom Ltd, 2009

```
const TERM_PLAYDIGITS  = 9;
const TERM_PLAYCPTONE  = 10;
const TERM_RECOG       = 11;
```

The following table gives a description of each of these terminating events and the function for which they apply:

Event Name	**Description**	**Applies to functions**
TERM_ERROR	An error of some kind was encountered	ALL
TERM_TONE	The function was terminated by a DTMF digit	SMplay(); SMplayh(); SMplayph(); SMplaypr(); SMrecord(); SMwaittones(); SMplaytone(); SMplaycptone()
TERM_MAXDTMF	The total number of DTMF digits requested has been received	SMwaittones()
TERM_TIMEOUT	A timeout has occurred	SMwaittones(); SMwaitrecog(); SMrecord()
TERM_INTERDELAY	The specified inter-digit delay timeout has occurred	SMwaittones()
TERM_SILENCE	The specified period of silence has occurred	SMrecord()
TERM_ABORT	The function was aborted by SMabort()	ALL
TERM_EODATA	End of file or data has been reached.	SMplay(); SMplayh(); SMplayph(); SMplaypr(); SMrecord()
TERM_PLAYTONE	The specified tone has finished playing	SMplaytone()
TERM_PLAYDIGITS	All specified digits have been played	SMplaydigits()
TERM_PLAYCPTONE	The specified call progresss tone has finished playing	SMplaycptone()
TERM_RECOG	A recognition event has been received (ASR,Grunt,CPtone etc)	SMgetrecognised()

Note: When called in blocking mode (the default mode for a channel), all of the above functions will also be terminated whenever a hangup signal is received that causes a jump to the *onsignal* function. However there is no specific terminating event code for this type of termination since the return value from the function can never be retrieved when a hangup signal is received, since the program execution will immediately jump to the *onsignal* routine.

In non-blocking mode (see 1.4 below), the above functions will not automatically be terminated by a jump to the *onsignal* function and must be manually aborted using the *SMabort()* function, or the application must manually wait for the function to complete by looping on the SMstate() function call and waiting for the state to return to 0 to indicate that the function has completed..

-o-

Blocking and Non-Blocking Mode

All of the above functions can be called in both blocking or non-blocking mode as specified by a call to *SMmode(vox_chan,nonblocking_flag)*. Under normal circumstances any call to the above blocking functions will cause the Telecom Engine task to block until the function is interrupted by one of the terminating events. However it is sometimes useful to allow program execution to continue while a SMplay() or other function continues in the background.

In order to allow this the voice channel can be put into non-blocking mode using the *SMmode(vox_chan,nonblocking_flag)* function. If the *nonblocking_flag* argument is set to a non zero value then any calls to the above speech functions will return immediately whilst the play, record, play tone etc proceeds in the background.

In this case it is up to the application to ensure that the current speech function has finished before attempting to call one of the other blocking speech functions. To do this there is a function *SMstate(vox_chan)* which returns the current function that running on the channel at the present time or 0 if there are no speech functions currently running.

NOTE: if *nonblocking_flag* is set to 1 then the SMmode() call will only apply to the next blocking speech function call. Once that function has completed then the channel mode will be set back to blocking mode. If a non-zero value other than 1 is given then the channel will stay in non-blocking mode until a call to SMmode() is made again with *nonblocking_flag* set to 0 to put the channel back into non-blocking mode.

There are two constants defined in ACULAB.INC for this purpose as shown below:

```
const MODE_BLOCKING                =0;
const MODE_NONBLOCKING_ONCEONLY=1;
const MODE_NONBLOCKNG              =2;
```

For example, the following code extract will play some music in the background whilst a database look-up occurs. Once the database lookup has completed the application will abort the music and wait for the channel to return to idle.

```
    // Prevent DTMF tones from interrupting playback
    SMtoneint(vox_chan,0);

    // Play "Please wait while we look up the information"
    SMplay(vox_chan,"PLSWAIT.VOX");
    // Change the mode to play in the background (non-blocking) for the next speech function only
(non-blocking_flag=1)
    SMmode(vox_chan,MODE_NONBLOCKING_ONCEONLY);
    // Play music in the backgroundwhile information is retrieved
    SMplay(vox_chan,"MUSIC.VOX");
    // Do the data retrieval whilst music is playing…
    data_retrieval_func();

    // Abort the music
    SMabort(vox_cha);
    // Loop waiting for chan state to return to 0 (should only take milliseconds..)
    while(SMstate(vox_chan))
          ;
    endwhile

    // Allow DTMF tones to interrupt SMplay() etc again..
```

```
SMtoneint(vox_chan,1);
…

onsignal
      // Check if hangup received during music playback (or other non-blocking operation
      if(SMstate(vox_chan))
            SMabort(vox_cha);
            // Loop waiting for chan state to return to 0 (should only take milliseconds..)
            while(SMstate(vox_chan))
                  ;
            endwhile
      endif

..
endonsignal
```

-o-

Aculab Prosody Speech Functions Quick Reference

SMgetmodules()
SMgetchannels()
SMgetcards()
SMcardinfo
(card,&pSerial,&pCard_type,&pCard_cap,&pSw_type.&pFirst_mod,&pNummods,&pStatus)
SMmodinfo(mod,&pfirst_chan,&pNumchans)
term_code=SMplay(vox_chan,filename[,filetype,sample_rate])
SMplayh(vox_chan,filehandle[bytes,filetype,sample_rate])
SMrecord(vox_chan,filename,[seconds,silence,filetype,sample_rate,beep])
SMsetrecparm(vox_chan,parmID,value);
SMgetrecparm(vox_chan,parmID);
SMabort(vox_chan)
handle=SMgetslot(vox_chan)
SMlisten(vox_chan, ts_handle)
SMunlisten(vox_chan)
SMctlDtmf(vox_chan,on_off[,AsDigit(=1),toneset,mode])
SMctlPulse(vox_chan,on_off)
SMctlCPtone(vox_chan,on_off)
SMctlGrunt(vox_chan,on_off,latency)
SMtoneint(vox_chan,on_off,toneset)
SMwaittones
(vox_chan,max_tones,first_delay10ths,inter_delay10ths[,term_digits,&pNnum_digits])
SMgettones(vox_chan[,max_tones])
SMclrtones(vox_chan)
SMplaytone(vox_chan,toneid,duration_ms)
SMplaydigits(vox_chan,digit_str,[inter_dur_ms,dig_dur_ms])
SMplayptone
(vox_chan,duration_ms,type,tone_id1,on_cad1_ms,off_cad1_ms[,tone_id2,on_cad2_ms,off_cad2_ms[,toneid3…]])
SMgetrecognised(vox_chan,timeout_10ths,&pType,&pParam0,&pParam1)
SMmode(vox_chan,non_blocking)
SMtrace(vox_chan,on_off)
item_id=SMaddASRvocab(module,filename)
SMclrASRvocabs(module)

SMsetASRchanparm(vox_chan,parmID,value[,parmID1,value1...])
SMaddASRitem(vox_chan,vocab_id,user_id)
SMclrASRitems(vox_chan) {
SMctlASR(vox_chan,off_on_cont)
conf_id=SMconfstart(module)
SMconfjoin(conf_id,vox_chan_out[,Type(1-listen only),[vox_chan_in,OutAgc,OutVol,InAGC,InVol])
SMconfleave(conf_id,vox_chan_out[,vox_chan_in])
SMconfend(conf_id)
SMdump() - Dumps various information about the state of the system
chan_state=SMstate(vox_chan)
num_items=SMdetected(vox_chan[,type])
SMword(vox_chan,prompt_no[,ipf_id])
term_event=SMplayph(vox_chan [,ipf_id, dataformat, samplerate])
term_event=SMplaypr(vox_chan,ipf_id, prompt_no [,dataformat, samplerate])
SMplaystrph(vox_chan,ipf_id,str_words[,dataformat,samplerate])
SMplaystrphm(vox_chan,str_words[,dataformat,samplerate])
SMcreateVMP(module_id,[local_addr])
SMtraceVMP(vmp_chan,tracelevel)
SMdestroyVMP(vmp_chan)
SMsetcodec(vmp_chan)
SMclrcodecs(vmp_chan)
SMcreateTDM(vox_chan)
SMtraceTDM(tdm_chan)
SMdestroyTDM(tdm_chan)
SMfeedlisten(chan1_id,chan1_type,chan2_id,chan2_type)
SMfeedunlisten(chan_id,chan_type)

-o-

Aculab Prosody Speech Function Reference

SMgetmodules

Synopsis:
SMgetmodules()

Description: This function returns the number of Digital Signal Processing (DSP) modules in the system.

Returns: Returns the number of DSP modules in the system.

-o-

SMgetchannels

Synopsis:
SMgetchannels()

Description: This function returns the number of voice channels available in the system.

Returns: Returns the number of voice channels in the system.

-o-

SMgetcards

Synopsis:
SMgetcards()

Description: This function returns the number of Prosody cards poresent in the syste,..

Returns: Returns the number of Prosody cards in the system.

-o-

SMcardinfo

Synopsis:

SMcardinfo(card,&pSerial,&pCard_type,&pCard_cap,&pSw_type.&pFirst_mod,&pNummods,&pStatus)

Arguments:

card – The card number for which to retrieve information (starting from 0)
pSerial – Pointer to variable that will hold the returned serial number
pCard_type – Pointer to variable that will hold the cards type
pCard_cap - Pointer to a variable that will hold the card capabilities
pSw_type – Pointer to variable that will hold the switching bus type
pFirst_mod – Pointer to a variable that will hold the first DSP module offset on the board
pNummods – Pointer to a variable that will hold the number of DSP modules on the board
pStatus – Pointer to a variable that will hold the board status

Description: This function returns information about the specified *card.* The *card* number ranges from 0 to one less than the total number of Prosody cards in the system. The remaining arguments are pointer to variables that will hold the returned card information values.

The p*Card_type* will hold the card type which will be one of the following types as defined in the ACULAB.INC file:

```
# types of card
const ACU_PROSODY_PCI_CARD              = 0x10;
const ACU_E1_T1_PCI_CARD                = 0x12;
const ACU_VOIP_PCI_CARD                 = 0x14;
const ACU_IP_TELEPHONY_PCI_CARD         = 0x14;
const ACU_PROSODY_CPCI_CARD             = 0x20;
const ACU_E1_T1_CPCI_CARD               = 0x21;
const ACU_PROSODY_S_CARD                = 0x22;
const ACU_E1_T1_CPCI_PMX_CARRIER_CARD   = 0x23;
const ACU_PROSODY_X_CARD                = 0x24;
```

The *pCard_cap* will hold the card capabilities which will be one or more of the following values

Or'ed together:

```
# functionality available on card
const ACU_RESOURCE_CALL          = 1;
const ACU_RESOURCE_SWITCH        = 2;
const ACU_RESOURCE_SPEECH        = 4;
const ACU_RESOURCE_IP_TELEPHONY  = 8;
const ACU_RESOURCE_ODPR          = 16;
const ACU_RESOURCE_TRM           = 32;
const ACU_RESOURCE_MG            = 64;
const ACU_RESOURCE_STUN          = 128;
```

The *pSw_type* will hold the type of switching resource on the card and will be one of the following:

```
# Switch types
const SWMODE_CTBUS_MVIP    =0;
const SWMODE_CTBUS_SCBUS   =1;
const SWMODE_CTBUS_H100    =2;
const SWMODE_CTBUS_PEB     =3;
const SWMODE_CTBUS_MC3     =4;
```

The *pFirst_Mod* will hold the module offset of the first DSP Module on the card. Modules are number 0 from the first module found on the first card and are then numbered sequentially depending on the order that the cards are opened.

