



DYACON®
RUGGED DATA SYSTEMS



Software Development Manual

CT6 Series

57-5103 Rev E
April 2009

Contents

CT6™ SERIES SOFTWARE DEVELOPMENT MANUAL	1
NOTICES.....	1
© Copyright 2007 Dyacon, Inc.....	1
CT6 FIRMWARE	2
SCOPE.....	2
OVERVIEW.....	2
<i>Firmware Block Diagram</i>	3
FILE SYSTEM.....	4
<i>Flash File System</i>	4
<i>TFS API</i>	4
CT6XX ENVIRONMENT.....	6
EXECUTABLES	6
<i>File Types and File Extensions</i>	7
SCRIPT INVOCATION	8
<i>Script-Specific Commands</i>	8
<i>Script Nesting</i>	8
COMMAND LINE REDIRECTION	9
SHELL VARIABLE USAGE	10
PRIVILEGE LEVELS FOR COMMANDS AND FILES	11
SHELL VARIABLES.....	12
<i>APPRAMBASE</i>	12
<i>ARGC</i>	13
<i>ARG'N'</i>	13
<i>BOOTROMBASE</i>	13
<i>CMDSTAT</i>	13
<i>CONSOLEBAUD</i>	13
<i>ENTRYPOINT</i>	13
<i>EXCEPTION_SCRIPT</i>	13
<i>EXCEPTION_TYPE</i>	14
<i>OSBUILT</i>	14
<i>NO_EXCEPTION_RESTART</i>	14
<i>PLATFORM</i>	14
<i>PSI</i>	14
<i>SCRIPT_IGNORE_ERROR</i>	14
<i>SCRIPTVERBOSE</i>	14
<i>VERSION_MAJ, VERSION_MIN</i>	14
SHELL COMMAND SET.....	15
SHELL COMMAND SET SUMMARY.....	15
SHELL COMMANDS.....	18
<i>call</i>	18
<i>cat</i>	19

<i>cp</i>	20
<i>date</i>	21
<i>dchk</i>	22
<i>defrag</i>	23
<i>df</i>	23
<i>delay</i>	24
<i>echo</i>	24
<i>edit</i>	25
<i>exit</i>	27
<i>finfo</i>	27
<i>flash</i>	28
<i>fsize</i>	28
<i>gosub</i>	29
<i>goto</i>	29
<i>gpio cfg (General Purpose IO Interface)</i>	30
<i>gpio info (General Purpose IO Interface)</i>	30
<i>gpio pin (General Purpose IO Interface)</i>	30
<i>gpio set (General Purpose IO Interface)</i>	31
<i>help</i>	31
<i>history</i>	32
<i>if</i>	33
<i>item</i>	34
<i>ld</i>	35
<i>ls</i>	35
<i>mem copy</i>	36
<i>mem dump</i>	37
<i>mem fill</i>	38
<i>mem put</i>	38
<i>mem srch</i>	39
<i>mem test</i>	39
<i>plvl</i>	40
<i>read</i>	41
<i>reset</i>	41
<i>return</i>	42
<i>rm</i>	42
<i>rs232 cfg</i>	43
<i>rs232 cts</i>	43
<i>rs232 dsr</i>	44
<i>rs232 dtr</i>	44
<i>rs232 rts</i>	45
<i>run</i>	45
<i>set</i>	46
<i>sleep</i>	46
<i>suspend</i>	47
<i>sysinfo</i>	47
<i>time</i>	47
<i>tfs add</i>	48
<i>tfs init</i>	48
<i>tfs ramdev</i>	49
<i>tfs stat [dev]</i>	49
<i>version</i>	49
<i>watchdog disable</i>	50
<i>watchdog enable</i>	50
<i>watchdog timeout</i>	50

<i>watchdog test</i>	50
<i>xrcv</i>	51
<i>xsnd</i>	51
OBC605 API	52
API SUMMARY.....	52
OBC605 APPLICATION PROGRAMMER'S INTERFACE.....	57
<i>appexit ()</i>	57
<i>cell_enable ()</i>	58
<i>cell_power ()</i>	58
<i>com2_autorts ()</i>	59
<i>com2_cfgcallback ()</i>	60
<i>com2_close ()</i>	61
<i>com2_cts ()</i>	61
<i>com2_getcfg ()</i>	62
<i>com2_open ()</i>	62
<i>com2_read ()</i>	63
<i>com2_rxavail ()</i>	63
<i>com2_rxpurge ()</i>	63
<i>com2_setcfg ()</i>	64
<i>com2_setrts ()</i>	65
<i>com2_txfree ()</i>	65
<i>com2_txpurge ()</i>	66
<i>com2_write ()</i>	66
<i>dcgood_cfgcallback ()</i>	67
<i>dcgood_wait ()</i>	67
<i>delay ()</i>	68
<i>dgtlin_cfg ()</i>	69
<i>dgtlin_get ()</i>	70
<i>dgtlout_set ()</i>	71
<i>docommand ()</i>	72
<i>getargv ()</i>	73
<i>getenv ()</i>	73
<i>getenvp ()</i>	74
<i>gps_enable ()</i>	75
<i>ints_disable ()</i>	75
<i>ints_enable ()</i>	76
<i>ints_restore ()</i>	76
<i>irq_enable ()</i>	77
<i>irq_gethandler ()</i>	77
<i>irq_sethandler ()</i>	78
<i>led_set ()</i>	78
<i>printmem ()</i>	79
<i>reset ()</i>	79
<i>rs232_break ()</i>	80
<i>rs232_brkdetect ()</i>	80
<i>rs232_cd ()</i>	81
<i>rs232_cfgcallback ()</i>	82
<i>rs232_close ()</i>	83
<i>rs232_cts ()</i>	83
<i>rs232_dsr ()</i>	84
<i>rs232_dtr ()</i>	84
<i>rs232_frmerror ()</i>	85
<i>rs232_getc ()</i>	85

<i>rs232_getcfg ()</i>	86
<i>rs232_open ()</i>	87
<i>rs232_ovrerror ()</i>	87
<i>rs232_prtyerror ()</i>	88
<i>rs232_putc ()</i>	88
<i>rs232_read ()</i>	89
<i>rs232_ri ()</i>	89
<i>rs232_rts ()</i>	90
<i>rs232_rxavail ()</i>	90
<i>rs232_rxpurge ()</i>	91
<i>rs232_setcfg ()</i>	91
<i>rs232_txfree ()</i>	92
<i>rs232_txpurge ()</i>	92
<i>rs232_write ()</i>	93
<i>rs485_autotxline ()</i>	93
<i>rs485_cfgcallback ()</i>	94
<i>rs485_close ()</i>	95
<i>rs485_cts ()</i>	95
<i>rs485_getcfg ()</i>	96
<i>rs485_open ()</i>	96
<i>rs485_read ()</i>	97
<i>rs485_rxavail ()</i>	97
<i>rs485_rxpurge ()</i>	98
<i>rs485_setcfg ()</i>	98
<i>rs485_txfree ()</i>	99
<i>rs485_txline ()</i>	99
<i>rs485_txpurge ()</i>	100
<i>rs485_write ()</i>	100
<i>rtc_getalarm ()</i>	101
<i>rtc_getdate ()</i>	102
<i>rtc_gettime ()</i>	103
<i>rtc_setalarm ()</i>	103
<i>rtc_setdate ()</i>	104
<i>rtc_settime ()</i>	104
<i>rtc_timestamp ()</i>	105
<i>setenv ()</i>	106
<i>sleep ()</i>	106
<i>suspend ()</i>	107
<i>sys_free ()</i>	107
<i>sys_getbytes ()</i>	108
<i>sys_getchar ()</i>	108
<i>sys_getline ()</i>	109
<i>sys_gotachar ()</i>	109
<i>sys_heap_extend ()</i>	110
<i>sys_malloc ()</i>	111
<i>sys_printf ()</i>	111
<i>sys_putbytes ()</i>	112
<i>sys_putchar ()</i>	112
<i>sys_puts ()</i>	113
<i>sys_realloc ()</i>	113
<i>sys_sprintf ()</i>	114
<i>sys_version ()</i>	114
<i>TFS RETURN CODES</i>	115
<i>tfsadd ()</i>	116

<i>tfsclose ()</i>	117
<i>tfseof ()</i>	118
<i>tfstat ()</i>	118
<i>tfgetline ()</i>	119
<i>tfinit ()</i>	119
<i>tfioctl ()</i>	120
<i>tfnext ()</i>	121
<i>tfsopen ()</i>	122
<i>tfsread ()</i>	124
<i>tfsseek ()</i>	125
<i>tfsstat ()</i>	125
<i>tfstell ()</i>	126
<i>tftruncate ()</i>	126
<i>tfsunlink ()</i>	127
<i>tfswrite ()</i>	128
<i>tmr_cfgfnct ()</i>	129
<i>tmr_check ()</i>	129
<i>tmr_reset ()</i>	130
<i>tmr_setmst ()</i>	130
<i>tmr_sett ()</i>	131
<i>tmr_start ()</i>	131
<i>tmr_stop ()</i>	132
<i>watchdog_enable ()</i>	132
<i>watchdog_service ()</i>	133
<i>watchdog_timeout ()</i>	133
J1708 – JBUS DEVICE	134
JBUS DEVICE	134
<i>Programming Considerations</i>	134
Opening/Closing the J1708 Port	134
Buffer	135
Printf and J1708	135
J1708 API	136
<i>j1708_add_filter ()</i>	136
<i>j1708_close ()</i>	136
<i>j1708_find_filter ()</i>	137
<i>j1708_get_errors ()</i>	137
<i>j1708_get_filter ()</i>	138
<i>j1708_get_message ()</i>	139
<i>j1708_get_specific_message ()</i>	140
<i>j1708_get_version ()</i>	141
<i>j1708_open ()</i>	142
<i>j1708_remove_filter ()</i>	142
<i>j1708_rxpurge ()</i>	143
<i>j1708_send_message ()</i>	143
<i>j1708_set_filter_type ()</i>	144
MANUAL HISTORY	145
57-5102-01A March, 2007	145
57-5102-02A May, 2007	145
57-5102-03A November, 2007	145
57-5102-04 July 2008	145

CT6™ Series Software Development Manual

Notices

© Copyright 2007 Dyacon, Inc.

All Rights Reserved

This publication is protected by copyright and all rights are reserved. Any reproduction of this manual, in part or in full, by any means, mechanical, electronic, or otherwise, is strictly prohibited without express written permission from Dyacon, Inc..

The information in this manual has been carefully checked and is believed to be accurate. However, Dyacon, Inc. assumes no responsibility for any inaccuracies that may be contained in this manual. All information is subject to change.

Trademark Acknowledgments

Wescor®, is a registered trademark of Wescor, Inc.

CT6™, RDT950™, RDT800™, Mobile Rx600™, Route Tracker™, Tracker-GPS™, RDTLink™, WTerm™, OBCLink™ and OBC600™ are recognized trademarks of Dyacon, Inc. (formerly Wescor Inc)

MS-DOS®, Windows® and Windows 95® and Windows CE® are registered trademarks of Microsoft Corporation®.

Enfora Enabler™ is a registered trademark of Enfora.

CDMA 2000™, MSM6050™, Qualcomm® are registered trademarks of Qualcomm.

All other trademarks are property of their respective owners.

CT6 Firmware

Scope

There are three manuals for the CT6 Series;

- (1) 57-5102-xx CT6 Series Reference Manual
- (2) 57-5103-xx CT6 Series Software Development Manual
- (3) 57-5104-xx CT6 Series Quick Start Guide

The content of this document is intended for software / application developers. The system integrator is responsible to provide specific operating instructions and manuals for end users.

Overview

Extensible Firmware Platform (EFP) is a very simple 32-bit operating system and does not support multiple users or multitasking. It is a target-resident environment that provides the developer with a suite of capabilities that enhance the development process and the environment for the application to execute.

EFP is the firmware that the CPU executes immediately after a reset or power-up. EFP resides in the non-volatile flash memory of the CT6xx. It is responsible for booting the CPU and getting the system to a state where a user can access the CT6xx through an RS-232 interface. After EFP initializes the system, it presents itself as a command line interface to the user.

The command interpreter (shell) provides a set of internal commands, variables, command line editing and history, command output redirection, user levels, and password protection.

EFP also configures flash memory as a file system (Tiny File System). The file system provides the capability of accessing flash memory as name space or address space. The files may be data files, compiled binary files in ELF format, script files, or configuration script file.

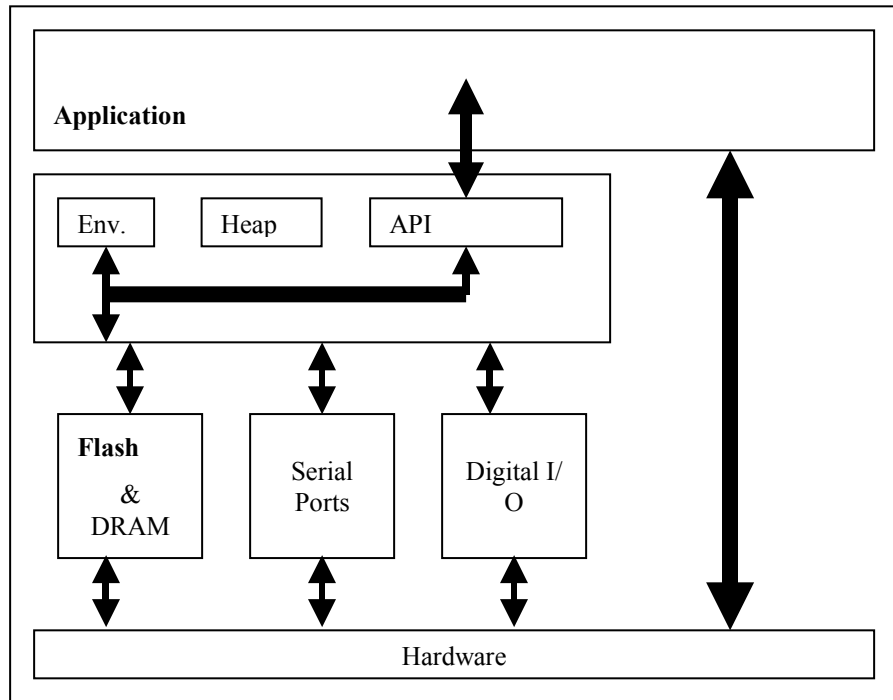
Executable files (ELF and scripts) may have an auto-bootable attribute, which the system uses to recognize files that need to be run at boot time. If more than one file has the auto-bootable attribute, they are executed sequentially and in alphabetical order.

In EFP, everything except the firmware itself is a file. When the application is running (as a result of it being loaded from TFS flash memory to DRAM by EFP), other files can be accessed by the active application.

EFP provides the capability of downloading and uploading files through a serial port. The transport protocol used is XMODEM and it also supports XMODEM 1K.

EFP has built in device drivers for all the system devices. The device drivers in turn interface with an application through a set of API's that are dynamically linked to the application at run time. The application has the capability to use the APIs or access the hardware directly if desired.

Firmware Block Diagram.



File System

A close look to the commands provided for the OBC605 file system reveals that there are no commands that change the file header, that is, commands to change file type, rename the file, etc.

The file system organizes the files within the flash in a contiguous one-way linked list. The initial portion of the file is a file header, which contains information about the file, pointer to the next file, and 32-bit CRCs of the header and data portion of the file. Maintaining unique CRC checks for the header and data allows the file system to more accurately detect corruption. File size is limited only by the amount of flash allocated to the file system. There is no restriction with regard to sector boundaries.

As files are created, they are appended to the end of the linked list of files. If a file is deleted from the list, it is simply marked as deleted. At some point, after several files have been deleted, it becomes necessary to clean up the file system flash space by running a defragmentation. This requires that a sector be dedicated to the defragmentation process and it also uses a small block of flash at the end of the file system flash space for maintaining a non-volatile state that can be retrieved in the event of an interrupted defragmentation (power hit or reset).

Note that the spare sector resides outside the file system's flash space.

Flash File System

Flash memory is composed of 256 Kbyte sectors. Individual bytes within these sectors can be written, but whole sectors must be erased at one time. This process is handled by the flash file system and is opaque to the application. While the write process is relatively fast, the erase process is slow.

To minimize the impact of erasing sectors during program execution, sectors are tagged for deletion and new or modified data is written to a fresh sector. These tagged sectors, or garbage, accumulate until, combined with active data, the drive is full. Accrual of these unnecessary files can also cause some delay when processing the linked file list, such as during boot or when retrieving files from flash memory.

Automatic garbage collection is initiated when the drive is full. However, this has the potential of stalling the current operation while drive defragmentation is in progress. As a preventative step, it is suggested that the application defragment the flash drive periodically.

The following command can be called in the application at a convenient time when any delays are unlikely to cause a conflict: system "defrag." The following commands could also be used within the application:

```
int _system (* char);  
_system ("defrag");
```

TFS API

The TFS (Tiny File System) API presents itself to the programmer in much the same way a standard OS file system would be seen. This may mislead application developers into thinking that the file system can be thought of as a disk that provides the user with the freedom to write/erase at any frequency. This is not the case. The TFS defragmentation mechanism does not use a floating spare sector. This means that the spare sector is the portion of flash that is likely to wear out first simply

because it is used as temporary storage for all other sectors when defragmentation is done. Assuming a worst-case defragmentation where all 61 TFS sectors are affected, a defragmentation run once a day on a part having a life expectancy of 100,000 erase cycles will be good for about 4 years ($100,000/61/365 = 4$). This value of 4 years makes the assumption that a defragmentation will be done daily, and that all sectors are affected by the defragmentation. The frequency of defragmentation and the number of sectors affected are very dependent on the application's use of the file system.

Some applications may wear out flash faster than others based on the way the sectors are utilized. When a file is deleted, it is simply marked as deleted (a bit in the header). A file is deleted by the command *rm* or through the API functions *tfsunlink()*, and *tfsclose()* when the file was opened for writing or appending. If a file that exists is opened for writing, the actual modification steps are deletion of the original file and re-write of the new file after the file that is currently the last file in physical flash space. Eventually this process reaches the end of the flash space used by TFS, so defragmentation must be run. At defragmentation, all of the space wasted by the deleted files is cleaned up and the current list of active files is placed end-to-end in the flash. This means that each sector used by TFS must be copied to the spare sector. That sector is then erased and only the active files in that sector are copied back to the original sector from the spare sector. This is repeated for each sector in TFS. The defragmentation attempts to make this as sector-erase-efficient as possible. If a sector is not affected by the defragmentation, then it is left untouched; hence, the number of sectors actually affected by a defragmentation depends on the amount of flash space currently being used by TFS and the position of the deleted files within the flash space.

In a typical project the files in TFS will be the application itself, an inittab file for system configuration, followed by other application-specific files. The application executable is likely to be the largest file and also the least likely to change at a high frequency. This means that the space taken up by the application executable is fairly static. The number of sectors occupied by the application will not contribute to the wear of the spare sector. This fact in and of itself increases the estimated life expectancy calculated previously. On the other hand, if the application executable is quite large relative to the total TFS space available, and if there are any files that are to be modified on a regular basis, defragmentation will be run more frequently. The point is that the actual wearing rate of the flash is application-specific and must be seriously considered and thought out during application development.

CT6xx Environment

Executables

There are two different types of executable files that can be stored in the CT6xx. The system supports binary and ASCII executables. Similar to other computer systems, files can be loaded and run as native program code (binary) or interpreted line-by-line and run as a script or batch file (ASCII).

The CT6xx supports binary load files in the ELF format. This means that a standard ELF file, as it is generated by off-the-shelf GNU tools (and Microcross GNU X-Tools), can be transferred to the CT6xx and run as is. The load file is considered to be absolute; the system does not do any relocation.

Executables can also be “autobootable.” There are three different types of autobootable files:

Autoboot with query

When the system starts up, the user is queried at the console to see if the automatic execution of the file should be aborted. If an ‘n’ is detected within about 2 seconds, it is aborted; otherwise the autoboot continues.

Autoboot without query

When the system starts up, this type of file is immediately executed without allowing the user to intervene.

Autoboot inittab

This is a special case autobootable. It is automatically executed as a script prior to the system being completely initialized; hence, environment variables established in the inittab file are used by the system initialization. When the inittab script is automatically executed, the CT6xx firmware is still initializing its own internals, so it isn’t ready for an application at that point.

Inittab should have the ‘s’ flag to be recognized. Do not set the ‘b’ or ‘q’ flags for this file. Inittab should be used for environmental initialization purposes only. No programs should be launched from here.

The system always sets the console speed to 38400 at boot and changes it, if necessary, after processing the inittab file. To change the baud rate of the console, add the following line to inittab:

```
Set CONSOLEBAUD baud
```

Where baud is a valid baudrate value.

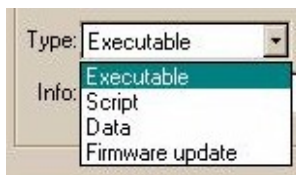
The inittab file is typically used to establish the interface baud rate, user level, and perhaps some other very basic environment setup. It cannot be aborted; thus, provides some guaranteed environment setup at startup. The ability to query the user is very handy during development because if the executable hangs the system, the query can be used to abort prior to the hang.

Notes

To prevent autoboot files from executing (inittab, or files with the 'b' or 'q' flag), type '^C' (control-C) at boot time.

Files with the 'q' flag will prompt. If any character is pressed at this time, the file is skipped; otherwise the file is executed.

File Types and File Extensions



There are four types of files that can be used with the CT6xx. The file types are: Executable, Script, Data, and Firmware update. File type must be specified for each file that gets uploaded to the CT6xx. This is done using OBCLink. A “type” pick list is available for selecting the type of file to upload, or file extensions can be used for the executable and script files to set the file type and the file attributes. The following file extensions are recognized by the OBCLink interface (the file extensions do not have any meaning on the CT6xx):

Extension	Description
.x	Executable (binary elf file)
.xb	Executable file and runs on boot
.xq	Executable file and queries on boot before running
.s	Script file
.sb	Script file and runs on boot
.sq	Script file and queries on boot before running

Notes:

- File extensions are case sensitive. All characters must be in lower case.
- Files with any other extension are considered data files.
- The extensions are used to set the file attributes and also can be used to remind the user what type of files they are and what attributes get set.
- If a file has no extension, the last flags settings on the dialog box are used.
- The file extensions do not have any meaning on the CT6xx, they are just for the OBCLink interface.

Script Invocation

A file is recognized as a script if it has the script (s) flag set. Scripts can be started up in two different ways. Like a UNIX or DOS shell, the script name can be typed on the command line directly and the command interpreter will find it. This implies that a script should not have the same name as any of the built-in commands in the command table because the command interpreter first looks through its own list of built-ins for a match.

Alternatively, the “run” command can be used. Running the script this way eliminates the concern of an executable having the same name as an internal built-in, plus it allows the user to specify that the script run with verbosity enabled. With verbosity enabled, each line is printed prior to its execution.

The shell variable SCRIPTVERBOSE can also be used to establish/modify script verbosity. Prior to running each line in the script, the shell variable is tested and the verbosity is adjusted accordingly. This allows the script itself to enable/disable verbosity.

A script is a readable ASCII file that contains commands that can be part of the shell (i.e. built-ins), or may be application programs. The following is a simple script:

```
# Sample script
# This script counts and displays it's count
# every two seconds until the console
# detects a character

set COUNT 0
echo "Count started"

# APP_LOOP:
sleep 2
echo Loop $COUNT
set COUNT=$COUNT+1
if -t ngc goto APP_LOOP

echo Done!
```

Script-Specific Commands

The following commands are applicable only within the context of a script:

IF, EXIT, GOSUB, GOTO, ITEM, READ, RETURN

All commands can be put in a script, but the above sets of commands are only useful when used in the context of a script.

Script Nesting

Script nesting is supported and is limited only by the amount of stack the system has.

The environment within a script (i.e. the shell variables) is not part of the script's stack. All shell variables are global. If script A sets VAR_1, then calls script B which modifies VAR_1, script A sees that modification when script B returns and script A regains control.

Command Line Redirection

At the command line the syntax for redirecting is similar, though not identical, to standard redirection on Unix. The difference is due to the fact that the redirection must be supplied with not only a file name, but also a buffer and buffer size. The idea here is that once supplied with a buffer, command output can be copied to this buffer and eventually the buffer will be transferred to a file. The syntax follows, with the redirection directives underlined.

```
>: echo this is some text >buffer,buffer_size[,filename]
```

This is the syntax for the single right arrow. A one or two comma delimited string containing a buffer address followed by the size of the buffer and an optional file name. If the filename is specified, then the output of the command is copied to the buffer (truncated at `buffer_size` if necessary) and then transferred to TFS as "filename" or serial port if the name refers to one of the system serial ports. The running buffer pointer is reset back to the base address of buffer. If filename is omitted, then the output of the command is copied to the buffer and the pointer into the buffer is left at the position just after the data copied (assuming `buffer_size` is not reached).

```
>: echo this is more text >>[filename]
```

This syntax is used to append the output of the command to the buffer that was created by the '`>`' syntax described above. If 'filename' is present, then the content of the buffer is transferred to TFS as "filename" and the running buffer pointer is reset back to the base address of the buffer. If "filename" is not specified, there is no transfer to a file, and the running pointer is incremented to the position just after the data copied (once again assuming `buffer_size` is not reached).

In both cases above, the "filename" string can contain up to 2 additional commas. These would be used to tell TFS the flags and info field to apply to the file when it is created, so the filename string could be "filename,flags,info."

It is the user's responsibility to make sure that the specified buffer is memory space that can be used. There are a few different ways to determine a buffer area. Typically, the buffer would simply be \$APPRAMBASE.

Shell Variable Usage

Similar to most other shells, shell variable names can contain alphanumeric characters and the underscore ('_'). Once assigned, the value within the variable can be accessed by preceding the variable name with a dollar sign (\$). The `${}` syntax is also supported. Use of the curly braces tells the command line processor to force the start and end point of a shell variable name that may otherwise not be seen as a variable because it is embedded within a larger string of characters. In addition, the dollar sign can be preceded by a backslash to negate its meaning as a shell variable starting delimiter.

Nested shell variables are supported, that is, variables based on other shell variables. For example, if there were three variables names `VAR_1`, `VAR_2`, and `VAR_3`, and a fourth variable called `IDX`, which contained the value 2 in it, then the expression `echo ${VAR_${IDX}}` outputs the contents of `VAR_2`.

Privilege Levels for Commands and Files

The CT6xx supports 4 privilege levels (0 through 3), with 3 being the highest privilege level (i.e. Superuser or Administrator). The files in TFS and the shell commands can be configured to require a certain privilege level to gain access to a file or the ability to execute the command.

At any given time, the system is running at some privilege level. The *plvl* command supports the ability to display the current privilege level as well as modify and/or configure the privilege levels. At startup, the target's default privilege level is at its highest (3) and all commands and files default to require privilege level 0 for access; hence, everything is accessible. If left untouched (via the *plvl* command), the privilege level remains at its max value and all facilities are accessible through the command line interface. If the running privilege level is lowered (by an autobootable script for example), then all accesses made to the hardware through the user interface after that is limited to the facilities (commands and files) that are available to the new privilege level. Commands and files can be configured to be accessible by some minimum privilege level.

At system startup, all commands default to require that the system be at privilege level 0 or higher to execute (in other words, all commands can be executed). If commands are to be restricted to different privilege levels, then the *plvl* command should be used in the inittab file to make all the necessary adjustments. For example, if the "xsnd" command is to be accessible only by privilege level 3, then in the inittab file (or some other autoboot-without-query script) the line "plvl -c xsnd,3" raises the required privilege level of the xsnd command and the line "plvl 0" lowers the current privilege level of the system to 0; hence, to run the xsnd command, the user needs to know the privilege-level-3 password. Note that a default system startup does not have any password file installed, so access to the various privilege levels is approved with any password. The "plvl -p" command must be used to create the three required passwords.

The usefulness of this feature depends on the needs of the application integrators, but it basically provides the system with a mechanism to protect some portion of the system from unauthorized users. The privilege level can be raised, but only by a user that knows the password to get to that particular privilege level. The passwords are encrypted and stored in a file in TFS that is automatically saved at the highest privilege level. The file is also unreadable by privilege levels lower than the max.

Shell Variables

Following is a summary of the shell variables:

Shell Variable	Description
APPRAMBASE	Starting point of the RAM space that is made available to the application.
ARGC	Argument count, includes argv[0] as an argument.
ARG'N'	Argument count, shell variables ARG0 through 'N' are automatically loaded with the argument list.
BOOTROMBASE	Address that the system sees as the starting point of the base flash device.
CMDSTAT	Status of the previous command within a script.
CONSOLEBAUD	Console baudrate – default is 38400
ENTRYPOINT	Is set by the ld command. Value corresponds to the entry point address of the application just loaded.
EXCEPTION_SCRIPT	If set, assumed to contain a script name that is to be executed when an exception occurs.
EXCEPTION_TYPE	If an exception occurs, this shell variable will contain the type of exception that occurred.
OSBUILT	Contains the string created by the concatenation of the following three strings at build time: DATE "@” TIME
NO_EXCEPTION_RESTART	At the time of an exception the system remains at the command prompt.
PLATFORM	Verbose description of the system for general use by the application.
PS1	Contains the string that the shell uses as the user prompt.
SCRIPT_IGNORE_ERROR	If present, then a running script does not stop when a command line error is detected.
SCRIPTVERBOSE	Contains the level of verbosity to be used during script execution.
VERSION_MAJ, VERSION_MIN	Initialized with the version number of the firmware.

APPRAMBASE

This shell variable is loaded with the starting point of the RAM space that is made available to the application. The system at startup automatically loads this variable. Certain facilities within the system use this value as a pointer to memory that is assumed to be accessible. The following is a list of system services that use the variable to accomplish its work:

```
edit
xrcv
cp
```

If your application uses these facilities at runtime, then the application must be mapped somewhere above the APPRAMBASE address so that these other facilities do not overwrite the application space. The value of the shell variable can be modified, and the modified value is then used by these facilities.

ARGC

Argument count. This variable is automatically loaded with the current argument count when a script is run. The count includes argv[0] as an argument.

ARG'N'

Argument content. The shell variables ARG0 through 'N' are automatically loaded with the argument list when a script is run. Note that since the shell variables within the system are global, these ARGN variables still exist after the script terminates.

BOOTROMBASE

This variable contains the address that the system sees as the starting point of the base flash device.

CMDSTAT

This variable is loaded with the status of the previous command within a script. Note that this is only populated by the command if it is within a script. The value is either "PASS" or "FAIL."

CONSOLEBAUD

This variable is used to allow an application to run with the same baudrate that the console is currently running at, plus it can be set in the inittab to override the default console baud rate. At initial startup, the system configures its COM port to some pre-defined baudrate (38400), then after the inittab file is run, the system looks for the presence of the CONSOLEBAUD shell variable. If set, the console baud rate is automatically set to the value stored in CONSOLEBAUD. If not set, then the system sets this variable to the default value.

To change the baud rate of the console, add the following line to inittab

```
Set CONSOLEBAUD baud
```

Where baud is a valid baudrate value.

ENTRYPOINT

This variable is set by the "ld" command. The value corresponds to the entry point address of the application just loaded. If a script uses "ld" to load an application to RAM, then execute "call \$ENTRYPOINT" as a verbose alternative to run the executable.

EXCEPTION_SCRIPT

If set, then the content of this shell variable is assumed to contain a script name that is to be executed when an exception occurs.

EXCEPTION_TYPE

If an exception occurs, this shell variable will contain the type of exception that occurred.

OSBUILT

This variable contains the string created by the concatenation of the following three strings at build time: `__DATE__` “@” `__TIME__`

NO_EXCEPTION_RESTART

If this shell variable is present, then at the time of an exception, the system does not reset. The system remains at the command prompt.

PLATFORM

This is just a verbose description of the system for general use by the application.

PS1

This variable contains the string that the shell uses as the user prompt. If not set, the default prompt is “#:” and this is loaded into the PS1 shell variable. At any time this shell variable can be changed and the prompt used by the command line changes to the content of PS1. (Example: Set PS1 :)

SCRIPT_IGNORE_ERROR

If this variable is present, then a running script does not stop when a command line error is detected.

SCRIPTVERBOSE

This variable, if present, contains the level of verbosity to be used during script execution. Valid values are 0, 1 & 2. 0 is no verbosity, 1 means the command line is echoed, 2 means the command line is echoed before and after shell processing. This shell variable is tested prior to each line of the script execution. If set, then it is used as the verbosity level; if not set, then the default level is used.

Note that this variable can be changed within a script to provide different levels of verbosity at different points in the script.

VERSION_MAJ, VERSION_MIN

These two variables are automatically initialized with the version number of the firmware. The version number is a 2-field, value (X,Y) that contains the major (X) and minor (Y) release numbers.

Shell Command Set

Shell Command Set Summary

Following is a summary of all of the shell commands:

Shell Command	Description
call	Calls an embedded function.
cat	Prints the specified file(s).
cp	Copies the named file to the new named file or RAM address.
date	Set or display the system's date.
dchk	Check the sanity of the files stored in TFS.
defrag	Defragment the TFS.
delay	Puts the processor in a delay loop for the specified time.
df	Return the amount of memory that is still available for use by TFS.
echo	Print the string to the local terminal.
edit	Edit the file or buffer.
exit	Exit a script.
finfo	Load the shell variable with the file information field.
flash	Flash memory operations (info, type, unlock).
fsize	Load the shell variable with the file size.
gosub	Call a subroutine.
goto	Branch to a file tag.
gpio cfg	Configure the corresponding port pin as an output or input.
gpio info	Display corresponding port configuration.
gpio pin	Return the pin state of the corresponding port.
gpio set	General Purpose Interface I/O - Set the corresponding port pin high

	or low.
help	Display help text for a specific command.
history	Displays the command history.
if	Conditional test with branching.
item	Processes a list of strings.
ld	Load the executable binary file from FLASH to RAM.
ls	List the current set of files in the file system.
mem copy	Copy memory from one location in memory to another.
mem dump	Display memory.
mem fill	Fill a memory range with a specified value or pattern.
mem put	Put to memory.
mem srch	Search through memory for a specified value, or block of data.
mem test	Runs walking ones and address-on-address test across the memory range.
plvl	Sets or configures the system's privilege level.
read	Load input data into specified variable.
reset	The reset is as close to a hard reset as can be supported by the firmware.
return	Return from subroutine.
rm	Remove the specified file(s).
rs232 cfg	Configure the specified port (COM1, GPS, CELL).
rs232 cts	Retrieve the state of the CTS line of the specified port.
rs232 dsr	Retrieve the state of the DSR line of the specified port.
rs232 dtr	Set the DTR line high or low for the specified port.
rs232 rts	Set the RTS line high or low for the specified port.
run	Run the specified file.
set	Set, clear, or adjust a shell variable.
sleep	Puts the CT6xx in a low power state for specified time.
suspend	Halts or suspends the system.
sysinfo	Displays system information.
time	Display or set the system's time.
tfs add	Append a file to TFS.
tfs init	Initialize the file system.
tfs ramdev	Create a temporary RAM-based TFS.
tfs stat [dev]	Dump the current state of TFS.
version	Displays the date/time at which the firmware was built.

watchdog disable	Disable the watchdog.
watchdog enable	Enable the watchdog.
watchdog timeout	Set the watchdog timeout in tenths of a second.
watchdog test	Enables the watchdog and stops petting the watchdog. Waits until the watchdog bites and then resets the system.
xrcv	Xmodem receive (Download) – the downloaded data is transferred to a file in the TFS.
xsnd	Xmodem send (Upload) – uploads a file or block of data.

Shell Commands

call

Calls the embedded function.

USAGE:

```
call [-aqv] address [arg1 [arg2] ...]
```

DESCRIPTION:

Allows the user to execute a function at a raw memory address. A maximum of 7 arguments can be specified. By default the function is called with the arguments converted to hex.

For example `call 0x12345 45 99` would be used to interface to a function located at address 0x12345 whose prototype is `func(int val1, int val2)`. The `-a` option allows `call` to work with functions whose prototype is `(int argc, char **argv)`. In this case, the `-a` option tells `call` to build an argument list and count that is then passed to the function.

OPTIONS:

-a	Build (argc,argv) then call function.
-q	Quiet mode.
-v {var}	Put return val in varname.

cat

Print the specified file(s).

USAGE:

```
cat [-mx] fname [fname ...]
```

DESCRIPTION:

Prints the specified file. Assumes the file is ASCII. If the `-m` option is used, 4 more lines of the file gets displayed with each press of the spacebar.

OPTIONS:

-m	Enable 'more'. Permits the user to display longer listings one page at a time. This switch is useful when displaying a long file or a command that scrolls the display too fast.
-x	Set exit flag if an error occurs.

cp

Copy the named file to the new named file or RAM address.

USAGE:

```
cp [-fix] fname {newfname | hexaddr}
```

DESCRIPTION:

Copies the named file to the new named file. If the destination begins with '0x,' then it is assumed to be a hex address pointing to RAM. The source file is first copied to memory pointed to by APPRAMBASE, then the new file is created from the data in RAM.