The *pNummods*will hold the number of DSP modules that are on the specified *card*.

The *pStatus* holds the card status which will be 0 for inactive/disabled or 1 for active.
Returns: Returns 0 on success or -1 if bad card number supplied

-o-

SMmodinfo

Synopsis:
SMmodinfo(mod,&pfirst_chan,&pNumchans)

Arguments:
mod – The module number (starting from 0)
pfirst_chan – The first voice channel on the specified module
pNumchans – The number of voice channels on the module.

Description: This function returns information about specified module. The information returned is the first voice channel number on the module and the total number of voice channels on the module.

Returns: 0 upon success or -1 if an invalid module number is supplied

-o-

SMplay

Synopsis:
term_code=SMplay(vox_chan,filename[,filetype,sample_rate])

Arguments:

vox_chan – The logical voice channel.
filename – The filename of the voice prompt to play
[filetype] – The type of voice prompt file to play
[sample_rate] – The sample rate of the voice prompt file.

Description: This function plays the speech file specified by *filename* on the given voice channel. If the *filetype* and *sample_rate* are not specified then it is assumed that the file type is SMDataFormatOKIADPCM and the sample rate is 6000.

Otherwise the *filetype* and *samperate* can be specifed as described below.

The full list of *filetypes* as defined in ACULAB.INC is shown below:

```
# File types for SMplay() etc
const SMDataFormatNone=0;
const SMDataFormatALawPCM=30;
const SMDataFormatULawPCM     =31;
const SMDataFormatOKIADPCM         =32;
const SMDataFormatACUBLKPCM   =33;
const SMDataFormat16bit            =34;
const SMDataFormat8bit        =35;
const SMDataFormatSigned8bit  =36;
const SMDataFormatIMAADPCM         =17;
```

The *sample_rate* is the sample rate in bits per second of the speech file. The valid sample rates are as follows:

8000 – The typical rate for telephone since it is the rate at which the telephone networks themselves operate.
6000 – A rate which reduces file size at the cost of some quality
11000 – a rate convenient for use with PC soundcards. This is sufficiently close to a quarter of the rate used by CDs and allows almost universal compatibility with cheap PC soundcards which can handle 11025 sampling.

Note that under normal circumstances the Telecom Engine will block the calling task until the playback is terminated by a terminating event of some kind. This may be the presence of a DTMF digit in the DTMF digit buffer for the channel, the end of the file, a call to SMabort() or any other terminating event.

The reason for the function terminating is returned as the return value of the function and may be one of the following values:

```
# Terminating events
const TERM_ERROR      = -1;
const TERM_TONE      = 1;
const TERM_ABORT      = 6;
const TERM_EODATA      =;7
```

(see 1.3 Terminating events)

Also whenever a jump to the *onsignal* function occurs (for example caused by a hangup signal after a call to CCuse()) and if the function is playing in blocking mode (see 1.4 Blocking and non-blocking mode) then the SMplay() will automatically be aborted. If playing in non-blocking mode then it is up to the application to abort the play and/or wait for it to complete.

Returns: Returns either an error code (E.g. if the file could not be opened) or the reason for the

function termination.

-o-

SMplayh

Synopsis:

SMplayh(vox_chan,filehandle[bytes,filetype,sample_rate])

Arguments:

vox_chan – The logical voice channel.
filehandle – A file handle returned from a call to *sys_fhopen()*
[bytes] – Number of bytes to play from the file
[filetype] – The type of voice prompt file to play
[sample_rate] – The sample rate of the voice prompt file.

Description: This function is similar to the *SMplay()*function except that it takes a file handle as returned from the *sys_fhopen()* function in the Telecom Engine standard system library (CXSYS.DLL). It is up to the application to open the file first and to ensure that the file handle is released after use.

If the optional argument *bytes* is specified then the function will terminate with the event TERM_EODATA once the specified number of *bytes* has been played from the file. If *bytes* is omitted or set to 0 then the function will continue playing from the file until the end of the file is reached or another terminating event causes the function to finish.

Just as for the SMplay() function the SMplayh() will return the reason for the termination of the function, which will be one of the following values:

```
# Terminating events
const TERM_ERROR       = -1;
const TERM_TONE        = 1;
const TERM_ABORT       = 6;
const TERM_EODATA       =;7
```

(see 1.3 Terminating events)

If the function is playing in blocking mode then a jump to *onsignal* will cause the playback to be aborted. In non-blocking mode the playback will continue even after a jump to *onsignal* and it is then up to the application to abort the playback and/or wait for it to complete.

Example:

```
int fh;
fh=sys_fhopen("HELLO.VOX","rs", SMDataFormatALawPCM,8000);
if(fh < 0)
      errlog("Error opening file: err=",fh);
   task_hangup(task_getpid());
endif

// If we get here then the file is open so play it
SMplayh(vox_chan,fh,0,);
sys_fhclose(fh);
```

Returns: Returns either an error code (E.g. invalid file handle) or the reason for the function

termination.

-o-

SMrecord

Synopsis:
SMrecord(vox_chan,filename,[seconds,silence,filetype,sample_rate,beep])
Arguments:
vox_chan – The voice channel
filename - The filename to record to
[seconds] – Optional number of seconds to record (default value is 0 (unlimited)
[silence] – Optional number of seconds of silence to end recording (default is 5 seconds)
[filetype] - Optional file type to record to (default is SMDataFormatOKIADPCM)
[sample_rate] - Optional sample rate (default is 6000)

Description: This function records to the given *filename* on the specified *vox_chan*. The number of *seconds* to record can be specified as an optional argument. If *seconds* is omitted or set to 0 then the recording will continue indefinitely until one of the other terminating events is received.

If the *filetype* and *sample_rate* are not specified then it is assumed that the file type is SMDataFormatOKIADPCM and the sample rate is 6000.

Otherwise the *filetype* and *samperate* can be specifed as described below.

The full list of *filetypes* as defined in ACULAB.INC is shown below:

```
# File types for SMplay() etc
const SMDataFormatNone=0;
const SMDataFormatALawPCM=30;
const SMDataFormatULawPCM      =31;
const SMDataFormatOKIADPCM          =32;
const SMDataFormatACUBLKPCM    =33;
const SMDataFormat16bit             =34;
const SMDataFormat8bit         =35;
const SMDataFormatSigned8bit   =36;
const SMDataFormatIMAADPCM          =17;
```

The *sample_rate* is the sample rate in bits per second of the speech file. The valid sample rates are as follows:

8000 – The typical rate for telephone since it is the rate at which the telephone networks themselves operate.
6000 – A rate which reduces file size at the cost of some quality
11000 – a rate convenient for use with PC soundcards. This is sufficiently close to a quarter of the rate used by CDs and allows almost universal compatibility with cheap PC soundcards which can handle 11025 sampling.

Note that under normal circumstances the Telecom Engine will block the calling task until the recording is terminated by a terminating event of some kind. This may be the presence of a DTMF digit in the DTMF digit buffer for the channel, or the maximum number of seconds has been reached or the maximum duration of silence has been detected etc

The reason for the function terminating is returned as the return value of the function and may be one of the following values:

```
# Terminating events
const TERM_ERROR      = -1;
const TERM_TONE       = 1;
const TERM_TIMEOUT    = 3;
const TERM_SILENCE    = 5;
const TERM_ABORT      = 6;
```

(see 1.3 Terminating events)

Also whenever a jump to the *onsignal* function occurs (for example caused by a hangup signal after a call to CCuse()) and if the function is playing in blocking mode (see 1.4 Blocking and non-blocking mode) then the SMplay() will automatically be aborted. If playing in non-blocking mode then it is up to the application to abort the play and/or wait for it to complete.

Returns: Returns either an error code (E.g. if the file could not be opened) or the reason for the function termination.

-o-

SMsetrecparm

Synopsis:
SMsetrecparm(vox_chan,parmID,value);
Arguments:
vox_chan - The voice channel
parmID - The parameter ID to set
value – The parameter value
Description: This function allows for certain recording parameters to be set for a particular channel. Each channel has it's own copy of a SM_RECORD_PARMS structure which is passed to the Aculab *sm_record_start()* function and these parameters can be set prior to a call to SMrecord() by using this function.

The list of record parameters that can be set is as follows:

ParmID	Maps to field:
SM_RECPARM_AGC	SM_RECORD_PARMS.agc
SM_RECPARM_VOLUME	SM_RECORD_PARMS.volume
SM_RECPARM_SILENC_ELIM	SM_RECORD_PARMS.silence_elimination
SM_RECPARM_TONE_ELIM_MODE	SM_RECORD_PARMS.tone_elimination_mode
SM_RECPARM_TONE_ELIM_SET	SM_RECORD_PARMS.tone_elimination_set_id
SM_RECPARM_ALTDATASOURCE	SM_RECORD_PARMS.alt_data_source
SM_RECPARM_ALTDATASOURCE_TYPE	SM_RECORD_PARMS.alt_data_source_type

The values for the parmIDs are defined in the ACULAB.INC file are are shown below:

```
const SM_RECPARM_AGC                    =1;
const SM_RECPARM_VOLUME                 =2;
const SM_RECPARM_SILENC_ELIM            =3;
const SM_RECPARM_TONE_ELIM_MODE         =4;
const SM_RECPARM_TONE_ELIM_SET          =5;
const SM_RECPARM_ALTDATASOURCE          =6;
const SM_RECPARM_ALTDATASOURCE_TYPE     =7;
```

Below is a description of these parameters:

SM_RECPARM_AGC – This is an Indicator of whether automatic gain control is to be enabled. Set to a non zero value to enable or zero to enable.
SM_RECPARM_VOLUME – Set this to the desired adjustment to the volume (dB). The range of gain supported is at least +8 to -22 dB
SM_RECPARM_SILENC_ELIM - The maximum duration (in mS) of silence to record. Silences longer than this are truncated to this length. The value zero disables silence elimination.
SM_RECPARM_TONE_ELIM_MODE – Indicaties what types of tones to eliminate from the recording. and can be one of these values (as defined in the ACULAB.INC file):

SMToneDetectionNone - Simple tones never recognised.

SMToneDetectionNoMinDuration - mple tone detection enabled, no minimum period. If the correct frequencies are detected with the correct signal to noise ratio, twist, etc. for however short a duration, the tone is considered to be present and is recognised.

kMToneDetectionMinDuration64 - Simple tone detection enabled, tone must be valid for minimum period to be detected. If the tone is valid for 64mS it will definitely be detected. Tones of shorter duration between 32mS and 64mS may be detected but cannot be guaranteed.