OPTIONS:

-f	Flags (see below).
-i	Information.
-x	Set exit flag if an error occurs.

Flags:

x	Executable binary file.
s	Executable script file.
b	Run at boot.
q	Query run at boot.
0-3	Privilege (plvl) level 0-3
u	File is not readable if privilege level requirement is not met; else it is read-only.

EXAMPLE:

```
cp -vxq hello helloworld
```

Copies hello to helloworld and makes it a boot-time executable with query.

date

Sets or displays the system's date.

USAGE:

```
date [-v] [mm[-dd][-[cc]yy]]
```

DESCRIPTION:

Sets or displays the system's date. If no date parameter is passed, the current date kept by the system is displayed.

mm	The month of the year (1 = Jan).
dd	The day of the month (1 – 31).
cc	The century (i.e. 19 for 1900, 20 for 2000).
yy	The year of the century.

OPTIONS:

-v {varname}	Create a variable 'varname' with output.
--------------	--

EXAMPLE:

```
date 06-02-2017
```

Sets the date to June 2, 2017.

```
Date (with no parameters passed)
```

```
Returns: Fri 06/02/2017
```

dchk

Disk check. Checks the sanity of the files stored in TFS.

USAGE:

```
dchk [-vx] [varname]
```

DESCRIPTION:

Checks the sanity of the files stored in TFS by running various tests. If a variable name is specified, then that shell variable is loaded with the string "PASS" or "FAIL" based on the result of the FS check.

OPTIONS:

-v	Enable verbosity.
-x	Set exit flag if an error occurs.

EXAMPLE:

```
dchk
tf . . . ok
inittab . . . ok
helloworld . . . ok
```

defrag

Defragment the tiny file system (TFS).

USAGE:

```
defrag [-vx] [dev]
```

DESCRIPTION:

Defragments the file system, in the device 'dev,' to free-up space. The default device is //FLASH//.

OPTIONS:

-v	Enable verbosity.
-x	Set exit flag if an error occurs.

EXAMPLE:

```
defrag -v
    TFS device '//FLASH/' defrag . . .
    . . .
    . . .
    . . .
    (equal to defrag -v //FLASH/)
```

df

Return the amount of memory that is still available for use by the tiny file system (TFS).

USAGE:

```
df [dev] [varname]
```

DESCRIPTION:

Returns (or stores in 'varname') the amount of flash memory that is still available for use by the TFS. The 'dev' argument specifies the device to list the memory that is available.

Note that since there is per-file overhead, the value returned here is the amount of data space available if one more file is stored in TFS; if additional files are to be stored, then the user must take into account the TFS overhead.

EXAMPLE:

```
df //FLASH/
    Oxed5565(15553893) bytes available to TFS //FLASH/
```

delay

Delay for a specified number of seconds (or milliseconds).

USAGE:

```
delay [-m] count
```

DESCRIPTION:

Puts the processor in a delay loop for a specified amount of time. If no options or arguments are specified, sleep returns the current loops-per-second count.

OPTIONS:

-m	Delay time is in milliseconds.
----	--------------------------------

echo

Print a string to the local terminal.

USAGE:

```
echo [arg1] ... [argn]
```

DESCRIPTION:

Prints its arguments (separated by blanks and terminated by new line) to the local terminal connection. If there are no arguments, a blank line is printed. The following backslash characters are accepted:

\b	Backspace.
\c	No new line.
\n	New line.
\r	Carriage return.
\t	Tab.
\x##	ASCII-coded hex.
\\	Backs.

EXAMPLE:

```
echo \n \r \t Text
      Text
```

edit

Edit a file or buffer.

USAGE:

```
edit [-bcfimrst] [filename]
```

DESCRIPTION:

Allows the user to edit ASCII files that are stored in TFS. It is a simple line-based file editor that supports line insertion, deletion, display and search. If the file already exists in TFS, then the content of that file is copied to a buffer in RAM space and all interaction and modification with the content of the file is done in the buffer. It is not until the 'q' (quit) command is issued, that the file is written to flash. If at any point during the edit session, the 'x' (exit) command is issued, then there is no change to the original file.

OPTIONS:

-b {addr}	Specifies the buffer base address that edit is to use for temporary storage of the file while being edited; if not specified, then edit assumes it owns all RAM on the board and the buffer starts at the address specified in the APPRAMBASE shell variable.
-c {cmd}	In-line command executed prior to entering interactive mode.
-f {flags}	Flags that are applied to the newly created file (See TFS for flag description).
-i {info}	Information field applied to the newly created file.
-m {size}	Specifies allocation size of the buffer to use for temporary storage.
-r	Edits defaults to automatically remove carriage returns from the file it is editing, this shuts off that automatic removal.
-s {size}	Size of the buffer to use for temporary storage.
-t	Converts tabs to spaces.

NOTES:

A typical usage of the edit command will put the user in an interactive mode that supports a basic set of editing commands. At the startup of the interactive mode, edit displays the address of the buffer it will be using for temporary storage, followed by the message "type ? for help". Following is a list of commands supported for interactive mode:

d{LRNG}	Delete line specified by "LRNG" (see below).
e#	Edit line # (uses the same line editor as is used by the command line editor).
i	Begin "insert" mode (use '.' to exit insert mode).
a	Begin "append" mode (use '.' to exit append mode).
P[LRNG]	Print entire buffer with line numbers prefixed.
p	Print entire buffer.
q[fname]	Quit edit, write file (with no fname specified) it writes to the file originally opened.
s[srchstr]	Go to the next line that contains "srchstr."
x	Exit edit, do not write file.
#	Go to line # (use '\$' to go to last line).
+/-#	Go to line relative to current position.

Where

represents a decimal number;

LRNG represents a line number or inclusive line range (# or #-#);

Flags:

x	Executable binary file.
s	Executable script file.
b	Run at boot.
q	Query run at boot.
0-3	Privilege (plvl) level 0-3
u	File is not readable if privilege level requirement is not met; else it is read-only.

exit

Exit a script.

USAGE:

```
exit [-r]
```

DESCRIPTION:

Provides a clean and simple way to terminate a script from anywhere within the script. If the -r option is specified, then the script is automatically deleted after exit is complete and the file has been closed under TFS.

OPTIONS:

-r	Delete script after exit.
----	---------------------------

finfo

Displays the file information field.

USAGE:

```
finfo fname [varname]
```

DESCRIPTION:

Displays the file information field. If the shell variable is specified ('varname'), load it with the information field stored with the file specified by 'fname'.

flash

Flash memory operations.

USAGE:

```
flash {operation} [args]
```

DESCRIPTION:

This command is used to obtain information about the flash device, and to unlock the firmware sectors so that the firmware can be updated.

The commands supported are:

info	Queries each bank configured in the hardware and displays the state that the system is assuming valid for the flash on board.
type	Displays the flash part and device ID.
unlock	This command unlocks the specified sector so that future writes will be legal.

fsize

Displays the file size.

USAGE:

```
fsize fname [varname]
```

DESCRIPTION:

Displays the size of the file 'fname', if the shell variable is specified (varname), then load the variable with the file size.

gosub

Call a subroutine.

USAGE:

```
gosub tagname
```

DESCRIPTION:

Branches to the named tag, and assumes that at some point the code branched to will execute a return. At that point, the script continues execution on the line after the gosub line.

goto

Branch to a file tag.

USAGE:

```
goto tagname
```

DESCRIPTION:

Allows the script to branch to specific tags within the script. The tag is simply a line starting with a # (pound sign), one blank and a tag. For example, # *TAG* on a line by itself is a target that could be branched to by the command *goto TAG*.

gpio cfg (General Purpose IO Interface)

USAGE:

```
gpio cfg {A|B|C|D}[bit#] [{o[utput]|i[nput]}]
```

DESCRIPTION:

Configures the corresponding port pin as an output or input. In order to be able to use this command, a working knowledge of the hardware and schematic diagrams are required. Typically this command is used when developing the hardware.

gpio info (General Purpose IO Interface)

USAGE:

```
gpio info [{A|B|C|D}]
```

DESCRIPTION:

Displays the corresponding port configuration. In order to be able to use this command, a working knowledge of the hardware and schematic diagrams are required. Typically this command is used when developing the hardware.

gpio pin (General Purpose IO Interface)

USAGE:

```
gpio pin {A|B|C|D}[bit#]
```

DESCRIPTION:

Returns the pin state of the corresponding port. In order to be able to use this command, a working knowledge of the hardware and schematic diagrams are required. Typically this command is used when developing the hardware.

gpio set (General Purpose IO Interface)

USAGE:

```
gpio set {A|B|C|D}[bit#] [{1|0}]
```

DESCRIPTION:

Sets the corresponding port pin high or low. In order to be able to use this command, a working knowledge of the hardware and schematic diagrams are required. Typically this command is used when developing the hardware.

help

Displays the command set.

USAGE:

```
help [-d] [cmdname]
```

DESCRIPTION:

Displays help text for a specific command (requires command name to be specified) or display a tabular listing of all commands available.

OPTIONS:

-d	When displaying all commands available, a per-command description is included.
----	--

Example:

```
>:help
Command Set:
call      cat      cp      date      dchk      defrag
df        echo      edit    exit      finfo     flash
fsize     gosub    goto    gpio      help      history
if        item     ld      ls        mem       read
reset     return   rm      rs232    rs485    run
set       sleep    time    tfs      sysinfo   plvl
version   watchdog xrcv    xsnd
```

history

Displays the command history.

USAGE:

```
history
```

DESCRIPTION:

Dumps the last 15 executed commands. Previously issued commands are accessible from the command line with the UP-ARROW to step back through the command history and the DOWN-ARROW to step forward through the command history.

if

Conditional test with branching.

USAGE:

```
if [-ptv] [arg1 compar arg2] action [else action]
```

DESCRIPTION:

Allows the user to build conditional tests that result in branches to different points within an executable script. The most common use of this is to make a string or numerical comparison between two different arguments. If the -t option is specified, then that overrides the basic comparison and the argument to -t is considered to be the test.

Numerical comparisons:

gt	True if arg1 is greater than arg2.
lt	True if arg1 is less than arg2.
le	True if arg1 is less than or equal to arg2.
ge	True if arg1 is greater than or equal to arg2.
eq	True if arg1 is equal to arg2.
ne	True if arg1 is not equal to arg2.

Logical comparisons:

and	True if arg1 AND arg2 is non-zero.
or	True if arg1 OR arg2 is non-zero.
xor	True if arg1 XOR arg2 is non-zero.

String comparisons:

seq	True if the string of arg1 is identical to the string of arg2.
sne	True if the string of arg1 is not identical to the string of arg2.
sin	True if the string of arg1 is within the string of arg2.

Actions:

goto tag	Jump to the location in the script specified by "tag".
gsub tag	Call the subroutine specified by "tag".
return	Return from the currently active subroutine.
exit	Terminate the currently active script.

OPTIONS:

-p pname	Specify the port name for option -t. The console is the default port if this option is not used.
-t testtype	Override the default "arg1 compare arg2" comparison. testtypes: gc "gotchar"... a character is present on the UART. ngc "not-gotchar"... a character is not present on the UART.
-v	Verbose mode. Simply prints "TRUE" or "FALSE" after the test or comparison.

item

Process a list of strings.

USAGE:

```
item idx stor_var [[item1] [item2 [...]]]
```

DESCRIPTION:

Allows the user to build scripts that conveniently process a list of strings (or items) – in conjunction with “if” and “goto.”

Note: idx=1 retrieves item1.

EXAMPLE:

```
item 3 letter a b c d e f
```

This would place 'c' in the shell variable "letter".

ld

Loads the executable binary file from FLASH to RAM.

USAGE:

```
ld [-Vvx] fname
```

DESCRIPTION:

Loads the executable binary file from FLASH to RAM. If verbosity is set to something greater than 1 (-vvv), then the load does not touch memory, it only displays what it would load to memory.

The verify option verifies that the executable binary image in flash space matches what is in RAM space.

OPTIONS:

-V	Verify.
-v	Enable verbosity.
-x	Set exit flag if an error occurs.

ls

Lists the current set of files in the file system.

USAGE:

```
ls [-almx] [[filter] [filter...]]
```

DESCRIPTION:

Lists the current set of files in the file system.

Specifying the filter can limit the number of files listed...

**filter* indicates a suffix match

*filter** indicates a prefix match

filter indicates a full filename match

OPTIONS:

-a	Display 'hidden' file.
-l	Display files in long format.
-m	Enable more. Works like the "more" DOD command. Permits the user to display longer listings one page at a time. This switch is useful when displaying a long file or a command that scrolls the display too fast.
-x	Set exit flag if error occurs.

mem copy

Copies memory.

USAGE:

```
mem copy [-24fv] src dst bytecnt
```

DESCRIPTION:

Copies memory from one location (source – src) in memory space to another (destination – dst). The size of the copy is specified by the count, which is always considered a byte-count (bytecnt).

OPTIONS:

-2	Short access.
-4	Long access.
-f	FIFO mode.
-v	Verify only.

mem dump

Displays memory.

USAGE:

```
mem dump [-24bdf1msv] addr [bytecnt]
```

DESCRIPTION:

Displays memory. This command attempts to support all modes of memory access through different options. Width can be specified for 8/16/32 bit access; the exact number of accesses can also be specified.

OPTIONS:

-2	Short access.
-4	Long access.
-b	Binary.
-d	Decimal.
-f	FIFO mode.
-l#	Size of line (in bytes).
-m	Enables 'more'. Permits the user to display longer listings one page at a time. This switch is useful when displaying a long file or a command that scrolls the display too fast.
-s	String.
-v {var}	Quietly load 'var' with element at addr.

mem fill

Fills memory.

USAGE:

```
mem fill [-24cinp] start {finish|bytecnt} {value|
pattern}
```

DESCRIPTION:

Fills a memory range with a specified value or pattern.

OPTIONS:

-2	Short access.
-4	Long access.
-c	arg2 is count (in bytes).
-i	Increment {value pattern}.
-n	No verification.
-p	arg3 is a pattern.

mem put

Puts to memory.

USAGE:

```
mem put [-24fsS] addr {val|string} [val] ...
```

DESCRIPTION:

Puts to memory

OPTIONS:

-2	Short access.
-4	Long access.
-f	FIFO mode.
-s	Stcpy.
-S	Streat.

mem srch

Searches memory.

USAGE:

```
mem srch [-24cnqsx] start {finish|bytecnt} srchfor
```

DESCRIPTION:

Searches through memory for a specified value, or block of data. This value can be a byte, short, long value or an ASCII-coded hex string or straight ASCII string.

OPTIONS:

-2	Short access.
-4	Long access.
-c	Arg2 is count (in bytes).
-n	Srchfor not (NA for -s or -x).
-q	Quit after first search hit.
-s	String srch.
-x	Hexblock srch.

mem test

Tests memory.

USAGE:

```
mem test [-CcqSstv] addr len
```

DESCRIPTION:

Runs walking ones and address-on-address test across the memory range specified.

OPTIONS:

-c	Continuous.
-C	Crc32 calculation.
-q	Quit on error.
-S	Determine size of on-board memory (see note below).
-s##	Sleep ## seconds between adr-in-addr write and readback.
-t##	Toggle data on '##'-bit boundary (##: 32 or 64).
-v	Cumulative verbosity... -v=<ticker>, -vv=<ticker + msg-per-error>

DYACON, Inc.

Memory test is walking ones followed by address-in-address.

plvl

Displays or modifies the current privilege level.

USAGE:

```
plvl [-chp] [new_level|min|max] [password]
```

DESCRIPTION:

Set or configures the system's privilege level. The privilege level determines what commands it can execute and what files are accessible. This command is hard coded to require only privilege-level 0 to execute.

The system supports 4 privilege levels (0 through 3), with 3 being the highest privilege level (i.e. Superuser or Administrator). The files in TFS and the shell commands can be configured to require a certain privilege level to gain access to a file or the ability to execute the command.

At startup, the target's default privilege level is at its highest (3) and all commands and files default to require privilege level 0 for access; hence, everything is accessible.

If the running privilege level is lowered (by an auto bootable script for example), then all accesses made to the hardware through the command line after that will be limited to the facilities (commands and files) that are available to the new privilege level. Commands and files can be configured to be accessible by some minimum privilege level. At system startup, all commands default to require that the system be at privilege level 0 or higher to execute (in other words, all commands can be executed). If commands are to be restricted to different privilege levels, then the *ulvl* command should be used in the inittab file to make all the necessary adjustments.

This feature basically provides a mechanism to protect some portion of the system from unauthorized users. The privilege level can be raised, but only by a user that knows the password to get to that particular privilege level. The passwords are encrypted and stored in a file in TFS that is automatically saved at the highest privilege level. The file is also unreadable by privilege levels lower than the max.

OPTIONS:

-c {cmd,lvl}	Set command's privilege level.
-h	Dump system header.
-p	Build new password file.

Note: cmd==ALL, applies action to all commands.

read

Load input data into the specified variable.

USAGE:

```
read [-ft] [port] [var1] [var2] ...
```

DESCRIPTION:

This command provides the ability of obtaining data from a serial port and loading a variable with the input data. If no 'port' is specified the console is used.

OPTIONS:

-f	Flush port input queue.
-t ###	Wait for input, but timeout after ### milliseconds. Timeout is restarted after each character is received.
-T ###	Wait for input, but timeout after ### milliseconds, timeout is cumulative (doesn't restart after each character).

reset

Resets the firmware.

USAGE:

```
reset [-x]
```

DESCRIPTION:

Resets the firmware. The reset is as close to a hard reset as can be supported by the firmware.

OPTIONS:

-x	Restart the firmware just as it would be restarted if an application had exited.
----	--

return

Returns from a subroutine.

USAGE:

```
return
```

DESCRIPTION:

Returns from a subroutine. There are no arguments or options associated with this command.

rm

Removes the specified file(s).

USAGE:

```
rm [-x] filter [filter ...]
```

DESCRIPTION:

Removes the specified file(s).

OPTIONS:

-x	Sets the exit flag if an error occurs.
----	--

rs232 cfg

Configures the RS-232 port specified.

USAGE:

```
rs232 cfg {com1 | gps | cell | com2} [baud parity wrdlen
stopbits]
```

DESCRIPTION:

Configures the specified port. Valid ports include COM1, the optional GPS port, or the optional cell phone port.

OPTIONS:

baud	Transfer speed (baud rate). Valid baud rates include: 1200, 2400, 4800, 9600, 38400, 57600, and 115200.
parity	O = Odd, E = Even, N = None.
wrdlen	Data size (7 or 8 bits).
stopbits	Number of stop bits (1 or 2).

rs232 cts

Retrieves the state of the specified RS-232 CTS line.

USAGE:

```
rs232 cts {com1 | gps | cell | com2}
```

DESCRIPTION:

Retrieves the state of the CTS line of specified port. Valid ports include COM1, the optional GPS port, or the optional cell phone port.

1	CTS line is set high.
0	CTS line is set low.

rs232 dsr

Retrieves the state of the specified RS-232 DSR line.

USAGE:

```
rs232 dsr {com1 | gps | cell | com2}
```

DESCRIPTION:

Retrieves the state of the DSR line of specified port. Valid ports include COM1, the optional GPS port, or the optional cell phone port.

1	DSR line is set high.
0	DSR line is set low.

rs232 dtr

Sets the specified RS-232 DTR line high or low.

USAGE:

```
rs232 dtr {com1 | gps | cell | com2} [1 | 0]
```

DESCRIPTION:

Sets the DTR line high or low for specified port. Valid ports include COM1, the optional GPS port, or the optional cell phone port.

OPTIONS:

1	Set the DTR line high.
0	Set the DTR line low.

rs232 rts

Sets the specified RS-232 RTS line high or low.

USAGE:

```
rs232 rts {com1 | gps | cell | com2} [1 | 0]
```

DESCRIPTION:

Sets the RTS line high or low for specified port. Valid ports include COM1, the optional GPS port, or the optional cell phone port.

OPTIONS:

1	Set the RTS line high.
0	Set the RTS line set low.

run

Runs the specified file.

USAGE:

```
run [-vx] fname
```

DESCRIPTION:

Runs the specified file based on the creation attributes. The file runs as either a script or an executable image. In the case of scripts, the `-v` option causes the TFS script runner to print the line of the script prefixed by the line number within the script.

OPTIONS:

-v	Enable verbosity.
-x	Set the exit flag if an error occurs.

set

Sets, clears or adjusts a shell variable.

USAGE:

```
set [-abcdefiox] [varname[=expression]] [value]
```

DESCRIPTION:

Sets shell variables that can then be used by other commands or by an application through the OBC605 *getenv()* call.

OPTIONS:

-a	Logically AND the content of the shell variable with the value specified.
-b	Set the console baudrate.
-c	Clear the environment.
-d	Decrease var by value (or 1 if value not specified).
-e	Build an environ string.
-f {file}	Create a script that recreates a subset of the current environment.
-i	Increase var by value (or 1 if value not specified).
-o	Logically OR the content of var with value.
-x	Output the result of the -I or -d option in hex (else decimal).

sleep

Delays for a specified number of seconds (or milliseconds).

USAGE:

```
sleep [-m] sec
```

DESCRIPTION:

With a call to sleep the current program is suspended from execution for the number of seconds specified by sec. The command uses the internal RTC alarm function (preserves any previous settings) to time the sleep. All interrupts are disabled during the sleep period except for the RTC alarm.

OPTIONS:

-m	Delay time is in milliseconds.
----	--------------------------------

suspend

Puts the system in a low power mode and halts it.

USAGE:

suspend

DESCRIPTION:

Puts the system in a low power mode and halts it. Activity in COM1 will make the system resume execution. In suspend mode all interrupts except the RTC day interrupt and COM1 interrupts are disabled

sysinfo

Displays system information.

USAGE:

sysinfo

DESCRIPTION:

Displays firmware version, platform, processor name, firmware built date, application start RAM address, and the API entry address. There are no arguments or options associated with this command.

time

Displays or sets the system's time.

USAGE:

time [-v] [hh[:mm[:ss]] [{A|P}]]

DESCRIPTION:

Displays the system time if no parameters are passed to it. If the command has parameters, it assumes that it is a new time and it updates the clock.

hh	Hour (0-23; 0=midnight) or (1-12).
mm	Minutes (0-59).
ss	Seconds (0-59).
A P	A = AM and P = PM. Used when specified hours are for a 12 hour clock.

OPTIONS:

-v {varname}	Create a variable named after 'varname' and put the output in it.
--------------	---

tfs add

Appends a file to TFS.

USAGE:

```
tfs add [-fix] fname srcaddr size
```

DESCRIPTION:

Creates the file named 'fname' to contain the data starting at location "srcaddr" of size "size". Options -f and -i can be used to specify the flags and information field associated with the newly created file.

OPTIONS:

-f	Flags.
-i	Info.
-x	Set the exit flag if an error occurs.

tfs init

Initializes the file system.

USAGE:

```
tfs init [-x] dev
```

DESCRIPTION:

Initializes the file system (remove all files and erase flash). The 'dev' parameter identifies the TFS device.

dev = //FLASH/	TFS in FLASH.
----------------	---------------

OPTIONS:

-x	Set the exit flag if an error occurs.
----	---------------------------------------

tfs ramdev

Creates a temporary ram-based TFS.

USAGE:

```
tfs ramdev [-x] name startaddr size
```

DESCRIPTION:

Creates a temporary ram-based TFS device occupying the ram space designated by 'start addr' and 'size'. The name of the new device (specified by 'name') is automatically wrapped with two leading slashes and one post slash.

OPTIONS:

-x	Set the exit flag if an error occurs.
----	---------------------------------------

tfs stat [dev]

Dumps the current state of the TFS.

USAGE:

```
tfs stat [dev]
```

DESCRIPTION:

Dumps the current state of TFS. The 'dev' parameter identifies the TFS device, by default dev = //FLASH/.

version

Displays version information.

USAGE:

```
version
```

DESCRIPTION:

Displays the date and time when the firmware was built. There are no arguments or options associated with this command.

watchdog disable

USAGE:

```
watchdog disable
```

DESCRIPTION:

Disables the watchdog. There are no arguments or options associated with this command.

watchdog enable

USAGE:

```
watchdog enable
```

DESCRIPTION:

Enables the watchdog. There are no arguments or options associated with this command.

watchdog timeout

USAGE:

```
watchdog timeout [seconds]  
(resolution 1 sec, valid range 1-63)
```

DESCRIPTION:

Sets the watchdog timeout in second intervals.

watchdog test

USAGE:

```
watchdog test
```

DESCRIPTION:

Enables the watchdog (if not previously enabled) and stops petting the watchdog. It waits until the watchdog bites and then resets the system.

xrcv

Xmodem receive (Download from the Host to the CT6xx).

USAGE:

```
xrcv [-acfikU] [fname [size]]
```

DESCRIPTION:

Places data in system RAM when downloading. Upon completion of the download, the downloaded data is transferred to a file in TFS. By default, the RAM address used is the value established by the system at boot time known as APPRAMBASE.

OPTIONS:

-a{#}	Address (overrides the default of APPRAMBASE).
-c	Use CRC (default = checksum).
-f{flags}	File flags.
-i{info}	File information.
-k	1K Xmodem.
-U	Firmware upgrade.

xsnd

Enables Xmodem send (Upload from the OBC605 to the host).

USAGE:

```
xsnd [-ck] fname
```

DESCRIPTION:

Uploads a file or block of data. If the upload is a file then the -F option must also be specified. If upload of raw data, then address and size must be specified on the command line.

OPTIONS:

-c	Use CRC (default = checksum).
-k	Force packet length to 1K.

OBC605 API

API Summary

Following is a summary of the API commands:

Command	Description
appexit ()	Allows the application to return control to the system.
cell_enable ()	Enables or disables the optional cell phone module.
cell_power ()	Turns on and off power to the option cell phone.
com2_autorts ()	Sets the RTS line under the control of the system or application.
com2_cfgcallback ()	Configures a callback for the COM2 serial port.
com2_close ()	Closes the COM2 serial port.
com2_cts ()	Retrieves the state of the CTS line of the COM2 serial port.
com2_getcfg ()	Retrieves the COM2 serial port configuration.
com2_open ()	Opens the COM2 serial port.
com2_read ()	Reads a block of data from the input queue of the COM2 serial port.
com2_rxavail ()	Retrieves the number of bytes available from reading COM2 port.
com2_rxpurge ()	Removes all available characters from the input queue.
com2_setcfg ()	Configures the COM2 serial port.
com2_setrts ()	Sets the state of the request to send line.
com2_txfree ()	Retrieves the space available in the output buffer of COM2 port.
com2_txpurge ()	Stops transmission (if any) and clears the output queue of COM2 port.
com2_write ()	Writes a block of data to the output queue of the COM2 serial port.
dcgood_cfgcallback ()	Configures a callback for the DC good signal.
dcgood_wait ()	Waits for DC good to become good again.
delay ()	Delays for a specified time (millisecond-resolution delay loop).

<code>dgtlin_cfg ()</code>	Configures the mode and a callback for the digital input.
<code>dgtlin_get ()</code>	Retrieves the state of the digital input.
<code>dgtlout_set ()</code>	Sets the state of the digital output.
<code>docommand ()</code>	Acts similarly to the “system” function in DOS and/or Unix.
<code>getargv ()</code>	Retrieves an argument list.
<code>getenv ()</code>	Acts similarly to standard <code>getenv()</code> . Retrieves the value that corresponds to a specified shell variable name.
<code>getenvp ()</code>	Retrieves a pointer to a string that contains one white space (newline) delimited “name-value” pair for each shell variable currently defined in the system.
<code>gps_enable ()</code>	Enables or disables the GPS module.
<code>ints_disable ()</code>	Allows the application to turn off the interrupts.
<code>ints_enable ()</code>	Allows the application to turn on interrupts.
<code>ints_restore ()</code>	Allows the application to restore interrupt state.
<code>irq_enable ()</code>	Enables or disables the specified IRQ interrupt.
<code>irq_gethandler ()</code>	Obtains the current interrupt handler function for the specified IRQ.
<code>irq_sethandler ()</code>	Registers an interrupt handler function for the specified IRQ.
<code>led_set ()</code>	Sets the state of the LEDs.
<code>printmem ()</code>	Prints a block of memory.
<code>reset ()</code>	Allows the application to reset the system.
<code>rs232_break ()</code>	Generates a break signal on the line of the specified serial port.
<code>rs232_brkdetect ()</code>	Retrieves the state of the break detect flag of the specified serial port.
<code>rs232_cd ()</code>	Retrieves the state of the CD line of the specified serial port.
<code>rs232_cfgcallback ()</code>	Configures a callback for one of the serial ports.
<code>rs232_close ()</code>	Closes one of the OBC605’s serial ports.
<code>rs232_cts ()</code>	Retrieves the state of the CTS line of the specified serial port.
<code>rs232_dsr ()</code>	Retrieves the state of the DSR line of the specified serial port.
<code>rs232_dtr ()</code>	Sets the state of the DTR line of the specified serial port.
<code>rs232_frmerror ()</code>	Retrieves the state of the framing error flag of the specified serial port.
<code>rs232_getc ()</code>	Retrieves one character from the specified serial port.
<code>rs232_getcfg ()</code>	Retrieves serial port configuration.
<code>rs232_open ()</code>	Opens one of the OBC605’s serial ports.
<code>rs232_ovrerror ()</code>	Retrieves the state of the overrun error flag of the specified serial port.
<code>rs232_prttyerror ()</code>	Retrieves the state of the parity error flag of the specified serial port.
<code>rs232_putc ()</code>	Puts one character in the queue of the specified serial port.
<code>rs232_read ()</code>	Reads a block of data from the input queue of the specified serial port.

rs232_ri ()	Retrieves the state of the RI line of the specified serial port.
rs232_rts ()	Sets the state of the RTS line of the specified serial port.
rs232_rxavail ()	Retrieves the number of bytes available for reading from a serial port.
rs232_rxpurge ()	Removes all available characters from the input queue.
rs232_setcfg ()	Configures a serial port.
rs232_txfree ()	Retrieves the space available in the output buffer of a serial port.
rs232_txpurge ()	Stops transmission (if any) and clears the output queue.
rs232_write ()	Writes a block of data to the output queue of the specified serial port.
rs485_autotxline ()	Sets the transmit control line of the RS-485 serial port under the control of the device driver.
rs485_cfgcallback ()	Configures a callback for the RS-485 serial port.
rs485_close ()	Closes the RS-485 serial port.
rs485_cts ()	Retrieves the state of the CTS line of the RS-485 serial port.
rs485_getcfg ()	Retrieves the RS-485 serial port configuration.
rs485_open ()	Opens the RS-485 serial port.
rs485_read ()	Reads a block of data from the input queue of the RS-485 serial port.
rs485_rxavail ()	Retrieves the number of bytes available for reading from the RS-485 serial port.
rs485_rxpurge ()	Removes all available characters from the input queue.
rs485_setcfg ()	Configures the RS-485 serial port.
rs485_txfree ()	Retrieves the space available in the output buffer of the RS-485 serial port.
rs485_txline ()	Sets the state of the transmit control line of the RS-485 serial port.
rs485_txpurge ()	Stops transmission (if any) and clears the output queue.
rs485_write ()	Writes a block of data to the output queue of the RS-485 serial port.
rtc_getalarm ()	Retrieves the real time clock alarm settings.
rtc_getdate ()	Retrieves the system date.
rtc_gettime ()	Retrieves the system time.
rtc_setalarm ()	Sets the real time clock alarm.
rtc_setdate ()	Sets the system date.
rtc_settime ()	Sets the system time.
rtc_timestamp ()	Obtains a time stamp from the system.
setenv ()	Establishes shell variables at the command line using the set command.
sleep ()	Puts the CT6xx in sleep mode.
suspend ()	Puts the CT6xx in suspend mode.
sys_free ()	Acts similarly to the standard free() command. Used to release memory.

sys_getbytes ()	Retrieves some number of bytes from the console port.
sys_getchar ()	Provides similar functionality as standard getchar().
sys_getline ()	Retrieves characters from the console.
sys_gotachar ()	Returns status of character presence on the console port.
sys_heap_extend ()	Extends the basic heap that is statically allocated by the system.
sys_malloc ()	Acts similarly to the standard malloc().
sys_printf ()	Console printf, acts similarly to printf() but limited in formatting capability.
sys_putbytes ()	Sends some number of bytes to the console port.
sys_putchar ()	Provides similar functionality as standard putchar().
sys_puts ()	Console puts, provides similar functionality as standard puts().
sys_realloc ()	Reallocates a block of memory from the system's heap.
sys_sprintf ()	Acts similarly to sprintf(), but limited in formatting.
sys_version ()	Retrieves the system's version information.
TFS Return Codes	Lists all TFS return codes.
tfsadd ()	Adds a new file to TFS.
tfsclose ()	Closes a TFS file that had previously been opened.
tfseof ()	Returns EOF (end of file) status on specified file.
tfsfstat ()	Populates a TFILE structure with the designated file's file header structure.
tfsgetline ()	Retrieves the next line from an assumed ASCII file.
tfsinit ()	Initializes the flash space that is used by TFS.
tfsioctl ()	Performs some type of control operation on TFS or a file in TFS.
tfsnext ()	Called to retrieve the "next" file in the TFS list.
tfsopen ()	Opens up a TFS file for read and/or write access.
tfsread ()	Accesses a file that has been previously opened for reading and retrieve data from that flash space.
tfsseek ()	Moves the internal pointer maintained by TFS to some specified position.
tfsstat ()	Returns a TFILE pointer to the file specified.
tfstell ()	Returns the current offset into the file referred to by the incoming descriptor.
tfstruncate ()	Truncates the size of a file that has been opened for append to a new size.
tfsunlink ()	Removes a file from TFS flash space.
tfswrite ()	Accesses a file that has been previously opened for writing and transfer data to TFS for eventual transfer to flash.
tmr_cfgfnct ()	Configures a callback for one of the timers.

DYACON, Inc.

<code>tmr_check ()</code>	Retrieves the timer timeout counter.
<code>tmr_reset ()</code>	Resets the timer timeout to its set timeout.
<code>tmr_setmst ()</code>	Sets the timer timeout in minutes, seconds, and tenths of a second.
<code>tmr_sett ()</code>	Sets the timer timeout in tenths of a second.
<code>tmr_start ()</code>	Starts a timer.
<code>tmr_stop ()</code>	Stops a timer.
<code>watchdog_enable ()</code>	Allows the application to enable or disable the system's watchdog.
<code>watchdog_service ()</code>	Allows the application to tickle the watchdog.
<code>watchdog_timeout ()</code>	Allows the application to set the watchdog's timeout.

OBC605 Application Programmer's Interface

appexit ()

Allows the application to return control to the system.

USAGE:

```
void appexit(int code);
```

DESCRIPTION:

Returns control to the system. This is a CT6xx-based application's equivalent of `exit()`. The exit status prints out when the system regains control. This does not automatically release any resources that may have been allocated by the application (files, heap space, environment variables, etc...).

PARAMETERS:

int code	The value considered to be the exit status. This value is printed by the system when this function is called.
----------	---

RETURN:

Void.

cell_enable ()

Enables or disables the cell phone module.

USAGE:

```
int cell_enable(int enable);
```

DESCRIPTION:

Enables or disables the cell phone module. The module by default is disabled. This function should be called to enable the cell phone module before the application tries to access the module. After the cell_enable function has been executed, the cell_power function must be used to apply power to the cell phone before the application tries to access the module.

Note: Does not control the ON/OFF enable pin on the cell phone.

PARAMETERS:

int enable	0 = Disable the cell phone module. 1 = Enable the cell phone module. -1 = Reads the current state of the module.
------------	--

RETURN:

Returns the module state.

cell_power ()

Turns on or off power to the cell phone.

USAGE:

```
int cell_power(int power);
```

DESCRIPTION:

Allows the application to apply power to the cell phone module. The module by default is disabled. This function should be called after the cell_enable. This function controls the ON/OFF enable pin on the cell phone.

PARAMETERS:

int power	0 = This command turns power off to the module. 1 = This command turns power on to the module.
-----------	---

RETURN:

Returns the module state.

com2_autorts ()

Sets the RTS line under the control of the system or application.

USAGE:

```
int com2_autorts(bool_t ctrlauto);
```

DESCRIPTION:

Tells the system to control the RTS line or tells the system that the RTS line will be controlled by the application. Consult Dyacon for availability of the RTS line on com2.

PARAMETERS:

bool_t ctrlauto	0 = Under application control. 1 = Under system control.
-----------------	---

RETURN:

Returns the previous state of the auto-control state.

com2_cfgcallback ()

Configures a callback for the COM2 serial port.

USAGE:

```
void com2_cfgcallback(void (*callback)(int argt,int
argv), uint8_t mask);
```

DESCRIPTION:

Configures a callback for the COM2 serial port.

The callback should be a function that takes two integer arguments and returns void. This function is invoked based on the mask set at the configuration function.

The function is invoked from within an interrupt, because of this, it should be short and to the point to prevent long periods of time with disabled interrupts.

The following table shows the valid values for the mask. They can be OR'd together to have more than one signal activated.