SMToneDetectionMinDuration40 - This mode uses a slightly more complex algorithm for analysing duration of a valid tone, and enables robust detection of tones with duration as short as 40mS.

SMToneEndDetectionNoMinDuration - This mode is like SMToneDetectionNoMinDuration but application notified when end of tone detected.

SMToneEndDetectionMinDuration64 - This mode is like SMToneDetectionMinDuration64 but application notified when end of tone detected.

SMToneEndDetectionMinDuration40 - This mode is like SMToneDetectionMinDuration40 but application notified when end of tone detected.

SMToneLenDetectionNoMinDuration - This mode is like SMToneEndDetectionNoMinDuration but returns additional tone duration information to application.

SMToneLenDetectionMinDuration64 - This mode is like SMToneEndDetectionMinDuration64 but returns additional tone duration information to application.

SMToneLenDetectionMinDuration40 - This mode is like SMToneEndDetectionMinDuration40 but returns additional tone duration information to application.

SMToneDetectionAsListenFor - This mode is only valid when a tone detection mode is currently active on the same channel, started by *SMctlDtmf()*. Any tones detected on the same channel as the recording will be eliminated from the recorded data.

SM_RECPARM_TONE_ELIM_SET – The tone set to use (NOT YET IMPLEMENTED)
SM_RECPARM_ALTDATASOURCE – An alternative voice channel whose input or output is to be record (set to 0 if no alternative voice channel is specified)
SM_RECPARM_ALTDATASOURCE_TYPE - If an SM_RECPARM_ALTDATASOURCE channel is specified, this defines which kind of data associated with that channel should be recorded. One of these values:

SMRecordAltSourceDefault – If the channel specified for SM_RECPARM_ALTDATASOURCE is an input only channel, then data switched to this

channel input will be recorded, otherwise the data being generated on this channel output will be recorded (this feature is normally used to record conferenced outputs). *SMRecordAltSourceInput* - Data switched to the alternative data source input will be recorded
SMRecordAltSourceOutput - Data generated on the alternative data source output will be recorded.

Returns: 0 if successful or negative error code.

-o-

SMgetrecparm

Synopsis:
SMgetrecparm(vox_chan,parmID);

Arguments:
vox_chan - The voice channel
parmID - The parameter ID to obtain the value of

Description: This function allows the value associate with the given record *parmID* to be retrieved for a particular channel. See CCsetparm*()* for a description of the *parmID* values and meanings

Returns: Returns the value associated with the specified parameters ID.

-o-

SMabort

Synopsis:
SMabort(vox_chan)

Arguments:
vox_chan – The voice channel

Description: This functions will abort a speech operation that is currently running on the given channel. This function works with all of the blocking type speech functions listed below:

SMplay()
SMplayh()
SMrecord()
SMwaittones()
SMplaytone()
SMplaydigits()
SMplayptone()
SMgetrecognised()
SMplayph()
SMplaypr()

If the SMabort() function is used to interrupt any of the above functions then the terminating event type returned by the function will be TERM_ABORT.

Returns: 0 if successful or -1 if an invalid voice channel is specified.

-o-

SMgetslot

Synopsis:
handle=SMgetslot(vox_chan)

Arguments:
vox_chan – The voice channel

Description: This function returns a logical handle to the external H.100 or SCBUS transmit timeslot for the specified *vox_chan*. The timeslot handle references the *transmit* timeslot of the given *vox_chan* which has been '*nailed'* to a H.100 or SCBUS timeslot by the library at start-up. This only applies to boards that have switching capabilities and not to the PROSODY_S type virtual boards which do not have any external switching capability.

The returned handle is actually obtained from the physical H.100 or SCBUS stream and timeslot from the following formula:

*handle = stream * 4096 + timeslot*

For the SCbus the *stream* is always 24 which is the internal fixed stream that is used by the ACULAB firmware when SCBUS is present.

For example, if a prosody board has two DSPs attached giving 300 channels of speech pricessing and is fitted with a H.100 bus, then upon startup the CXACUDSP.DLL library will automatically ' nail' the transmit timeslots from the voice channels to the H.100 bus starting at stream 8, timeslot 0 (or as defined by the PROSODY_TSOFFS environment variable) . Since each H.100 stream can have 128 timeslots then three streams will be used by the 300 voice channels on this card.

Voice channels 1 to 128 would be nailed to the timeslots of stream 8, voice channels 129 to 256 would be nailed to the timeslots of stream 9 and voice channels 257 to 300 would be nails to the first 64 timeslots of stream 10 on the H.100 bus.

From the formula shown above, the handles returned by the SMgetslot() function would range from 32768 though to 32895 for first 128 voice channels, 36864 through to 36991 for channels 129 to 256 and 40960 through to 41023 for the last 64 voice channels.

The external bus *handle* returned by SMgetslot() can be used in the SMlisten() function to allow the 'receive' timeslot of one channel to '*listen'* to the 'transmit' timeslot from another channel via the external H.100 or SCBUS.

For example, if there was an inbound call on E1 port 0, channel 1 then the voice channel number 1 could be connected to this network channel in a full duplex connect, via the external bus.

```
# Make a full duplex connection between the voice channel and the network channel
CClisten(0,1,SMgetslot(1));
SMlisten(1,CCgetslot(0,1);
```

Returns: This function returns the logical timeslot handle for the given *vox_chan* or a negative error code.

-o-

SMlisten

Synopsis:
SMlisten(vox_chan, ts_handle)
Arguments:
port – The logical E1/T1 port number.
ts_handle – The logical timeslot handle returned from SMgetslot() (or using the formula shown in SMgetslot())
Description: This function causes the receive timeslot of the given voice channel to 'listen' to the transmit timeslot that has been nailed to the external H.100 or SCBUS. The *ts_handle* is a logical handle that references an external H.100 or SCBUS stream/timeslot as returned by CCgetslot() or SMgetslot() or which can be obtained by using the formula:

*handle = stream * 4096 + timeslot*

Where *stream* and *timeslot* are the stream and timeslot on the external H.100 or SCBUS. For the SCBUS the *stream* is hardcoded to 24.

For example the following code makes voice channel 1 listen to the network channel on port 0, channel 1 that any DTMF or audio from the caller will be heard by the voice channel for DTMF detection or recording.

x=SMlisten(1,CCgetslot(0,1);

Returns: This function returns 0 upon success or a negative error code.

-o-

SMunlisten

Synopsis:
SMunlisten(vox_chan)

Arguments:
vox_chan – The voice channel

Description: This function stops the receive timeslot of the given *vox_chan* from listening to any H.100 or SCBUS transmit timeslot:

Returns: This function returns 0 upon success or a negative error code.

-o-

SMctlDtmf

Synopsis:

SMctlDtmf(vox_chan,on_off[,AsDigit(=1),toneset,mode])

Arguments:

vox_chan – The voice channel
on_off - Set to 0 to turn off DTMF tone detection, or 1 to turn it on.
AsDigit – Set to 0 if there is no digit mapping or 1 (default) if the digits are to be mapped to digit codes.
toneset - The tone set ID to use (one of the internal Aculab tonesets)
mode - The tone detection mode.

Description: This function controls how the specified *vox_chan* handles the detection of DTMF digits. Detection of DTMF can be turned on or off by setting the *on_off* flag to 1 or 0 respectively.

The *AsDigit* argument defines whether the received DTMF will be converted to the corresponding DTMF character (E.g. #, *, 0 etc) or whether the tone ID within the tone set will be returned instead. By default *AsDigit* is set to 1 which means convertion will take place. Set this to 0 to receive the tone id instead.

Returns: 0 if successful or -1 if bad *vox_chan* was specified.

-o-

SMctlPulse

Synopsis:

SMctlPulse(vox_chan,on_off)

Arguments:

vox_chan: The voice channel
on_off - Set to 0 to turn pulse detection off (default) or 1 to turn pulse detection on

Description: This function allows pulse detection to be switched on or off.

Returns: Returns 0 on success or -1 if a bad *vox_chan* was specified.

-o-

SMctlCPtone

Synopsis:

SMctlCPtone(vox_chan,on_off)

Argument:

vox_chan – The voice channel
on_off – set to 0 to turn call progress tone detection off (default), 1 to turn it on

Description: This function turns on (1) or off (0) call progress tone detection that corresponds to a member of the set of call-progress tones currently recognisable by the module. Note that call-progress tone detection may not be used simultaneously with tone or digit detection on the

same channel.

See Prosody API documentation for sm_listen_for() for the set of preloaded call progress tones.

Returns: 0 if successful or -1 if invalid voice channel speicified.

-o-

SMctlGrunt

Synopsis:
SMctlGrunt(vox_chan,on_off,latency)

Arguments:
vox_chan – The voice channel
on_off – set to 0 to turn grunt detection off, 1 to turn it on
latency- The duration in ms required before a signal is considered to be silent

[(n.b. need *min_noise_level* and *grunt_level* to be implemented)]

Description: This function turns on (1) or off (0) grunt detection on the given channel. The *latency* can be set to the number of milliseconds required before a signal is considered to be silent.

Returns: Returns 0 if successful or 01 if bad *vox_chan* specified.

-o-

SMtoneint

Synopsis:
SMtoneint(vox_chan,on_off,toneset)

Arguments:
vox_chan – The voice channel
on_off – Set to 0 to prevent DTMF tones from interrupting speech functions, set to 1 to allow DTMF to interrupt speech
toneset – Which of the internal Aculab tonesets to use

Description: This function specifies whether DTMF tones will interrupt blocking speech functions such as SMplay(), SMrecord() etc. By default any DTMF tone received will interrupt these functions with a terminating event of TERM_TONE, to prevent this *on_off* should be set to 0.

toneset can be set to one of the internal Aculab tonesets to specifiy which toneset can interrupt a speech function.

Returns: Returns 0 if successful or -1 if bad *vox_chan* specified

-o-

SMwaittones

© Zentel Telecom Ltd, 2009

Synopsis:

SMwaittones(vox_chan,max_tones,first_delay10ths,inter_delay10ths[,term_digits,&pNnum_digits])

Arguments:

vox_chan – The voice channel
max_tones – The maximum number of tones to receive
first_delay10ths – The time to wait for the first digit to be entered (in 10ths of a second)
inter_delay10ths – The maximum time between digits (in 10ths of a second)
[term_digits] – Optional argument specifying a string of DTMF digits that would terminate the input
pNum_digits – Pointer to a variable to hold the number of digits actually received before the function terminated

Description: This function allows the application to block waiting for DTMF input and also copies any DTMF digits received to the internal DTMF buffer belonging to the channel. Each channel maintains two buffers for collecting DTMF digits, one background buffer which collects all DTMF digits detected on the channel, and one foreground buffer where tones are copied after a call to SMwaittones(). It is the foreground buffer that is returned by a call to the SMgetttones() functions.

Therefore the SMwaittones() and SMgetttones() functions work in conjunction with each other. The SMwaittones() function sets conditions for which tones to wait for and the conditions which will cause the SMwaittones() to terminate. After termination the SMwaittones() will copy any DTMF digits it received, up to and including the terminating event, to the foreground buffer.

The *max_tones* argument specifies the maximum number of DTMF tones to receive before terminating the SMwaittones() function with a TERM_MAXDTMF event. Note that if the background buffer already holds the number of tones specified by *max_tones* then the function will terminate immediately with TERM_MAXDTMF and these tones will be copied over to the foreground buffer.

first_delay10ths specifies the maximum time (in 1/10ths second) that the function will wait for the first input tone to be received. If this timeout is exceeded then the function will terminate with a TERM_TIMEOUT event and no digits will be copied to the foreground buffer. IF there are already one or more digits in the background buffer then the *first_delay10ths* expires immediately and the *inter_delay10ths* timer is started. If *first_delay10ths* is set to 0 (or negative) value then the function will terminate immediately with TERM_TIMEOUT if there was not already a digit in the background buffer.

the *inter_delay10ths* timer is started after the first digit has been received and specifies the maximum time allowed between all successive received digits. If the *inter_delay10ths* timer is exceeded then the function will be terminated with a TERM_INTERDELAY event and the digits received so far are copied to the foreground buffer.

The optional *term_digits* argument allows the input to be terminated upon receipt of one of a set of DTMF digits specified as a string of digits. For example if the input is to be terminated by either a '*' or '#' digit then the *term_digits* string should be set to "*#". As soon as either of these digits is received then the function terminates with a TERM_TONE event and all the digits received so far (including the terminating digit) are copied to the foreground buffer.

If the optional *pNum_digits* variable is specified then the variable that this points to will be set to

the total number of digits copied to the foreground buffer upon termination of this function.

Examples:

```
// This will wait for upto 4 digits to be received with the first and inter digit delay set to 4 seconds each
x=SMwaittones(vox_chan,4,40,40);
//  Get the digits received from the foreground buffer..
tones=SMgettones(vox_chan);


// This will wait for upto 4 digits to be received unless a * or # is received, with the first and inter digit delay
set to 4 seconds each
x=SMwaittones(vox_chan,4,40,40,"*#");
//  Get the digits received from the foreground buffer..
tones=SMgettones(vox_chan);


// This will return immediately with TERM_TIMEOUT unless there is already a DTMF digit already in the
background buffer (first_delay10th set to 0)
x=SMwaittones(vox_chan,1,0,0);
//  Get the digits received from the foreground buffer..
tones=SMgettones(vox_chan);
```

Returns: Returns the terminating event or a negative error code.

-o-

SMgetttones

Synopsis:

SMgettones(vox_chan[,max_tones])

Arguments:

vox_chan – The voice channel

[max_tones] – optional argument specifying the maximum number of tones to return from the foreground buffer

Description: This function works in conjunction with the SMwaittones() function to receive DTMF input. The SMgettones() function will return all of the digits that have been copied to the foreground buffer by a call to SMwaittones(), or the number of tone specified by *max_tones* if this argument is specified.

Returns: Returns the string of DTMF digits from the foreground buffer or -1 if an invalid *vox_chan* was specified.

-o-

SMclrtones

Synopsis:

SMclrtones(vox_chan)

Arguments:

vox_chan – The voice channel

Description: This function clears both the foreground and background buffers of all received

DTMF digits.

Returns: Returns 0 upon success or -1 if invalid *vox_chan* specified

-o-

SMplaytone

Synopsis:
SMplaytone(vox_chan,toneid,duration_ms)

Arguments:
vox_chan – The voice channel
toneid - One of the predefined toneIDs to play
durations_ms – The duration of the tone in milliseconds

Description: This function allows one of the Aculab predefined simple output tones to be played on the given *vox_chan*. (See Prosody document for the list of predefine output tones)

Returns: Returns 0 on success or negative error code

-o-

SMplaydigits

Synopsis:
SMplaydigits(vox_chan,digit_str,[inter_dur_ms,dig_dur_ms])
Arguments:
vox_chan – The voice channel
digit_str – The string of DTMF digits to play
[inter_dur_ms] – The gap between digits (in millseconds)
[digit_dur_ms] – The digit duration in milliseconds

Description: This function plays the string of DTMF digits specified in the *digit_str* argument. The optional *inter_dur_ms* and *dig_dur_ms* allow for the delay between digits and the digit duration to be specified (in milliseconds). If these are omitted or set to 0 then the Aculab default durations are used.

Returns: Returns 0 upon success or negative error code.

-o-

SMplayptone

Synopsis:

SMplaycptone(vox_chan,duration_ms,type,tone_id1,on_cad1_ms,off_cad1_ms[,tone_id2,on_cad2_ms,off_cad2_ms[,toneid3…]])
Arguments:
vox_chan – The voice channel

duration_ms – The duration of the tone (in millseconds) or the number of times the tone should be played (for type repeat).
type – The tone type (0=Once, 1=Repeat, 2=Continuous)
tone_id1 – The tone id of the first tone frequency
on_cad1 – The on time of the fist tone frequency (in ms)
off_cad1 – The off time of the first tone frequency (in ms)
[tone_id2] – The tone id of the second tone frequency
[on_cad2] – The on time of the second tone frequency (in ms)
[off_cad2] – The off time of the second tone frequency (in ms)
etc

Description: This function enables call progress tones to be played made up of one or more frequencies and cadences on the specified *vox_chan*.
The *duration_ms* argument specified the number of milliseconds that the tone will be played for. If it is set to 0 then this means that the tone should be played indefinitely until a terminating event occurs (such as SMabort() or hangup signal).

Note that If the *type* is set to *SMPlayCPToneTypeOneShot* then the *duration_ms* is ignored and the set of tone cadences will be played just once.

The *type* argument can take one of the following values as defined in ACULAB.INC:

```
# Call Progress tone types
const SMPlayCPToneTypeOneShot        =0;
const SMPlayCPToneTypeRepeat         =1;
const SMPlayCPToneTypeContinuous     =2;
```

If the *type* is set to SMPlayCPToneTypeRepeat or SMPlayCPToneTypeContinuous then the *duration_ms* defines the number of milliseconds that the tone will play before it terminates with a TERM_CPTONE event.

The *type* SMPlayCPToneTypeContinuous is special (an depreciated by SMplaytone()) since it only plays the first tone specified by *tone_id1* as a continuous tone (cadences and subsequent tones are ignore).

The function also takes the set of tones and cadences that make up the tone through the arguments: *tone_id1, on_cad1_ms, on_cad1_ms* and there may be a number of further tones that make up the call progress tone through the optional arguments: *tone_id2, on_cad2_ms, on_cad2_ms* , *tone_id3, on_cad3_ms, on_cad3_ms* and so on..

The *tone_id* specifies the tone ID of the tone to play, the *on_cad_ms* specifies the duration (in ms) that the tone is on for, and *off_cad_ms*defines the duration (in ms) of silence after this tone.

Returns: Returns 0 if successful or a negative error code.

-o-

SMgetrecognised

Synopsis:
SMgetrecognised(vox_chan,timeout_10ths,&pType,&pParam0,&pParam1)
Arguments:
vox_chan – The voice channel

timeout_10ths – The time to wait for a recognition event
pType – Pointer to a variable that will hold the returned recognition event type
pParam0 – Pointer to a variable that will hold the returned recognition event parameter 0 data
pParam1 - Pointer to a variable that will hold the returned recognition event parameter 1 data

Description: This function waits for and returns details of a single recognition event that occurred as a result of a call to sm_wait_for() initiated by one of the functions such as SMctlPulse(), SMctlCPtone(), SMctlGrunt() and SMctlASR() (Note that SMctlDTMF causes DTMF digits to be copied to a separate channel buffer and should be retrieved using the SMwaittones() and SMgettones() function instead).

If detection of pulse, CP tones, Grunt or ASR have been initiated through one of the above functions then this function can wait for and collect the received recognition events caused by detection of these events.

Only a signal recognition event at a time can be retrieved by this function and the *timeout_10ths* argument defines how long (in 1/10th seconds) that the function will wait before terminating with a TERM_TIMEOUT event. If there is already a recognition event in the channel buffer then the function will terminate immediately with a TERM_RECOG event and the variables pointed to by the *pType, pParam0* and *pParam1* will hold the information about the received recognition event.

The variable pointed to by the pType argument will be set to one of the following values as defined in ACULAB.INC:

```
# Recognition event types
const SMRecognisedNothing        =0;
const SMRecognisedTrainingDigit  =1;
const SMRecognisedDigit          =2;
const SMRecognisedTone                =3;
const SMRecognisedCPTone         =4;
const SMRecognisedGruntStart     =5;
const SMRecognisedGruntEnd       =6;
const SMRecognisedASRResult      =7;
const SMRecognisedASRUncertain =8;
const SMRecognisedASRRejected         =9;
const SMRecognisedASRTimeout          =10;
const SMRecognisedCatSig         =11;
const SMRecognisedOverrun        =12;
const SMRecognisedANS            =13;
```

Note however that in the current version of the library the following recognition types will not be received:

```
const SMRecognisedCatSig         =11;
const SMRecognisedANS            =13;
```

Also the SMRecognisedDigit events are handled by the SMwaittones() and SMgettones() functions and will not be returned by this function, and the SMRecognisedNothing event is ignored.

The variables pointed to by *pParam0* and *pParam1* will be set to values depending on the *pType*. See the Prosody Speech API guide for details (sm_get_recognised()) for details of what these

parameters will be set to for each type.

Returns: Returns 0 upon success or negative error code.

-o-

SMmode

Synopsis:
SMmode(vox_chan,non_blocking)

Arguments:
vox_chan – The voice channel
non_blocking – Set to 0 for blocking mode, 1 to one-shot non-blocking mode, 2 for continuous non_blocking mode.

Description: This function allows for certain speech functions (such as SMplay(), SMrecord() etc) to be carried out in either blocking or non-blocking mode. All of the speech functions that operate in this way are known as *blocking speech functions*. In blocking mode the calling Telecom Engine task will *block* until the speech function has completed after which the function will return with the terminating event that caused the *blocking speech function* to complete. In non-blocking mode the function will return immediately and the speech function will continue playing in the background while the Telecom Engine task continues processing. In this case it is up to the program to wait for the operation to complete (using SMstate()) or to specifically abort the speech operation using SMabort().

See Blocking and Non-blocking Mode for more information.

If the *non_blocking* argument is set to 0 then this causes all subsequent *blocking speech functions* to block the calling telecom Engine task (this is the default behaviour).

If the *non-blocking* flag is set to 1 then the next *blocking speech function* will operate in non-blocking mode after which further calls to *blocking speech functions* will block the calling task as normal.

If the *non-blocking* flag is set to 1 then the next *blocking speech function* will operate in non-blocking mode and a call to SMmode() with the *non-blocking* flag set to 0 will be required to set the channel back to blocking mode again.

Returns: Returns 0 is successful or a negative error code.