Mask Values	Definition
RS232_RXR	Signal when one or more characters are available for reading.
RS232_TXR	Signal when the last character has been transmitted.
RS232_LSR	Signal when a change of state has occurred in one of the port lines.
RS232_MSR	Signal when a change of state has occurred in one of the modem control lines.

PARAMETERS:

void (*callback)(int,int)	Pointer to function provided by the application.
	The first parameter indicates why the function was called.
	The second parameter can be one of the following:
	If argt= RS232_RXR then argv=characters available.
	If argt = RS232_TXR then argv=0.
	If argt = RS232_LSR then argv=lsr.
	If argt = RS232_MSR then argv= msr.
uint8_t mask	Type of signal(s) to be activated for the callback.

RETURN:

Void.

com2_close ()

Closes the COM2 serial port.

USAGE:

```
void com2_close(void);
```

DESCRIPTION:

Closes the COM2 serial port.

RETURN:

Void.

com2_cts ()

Retrieves the state of the CTS line of the COM2 serial port.

USAGE:

```
int com2_cts(void);
```

DESCRIPTION:

Obtains the state of the CTS line of the COM2 serial port.

RETURN:

Returns the CTS line state (1 = asserted, 0 = de-asserted).

com2_getcfg ()

Retrieves the COM2 serial port configuration.

USAGE:

```
void com2_getcfg(uint32_t *baud, char *parity, uint8_t *dbits, uint8_t *sbits);
```

DESCRIPTION:

Retrieves the configuration of the RS-232 serial port.

PARAMETERS:

uint32_t *baud	Pointer where to store the baud rate value.
char *parity	Pointer where to store the parity value.
uint8_t *dbits	Pointer where to store the word length value.
uint8_t *sbits	Pointer where to store the stop bits value.

RETURN:

Void.

com2_open ()

Opens the COM2 serial port.

USAGE:

```
int com2_open(void);
```

DESCRIPTION:

Activates the COM2 serial port. Normally the application should not call this function unless it has called the com2_close() function first, that is because the COM2 serial port is opened when the application gains control of the system.

RETURN:

Returns com2_OK if successful.

com2_read ()

Reads a block of data from the input queue of the COM2 serial port.

USAGE:

```
int com2_read(void *buf, uint16_t count);
```

DESCRIPTION:

Reads a block of data from a serial port.

PARAMETERS:

void* buff	Pointer to data to be transmitted.
uint16_t count	Number of bytes to transmit.

RETURN:

Returns the number of characters read.

com2_rxavail ()

Retrieves the number of bytes available for reading from the COM2 serial port.

USAGE:

```
int com2_rxavail(void);
```

DESCRIPTION:

Obtains the number of characters stored in the input buffer of the COM2 serial port.

RETURN:

Returns the number of bytes available.

com2_rxpurge ()

Removes all available characters from the input queue.

USAGE:

```
void com2_rxpurge(void);
```

DESCRIPTION:

Clears the input queue of the COM2 serial port.

RETURN:

Void.

com2_setcfg ()

Configures the COM2 serial port.

USAGE:

```
int com2_setcfg(uint32_t baud, char parity, uint8_t dbits, uint8_t sbits);
```

DESCRIPTION:

Configures the COM2 serial port.

PARAMETERS:

uint32_t baud	Baud rate value (Valid baud rates include: 1200, 2400, 4800, 9600, 38400, 57600, and 115200.)
char parity	Parity ('N' = none, 'E' = even, 'O' = odd).
uint8_t dbits	Word length (7 or 8).
uint8_t sbits	Stop bits (1 or 2).

RETURN:

If successful, it returns RS232_OK ; else

RS232_ERROR_BADPARITY

RS232_ERROR_BADWRDSIZE

RS232_ERROR_BADSTOPBITS

RS232_ERROR_PORTNOTFOUND

RS232_ERROR_BADPARAM

com2_setrts ()

Sets the state of the request to send line.

USAGE:

```
int com2_setrts(int setting);
```

DESCRIPTION:

Sets the RTS line high or low.

PARAMETERS:

int setting	0 = set request to send line low. 1 = set request to send line high..
-------------	--

RETURN:

Returns the current state of the RTS line.

com2_txfree ()

Retrieves the space available in the output buffer of the COM2 serial port.

USAGE:

```
int com2_txfree(void);
```

DESCRIPTION:

Obtains the space available in the transmit queue (in bytes) for the COM2 serial port.

RETURN:

Returns the number of bytes available.

com2_txpurge ()

Stops transmission (if any) and clears the output queue of the COM2 serial port.

USAGE:

```
void com2_txpurge(void);
```

DESCRIPTION:

Stops a transmission and clear the output queue of the COM2 serial port.

RETURN:

Void.

com2_write ()

Writes a block of data to the output queue of the COM2 serial port.

USAGE:

```
int com2_write(void *buf, uint16_t count);
```

DESCRIPTION:

Transmits a block of data.

PARAMETERS:

void *buf	Pointer to data to be transmitted.
uint16_t count	Number of bytes to transmit.

RETURN:

Returns the number of characters transferred.

dcgood_cfgcallback ()

Configures a callback for the DC good signal.

USAGE:

```
void dcbgood_cfgcallback(void (*callback)(int));
```

DESCRIPTION:

Configures a callback for the DC good signal.

The callback should be a function that takes an integer argument and return void. This function is invoked when the state of the dc good changes.

The function is invoked from within an interrupt, because of this, it should be short and to the point to prevent long periods of time with disabled interrupts.

PARAMETERS:

void (*callback)(int)	Pointer to function provided by the application.
	The parameter indicates the dc good state. 1 = OK 0 = BAD

RETURN:

Void.

dcgood_wait ()

Waits for DC good to become good again.

USAGE:

```
void dcbgood_wait(void)
```

DESCRIPTION:

Waits for the dc good to become good again.

RETURN:

Void.

delay ()

Delays for a specified period of time (millisecond-resolution delay loop).

USAGE:

```
void delay(int milliseconds);
```

DESCRIPTION:

Provides a delay loop for the specified number of milliseconds.

PARAMETERS:

int milliseconds	Number of milliseconds to delay.
------------------	----------------------------------

RETURN:

Void.

dgtlin_cfg ()

Configures the mode and a callback for the digital input.

USAGE:

```
int dgtlin_cfg(unsigned n, int mode, void (*callback)
(int input));
```

DESCRIPTION:

Configures the mode and a callback for the digital input.

The callback should be a function that takes an integer argument and returns a void. This function is invoked when the input is asserted.

The function is invoked from within an interrupt, because of this, it should be short and to the point to prevent a long period of time with disabled interrupts.

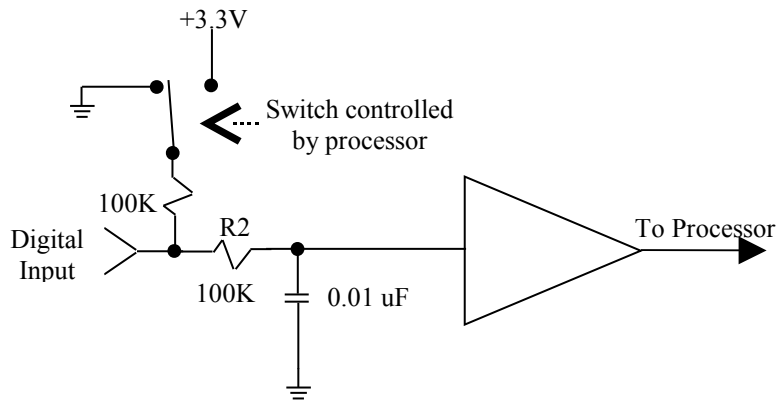
PARAMETERS:

unsigned n	Which digital input (DI_1 or DI_2).
int mode	Mode for the digital input:
	DI_ACTIVE_LOW.
	DI_ACTIVE_HIGH.
void (*callback)(int)	Pointer to function provided by the application.
	The parameter indicates the digital input state. 1 = high 0 = low

RETURN:

Void.

Basic Diagram:



dgtlin_get ()

Retrieves the state of the digital input.

USAGE:

```
int dgtlin_get(unsigned n)
```

DESCRIPTION:

Obtains the state of the digital input.

PARAMETERS:

unsigned n	Which digital input (DI_1 or DI_2).
------------	-------------------------------------

RETURN:

Returns DI_LOW or DI_HIGH if successful; otherwise -1.

dgtlout_set ()

Sets the state of the digital output.

USAGE:

```
int dgtlout_set(unsigned n, int setting);
```

DESCRIPTION:

Sets the state of the digital output.

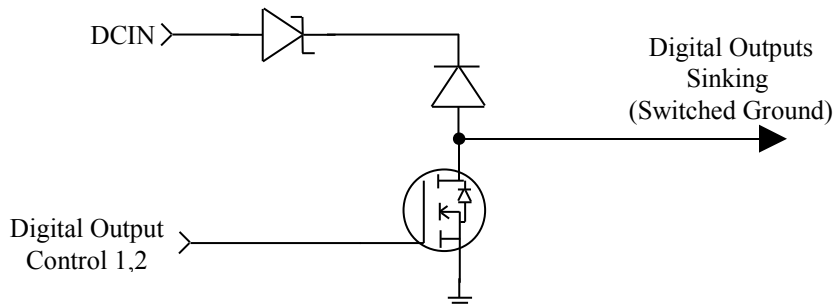
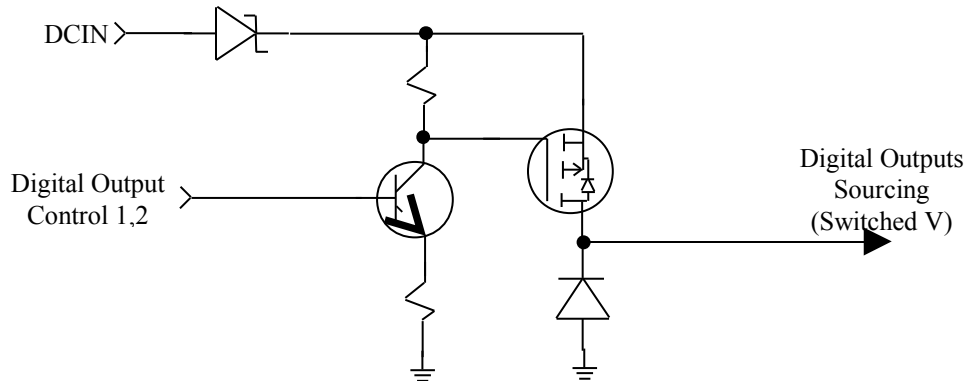
PARAMETERS:

unsigned n	Which digital input (DO_1 or DO_2).	
int setting	State to which the digital output should be set:	
	DO_FLOAT	Digital output floats.
	DO_LOW	Digital output set to low.
	DO_HIGH	Digital output set to high.

RETURN:

Returns if successful; otherwise -1.

Basic Diagrams:



docommand ()

Acts similarly to the “system” function in DOS and/or Unix.

USAGE:

```
int docommand (char *cmdline, int verbose);
```

DESCRIPTION:

Allows the application code to execute commands that are normally executable from the command line interface of the CT6xx.

PARAMETERS:

char *cmdline	String of characters that would be typed as if they were a command entered at the console interface.
int verbose	If non-zero, the CT6xx’s command interpreter prints the command string after processing the line for environment substitutions (shell variables and symbols).

RETURN:

These return values are found in cli.h.

CMD_SUCCESS:	Everything worked ok.
CMD_FAILURE:	Command parameters are valid, but command itself failed for some other reason.
CMD_LINE_ERROR:	Command line itself was invalid. Too many args, invalid shell var syntax, etc.
CMD_PARAM_ERROR:	Command line did not parse properly. There was a syntax error on the command line that did not even allow the command function to get going.
CMD_NOT_FOUND:	Indicates that docommand() could not find the command.

getargv ()

Retrieves an argument list.

USAGE:

```
void getargv(int *argc, char ***argv);
```

DESCRIPTION:

Provides the hook needed by the application to retrieve an argument list that was created previously by command line invocation. The integer pointed to by `argc` is loaded with the argument count and the pointer to a character array is loaded with the location of the `char *argv[]` table.

PARAMETERS:

<code>int *argc</code>	Pointer to an integer that is to be loaded with the number of arguments.
<code>char ***argv</code>	Pointer to an array of string pointers that is loaded with the <code>char *argv[]</code> table.

RETURN:

Void.

getenv ()

Acts similarly to standard `getenv()`.

USAGE:

```
char *getenv(const char *name);
```

DESCRIPTION:

The system can establish shell variables at the command line using the `'set'` command or by another application using the `setenv()` API. This function allows an application to retrieve the value that corresponds to a specified shell variable name.

PARAMETERS:

<code>char *name</code>	The name of the shell variable of which to retrieve the content.
-------------------------	--

RETURN:

Returns the pointer to a NULL-terminated string representing the content of the shell variable specified by `varname`, or `(char *)NULL` if the variable does not exist.

getenvp ()

Retrieves a pointer to a string that contains one white space (newline) delimited "name=value" pair for each shell variable currently defined in the system.

USAGE:

```
char *getenvp(void);
```

DESCRIPTION:

The system can establish shell variables at the command line using the *set* command or by another application using the *setenv()* API function. To retrieve the content of just one known environment variable, use *getenv(char *varname)*. To retrieve a string that contains all "name=value" pairs, use this function. The format of the returned string is:

```
NAME=VALUE NL NAME=VALUE NL NAME=VALUE NL NAME=VALUE NL NULL
```

Where

NL is the newline character.

NULL is the terminating character.

The number of "name=value" pairs is limited only by the amount of heap space available in the system. Note that this command does a *malloc()* to allocate space for the string. It is the responsibility of the caller to free that space (*free()*) when finished with the string.

RETURN:

Returns the string as specified above or (char *)NULL if no shell variables are set.

gps_enable ()

Enables or disables the GPS module.

USAGE:

```
int gps_enable(int enable);
```

DESCRIPTION:

Enables or disables the GPS module. The module by default is disabled.

This function should be called to enable the GPS module before the application tries to access the module.

PARAMETERS:

bool_t enable	0 = Disable. 1 = Enable. -1 = Read state only.
---------------	--

RETURN:

Returns the module state.

ints_disable ()

Allows the application to turn off interrupts.

USAGE:

```
long ints_disable(void);
```

DESCRIPTION:

Allows the application to turn off interrupts.

RETURN:

The value returned can be used as an argument to ints_restore() to re-establish previous interrupt state.

ints_enable ()

Allows the application to turn on interrupts.

USAGE:

```
void ints_enable(void);
```

DESCRIPTION:

Allows the application to turn on interrupts.

RETURN:

Void.

ints_restore ()

Allows the application to restore interrupt state.

USAGE:

```
void ints_restore(long cps);
```

DESCRIPTION:

Allows the application to restore interrupts.

PARAMETERS:

long cps	CPU-specific value that was previously returned by <code>ints_disable()</code> .
----------	--

RETURN:

Void.

irq_enable ()

Enables or disables the specified IRQ interrupt.

USAGE:

```
int irq_enable(unsigned irq, int enable);
```

DESCRIPTION:

Enables or disables the specified IRQ interrupt.

PARAMETERS:

unsigned irq	IRQ number that the function will handle its interrupt.
int enable	Non-zero to enable, zero to disable.

RETURN:

Returns 0 on success, -1 if an invalid IRQ number was passed.

irq_gethandler ()

Obtains the current interrupt handler function for the specified IRQ.

USAGE:

```
void *irq_gethandler(unsigned irq);
```

DESCRIPTION:

Gets the pointer to the current IRQ interrupt handler. This function is typically used in conjunction with the irq_sethandler().

PARAMETERS:

unsigned irq	IRQ number that the function will handle its interrupt.
--------------	---

RETURN:

Returns a pointer to the function that currently is handling the interrupt or NULL if an invalid IRQ number was passed.

irq_sethandler ()

Registers an interrupt handler function for the specified IRQ.

USAGE:

```
int irq_sethandler(unsigned irq, void *pfunct);
```

DESCRIPTION:

Sets a function as an interrupt handler. Subsequently, whenever the irq interrupt is generated, the handler routine is called. The function does not require to preserve the CPU registers, but should clear the interrupt that it is handling before exiting.

PARAMETERS:

unsigned irq	IRQ number that the function will handle its interrupt.
void *pfunct	Pointer to the function that handles the interrupt.

RETURN:

Returns 0 if successful; else -1 if an invalid irq number was passed.

led_set ()

Sets the state of the LEDs.

USAGE:

```
int led_set (int led, int setting);
```

DESCRIPTION:

Sets the state of the red and green LEDs.

PARAMETERS:

int led	LED_RED – controls the red LED. LED_GREEN – controls the green LED.
int setting	1 = turn on the LED. 0 = turn off the LED.

RETURN:

Returns the current state of the LED.

printmem ()

Prints a block of memory.

USAGE:

```
void printmem(uint8_t *base, int size, int showascii);
```

DESCRIPTION:

Provides the application with a simple interface to display a block of memory to the console port in ASCII-coded hex.

PARAMETERS:

uint8_t *base	Starting point of the memory block to be displayed.
int size	Size of the block of memory to be printed.
int showascii	If set to 1, then the memory is displayed in ASCII as well as ASCII-code hex. For each character that is non-printable, the ASCII output displays it as a dot.

RETURN:

Returns the size of the block printed.

reset ()

Allows the application to reset the system.

USAGE:

```
void reset(void);
```

DESCRIPTION:

Performs a system warm boot within 2 seconds, regardless of the watchdog state. The function disables all interrupts to prevent interrupt service routines from feeding the watchdog timer.

Caution: The watchdog timer should not be serviced inside of interrupt service routines.

RETURN:

Void.

rs232_break ()

Generates a break signal on the line of the specified serial port.

USAGE:

```
int rs232_break(unsigned port, long msec);
```

DESCRIPTION:

Generates a break signal on the line.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
long msec	Time in milliseconds to have the break signal active.

RETURN:

If successful, it returns RS232_OK.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_brkdetect ()

Retrieves the state of the break detect flag of the specified serial port.

USAGE:

```
int rs232_brkdetect(unsigned port, int reset);
```

DESCRIPTION:

Obtains the state of the BREAK flag of a serial port. The state is reset when the 'reset' parameter is set to a non-zero value.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
int reset	Clear the BREAK flags if non-zero.

RETURN:

If successful, it returns BREAK state.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_cd ()

Retrieves the state of the CD line of the specified serial port.

USAGE:

```
int rs232_cd(unsigned port);
```

DESCRIPTION:

Obtains the state of the CD line of a serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns CD line state (1 = asserted, 0 = de-asserted).

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_cfgcallback ()

Configures a callback for one of the serial ports.

USAGE:

```
int rs232_cfgcallback(unsigned port, void (*callback)
(int argt,int argv), uint8_t mask);
```

DESCRIPTION:

Configures a callback for one of the CT6xx's serial ports.