-o-

SMtrace

Synopsis:
SMtrace(vox_chan,on_off)

Arguments:
vox_chan – The voice channel
on_off – set to 1 to turn event tracing on, 0 to turn it off

Description: This function allows for event tracing to be turned on or off for a particular *vox_chan*. If turned on then all events on the specified channel will be logged to the system trace log.

Returns: Return 0 on success or -1 is bad channel is specified.

-o-

SMaddASRvocab

Synopsis:

item_id=SMaddASRvocab(module,filename)

Arguments:

module – The module ID upon which to load the vocabulary (numbered from 0 in the order that the boards are opened)

filename – The name of the vocabulary file to load to the module.

Description: This function downloads an ASR vocabulary to the specified *module* modules are numbered from 0 from the first module on the first speech boards and are then numbered sequentially across all boards in the order that the boards are opened.

The *filename* is the path to the file containing the vocabulary item.

Returns: Returns 0 if successful or negative error code

-o-

SMclrASRvocabs

Synopsis:

SMclrASRvocabs(module)

Arguments:

module - The module ID

Description: This function clears any vocabulary files that have previously been downloaded to the specified speech module.

Returns: Returns 0 if successful or -1 if an invalid module was specified

-o-

SMsetASRchanparm

Synopsis:

SMsetASRchanparm(vox_chan,parmID,value[,parmID1,value1...])

Arguments:

vox_chan – The voice channel

parmID - The ID of the parameter to set

value – The value to set the parameter to
[parmID1] – Multiple parameters can be set through these optional arguments
[value1] - Multiple parameters can be set through these optional arguments
etc

Description: This function allows the ASR parameters to be set for a particular *vox_chan* prior to a call to SMctlASR(). The *parmID* argument specifies which parameter is to be set and can be one of the following values as defined in the ACULAB.INC:

```
# ASR parameters (as used by SMsetASRchanparm())
const ASRP_VFR_MAX_FRAMES     =0;
const ASRP_VFR_DIFF_THRESHOLD      =1;
const ASRP_PSE_MAX_FRAMES     =2;
const ASRP_PSE_MIN_FRAMES          =3;
const ASRP_VIT_SOFT_THRESHOLD      =4;
const ASRP_VIT_HARD_THRESHOLD      =5;
const ASRP_VIT_SNR_ADJUST          =6;
```

The above parameters map to one of the fields of the SM_ASR_CHARACTERISTICS structure as follows:

ParmID	**Maps to Field:**
ASRP_VFR_MAX_FRAMES	SM_ASR_CHARACTERISTICS.vfr_max_frames
ASRP_VFR_DIFF_THRESHOLD	SM_ASR_CHARACTERISTICS.vfr_diff_threshold
ASRP_PSE_MAX_FRAMES	SM_ASR_CHARACTERISTICS.pse_max_frames
ASRP_PSE_MIN_FRAMES	SM_ASR_CHARACTERISTICS.pse_min_frames
ASRP_VIT_SOFT_THRESHOLD	SM_ASR_CHARACTERISTICS.vit_soft_threshold
ASRP_VIT_HARD_THRESHOLD	SM_ASR_CHARACTERISTICS.vit_hard_threshold
ASRP_VIT_SNR_ADJUST	SM_ASR_CHARACTERISTICS.vit_snr_adjust

See the description of the Aculab sm_listen_for_asr() function for more information about these parameters and their meanings.

Returns: Returns 0 if successful or a negative error code.

-o-

SMaddASRitem

Synopsis:
SMaddASRitem(vox_chan,vocab_id,user_id)

Arguments:
vox_chan – The voice channel
vocab_id – The Vocabulary ID to add to the set of ASR items that can be detaected on the

channel.

user_id - The user defined ID for the vocabulary item

Description: Each channel can have a set of Active Speech Recognition (ASR) vocabulary items defined for it that may be recognised after a call to SMctlASR(). This function allows the *vocab_id* and associated *user_id* to be added to this set of ASR vocab items for the channel. The set of vocabulary items on a channel will only become active upon the next call to SMctlASR ().

Returns: Returns 0 if successful or a negative error code.

-o-

SMclrASRitems

Synopsis:

SMclrASRitems(vox_chan) {

Arguments:

vox_chan – The voice channel

Description: This function allows for the current set of vocabulary items to be cleared on the specified *vox_chan*. Note that the previous set of vocabulary items will still be active on the *vox_chan* if a previous call was made to SMctlASR() with a vocabulary set defined and the *off_on_cont* flag set to 1 or 2. To stop ASR events from being recoignised then a call to SMctlASR() should be made with *off_on_cont* flag set to 0.

Returns: Returns 0 if successful or a negative error code.

-o-

SMctlASR

Synopsis:

SMctlASR(vox_chan,off_on_cont)

Arguments:

vox_chan – The voice channel

off_on_cont – Set to 0 to turn ASR recognition off, 1 to turn it on for a single recognition event, 2 to turn it on contimuously

Description: This function allows Active Speech Recognition (ASR) to be turned on or off on the specified *vox_chan*. The set of vocabulary items that will be recognised are defined by calls to the SMaddASRitem() function. If *off_on_cont* is set to 0 the ASR detection is turned off. If *off_on_cont* is set to 1 then a signal recognition event will be activated for the vocabulary set on the channel. Once one of these vocabulary items has been recognised then the ASR will be turned off again.

For continous ASR vocabulary recognition the *off_on_cont* should be set to 2.

Returns: Returns 0 if successful or a negative error code.

-o-

SMconfstart

Synopsis:

conf_id=SMconfstart(module)

Arguments:

module – The module id upon which to start the conference.

Description: This function creates a conference on the specified *module*. Speech module IDs are numbered from 0 starting from the first module on the first prosody speech boards and then opened in sequence across the boards in the order that they are opened. If successful the function will return a unique conference ID (*conf_id)* that can be used in all subsequent conference calls .

Returns: Returns a unique conference ID or a negative error code.

-o-

SMconfjoin

Synopsis:

SMconfjoin(conf_id,vox_chan_out[,Type(1-listen only),[vox_chan_in,OutAgc,OutVol,InAGC,InVol])

Arguments:

conf_id – The conference ID
vox_chan_out – The voice channel whose output side will carry the conference audio
[Type] – Set to 0 for full duplex (default) or 1 for listen only
[vox_chan_in] – The voice channel whose input side will be inserted into the conference (by default same as *vox_chan_out*)
[OutAgc] – Set to 1 (default) to enable automatic gain control on the output from the conference
[OutVol] – Set to the output volume in DB (0 by default)
[InAGC] – Set to 1 (default) to enable automatic gain control on the input to the conference
[InVol] - Set to the output volume in DB (0 by default)

Description: This function allows the transmit side of the voice channel specified by *vox_chan_out* to carry the audio output from the conference. Since the transmit side of a voice channel is automatically 'nailed' to the external H.100 or SCBUS timeslot then another device (such as a port on on E1 network channel) can then 'listen' to this timeslot to receive the output from the conference.

The optional *type* argument can specify whether the conference is to be joined in 'listen only' mode (in which case the *vox_chan_in* argument is ignored) or the *type* is set to 0 then the conference in joined in 'full duplex' mode in which case the *vox_chan_in* defines the channel whose output will be inserted into the conference.

By default the optional *vox_chan_in* argument is set to the same voice channel as specified by *vox_chan_out*but it can be a different voice channel if required.

The optional *OutAgc* argument can be set to 1 (default) if automatic gain control (AGC) is to be enabled on the conference output to *vox_chan_out*, or 0 to disable AGC on the output from the conference.

The optional OutVol can be set to the volume of the output from the conference in DB (default 0).

The optional In*Agc* argument can be set to 1 (default) if automatic gain control (AGC) is to be enabled on the conference input from *vox_chan_in*, or 0 to disable AGC on the input to the conference.

The optional InVol can be set to the volume of the input to the conference in DB (default 0).

Below is an example:

```
 #  Assume that a call has been received and accepted on port and chan of the network card.
 # Cause jump to onsignal if hanghup occurs.
 CCuse(port,chan);
 vox_chan=1;

 # Now make a full duplex switch between the network chan and the prosody chan
 CClisten(port,chan,SMgetslot(vox_chan));
 SMlisten(vox_chan,CCgetslot(port,chan);

 # The inbound caller will hear this welcome message first
 SMplay(vox_line,"welcome.vox");

 ## Create a conference on module 0.
 conf_id=SMconfstart(0);
if(conf_id < 0)
   errlog("Error creating conference on module 0");
   stop;
end

 ## Add Vox channel to conference..
 x=SMconfjoin(conf_id,vox_chan);
if(x < 0)
   errlog("Error joining conference id=",conf_id," err=",x);
   stop;
end

 # Two more conference channels
 vox_chan2=2;
 vox_chan3=3;

 # Two spare voice channels to play prompts
 vox_chan4=4;
 vox_chan5=5;


 ## add these Vox channels to conference..
 x=SMconfjoin(conf_id,vox_chan2);
if(x < 0)
   errlog("Error joining conference id=",conf_id," err=",x);
   stop;
end

 x=SMconfjoin(conf_id,vox_chan3);
if(x < 0)
   errlog("Error joining conference id=",conf_id," err=",x);
   stop;
end
```

```
    # Make these two channels listen to vox channels 4 and 5
    # Since anything that vox_chan, vox_chan2 and vox_chan3 listen to will automatically be inserted into the
conference
    SMlisten(vox_chan2,SMgetslot(vox_chan4));
    SMlisten(vox_chan3,SMgetslot(vox_chan5));

    # Now loop playing prompts on channels 4 and 5 which will then be inserted into the conference as
channels 2 and 3 are listening to them..
    # only a hangup by inbound caller will interrupt this loop
   while(1)
          # Play in non-blocking mode  (single shot)
          SMmode(vox_chan4);
          SMmode(vox_chan5);
          # These two prompts will be mixed together in the conference and heard by the inbound caller
          SMplay(vox_chan4,"Prompt1.vox");
          SMplay(vox_chan4,"Prompt2.vox");
          # wait for both channels to finish
           while(not SMstate(vox_chan4) and not SMstate(vox_chan5))
               sleep(30);
          endwhile
    endwhile
… etc

onsignal
      # Abort play on channels 4 and 5
      SMabort(vox_chan4);
      SMabort(vox_chan5);
     # kill the conference
      confend(conf_id);

      .. etc
endonsignal
```

Returns: Returns 0 if successful or a negative error code.

-o-

SMconfleave

Synopsis:
SMconfleave(conf_id,vox_chan_out[,vox_chan_in])

Arguments:
conf_id – The conference ID
vox_chan_out – The voice channel whose output carries the conference audio
[vox_chan_in] - The voice channel whose input will be inserted to the conference (default the same as *vox_chan_out*)

Description: This function allows a voice channel to leave the specified conference. The *vox_chan_out* and *vox_chan_in* must match the channels specified in the SMconfjoin() function. By default the optional *vox_chan_in* will be set the same as the *vox_chan_out* argument.