The callback should be a function that takes two integer arguments and returns void. This function is invoked based on the mask set the configuration function.

The function is invoked from within an interrupt, because of this, it should be short and to the point to prevent long periods of time with disabled interrupts.

The following table shows the valid values for the mask. They can be OR'd together to have more than one signal activated.

Mask Values	Definition
RS232_RXR	Signal when one or more characters are available for reading.
RS232_TXR	Signal when the last character has been transmitted.
RS232_LSR	Signal when a change of state has occurred in one of the port lines.
RS232_MSR	Signal when a change of state has occurred in one of the modem control lines.

PARAMETERS:

unsigned port	RS232 port to access. Valid parameters: COM1, GPS, CELL
void (*callback)(int argt, int argv),	Pointer to function provided by the application. The first parameter indicates why the function was called. The second parameter can be one of the following: if argt= RS232_RXR then argv=characters available. if argt = RS232_TXR then argv=0. if argt = RS232_LSR then argv=lsr. if argt = RS232_MSR then argv= msr.
uint8_t mask	Type of signal(s) to be activated for the callback.

RETURN:

If successful, it returns RS232_OK.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_close ()

Closes one of the CT6xx's serial ports.

USAGE:

```
int rs232_close(unsigned port);
```

DESCRIPTION:

Deactivates one the CT6xx's serial ports.

PARAMETERS:

unsigned port	RS232 port to close.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns RS232_OK.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_cts ()

Retrieves the state of the CTS line of the specified serial port.

USAGE:

```
int rs232_cts(unsigned port);
```

DESCRIPTION:

Obtains the state of the CTS line of a serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns CTS line state (1 = asserted, 0 = de-asserted).

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_dsr ()

Retrieves the state of the DSR line of the specified serial port.

USAGE:

```
int rs232_dsr(unsigned port);
```

DESCRIPTION:

Obtains the state of the DSR line of the specified serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns DSR line state (1 = asserted, 0 = de-asserted).

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_dtr ()

Sets the state of the DTR line of the specified serial port.

USAGE:

```
int rs232_dtr(unsigned port, int setting);
```

DESCRIPTION:

Sets the state of the DTR line of the specified serial port. A non-zero value in the 'setting' parameter causes the line to be asserted.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
int setting	Assert the line if non-zero.

RETURN:

If successful, it returns RS232_OK.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_frmerror ()

Retrieves the state of the framing error flag of the specified serial port.

USAGE:

```
int rs232_frmerror(unsigned port, int reset);
```

DESCRIPTION:

Obtains the state of the framing error flag of the specified serial port. The flag is reset when the 'reset' parameter is set to a non-zero value.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
int reset	Clear the flags if non-zero.

RETURN:

If successful, it returns framing error flag (1 or 0).

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_getc()

Retrieves one character from the specified serial port.

USAGE:

```
int rs232_getc(unsigned port);
```

DESCRIPTION:

Retrieves one character from the input queue of the specified serial port. If the queue is empty, it returns a -1.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns the next available character from the input queue.

If the queue is empty it returns -1.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_getcfg ()

Retrieves the specified serial port configuration.

USAGE:

```
int rs232_getcfg(unsigned port, uint32_t *baud, char *parity, uint8_t *dbits, uint8_t *sbits);
```

DESCRIPTION:

Retrieves the configuration of the specified serial port.

PARAMETERS:

unsigned port	RS232 port to access. Valid parameters: COM1, GPS, CELL
uint32_t *baud	Pointer where to store the baud rate value.
char *parity	Pointer where to store the parity value.
uint8_t *dbits	Pointer where to store the word length value.
uint8_t *sbits	Pointer where to store the stop bits value.

RETURN:

If successful, it returns RS232_OK.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_open ()

Opens one of the CT6xx's serial ports.

USAGE:

```
int rs232_open(unsigned port);
```

DESCRIPTION:

Activates one of the CT6xx's serial ports. Normally the application should not call this function unless it has called the rs232_close() function first, that is because all serial ports are opened when the application gains control of the system.

PARAMETERS:

unsigned port	RS232 port to open.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns RS232_OK.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_ovrerror ()

Retrieves the state of the overrun error flag of the specified serial port.

USAGE:

```
int rs232_ovrerror(unsigned port, int reset);
```

DESCRIPTION:

Obtains the state of the overrun error flag of the specified serial port. The flag is reset when the 'reset' parameter is set to a non-zero value.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
int reset	Clear the flags if non-zero.

RETURN:

If successful, it returns overrun error flag (1 or 0).

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_prterror ()

Retrieves the state of the framing error flag of the specified serial port.

USAGE:

```
int rs232_prterror(unsigned port, int reset);
```

DESCRIPTION:

Obtains the state of the parity error flag of the specified serial port. The flag is reset when the 'reset' parameter is set to a non-zero value.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
int reset	Clear the flags if non-zero.

RETURN:

If successful, it returns parity error flag (1 or 0).

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_putc ()

Puts one character in the output queue of the specified serial port.

USAGE:

```
int rs232_putc(unsigned port, int c);
```

DESCRIPTION:

Transmits a character by placing it in the output queue of the specified serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
int c	Character to be transmitted.

RETURN:

If successful, it returns the character.

If the queue is full, it returns -1.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_read ()

Reads a block of data from the input queue of the specified serial port.

USAGE:

```
int rs232_read(unsigned port, void *buf, uint16_t
count);
```

DESCRIPTION:

Reads a block of data from the specified serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
void *buf	Pointer where to store the data.
uint16_t count	Number of bytes to retrieve.

RETURN:

If successful, it returns the number of characters read.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_ri ()

Retrieves the state of the RI line of the specified serial port.

USAGE:

```
int rs232_ri(unsigned port);
```

DESCRIPTION:

Obtains the state of the RI (ring) line of the specified serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns RI line state (1 = asserted, 0 = de-asserted).

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_rts ()

Sets the state of the RTS line of the specified serial port.

USAGE:

```
int rs232_rts(unsigned port, int setting);
```

DESCRIPTION:

Sets the state of the RTS line of the specified serial port. A non-zero value in the 'setting' parameter causes the line to be asserted.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
int setting	Assert the line if non-zero.

RETURN:

If successful, it returns RS232_OK.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_rxavail ()

Retrieves the number of bytes available for reading from the specified serial port.

USAGE:

```
int rs232_rxavail(unsigned port);
```

DESCRIPTION:

Obtains the number of characters stored in the input buffer of the specified serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns the number of bytes available.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_rxpurge ()

Removes all available characters from the input queue of the specified serial port.

USAGE:

```
int rs232_rxpurge(unsigned port);
```

DESCRIPTION:

Clears the input queue of the specified serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns RS232_OK.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_setcfg ()

Configures the specified serial port.

USAGE:

```
int rs232_setcfg(unsigned port, uint32_t baud, char parity, uint8_t dbits, uint8_t sbits);
```

DESCRIPTION:

Configures one of the CT6xx's serial ports.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
uint32_t baud	Baud rate value (Valid baud rates include: 1200, 2400, 4800, 9600, 38400, 57600, and 115200).
char parity	Parity ('N' = none, 'E' = even, 'O' = odd).
uint8_t dbits	Word length (7 or 8).
uint8_t sbits	Stop bits (1 or 2).

RETURN:

If successful, it returns RS232_OK.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_txfree ()

Retrieves the space available in the output buffer of the specified serial port.

USAGE:

```
int rs232_txfree(unsigned port);
```

DESCRIPTION:

Obtains the space available in the transmit queue (in bytes) for the specified serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns the number of bytes available.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_txpurge ()

Stops transmission (if any) and clears the output queue of the specified serial port.

USAGE:

```
int rs232_txpurge(unsigned port);
```

DESCRIPTION:

Stops a transmission and clears the output queue of the specified serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL

RETURN:

If successful, it returns RS232_OK.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs232_write ()

Writes a block of data to the output queue of the specified serial port.

USAGE:

```
int rs232_write(unsigned port, void *buf, uint16_t
count);
```

DESCRIPTION:

Transmits a block of data to the specified serial port.

PARAMETERS:

unsigned port	RS232 port to access.
	Valid parameters: COM1, GPS, CELL
void *buf	Pointer to data to be transmitted.
uint16_t count	Number of bytes to transmit.

RETURN:

If successful, it returns the number of characters transferred.

If an invalid port was specified, it returns RS232_ERROR_PORTNOTFOUND.

rs485_autotxline ()

Sets the transit control line of the RS-485 serial port under the control of the device driver.

USAGE:

```
int rs485_autotxline(bool_t ctrlauto);
```

DESCRIPTION:

This function allows the application to obtain full control or let the RS-485 device driver control the transmit line. By default the transmit line is under control of the device driver.

PARAMETERS:

bool_t ctrlauto	0 = Under application control.
	1 = Under driver control.

RETURN:

DYACON, Inc.

Returns the previous state of the auto-control state.

rs485_cfgcallback ()

Configures a callback for the RS-485 port.

USAGE:

```
void rs485_cfgcallback(void (*callback)(int argt,int
argv), uint8_t mask);
```

DESCRIPTION:

Configures a callback for the RS-485 serial port.

The callback should be a function that takes two integer arguments and returns void. This function is invoked based on the mask set at the configuration function.

The function is invoked from within an interrupt, because of this, it should be short and to the point to prevent long periods of time with disabled interrupts.

The following table shows the valid values for the mask. They can be OR'd together to have more than one signal activated.

Mask Values	Definition
RS485_RXR	Signal when one or more characters are available for reading.
RS485_TXR	Signal when the last character has been transmitted.
RS485_LSR	Signal when a change of state has occurred in one of the port lines.
RS485_MSR	Signal when a change of state has occurred in one of the modem control lines.

PARAMETERS:

void (*callback)(int,int)	Pointer to function provided by the application.
	The first parameter indicates why the function was called.
	The second parameter can be one of the following:
	If argt= RS485_RXR then argv=characters available.
	If argt = RS485_TXR then argv=0.
	If argt = RS485_LSR then argv=lsr.
	If argt = RS485_MSR then argv= msr.
uint8_t mask	Type of signal(s) to be activated for the callback.

RETURN:

Void.

rs485_close ()

Closes the RS-485 serial port.

USAGE:

```
void rs485_close(void);
```

DESCRIPTION:

Deactivates the RS-485 serial port.

RETURN:

Void.

rs485_cts ()

Retrieves the state of the CTS line of the RS-485 serial port.

USAGE:

```
int rs485_cts(void);
```

DESCRIPTION:

Obtains the state of the CTS line of the RS-485 serial port.

RETURN:

Returns the CTS line state (1 = asserted, 0 = de-asserted).

rs485_getcfg ()

Retrieves the RS-485 serial port configuration.

USAGE:

```
void rs485_getcfg(uint32_t *baud, char *parity, uint8_t *dbits, uint8_t *sbits);
```

DESCRIPTION:

Retrieves the configuration of the RS-485 serial port.

PARAMETERS:

uint32_t *baud	Pointer where to store the baud rate value.
char *parity	Pointer where to store the parity value.
uint8_t *dbits	Pointer where to store the word length value.
uint8_t *sbits	Pointer where to store the stop bits value.

RETURN:

Void.

rs485_open ()

Opens the RS-485 serial port.

USAGE:

```
int rs485_open(void);
```

DESCRIPTION:

Activates the RS-485 serial port. Normally the application should not call this function unless it has called the rs485_close() function first, that is because the RS-485 serial port is opened when the application gains control of the system.

RETURN:

If successful, it returns RS485_OK.

rs485_read ()

Reads a block of data from the input queue of the RS-485 serial port.

USAGE:

```
int rs485_read(void *buf, uint16_t count);
```

DESCRIPTION:

Reads a block of data from the input queue of the RS-485 serial port.

PARAMETERS:

void* buff	Pointer to data to be transmitted.
uint16_t count	Number of bytes to transmit.

RETURN:

Returns the number of characters read.

rs485_rxavail ()

Retrieves the number of bytes available for reading from the RS-485 serial port.

USAGE:

```
int rs485_rxavail(void);
```

DESCRIPTION:

Obtains the number of characters stored in the input buffer of the RS-485 serial port.

RETURN:

Returns the number of bytes available.

rs485_rxpurge ()

Removes all available characters from the input queue.

USAGE:

```
void rs485_rxpurge(void);
```

DESCRIPTION:

Clears the input queue of the RS-485 serial port.

RETURN:

Void.

rs485_setcfg ()

Configures the RS-485 serial port.

USAGE:

```
int rs485_setcfg(uint32_t baud, char parity, uint8_t dbits, uint8_t sbits);
```

DESCRIPTION:

Configures the RS-485 serial port.

PARAMETERS:

uint32_t baud	Baud rate value (Valid baud rates include: 1200, 2400, 4800, 9600, 38400, 57600, and 115200.)
char parity	Parity ('N' = none, 'E' = even, 'O' = odd).
uint8_t dbits	Word length (7 or 8).
uint8_t sbits	Stop bits (1 or 2).

RETURN:

If successful, it returns RS485_OK ; else

RS485_ERROR_BADPARITY

RS485_ERROR_BADWRDSIZE

RS485_ERROR_BADSTOPBITS

RS485_ERROR_BADPARAM

rs485_txfree ()

Retrieves the space available in the output buffer of the RS-485 serial port.

USAGE:

```
int rs485_txfree(void);
```

DESCRIPTION:

Obtains the space available in the transmit queue (in bytes) for the RS-485 serial port.

RETURN:

Returns the number of bytes available.

rs485_txline ()

Sets the state of the transmit control line of the RS-485 serial port.

USAGE:

```
int rs485_txline(int setting);
```

DESCRIPTION:

Sets the state of the transmit control line of the RS-485 serial port. A non-zero value in the 'setting' parameter causes the line to be asserted.

PARAMETERS:

int setting	0 = De-assert. -1 = Read state only. other = Assert.
-------------	--

RETURN:

Returns transmit line state.

rs485_txpurge ()

Stops transmission (if any) and clears the output queue of the RS-485 serial port.

USAGE:

```
void rs485_txpurge(void);
```

DESCRIPTION:

Stops a transmission and clear the output queue of the RS-485 serial port.

RETURN:

Void.

rs485_write ()

Writes a block of data to the output queue of the RS-485 serial port.

USAGE:

```
int rs485_write(void *buf, uint16_t count);
```

DESCRIPTION:

Transmits a block of data.

PARAMETERS:

void *buf	Pointer to data to be transmitted.
uint16_t count	Number of bytes to transmit.

RETURN:

Returns the number of characters transferred.

rtc_getalarm ()

Gets the real time clock alarm settings.

USAGE:

```
int rtc_getalarm (alarm_t *palarm);
```

DESCRIPTION:

Reads back the RTC alarm settings. The alarm_t structure is defined as follows:

```
Typedef struct alarm alarm_t;
struct alarm
{
    unsigned char  sec;    // seconds
    unsigned char  min;    // minutes
    unsigned char  hr;     // hours
    unsigned short day;    // day of the year
    void (*callback) (void); // pointer to callback function
};
```

RETURN:

Returns zero if the alarm is enabled; otherwise it returns a non-zero value.

rtc_getdate ()

Gets the system date.

USAGE:

```
void rtc_getdate (rtcdate_t *pdt);
```

DESCRIPTION:

Fills in the `rtcdate_t` structure (pointed by `pdt`) with the system's current date. The `rtcdate_t` structure is defined as follows:

```
Typedef struct _rtcdate rtcdate_t;
struct _rtcdate
{
    unsigned char    day;    // day of the month
    unsigned char    month; // month (1 = Jan)
    unsigned short   year;   // current year
    unsigned char    dow;   // Day Of Week (0 is Sunday)
};
```

RETURN:

Function does not return a value.

rtc_gettime ()

Gets the system time.

USAGE:

```
void rtc_gettime (rtctime_t *ptm);
```

DESCRIPTION:

Fills the `rtctime_t` structure, pointed to by `ptm`, with the system's current time. The `rtctime_t` structure is defined as follows:

```
Typedef struct _rtctime rtctime_t;
struct _rtctime
{
    unsigned char  sec;    // seconds
    unsigned char  min;    // minutes
    unsigned char  hr;     // hours
};
```

RETURN:

This function does not return a value.

rtc_setalarm ()

Sets the real time clock alarm.

USAGE:

```
void rtc_setalarm (alarm_t *palarm, int enable);
```

DESCRIPTION:

Sets the RTC alarm with the values passed in the structure pointed by `*palarm`. The second parameter specifies if the alarm should be enabled or disabled.

RETURN:

Void.

rtc_setdate ()

Sets the system date.

USAGE:

```
void rtc_setdate (rtcdate_t *pdt);
```

DESCRIPTION:

Sets the system date (month, day, and year) to that in the rtcdate structure pointed to by *pdt.

RETURN:

Void.

rtc_settime ()

Sets the system time.

USAGE:

```
void rtc_settime (rtctime_t *ptm);
```

DESCRIPTION:

Sets the system time to that in the rtctime structure pointed to by *ptm.

RETURN:

Void.

rtc_timestamp ()

Obtains a time stamp from the system.

USAGE:

```
ts_t rtc_timestamp (void);
```

DESCRIPTION:

Obtains a time stamp from the system.