Returns: Returns 0 if successful or a negative error code if the *conf_id* is invalid or if the *vox_chan_out* and/or *vox_chan_in*do not match a channel that was added using the SMconfjoin() function.

-o-

SMconfend

Synopsis:
SMconfend(conf_id)

Arguments:
conf_id – The conference ID

Description: This function will force all channels to leave a conference and will then delete the conference. The conference ID will no longer be valid after a call to this function/

Returns: Returns 0 if successful or -1 if the *conf_id* was invalid.

-o-

SMdump

Synopsis:
SMdump() - Dumps various information about the state of the system

Description: This function is used for debugging purposes and will dump information about the Aculab speech channels to the system tracelog.

Returns: Nothing

-o-

SMstate

Synopsis:
chan_state=SMstate(vox_chan)

Arguments:
vox_chan – The voice channel

Description: This function returns the blocking speech function type currently running on the specified *vox_chan* or 0 if the channel is idle. If the channel is currently running a blocking speech function then the *chan_state* returned by the function will be one of the following as defined in the ACULAB.INC file:

```
const MTF_PLAY            = 1;
const MTF_RECORD          = 2;
const MTF_WAITTONE        = 3;
const MTF_PLAYTONE        = 4;
const MTF_PLAYDIGITS      = 5;
```

```
const MTF_PLAYCPTONE      = 6;
const MTF_WAITRECOG       = 7;
const MTF_PLAYIPF         = 8;
```

This function can be used to determine whether a speech function has completed on a channel and is used in particular in non-blocking mode (see SMmode()).

Returns: Returns the channel state or -1 if a bad channel was specified

-o-

SMdetected

Synopsis:
num_items=SMdetected(vox_chan[,type])

Arguments:
vox_chan – The voice channel
type – Set to 0 to return the number of DTMF digits, 1 to return the number of other recognition events detected.

Description: This function allows for the number of DTMF (if *type=0*) or other recognition events (if *type=1)* that have been received on the specified *vox_chan.*

Returns: Returns the number of detected items in the appropriate channel buffer depending on the *type* argument, else returns -1 if an invalid channel was specified.

-o-

SMword

Synopsis:
SMword(vox_chan,prompt_no[,ipf_id])

Arguments:
vox_chan – The voice channel
prompt_no - The prompt number to add to the list of words to play (ranging from 1 upwards)
[ipf_id] - The optional IPF number (ranging from 1 upwards).

Description: See ***Index Prompt Files*** for more information about index prompt file initialisation.

This function allows for words to be added one at a time to the list of words from the Index Prompt Files to play on the specified *vox_chan* (with a call to SMplayph()).

To start a new list of words the function should be called with a prompt number of 0, otherwise the word will be added to the existing list that is being built up.

The *prompt_no* identifies the index into the IPF file of the prompt to play (starting at 0).

The optional *ipf_id* allows a specific IPF_ID to be specified (as defined in the PR.PAR file). If no *ipf_id* is specified then the one given in the SMplayph() will be used. IF an *ipf_id* is specified then the one given in he SMplayph() will be ignored for this word and the one specified here will be used instead.

Returns: Returns 0 if successful or a negative error code.

-o-

SMplayph

Synopsis:

term_event=SMplayph(vox_chan [,ipf_id, dataformat, samplerate])

Arguments:

vox_chan – The voice channel
[ipf_id] – Optional ipf_id (defaults to 1 if not specified)
[dataformat] – Optional voice prompt format (defaults to SMDataFormatOKIADPCM)
` *[samplerate]* – The sample rate of the file (defaults to 6000)

Description: This function plays the set of words/phrases that have been added to the list with the SMword() function. If the SMword() function was called specifying a ipf_id then any *ipf_id* specified in this fuction will be ignored for that particular word.

If the optional argument *ipf_id* is not specified then it defaults to 1. By default the words added using the SMword() function will play from the given *ipf_id* file (unless a specific one was specified in a call to SMword()).

The optional *dataformat* argument defines what the speech file format in the IPF file, and defaults to SMDataFormatOKIADPCM (see SMplay() for a list of valid speech file formats).

The optional *samplerate* can specify the same rate of the data in the IPF file and defaults to 6000 if not specified (see SMplay for a list of valid sample rates).

Returns: The function will return the terminating event that caused the play function to complete which will be one of the following as defined in ACULAB.INC

```
# Terminating events
const TERM_ERROR       = -1;
const TERM_TONE        = 1;
const TERM_ABORT       = 6;
const TERM_EODATA      =;7
```

The function may also return other negative error codes if invalid parameters were specified.

-o-

SMplaypr

Synopsis:

term_event=SMplaypr(vox_chan,ipf_id, prompt_no [,dataformat, samplerate])

Arguments:

vox_chan – The voice channel
ipf_id – The ipf file ID (as specified in the PR.PAR)
prompt_no – The prompt number withing the IPF file to play (starting from 1)
[dataformat] – Optional format of ipf file
[samplerate] – Optional sample rate of ipf file

Description: This function allows a single word/phrase to be played from a specific IPF file specified by the *ipf_id.* The word/phrase to play is specified by the *prompt_no* argument,

The optional *dataformat* argument defines what the speech file format in the IPF file, and defaults to SMDataFormatOKIADPCM (see SMplay() for a list of valid speech file formats).

The optional *samplerate* can specify the same rate of the data in the IPF file and defaults to 6000 if not specified (see SMplay for a list of valid sample rates).

Returns: The function will return the terminating event that caused the play function to complete which will be one of the following as defined in ACULAB.INC

```
# Terminating events
const TERM_ERROR    = -1;
const TERM_TONE     = 1;
const TERM_ABORT    = 6;
const TERM_EODATA   = 7;
```

The function may also return other negative error codes if invalid parameters were specified.

-o-

SMplaystrph

Synopsis:

SMplaystrph(vox_chan,ipf_id,str_words[,dataformat,samplerate])

Arguments:

vox_chan – The voice channel
ipf_id – The IPF id as specified in the PR.PAR file
str_words – A string containing the list of words to play from the IPF file (as ASCII values)
[dataformat] – Optional format of ipf file
[samplerate] – Optional sample rate of ipf file

Description: This function allow for a string of words/phrases from the IPF file specified by *ipf_id* to be played. The *string_words* string contains the string of ascii values each of which represents the id of a prompt within the IPF file to play.

For example:

x=SMplaystrph(vox_chan,1,"`01`04`06");

The above function call will play the first, fourth and sixth prompts from IPF file with *ipf_id* = 1

Returns: The function will return the terminating event that caused the play function to complete which will be one of the following as defined in ACULAB.INC

```
# Terminating events
const TERM_ERROR    = -1;
const TERM_TONE     = 1;
const TERM_ABORT    = 6;
const TERM_EODATA   = 7;
```

The function may also return other negative error codes if invalid parameters were specified.

-o-

SMplaystrphm

Synopsis:
SMplaystrphm(vox_chan,str_words[,dataformat,samplerate])

Arguments:
vox_chan – The voice channel
str_words – A string containing the list of pairs of *ipf_ids* and *words* to play (as ASCII values)
[dataformat] – Optional format of ipf file
[samplerate] – Optional sample rate of ipf file

Description: This function allow for a string of words/phrases from different IPF files to be specified by pairs of values in the *str_words* argument. The *string_words* string contains the string of pairs of ascii values where each pair represents the *ipf_id* and the *prompt_num* for each word to be played.

For example:

x=SMplaystrph(vox_chan,,"`01`04`06`02");

The above function call will play the 4th prompt from the 1st IPF and the 2nd prompt from the 6th IPF.

Returns: The function will return the terminating event that caused the play function to complete which will be one of the following as defined in ACULAB.INC

```
# Terminating events
const TERM_ERROR    = -1;
const TERM_TONE     = 1;
const TERM_ABORT    = 6;
const TERM_EODATA   = 7;
```

The function may also return other negative error codes if invalid parameters were specified.

/

-o-

SMcreateVMP

Synopsis:
SMcreateVMP(module_id,[local_ip_addr])

Arguments:
module_id – The module number
local_ip_addr - Optional local IP address

Description: This function creates a Virtual Media Processing port on the specified DSP *module_id*. Modules are numbered from 0 in the order that they are opened when the CXACUDSP.DLL library initialises. If an ACUCFG.CFG is supplied then this defines the order that the modules are opened, otherwise the acu_get_system_snapshot() will define the order that the cards (and thus the DSP modules) are opened.

The optional *local_ip_addr* argument allows a local ip address to be specified to identify a network adaptor within the system in the case when there are multiple network adaptors to choose from.

The function creates **vmprx** and **vmptx** virtual media processing end-points on the specified module to allow RTP data to be transmitted to and received from a remote ip destination. This function maps to the Aculab **sm_vmprx_create()** and **sm_vmptx_create()** functions.

Upon success the function will return a *vmp_handle* which can be used in subsequent calls that refer to this VMP channel.

For IP calls the VMP channel should be specified in call parameters for the CCaccept() call as follows:

```
port=0;
chan=1;
vox_chan=1;
module_id=0;
  vmp_chan=SMcreateVMP(module_id);

 // specify a codec on the vmp channel
  SMsetcodec(vmp_chan,0,G711_ALAW);

....

while(1)
      x=CCwait(port,chan,WAIT_FOREVER,&state);
      if(state eq CS_INCOMING_CALL_DET)
          CCalerting(port,chan);   // send INCOMING_RINGING event
      else if(state eq CS_WAIT_FOR_ACCEPT)
           CCclrparms(port,chan,PARM_TYPE_ACCEPT);
             CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPRXID,vmp_chan);
             CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPTXID,vmp_chan);
             CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_CODECS,vmp_chan);
         CCaccept(port,chan);
   endif endif
endwhile

 // make full duplex connection between VMP and a voice channel datafeeds
  SMfeedlisten(vmp_chan,TYPE_VMP,vox_chan,TYPE_VOX);
  SMfeedlisten(vox_chan,TYPE_VOX,vmp_chan,TYPE_VMP);
```

```
    SMplay(vox_chan,"TEST.VOX");
  etc...
```

Returns: Upon success this function returns a VMP channel handle, otherwise it returns a negative error code..

-o-

SMtraceVMP

Synopsis:
SMtraceVMP(vmp_chan,tracelevel)

Arguments:
vmp_chan – The VMP channel handle (as returned from SMcreateVMP())
tracelevel - The trace level (0=Off)

Description: This function allows trace to be turned on or off on the VMP channel specified by *vmp_chan.* Set the *tracelevel* to 0 to turn trace off or a non-zero value to turn trace on.

Returns: 0

-o-

SMdestroyVMP

Synopsis:
SMdestryVMP(vmp_chan)

Arguments:
vmp_chan – The VMP channel handle (as returned from SMcreateVMP())

Description: This function releases the VMP end-point specified by *vmp_chan* and frees the handle. It is up to the programmer to decide whether VMP channels are allocated once upon start-up or whether they are allocated dynamically as and when required (e.g. when an inbound IP call is detected).

The following example shows a VMP being created dynamically when an inbound call is received on an IP channel after which it is destroyed with SMdestroyVMP():

```
$include "aculab.inc"
main
  int port,chan,vox_chan,module_id,x,state;
    port=0;
    chan=1;
    vox_chan=1;
    module_id=0;

    while(1)
         x=CCwait(port,chan,WAIT_FOREVER,&state);
         if(state eq CS_INCOMING_CALL_DET)
             CCalerting(port,chan);   // send INCOMING_RINGING event
         else if(state eq CS_WAIT_FOR_ACCEPT)
             vmp_chan=SMcreateVMP(module_id);
```

```
                // specify a codec on the vmp channel
                  SMsetcodec(vmp_chan,0,G711_ALAW);


                  CCclrparms(port,chan,PARM_TYPE_ACCEPT);
                    CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPRXID,vmp_chan);
                    CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPTXID,vmp_chan);
                    CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_CODECS,vmp_chan);
                CCaccept(port,chan);
           endif endif
        endwhile

        // make full duplex connection between VMP and a voice channel datafeeds
         SMfeedlisten(vmp_chan,TYPE_VMP,vox_chan,TYPE_VOX);
         SMfeedlisten(vox_chan,TYPE_VOX,vmp_chan,TYPE_VMP);

        // play a prompt
         SMplay(vox_chan,"TEST.VOX");

        // Stop the full duplex listen to datafeeds
          SMfeedunlisten(vmp_chan,TYPE_VMP);
         SMfeedunlisten(vox_chan,TYPE_VOX);
        // release the VMP
         SMdestroyVMP(vmp_chan);

        // Clear down the call
         CChangup(port,chan);
         while(CCstate(port,chan)!=CS_IDLE)
           sleep(1);
        endwhile
        CCrelease(port,chan);

        // restart program to wait for next call..
       restart;
endmain
```

Returns: 0 upon success or a negative error code.

-o-

SMsetcodec

Synopsis:

SMsetcodec(vmp_chan,array_index,codecID[,vad[,fpp[,options]]])
)

Arguments:

vmp_chan	– The VMP channel handle (as returned from SMcreateVMP())
array_index	- Index of element in the codec array to set
codecID	- The Id of the codec to set in the codec array
vad	- (Optional) Turn Voice Activity Detection on or off (0=Off)
fpp	- (Optional) number of frames per packet
options	- (Optional) options

Description: This function allows elements of the array of supported codecs to be set on the VMP specified by *vmp_chan*.
Each VMP channel has an array of up to 11 codecs which it can accept which are indexed from 0 through to 10 by the *array_index* argument.

The codecID should be set to one of the following values as specified in **aculab.inc**:

```
const NOT_INITIALISED    =    0;
const G711_ALAW          =    1;
const G711_ULAW          =    2;
const G723               =    3;
const G729               =    4;
const G729A                   =    5;
const T38                =    6;
const G726               =    7;
const GSM_FR             =    8;
const iLBC               =    9;
const RFC4040            =    10;
const G728               =    11;
const AMR_NB             =    12;
const AMR_WB             =    13;
const EVRC               =    14;
const EVRC0                   =    15;         // headerless EVRC
const SMV                =    16;
const SMV0               =    17;         // headerless SMV
```

If no valid codecs are specified then the system codec list will be used. If this also contains no valid codecs then calls will be failed with a parameter error.

The optional *vad* argument allows the voice activity detector to be turned on for this call. Turning on the voice activity detector allows the IP Telephony card to perform silence suppression for that call. Permitted values are :

const VAD_ON= 1;
const VAD_OFF= 0;

The optional *fpp* can be used in API calls, prior to call connection, to specify the actual number of frames per
packet. The minimum value is 1 and the maximum value is 3, with the default being 2 frames per packet.

The optional *options* argument is not currently used and will be ignored.

Returns: 0 upon success or a negative error code.

-o-

SMclrcodecs

Synopsis:
SMcrlcodec(vmp_chan)

Arguments:
vmp_chan – The VMP channel handle (as returned from SMcreateVMP())

Description: This function clears all of the entries in the codec array for the specified VMP channel.

Returns: 0 upon success or -1 if an invalid *vmp_chan* is specified.

-o-

SMcreateTDM

Synopsis:
tdm_chan=SMcreateTDM(vox_chan)
Arguments:
vox_chan – The voice channel

Description: This function creates a TDM endpoint for the voice channel specified by *vox_chan.* The *vox_chan* must reside on a module on a board the supports TDM end-points (i.e Prosody X) and upon success it returns a TDM channel handle which should be used in all future function calls that reference this TDM. The internal Prosody stream and time slot and module id are obtained from the internal data associated with the *vox_chan* and are used when creating the TDM endpoint.

The function creates **tdmprx** and **tdmtx** end-points to allow RTP data to be transmitted to and received from a data feed and transmittted onto the internal TDM stream and timeslot. This function maps to the Aculab **sm_tdmrx_create()** and **sm_tdmtx_create()** functions.

To create a TDM endpoint for an E1 channel you should use the CCcreateTDM(port,chan) function.

Returns: Upon success this function returns a TDM channel handle, otherwise it returns a negative error code..

-o-

SMtraceTDM

Synopsis:
SMtraceTDM(tdm_chan,tracelevel)

Arguments:
tdm_chan – The TDM channel handle (as returned from SMcreateTDM() or CCvreateTDM())
tracelevel - The trace level (0=Off)

Description: This function allows trace to be turned on or off on the TDM channel specified by *tdm_chan.* Set the *tracelevel* to 0 to turn trace off or a non-zero value to turn trace on.

Returns: 0

-o-

SMdestroyTDM

Synopsis:

SMdestryTDM(tdm_chan)

Arguments:

tmd_chan – The TDM channel handle (as returned from SMcreateTDM() or CCcreateTDM())

Description: This function releases the TDM end-point specified by *tdm_chan* and frees the handle. It is up to the programmer to decide whether TDM channels are allocated once upon start-up or whether they are allocated dynamically as and when required.

Returns: 0 upon success or a negative error code

-o-

SMfeedlisten

Synopsis:

SMfeedlisten(chan1_id,chan1_type,chan2_id,chan2_type)

Arguments:

chan1_id – The channel into which the feed will be fed
chan1_type - The channel type (TYPE_VOX, TYPE_VMP, TYPE_TDM)
chan2_id - The channel whose feed to listen to
chan2_type - The channel type (TYPE_VOX, TYPE_VMP, TYPE_TDM)

Description: This function makes the channel specified by the arguments *chan1_id* and *chan1_type* 'listen' to the datafeeds provided by the channel specified by the arguments *chan2_id* and *chan2_type*.

The specified channels can be one of the following types as defined in **aculab.inc:**

```
const TYPE_VOX=0;
const TYPE_VMP=1;
const TYPE_TDM=2;
```

Channel types TYPE_VMP and TYPE_TDM must be have been previously created using the appropriate functions (SMcreateVMP() or SMcreateTMD()), whereas TYPE_VOX channels are allocated at startup.

The first time a TYPE_VOX, TYPE_VMP, or TYPE_TDM channel is used as a feed (i.e as *chan2_id* and *chan2_type* arguments) then the datafeed for that channel is created and stored in the internal structure associated with that channel.

For example, if you make a VMP channel listen to a VOX channel datafeed then any prompts or tones played on that VOX channel will be output by the VMP channel. For Example:

```
int vox_chan, port, chan, vmp_chan, module_id;
vox_chan=1;
module_id=0;
```

```
port=0;
chan=1;

 vmp_chan=SMcreateVMP(module_id);
// The vmp_chan will now listen to the vox_chan datafeed
 SMfeedlisten(vmp_chan,TYPE_VMP,vox_chan,TYPE_VOX);

// Wait for an inbound call (using the newly created VMP when we accept)
CCenablein(port,chan);
while(1)
        x=CCwait(port,chan,WAIT_FOREVER,&state);
        if(state eq CS_INCOMING_CALL_DET)
            CCalerting(port,chan);   // send INCOMING_RINGING event
        else if(state eq CS_WAIT_FOR_ACCEPT)
             vmp_chan=SMcreateVMP(module_id);

           // specify a codec on the vmp channel
             SMsetcodec(vmp_chan,0,G711_ALAW);


             CCclrparms(port,chan,PARM_TYPE_ACCEPT);
               CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPRXID,vmp_chan);
               CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPTXID,vmp_chan);
               CCsetparm(port,chan,PARM_TYPE_ACCEPT,CP_IPTEL_CODECS,vmp_chan);
           CCaccept(port,chan);
     endif endif
 endwhile

 // play a prompt which because of the above SMfeedlisten() will be heard by the caller..
  SMplay(vox_chan,"TEST.VOX");
...etc
```

To make a full duplex connection then make each channel listen to the other:

```
// The vmp_chan will now listen to the vox_chan datafeed
 SMfeedlisten(vmp_chan,TYPE_VMP,vox_chan,TYPE_VOX);
// The vox_chan will now listen to the vmp_chan datafeed
 SMfeedlisten(vox_chan,TYPE_VMP,vox_chan,TYPE_VMP);
```

For IP to TDM calls then the datafeeds from the VMP and the TDM can be connected as follows:

```
 int vox_chan, port, chan, vmp_chan, module_id;
vox_chan=1;
module_id=0;
e1port=0;
e1chan=1;

ipport=8;
ipchan=1;

 tdm_chan=CCcreateTDM(e1port,e1chan);    // Get a TDM endpoint for an E1 channel
 vmp_chan=SMcreateVMP(module_id);        // Create a VMP channel
// specify a codec on the vmp channel
 SMsetcodec(vmp_chan,0,G711_ALAW);

// Wait for an inbound call on IP channel(using the newly created VMP when we accept)
CCenablein(ipport,ipchan);
while(1)
        x=CCwait(ipport,ipchan,WAIT_FOREVER,&state);
        if(state eq CS_INCOMING_CALL_DET)
            CCalerting(ipport,ipchan);   // send INCOMING_RINGING event
        else if(state eq CS_WAIT_FOR_ACCEPT)
```

```
            // specify a codec on the vmp channel
              SMsetcodec(vmp_chan,0,G711_ALAW);


              CCclrparms(ipport,ipchan,PARM_TYPE_ACCEPT);
                CCsetparm(ipport,ipchan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPRXID,vmp_chan);
                CCsetparm(ipport,ipchan,PARM_TYPE_ACCEPT,CP_IPTEL_VMPTXID,vmp_chan);
                CCsetparm(ipport,ipchan,PARM_TYPE_ACCEPT,CP_IPTEL_CODECS,vmp_chan);
            CCaccept(ipport,ipchan);
        endif endif
    endwhile

    // Now make outbound call on E1 port/channel
    x=CCmkcall(e1port,e1chan,"123456"."987654");
   while(1)
          x=CCwait(e1port,e1chan,WAIT_FOREVER,&state);
          if(state eq CS_OUTGOING_RINGING)
             applog("Outgoing ringing")   // send INCOMING_RINGING event
          else if(state eq CS_CALL_CONNECTED)
          break;
          else if(state eq CS_CALL_DISCONNECTED)
            // clear down the calls here and restart..
           cleardown_calls();
          restart;
        endif endif
    endwhile

    // ** Now connect the feeds from the VMP and TDM endpoint to make
   // ** a full duplex connection to connect the conversations from the IP to E1 calls

    // The vmp_chan will now listen to the tdm_chan datafeed
     SMfeedlisten(vmp_chan,TYPE_VMP,tdm_chan,TYPE_TDM);
    // The tdm_chan will now listen to the vpm_chan datafeed
     SMfeedlisten(tdm_chan,TYPE_TDM,vpm_chan,TYPE_VPM);

    // The conversations are now connected..
   etc
```

Returns: Upon success it returns 0, otherwise a negative error code.