RETURN:

The time stamp returned from the system is an unsigned 32 bit value formatted as follows:

```
// Field: -----Year -----Month--- ---Day----- ---Hours---- --- Minutes--- ---Seconds---  
// Bit#:  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

The year is based from 2000. That is, if bits 31..26 contain 0 it really means that the year is 2000. If bits 31..26 contain 12, the year is 2012.

Bits	Description
0-5	Seconds bits
6-11	Minutes bits
12-16	Hours bits
17-21	Day bits
22-25	Month bits
26-31	Year bits

setenv ()

Establishes shell variables at the command line of the CT6xx.

USAGE:

```
void setenv (const char *name, const char *value, int
notused);
```

DESCRIPTION:

Establishes shell variables at the command line of the CT6xx. This function allows an application to also establish a shell variable.

PARAMETERS:

char *name	The name of the shell variable to be created.
char *value	The value that the shell variable represents. If this pointer is NULL, the shell variable with the name is removed from the environment.
int notused	This parameter is for compatibility with the standard library version of setenv. It is not used and should be set to zero.

RETURN:

Returns zero if successful; otherwise it returns a non-zero value.

sleep ()

Suspends execution for an interval (seconds).

USAGE:

```
void sleep (unsigned int sec);
```

DESCRIPTION:

Suspends execution for an interval (seconds). With a call to sleep, the current program is suspended from execution for the number of seconds specified by the argument sec. This function uses the internal RTC alarm function (preserves any previous settings) to time the sleep. All interrupts are disabled during the sleep period except for the RTC alarm.

PARAMETERS:

unsigned int sec	Interval in seconds that execution is suspended.
------------------	--

RETURN:

Void.

suspend ()

Puts the system in a low power mode and halts it.

USAGE:

```
void suspend (void);
```

DESCRIPTION:

Puts the system in a low power mode and halts it. A call to suspend puts the system in a low power mode and halts it. Activity in COM1 will make the system resume execution. In suspend mode all interrupts except the RTC day interrupt and COM1 interrupts are disabled.

RETURN:

Void.

sys_free ()

Acts similarly to standard free().

USAGE:

```
void sys_free(char *cp);
```

DESCRIPTION:

Acts similarly to standard free(). Used to release memory that was previously allocated by sys_malloc() or sys_realloc(). If the pointer passed to sys_free() is not a pointer that was previously returned by sys_malloc() an error results.

PARAMETERS:

char *cp	Pointer to a buffer that was previously allocated by sys_malloc() or sys_realloc().
----------	---

RETURN:

Void.

sys_getbytes ()

Retrieves some number of bytes from the console port.

USAGE:

```
sys_getbytes (char *buf, int cnt, int block);
```

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still read in characters from the serial port designated as the console.

PARAMETERS:

char *buf	Pointer to data buffer.
int cnt	Number of characters to transmit
int block	

RETURN:

The number of characters retrieved.

sys_getchar ()

Provides similar functionality as standard getchar().

USAGE:

```
int sys_getchar(void);
```

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still receive characters from the console port. This function will block, waiting for the next incoming character into the console port. If a character is already there prior to the call, then the return is immediate.

RETURN:

Returns the character read on the console port.

sys_getline ()

Retrieves characters from the console.

USAGE:

```
int sys_getline(char *buf, int max, int ledit);
```

DESCRIPTION:

Retrieves characters from the console port until a carriage return or line feed is received (or 'max' characters are received). The 'ledit' option enables the use of the command line editing facilities.

PARAMETERS:

char *buf	A pointer to the space into which the incoming characters are to be placed.
int max	The maximum number of bytes to place into the buffer.
int ledit	If set to 1, then enable the use of the line-editing functions while retrieving the line.

RETURN:

Returns the number of bytes retrieved.

sys_gotachar ()

Returns status of character presence on the console port.

USAGE:

```
int sys_gotachar(void);
```

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still query for the presence of a character on the console. This query does not affect the status of the console's incoming character stream.

RETURN:

Returns a 1 if there is a character present; else 0.

sys_heap_extend ()

Extends the basic heap that is statically allocated to the system.

USAGE:

```
int sys_heap_extend(char *start, int size);
```

DESCRIPTION:

Extends the basic heap that is statically allocated to the system. The system has a heap for its own use of `sys_malloc`. The memory allocation used by the system supports a non-contiguous heap. This means that a second chunk of memory can be given to the system for additional heap space and it does not have to be memory contiguously appended to the end of its first allocated block. This API call allows an application to allocate a second chunk of memory to the system's heap.

PARAMETERS:

char *start	Pointer to the starting location of a new block of heap memory space.
int size	Size of the block of memory. If this size is set to -1, then the current heap expansion is released.

RETURN:

Returns 0 if the request was accepted; else -1 indicating a failure.

sys_malloc ()

Acts similarly to standard malloc().

USAGE:

```
char *sys_malloc(int size);
```

DESCRIPTION:

Acts similarly to standard malloc(). This provides the application with a memory allocator whose heap is maintained by the system. At each allocation/deallocation, the entire control structure of the heap is checked for sanity. Also, in addition to control verification, overrun and under-run checks are made because the space returned by sys_malloc also has additional wrapper space that is checked to verify that the space is not used beyond (or prior to) its limit. This implies overhead, which may or may not be desired, but does provide a reasonable amount of error checking.

The system is built with an amount of heap space that is needed for the system itself plus a bit extra for application space. To allow additional memory allocations to be made, the heapextend() API function allows the user to define a starting point and size of the additional space.

PARAMETERS:

int size	The size of the block of memory to be allocated.
----------	--

RETURN:

Returns a pointer to the block of memory or (char *)NULL if an error occurs.

sys_printf ()

Acts similarly to printf() but limited in the formatting capability (console printf()).

USAGE:

```
int sys_printf(const char *fmt, ...);
```

DESCRIPTION:

Provides the application with a small and simple printf() with limited formatting capability. This function does not support floating point.

PARAMETERS:

char *format	Pointer to a format buffer.
...	argN arguments referred to (if any) by the format buffer.

RETURN:

Returns the size of the final string printed out the console port.

sys_putbytes ()

Sends some number of bytes to the console port.

USAGE:

```
int sys_putbytes (char *buf, int cnt);
```

DESCRIPTION:

Sends some number of bytes to the console port. Allows the application to be unaware of the console port interface, but still write characters to the serial port designated as the console.

PARAMETERS:

char *buf	Pointer to the data buffer.
int cnt	Number of characters to transmit.

RETURN:

Returns the number of characters sent.

sys_putchar ()

Provides similar functionality as standard putchar().

USAGE:

```
int sys_putchar(int c);
```

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still transfer characters to the console.

PARAMETERS:

int c	The character destined for the console port.
-------	--

RETURN:

Returns the same character that was passed to the function is returned.

sys_puts ()

Provides similar functionality as standard puts() (console puts()).

USAGE:

```
int sys_puts(const char *str);
```

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still transfer characters to the console.

PARAMETERS:

char *str	NUL terminated string destined for the console port.
-----------	--

RETURN:

Returns the same character that was passed to the function is returned.

sys_realloc ()

Reallocates a block of memory from the system's heap.

USAGE:

```
char *sys_realloc(char *buf, int newsize);
```

DESCRIPTION:

Allows the application to adjust the size of a block of memory that was previously allocated from the system's heap.

PARAMETERS:

char * buf	Pointer to the current buffer.
int newsize	Size to make the new buffer.

RETURN:

If successful, a non-zero pointer is returned; else NULL.

sys_sprintf ()

Acts similarly to sprintf(), but limited in the formatting capability.

USAGE:

```
int sys_sprintf(char *buf, char *fmt, ...);
```

DESCRIPTION:

Provides the application with a small and simple sprintf() with limited formatting capability. The %s, %x, %c, and %d formats are supported to some degree. To keep it small and simple, no floating point conversion is supported. If %s points to a NULL, the string "NULL_POINTER" is printed.

Non-standard format conversions for time stamp format is supported. %T assumes the argument is a ts_t type (long) and it is converted to a string of the form

"MM-DD-YY HH:MM:SS"

PARAMETERS:

char *buffer	Buffer into which the format conversion is to be placed.
char *format	Pointer to a format buffer.
...	argN arguments referred to (if any) by format buffer.

RETURN:

Returns the size of the final string printed out the console port.

sys_version ()

Retrieves the system's version information.

USAGE:

```
char *sys_version(void);
```

DESCRIPTION:

Retrieves the system's version information. The version is a 2-digit, dot-delimited number: X.Y where "X.Y" is the major and minor version of the firmware. Note that the version information and the build-date of the firmware are also available through the shell variables VERSION_MAJ and VERSION_MIN.

RETURN:

Returns a null terminated string containing the "X.Y" value discussed above.

TFS RETURN CODES

Note that all return error codes are negative except TFS_OKAY.

ERROR CODE RETURNED	Value	Error Meaning
TFS_OKAY	0	No error.
TFSERR_NOFILE	-1	File not found.
TFSERR_NOSLOT	-2	Maximum number of files opened.
TFSERR_EOF	-3	End of file.
TFSERR_BADARG	-4	Bad argument.
TFSERR_NOTEXEC	-5	Not executable.
TFSERR_BADCRC	-6	Bad crc.
TFSERR_FILEEXISTS	-7	File already exists.
TFSERR_FLASHFAILURE	-8	Flash operation failed.
TFSERR_WRITEMAX	-9	Max write count exceeded.
TFSERR_RDONLY	-10	File is read-only.
TFSERR_BADFD	-11	Invalid descriptor.
TFSERR_BADHDR	-12	Bad binary executable header.
TFSERR_CORRUPT	-13	Corrupt file.
TFSERR_MEMFAIL	-14	Memory failure.
TFSERR_NOTIPMOD	-15	File is not in-place-modifiable.
TFSERR_FLASHFULL	-16	Out of flash space.
TFSERR_USERDENIED	-17	User level access denied.
TFSERR_NAMETOOBIG	-18	Name or info field too big.
TFSERR_FILEINUSE	-19	File in use.
TFSERR_NOTAVAILABLE	-20	Not Available
TFSERR_BADFLAG	-21	Bad flag.
TFSERR_CLEANOFF	-22	Defragmentation is disabled.
TFSERR_FLAKEYSOURCE	-23	Dynamic source data.
TFSERR_BADEXTENSION	-24	Bad Extension
TFSERR_LINKERROR	-25	File link error.
TFSERR_BADPREFIX	-26	Invalid device prefix.
TFSERR_NORUNINITTAB	-27	
TFSERR_UNKNOWN	-99	
TFSERR_MIN	-100	

tfsadd ()

Adds a new file to TFS.

USAGE:

```
int tfsadd(char *name, char *info, char *flags, uint8_t
*src, int size);
```

DESCRIPTION:

Adds a new file to TFS. The application can use this function to create a file from some block of memory without having to go through a typical open, write, close scenario. If the file already exists, it first sees if the incoming data is identical to that of the file already in TFS; if it is, then no flash operation is performed and TFS_OKAY is returned. If there are differences, then the new file is added and verified, then the old file is deleted.

PARAMETERS:

char *name	Name of the file being created.
char *info	Content to be placed in the info field of the header (or NULL).
char *flags	Flags to be assigned to the file (or NULL).
uint8_t *src	Location of the data to become the file content
int size	Size of the data copied to the file.

RETURN:

Returns TFS_OKAY if successful; else

TFSERR_BADARG

TFSERR_CORRUPT

TFSERR_FILEEXISTS

TFSERR_FLASHFULL

TFSERR_FLASHFAILURE

TFSERR_BADCRC

TFSERR_FLAKEYSOURCE

tfsclose ()

Closes a TFS file that had previously been opened.

USAGE:

```
int tfsclose(int tfd, char *info);
```

DESCRIPTION:

Closes a TFS file that had previously been opened. When all interaction with an opened file is complete, `tfsclose()` must be called to release the file descriptor used with the opened file and possibly initiate a transfer of the data to flash (if the file was opened for some type of modification). The *tfd* argument is the value that was returned from the initial `tflopen()`, and the *info* argument is a string (optionally NULL) that is used as the "info" field of the file header (if it is being modified or created).

NOTE:

This is a significant difference between TFS and a standard open/close/read/write model for file IO. TFS does not actually write any data to flash until the file interaction is completed (i.e. when `tfsclose()` is called).

PARAMETERS:

int tfd	The same value that was returned when the initial <code>tflopen()</code> was called.
char *info	A pointer to a string that is to be stored in the "info" field of the file header.

RETURN:

Returns `TFS_OKAY` if successful; else

`TFSERR_BADARG`

`TFSERR_BADFD`

`TFSERR_FLASHFAILURE`

tfseof ()

Returns EOF (end of file) status on specified file.

USAGE:

```
int tfseof(int fd);
```

DESCRIPTION:

Allows the application to check to see if a currently opened-for-read file has reached the end-of-file.

PARAMETERS:

int fd	TFS file descriptor returned from a previous call to <code>tfsopen()</code> .
--------	---

RETURN

Returns a 1 if TFS's internal pointer has reached the end of the file;

0 if not at the end of file;

else negative indicating some error:

TFSERR_BADARG

TFSERR_BADFD

tfsfstat ()

Populates a TFILE structure with the designated file's file header structure.

USAGE:

```
int tfsfstat(char *filename, TFILE *tfsstruct);
```

DESCRIPTION:

Retrieves a TFILE structure (struct `tfshdr`) attached to the specified file (if it exists).

PARAMETERS:

char *filename	Name of file in TFS.
TFILE *tfsstruct	Pointer to a TFILE structure that <code>tfsfstat</code> populates if the file exists.

RETURN:

Returns 0 if the file exists; else -1.

tfsgetline ()

Retrieves the next line from an assumed ASCII file.

USAGE:

```
int tfsgetline (int fd, char *buf, int max);
```

DESCRIPTION:

Retrieves the next line of characters from an opened ASCII-readable file. Retrieval continues until either 'max-1' characters are loaded or a CR and/or LF is found. If CR and/or LF is loaded, it is followed by a NULL. The returned buffer is always NULL terminated.

PARAMETERS:

int fd	The descriptor of the file, returned previously by tfsopen().
char *buffer	A pointer to the space into which TFS is to place the specified number of bytes.
int max	The maximum number of bytes to place into the buffer.

RETURN:

Returns the number of bytes retrieved if successful; else the error returned from tfsread().

tfsinit ()

Initializes the flash space that is used by TFS.

USAGE:

```
int tfsinit(void);
```

DESCRIPTION:

Initializes the flash space that is used by TFS. All of the flash space is erased by this function.

RETURN

Returns TFS_OKAY if successful; else negative indicating a flash operation error (this condition should not happen).

tfsioctl ()

Performs some type of control operation on TFS or a file in TFS.

USAGE:

```
long tfsioctl(int rqst, long arg1, long arg2);
```

DESCRIPTION:

Acts similarly in purpose to a standard ioctl() system call. Performs some type of control operation on TFS or a file in TFS.

PARAMETERS:

int rqst	Type of control function to be performed.
long arg1	Depending on the value of rqst, this argument may or may not be used.
long arg2	Depending on the value of rqst this argument may or may not be used.

VALID RQST VALUES:

TFS_ERRMSG	Returns a pointer to a character string that corresponds to the verbose description of the error. The value in <i>arg1</i> is some error value that was returned by some other TFS system call.
TFS_MEMUSE	Returns the amount of flash memory that is currently being used by files in TFS. This includes space used by active and deleted files.
TFS_MEMAVAIL	Returns the amount of flash memory that is still available for use by TFS.
TFS_MEMDEAD	Returns the amount of flash memory that is currently being used by deleted files.
TFS_DEFRAG	Runs a TFS defragmentation, to remove any "dead" flash space taken up by deleted files. If <i>arg1</i> is non-zero, then after defragmentation, the OBC605 is reset; the value of <i>arg2</i> is considered the verbosity level to use during defragmentation.
TFS_UNOPEN	If a TFS file was previously opened for creation or append, and for some reason, the need to create/modify the file no longer exists, this function essentially calls <code>tfsclose()</code> but does not make any modifications to the flash. The value of <i>arg1</i> is the file descriptor returned by the initial call to <code>tfsopen()</code> .
TFS_FCOUNT	Returns the number of files in TFS or within one device within TFS space. The value of <i>arg1</i> (if non-zero) is assumed to be the name of the TFS device; if NULL, then a count of all files (regardless of device) is returned.

TFS_TELL	Returns the offset into the file specified by <i>arg1</i> which is the file descriptor returned by <code>tfsopen()</code> sometime prior.
TFS_DEFRAGDEV	<i>Arg1</i> is a char pointer to the name of the TFS device to be defragmented. Returns TFS_OKAY if successful.
TFS_INITDEV	<i>Arg1</i> is a char pointer to the name of the TFS device to be initialized. Returns TFS_OKAY if successful.
TFS_CHECKDEV	<i>Arg1</i> is a char pointer to the name of the TFS device to be checked. Returns TFS_OKAY if file system on the specified device is not found to be corrupt.
TFS_RAMDEV	<i>Arg1</i> is TRAMDEV pointer (see OBC605.h) which must contain the new RAM device configuration. Returns TFS_OKAY if successful.

tfsnext ()

Retrieves the “next” file in the TFS list.

USAGE:

```
TFILE *tfsnext (TFILE *fp);
```

DESCRIPTION:

Retrieves the “next” file in the TFS list.

PARAMETERS:

TFILE *fp	Pointer to a TFILE structure.
-----------	-------------------------------

RETURN:

If the incoming argument is NULL, then return the first file in the list. If no more files, return NULL; else return the `tfshdr` structure pointer to the next (or first) file in the TFS.

tfsopen ()

Opens a TFS file for read and/or write access.

USAGE:

```
int tfsopen(char *file, long flagmode, char *buf);
```

DESCRIPTION:

Acts similarly to a standard open() of a file, this function allows the user to open a TFS file for access. The final buffer argument is needed only for files that are to be created or modified. This is the space that is used by TFS for building the file. As multiple tfswrite() calls are made, the data written is placed in this buffer; then when tfsclose() is called to complete the file transaction, the buffer is transferred to flash to become a permanent part of the file system.

Note that the final buffer argument should be NULL if the file is opened for read-only.

PARAMETERS:

char *file	Name of the file to be read, written or created.
long flagmode	The flags to be applied to the file when closed and the mode that the file is to be opened with.
char *buf	A pointer to memory space that is used by TFS while the file is being generated.

Valid modes:

TFS_RDONLY	File is assumed to already exist, and it is being opened for read only.
TFS_APPEND	File is assumed to already exist, and it is being opened to append to the end of the current file. If the file does not exist, then an error (TFSERR_NOFILE) is returned.
TFS_CREATE	File is assumed to not exist, and it is being created. If the file does exist, an error (TFSERR_FILEEXISTS) is returned.

In general, only one mode should be specified. An exception to this is TFS_APPEND|TFS_CREATE. If both of these modes are specified, then TFS modifies the mode based on the presence of the file. If the file exists, then it is opened with TFS_APPEND; if the file doesn't exist, it is opened with TFS_CREATE.

Valid Flags:

TFS_SCRIPT	Executable.
TFS_BRUN	To be automatically executed at boot time.
TFS_QRYBRUN	To be automatically executed at boot time, after querying at the console.
TFS_ELF	Loadable executable in ELF format.
TFS_UNREAD	File cannot even be read at a privilege level lower than its own.
TFS_PLVLN	File is accessible by privilege level N and above, where N can be 0-3.

In general, multiple flags are specified for a file. For example:

TFS_SCRIPT | TFS_QRYBRUN | TFS_ELF | TFS_PLVL2

is a valid flag specification indicating that the file is executable ELF that will autoboot with query and will only be executable by privilege levels greater than or equal to 2.

RETURN:

Returns any number greater than or equal to zero if successful; else

TFSERR_NOFILE

TFSERR_USERDENIED

TFSERR_FILEEXISTS

TFSERR_BADARG

TFSERR_MEMFAIL

TFSERR_NOSLOT

tfsread ()

Accesses a file that has been previously opened for reading and retrieve data from that flash space.

USAGE:

```
int tfsread(int fd, char *buf, int cnt);
```

DESCRIPTION:

Acts similarly to a standard read() of a file, this function allows the user to retrieve data from a file that has been previously opened.

PARAMETERS:

int fd	The descriptor of the file, returned previously by tfsopen().
char *buf	A pointer to the space into which TFS is to place the specified number of bytes.
int cnt	The number of bytes to place into the buffer.

RETURN:

Returns the number of bytes retrieved if successful; else negative

TFSERR_BADARG

TFSERR_BADFD

TFSERR_EOF

TFSERR_MEMFAIL

tfsseek ()

Moves the internal pointer maintained by TFS to some specified position.

USAGE:

```
int tfsseek(int fd, int offset, int whence);
```

DESCRIPTION:

Acts similarly to a standard lseek() of a file, this function allows the user to adjust the current pointer maintained by TFS for the specified file.

PARAMETERS:

int fd	Descriptor of the file whose pointer is to be adjusted.
int offset	Offset relative to location specified by 'whence'.
int whence	Base position from which the offset is assumed.

Valid values for whence:

TFS_BEGIN	Specified offset is relative to the beginning of the file.
TFS_CURRENT	Specified offset is relative to the current position in the file.

RETURN:

Returns the offset into the file if successful; else negative.

TFSERR_BADARG

TFSERR_EOF

tfsstat ()

Returns a TFILE pointer to the file specified.

USAGE:

```
struct tfshdr *tfsstat(char *filename);
```

DESCRIPTION:

Retrieves a TFILE pointer (struct tfshdr *) to the specified file (if it exists).

PARAMETERS:

char *filename	Name of file in TFS.
----------------	----------------------

RETURN:

Returns a pointer to the header of the specified file or (struct tfshdr *)NULL.

tfstell ()

Returns the current offset into the file referred to by the incoming descriptor.

USAGE:

```
long tfstell(int fd);
```

DESCRIPTION:

Determines current offset into the specified file.

PARAMETERS:

int fd	Descriptor (returned by <code>tfopen()</code>) of the file.
--------	--

RETURN:

Returns `TFSERR_BADARG` if failure; else the offset.

tfstruncate ()

Truncates the size of a file that has been opened for append to a new size.

USAGE:

```
int tfstruncate (int fd, int size);
```

DESCRIPTION:

Truncates the size of a file that has been opened for append to a new size. If a file is opened for writing (`TFS_APPEND` flag passed to `tfopen`), and the modifications to this file require that the new file size be smaller, then the file size must be truncated. This function provides that capability.

PARAMETERS:

int fd	Is the file descriptor returned by <code>tfopen</code> .
int size	Is the smaller file size.

RETURN:

Returns zero if successful.

tfsunlink ()

Removes a file from TFS flash space.

USAGE:

```
int tfsunlink(char *name);
```

DESCRIPTION:

Removes a file from TFS flash space.

PARAMETERS:

char *name	Name of the file to be removed.
------------	---------------------------------

RETURN:

Returns TFS_OKAY if successful; else

TFSERR_NOFILE

TFSERR_USERDENIED

TFSERR_FLASHFAILURE

tfswrite ()

Accesses a file that has been previously opened for writing and transfer data to TFS for eventual transfer to flash.

USAGE:

```
int tfswrite(int fd, char *buf, int cnt);
```

DESCRIPTION:

Acts similarly to a standard write() of a file, this function allows the user to place data into a file that was previously opened for writing.

PARAMETERS:

int fd	The descriptor of the file, returned previously by tfsopen().
char *buf	A pointer to the space from which TFS is to copy the specified number of bytes.
int cnt	The number of bytes for TFS to copy from the buffer.

RETURN:

Returns TFS_OKAY if successful; else

TFSERR_BADARG

TFSERR_RDONLY

TFSERR_MEMFAIL

tmr_cfgfnct ()

Configures a callback for one of the timers.

USAGE:

```
void tmr_cfgfnct(unsigned n, void (*fnct)(void *), void *arg);
```

DESCRIPTION:

Configures a callback for one of the 10 timers.

The callback should be a function that takes a pointer to a void argument and return void. This function is invoked when the time of the timer expires, and it is passed 'arg' as the parameter.

The function is invoked from within an interrupt, because of this, it should be short and to the point to prevent a long period of time with disabled interrupts.

PARAMETERS:

unsigned n	Which timer (0 – 9).
void (*fnct) void *)	Pointer to function provided by the application.
void* arg	Pointer to callback parameter.

RETURN:

Void.

tmr_check ()

Retrieves the timer timeout counter.

USAGE:

```
uint32_t tmr_check(unsigned n);
```

DESCRIPTION:

Retrieves the timer timeout counter. When the counter is zero, the timer has timed out.

PARAMETERS:

unsigned n	Which timer (0 – 9).
------------	----------------------

RETURN:

Returns the timer timeout counter.

tmr_reset ()

Resets the timer timeout to its set timeout.

USAGE:

```
void tmr_reset(unsigned n);
```

DESCRIPTION:

Resets the timer to its timeout.

PARAMETERS:

unsigned n	Which timer (0 – 9).
------------	----------------------

RETURN:

Void.

tmr_setmst ()

Sets the timer timeout.

USAGE:

```
void tmr_setmst(unsigned n, uint16_t min, uint16_t sec,  
uint16_t tenths);
```

DESCRIPTION:

Sets the timer timeout in minutes, seconds, and tenths of a second. This function should be called before starting the timer.

PARAMETERS:

unsigned n	Which timer (0 – 9).
uint16_t min	Time in minutes.
uint16_t sec	Time in seconds.
uint16_t tenths	Time in tenths of a second.

RETURN:

Void.

tmr_sett ()

Sets the timer timeout.

USAGE:

```
void tmr_sett(unsigned n, uint32_t tenths);
```

DESCRIPTION:

Sets the timer timeout in tenths of a second. This function should be called before starting the timer.

PARAMETERS:

unsigned n	Which timer (0 – 9).
uint32_t tenths	Time in tenths of a second.

RETURN:

Void.

tmr_start ()

Starts a timer.

USAGE:

```
void tmr_start(unsigned n);
```

DESCRIPTION:

Starts a timer.

PARAMETERS:

unsigned n	Which timer (0 – 9).
------------	----------------------

RETURN:

Void.

tmr_stop ()

Stops a timer.

USAGE:

```
void tmr_stop(unsigned n);
```

DESCRIPTION:

Stops a timer.

PARAMETERS:

unsigned n	Which timer (0 – 9).
------------	----------------------

RETURN:

Void.

watchdog_enable ()

Enables or disables the system's watchdog.

USAGE:

```
void watchdog_enable(int enable);
```

DESCRIPTION:

Enables or disables the system's watchdog.

PARAMETERS:

int enable	Non-zero value enables the watchdog; else it disables the watchdog.
------------	---

RETURN:

Void.

watchdog_service ()

Tickles or services the watchdog.

USAGE:

```
void watchdog_service(void);
```

DESCRIPTION:

Tickles or services the watchdog. When the system's watchdog is enabled, the application needs to periodically tickle the watchdog and to do so, it should call this function.

Note: The watchdog is automatically serviced during flash defragment and flash write (tsf_close). The duration of these two functions can't be controlled by the application. Delay () must be less than the watchdog timeout interval or a reset will occur.

RETURN:

Void.

watchdog_timeout ()

Sets the watchdog's timeout.

USAGE:

```
int watchdog_timeout(int t);
```

DESCRIPTION:

Sets the watchdog's timeout or retrieves the timeout value from the watchdog. If used to set the timeout, it should be called before the watchdog is enabled with the watchdog_enable() function. The timeout is specified in 1 second intervals. Reset is guaranteed to be no less than the time set.

To retrieve the timeout value from the watchdog, a -1 should be passed as a parameter.

PARAMETERS:

int t	Timeout in seconds (valid range: 1 – 63).
-------	---

RETURN:

If a valid time is passed, it returns the watchdog's timeout value prior to the timeout change.

If a -1 is passed as the argument, the current watchdog's timeout is returned.

J1708 - JBUS Device

JBUS Device

The JBUS device looks at all messages on the J1708 bus. Messages with multiple PID's are broken into smaller messages so that they conform to the following format:

1. Message Identification Character (MID).
2. Parameter Identification Character (PID).
3. PID Data Characters.

Message filtering resides on the J1708 controller. Up to 30 messages may be stored in the filter table on the controller. This avoids excessive interrupts and processing burden on the main processor.

When the J1708 port is opened, the filter table is empty and the default is in “allow” mode. As messages are added to the filter, any matches received on the J1708 data bus will be passed to the main processor and begin filling the buffer on the main processor, which can hold 2000 messages. `J1708_get_message` returns the first message received.

Each message MID and PID is compared against the filter table and if a match is found, depending on the type of filter table, one of the following occurs:

- If the table type is `FILTER_DISABLE`, then all messages present on the J1708 data bus are passed to the main processor message buffer.
- If the table type is `FILTER_ALLOW`, then only messages with their MID and PID in the table are passed to the main processor message buffer.
- If the table type is `FILTER_DENY`, then only the messages with their MID and PID in the table are ignored and the rest are passed to the main processor message buffer.

Programming Considerations

Opening/Closing the J1708 Port

Opening the J1708 port initializes communication between the main processor and the J1708 controller. All parameters related to J1708 must be reset when the port is opened. Closing the port clears the filter table, settings, and buffer.

Buffer

Once the J1708 port is opened and the filter mode is set, data that is received by the J1708 controller and passes the filter settings is sent to the main processor. Messages are buffered, whether or not they are read. A read of the data may return “old” data values.

For time critical data, it is recommended to purge the buffer using `j1708_rxpurge()` or use `j1708_get_specific_message()`.

Printf and J1708

The `printf` or `sys_printf` functions format and send messages on the console port. These functions will cause additional delay and may slow operation of the J1708 functions.

J1708 API

j1708_add_filter ()

DESCRIPTION:

Adds a MID,PID pair to the filter table. It is possible to add PID's that have values greater than 255. Filters are added to the next available table position.

USAGE:

```
int j1708_add_filter(int mid, int pid);
```

PARAMETERS:

int mid	Message ID (MID). It is possible to add a MID wildcard for filtering. MID = 0 (zero) allows any specified PID to pass the filter, regardless of the MID of the message.
int pid	Process ID (PID).

RETURN:

Returns the index in the filter table where the (MID,PID) pair has been saved; otherwise -1 if it failed to add the filter.

j1708_close ()

DESCRIPTION:

Powers down the JBUS device. After calling this function, all communications stops and the filter table is erased.

USAGE:

```
void j1708_close(void);
```

j1708_find_filter ()

DESCRIPTION:

Locates a filter in the filter table and returns its index.

USAGE:

```
int j1708_find_filter(int mid, int pid);
```

PARAMETERS:

int mid	Message ID. It is possible to add a MID wildcard for filtering. MID = 0 (zero) allows any specified PID to pass the filter, regardless of the MID of the message.
int pid	Process ID.

RETURN:

Returns the filter entry index (0-29) if found, 254 for BAD DATA input values, and 255 for no communication with J1708 controller.

j1708_get_errors ()

DESCRIPTION:

Reads the errors from the J1708 system. Used for engineering debug and hardware troubleshooting only. Errors do not indicate an application problem or API failure.

USAGE:

```
int j1708_get_errors(int clear_errors);
```

PARAMETERS:

int clear_errors	1 = clear the errors after reading 0 = does not clear the errors
------------------	---

RETURN:

Returns a 32-bit number describing the errors in a bit field. See j1708.h for the possible errors.

j1708_get_filter ()

DESCRIPTION:

Reads the filter values out of the given position from the filter table. This function can be used to read the contents of the whole filter table by incrementing through each table position.

USAGE:

```
int j1708_get_filter(int pos, int *mid, int *pid);
```

PARAMETERS:

int pos	Position of table to retrieve filter values.
int *mid	Pointer to a location to store the MID value from the filter table.
int *pid	Pointer to a locaton to store the PID value from the filter table.

RETURN:

Returns the table position (0 – 29) if the filter was found, 254 = end of table (BAD_DATA) and -1 = No Communication.

Example:

```
// Read filter table until empty and print the values
for(pos = 0; pos<100; pos++)
{
    chkpos = j1708_get_filter(pos, &mid, &pid);

    if(chkpos == 254)
    {
        printf("Done.\n");
        return;
    }

    if(chkpos == -1)
    {
        printf("COMMUNICATION_FAILED.\n");
        return;
    }

    printf("Pos %d contains MID %d PID %d.\n",chkpos,mid,pid);
}
```

j1708_get_message ()

DESCRIPTION:

Retrieves a message from the message queue. The message queue is a circular buffer of 2048 messages. The FIFO discards old data when it overflows. (Note: the previous buffer would discard new messages when it was full.)

USAGE:

```
int j1708_get_message(int *mid, void *data);
```

PARAMETERS:

int *mid	Pointer to a location where the message MID is saved.
void *data	Pointer to a location where the message data is saved. This location should be large enough to receive the largest message expected. Note: There are no checks for overflowing the data space and the first byte of the data is the PID.

RETURN:

Returns the number of bytes saved in the data space pointed by the “data” parameter or -1 if no message is available.

j1708_get_specific_message ()

DESCRIPTION:

Searches through the last 2048 messages and returns the newest message that meets the specified MID, PID combination. This function always searches through the entire 2048 position buffer, even if the messages have been read out of the buffer by j1708_get_message(). To clear the buffer for j1708_get_specific_message() use j1708_rxpurge().

USAGE:

```
int j1708_get_specific_message(int *mid, int pid, void
*data);
```

PARAMETERS:

int *mid	Pointer to a location where the message MID to search for. It is possible to use a MID wildcard for searching. MID = 0 (zero) allows any specified PID to pass the search, regardless of the MID of the message.
int pid	The PID to search for.
void *data	Pointer to a location where the message data is saved. This location should be large enough to receive the largest message expected (currently 20 bytes). Note: There are no checks for overflowing the data space.

RETURN:

Returns the number of bytes saved in the data space pointed by the “data” parameter or -1 if no message is found.

Example Code:

```
int mid, pid, len;
unsigned char data[20]

//set data to search for
mid = 0;
pid = 84;
len = j1708_get_specific_message(&mid, pid, data);
```

Example Output:

When it returns and len is > 0, the data might look like

```
len = 1
mid = 128
pid = 84 (This was not modified, but since the function succeeded,
it is the pid returned.)
data[0] = data value
```

Note: The j1708_get_specific_message() was designed to be used with the “allow” filter mode. There must be a filter that matches what is being requested with this function. For example, to use this function to search for 128, 84, there must be a filter of 128, 84 or 0, 84.

j1708_get_version ()

DESCRIPTION:

j1708_get_version returns the version number of the j1708 controller firmware and the library the application was compiled with. The function is only effective as of software release SW0149-08 and later.

The version is a null terminated character string. The pointer should be declared as a 15 byte array.

USAGE:

```
int j1708_get_version(char *v);
```

PARAMETERS:

char *v	Pointer to a 15 byte array.
---------	-----------------------------

RETURN:

Return 0 is successful, otherwise no version data is available.

Example Code:

```
char data[15];
if(j1708_get_version(data) == 0)
    printf("J1708 Processor Ver: %s\n", data);
else
    printf("J1708 Processor Ver: Unknown\n");
```

Example Output:

J1708 Processor Ver: J3.0 L3.0

j1708_open ()

DESCRIPTION:

Powers on the JBUS device and instructs it to connect to the JBUS. The filter type is cleared and set to FILTER_ALLOW. Since the table is empty, no messages will be passed to the application until a MID, PID pair is entered or the filter type is changed. This function should be called before any other J1708 functions or modules.

USAGE:

```
int j1708_open(void);
```

RETURN:

Returns 0 (zero) if successful; otherwise -1.

j1708_remove_filter ()

DESCRIPTION:

Removes a MID, PID pair from the filter table, or if the index is 255, it clears the table. When filters are removed from the middle of the filter table, the filters are renumbered sequentially so that there are no vacant entries except at the end of the table.

USAGE:

```
int j1708_remove_filter(int index);
```

PARAMETERS:

int index	Index to the filter table obtained when the filter was added. FILTER_REMOVE_ALL = clears the filter table.
-----------	---

```
int j1708_remove_filter(FILTER_REMOVE_ALL); clears the filter table.
```

RETURN:

Returns 0 (zero) if successful; otherwise -1 if failed to communicate, or BAD_DATA if the index is invalid.

j1708_rxpurge()

DESCRIPTION:

Clears the receive message buffer.

USAGE:

```
int j1708_rxpurge(void);
```

RETURN:

Returns 0 (zero) if successful; otherwise -1.

j1708_send_message ()

DESCRIPTION:

Sends a message to a JBUS module through the JBUS using J1708 protocol.

USAGE:

```
int j1708_send_message(int mid, int priority, void  
*data, int length);
```

PARAMETERS:

int mid	Message ID.
int priority	Message priority (1-8 with 8 being the lowest priority).
void *data	Pointer to message contents. The first byte of the data is the PID.
int length	Number of bytes in the data portion of the message.

RETURN:

Returns 0 (zero) if message was successfully sent; otherwise -1.

j1708_set_filter_type ()

DESCRIPTION:

Specifies the behavior of the filter.

USAGE:

```
int j1708_set_filter(int type);
```

PARAMETERS:

int type	Filter type constant. The type can be one of the following filter type constants:	
	FILTER_DISABLE	Disables the filter (all messages are passed).
	FILTER_ALLOW	Only allow the messages listed to pass.
	FILTER_DENY	Only deny messages listed from passing.

RETURN:

Returns 0 (zero) if successful; otherwise -1 if failed to communicate, or BAD_DATA if the type is invalid.

Manual History

57-5102-01A March, 2007

Initial release of the 57-5102-01A CT6 Series Software Development Manual.

57-5102-02A May, 2007

Changes made in the manual correspond to changes made to the operating system and obc605 libraries and header files. Release -06 (TOS version 1.6).

1. Added information about using J1708.
2. Replaced com2_ctrlrts with int com2_autorts.
3. Added int j1708_rxpurge(void).
4. Removed the AddCommand() and tfsrun() commands. These are no longer supported.
5. Updated information related to the watchdog timer. Watchdog timer resolution is 1 second.

57-5102-03A November, 2007

Changes made in the manual correspond to changes made to the operating system and obc605 libraries and header files. Release -10 (TOS version 1.7)

1. Added j1708_get_version.
2. Added j1708_get_specific message.
3. Added j1708_get_filter.
4. Added j1708_get_errors.
5. Added additional information to the j1708_get_message(), j1708_add_filter(), j1708_find_filter(), and j1708_remove_filter().

57-5102 Rev E April, 2009

Changed to Dyacon name.