-o-

SMfeedunlisten

Synopsis:

SMfeedunlisten(chan_id,chan_type)

Arguments:

chan_id – The channel to stop listening to the feed
chan_type - The channel type (TYPE_VOX, TYPE_VMP, TYPE_TDM)

Description: This function stops the channel specified by the arguments *chan_id* and *chan_type* from listening to a datafeeds previously connected through a SMfeedlisten() call.

The specified channels can be one of the following types as defined in **aculab.inc:**

```
const TYPE_VOX=0;
const TYPE_VMP=1;
const TYPE_TDM=2;
```

Returns: 0 upon success or a negative error code.

-o-

Index

- A -

A First Look at the Language 12
Aculab Call Control Quick Reference 236
ADO Library Function Quick Reference 138
adoBlockMode 140
adoBusyState 140
adoConnClose 145
adoConnection 141
adoConnGetHandle 146
adoConnOpen 142
adoConnParmGet 143
adoConnParmSet 145
adoConnState 146
adoConnTransBegin 147
adoConnTransCancel 148
adoConnTransCommit 147
adoErrClear 169
adoErrCount 167
adoErrMessage 168
adoErrNative 169
adoErrValue 168
adoErrVerbose 139
adoFldCount 164
adoFldGetName 164
adoFldGetValue 165
adoFldParmGet 166
adoFldParmSet 167
adoFldSetValue 165
adoLastError 140
adoRecordSet 148
adoRSetAddNew 160
adoRSetCancelBatch 161
adoRSetCancelUpd 161
adoRSetClose 156
adoRSetCmd 152
adoRSetDelete 162
adoRSetGetHandle 156
adoRSetIsBOF 163
adoRSetMove 157
adoRSetMoveFirst 158
adoRSetMoveLast 159
adoRSetMoveNext 159
adoRSetMovePrev 159
adoRSetParmGet 154
adoRSetParmSet 156
adoRSetQuery 149
adoRSetRecCount 157
adoRSetRequery 153
adoRSetResync 152
adoRSetState 162
adoRSetUpdate 160
adoRSetUpdBatch 161
adoTrace 139
Ambiguous Operators 52
applog 112

Arithmetic Expressions 40
Arithmetic Operators 48
Assignment Expression 26
Assignment Expressions 44
Assignment Operators 49

- B -

Blocking and Non-Blocking Mode 292
Blocking or non-blocking mode 133
Board Opening Order 288
Break Statement Definition 36
Break Statement Example 36

- C -

CCabort 245
CCaccept 246
CCalarm 240
CCalerting 267
CCanscode 271
CCclrparms 262
CCclrxparms 278
CCcreateTDM 283
CCdisconnect 247
CCenablein 246
CCenquiry 274
CCgetaddr 271
CCgetcause 267
CCgetcharge 268
CCgetcnctless 279
CCgetparm 263
CCgetslot 241
CCgetxparm 276
CChold 273
CCinttohex 281
CCkeypad 273
CClisten 242
CCmkcall 247
CCmkxcall 279
CCnotify 272
CCnports 237
CCoverlap 268
CCproceed 270
CCprogress 270
CCputcharge 272
CCreconnect 274
CCrelease 248
CCsendfeat 280
CCsetparm 249
CCsetparty 275
CCsetupack 269
CCsetxparm 278
CCsiginfo 238
CCsigtype 237
CCsndcnctless 280
CCstrtohex 281
CCtrace 241
CCtransfer 275
CCtrunktype 239
CCunlisten 243
CCunstohex 282

CCuse 244
CCwait 244
CCwatchdog 240
Clipper Database Library Quick Reference 200
Command Line Options 68
Comparison Operators 48
Compiler Directives 57
Compiler Options 60
Constant Expressions 45
Continue Statement Definition 37
Continue Statement Example 37

- D -

db_append 204
db_close 206
db_fget 205
db_first 208
db_flock 211
db_fname 207
db_fput 205
db_fwidth 207
db_get 204
db_ixopen 203
db_key 210
db_next 209
db_nfields 207
db_nrecs 206
db_open 203
db_prev 210
db_recnum 211
db_rls 206
db_rlsall 208
Declaration Block Definition 19
Declaration Block Examples 19
Do Statement Definition 33
Do Statement Example 34

- E -

Environment Variables 64
errlog 114
Error Codes 135
Example Expression statements 27
Expression Statement Definition 24
Expression Types 40

- F -

Floating Point Library Quick Reference 212
For Statement Definition 34
For Statement Example 35
fp_add 212
fp_decs 212
fp_div 214
fp_mul 213
fp_pow 214
fp_rnd 215
fp_sub 213
Function Block Definition 21

Function Block Examples 22
Function Declaration Definition 22
Function Expression 26
Function Expressions 47
Function Name Resolution 66

- G -

glb_set 192
Global Array Library Quick Reference 192
Goto Statement Example 27
Goto Statment Definition 27

- I -

If Statement Definitions 37
If Statement Example 38
Indexed Prompt Files (IPFs) 289
Indirection Operators 49
Inter-task Messaging Libary Quick Reference 187
Introduction 12, 16, 191, 215, 212, 200, 60, 129, 76, 78, 87, 108, 224, 284, 186, 67
Introduction 12, 16, 191, 215, 212, 200, 60, 129, 76, 78, 87, 108, 224, 284, 186, 67
Introduction 12, 16, 191, 215, 212, 200, 60, 129, 76, 78, 87, 108, 224, 284, 186, 67

- J -

Jump Statement Definition 28
Jump Statement Example 29

- L -

Label Statement Definition 29
Label Statement Example 30
Library Configuration Tab 73
Library Limits and Defaults 87
Library Quick Reference 79
Loading DLLs and .DEF files 65
Logical Expressions 42
Logical Operators 49

- M -

Manual Conventions 77
Miscellaneous Operators 52
More on Arrays 56
More on Functions 53
More on Variables 55
msg_flush 189
msg_freecount 191
msg_read 188
msg_send 189
msg_senderid 190
msg_sendername 190
msg_setname 188

- N -

Nailing transmit timeslots to H.100 or SCBUS 288
Notes on Style 15

- O -

Onsignal Declaration Definition 22

- P -

Performance and blocking calls 133
Private and Public Objects 134
Program Structure 18

- R -

Registry Settings 69
Restart Statement Definition 31
Restart Statement Example 31
Return Statement Definition 30
Return Statement Example 30
Run-time Initialisation and configuration 228

- S -

Saccept 220
Scheck 221
Sclose 217
Sconnect 216
Scrolling Log Tabs 70
Semaphore Library Quick Reference 198
Shostname 222
Side Effects From Signals 88
Simple VOIP -> TDM example 233
Simple VOIP example 286
Slisten 219
SMabort 302
SMaddASRitem 314
SMaddASRvocab 313
SMcardinfo 295
SMclrASRitems 315
SMclrASRvocabs 313
SMclrtones 308
SMconfend 319
SMconfjoin 316
SMconfleave 318
SMconfstart 316
SMcreateTDM 328
SMctlASR 315
SMctlCPtone 305
SMctlDtmf 305
SMctlGrunt 306
SMctlPulse 305
SMdestroyTDM 329
SMdestroyVMP 325
SMdetected 320

SMdump 319
SMfeedunlisten 331
SMgetcards 295
SMgetchannels 294
SMgetmodules 294
SMgetrecparm 302
SMgetrecrognised 310
SMgetslot 303
SMgetttones 308
SMlisten 304
SMmode 312
SMmodinfo 296
SMplay 296
SMplaydigits 309
SMplayh 298
SMplayptone 309
SMplaytone 309
SMrecord 299
SMsetASRchanparm 313
SMsetrecparm 300
SMstate 319
SMtoneint 306
SMtrace 312
SMtraceTDM 328
SMtraceVMP 325
SMunlisten 304
SMwaittones 307
SMword 320
Some Simple Examples 130, 230, 284
SopenDGRAM 222
Srecv 218
SrecvDGRAM 223
Ssend 220
SsendDGRAM 222
Statement Block Definition 23
Stop Statement Definition 32
Stop Statement Example 32
Strace 224
String Concatenation Operator
String Library Quick Reference 170
Switch Statement Definition 39
Switch Statement Example 39
SWmode 282
SWquery 283
SWset 283
Syntax definitions 17
syslog 113
System Library Quick Reference 88
sys_bufcopy 90
sys_bufget 91
sys_bufmove 91
sys_bufrls 90
sys_bufrlsall 90
sys_bufset 91
sys_bufuse 89
sys_date 103
sys_dateadd 105
sys_datecvt 107
sys_dirfirst 100
sys_dirmake 102
sys_dirnext 101
sys_dirremove 99
sys_diskfree 101
sys_exit 107
sys_fcopy 99

sys_fdelete 102
sys_fhclose 93
sys_fhcloseall 93
sys_fheof 97
sys_fhgetline 96
sys_fhlock 98
sys_fhopen 92
sys_fhputline 96
sys_fhreadbuf 94
sys_fhseek 94
sys_fhsetsize 98
sys_fhunlock 98
sys_fhwritebuf 95
sys_fhwrites 97
sys_finfo 103
sys_frename 100
sys_getenv 108
sys_gethandle 102
sys_settime 106
sys_ticks 104
sys_time 104
sys_timeadd 104
sys_timesub 105
sys_tmrsecs 106
sys_tmrstart 106

- T -

task_arg 86
task_chain 81
task_clrdefer 85
task_defersig 84
task_exec 81, 78
task_getpid 86
task_hangup 83
task_kill 86
task_parentid 82
task_return 82
task_sleep 83
task_spawn 80
TE Language Operators 47
Teminating Events 290
Terminal Console Library Quick Reference 111
term_attr_def 120
term_box 118
term_clear 124
term_colour 119
term_cur_pos 117
term_errctl 115
term_fill 123
term_kbedit 128
term_kbget 125
term_kbgetx 125
term_kbqsize 127
term_log 115
term_print 118
term_put_nch 123
term_resize 116
term_scroll_area 117
term_size 116
term_write 117
The $if Directive 58
The $include Directive 57
The ACUCFG.CFG Configuration file 224

The Conditional Expression Operator 53
Tools Tab 75
tracelog 114

- W -

While statement Definition 32
While Statement Example 33

© Zentel Telecom Ltd., 2009
www.telecom-engine.com