

**PC\_UNOS**

**A Real-Time Operating System for PCs**

**Reference Manual**

# **PC\_UNOS**

## **Reference Manual**

**Prepared for TUNRA Ltd**

**by**

**MUVELIN**

Some information contained in this Manual has been drawn from:-

UNOS User Manual: Authors R Betts & L Sciacca  
Design documents and source code owned by TUNRA Ltd  
PC\_UNOS source code prepared by MUVELIN  
Borland® C++ user documents

Editor: DJL Brown BE JSSC

July, 1999

Copyright © 1999. by TUNRA Ltd. All rights reserved.

® Borland is a registered trademark of Borland International, Inc

# CONTENTS

<b>1. AMENDMENTS</b>	<b>5</b>	current_key	33
<b>2. INTRODUCTION</b>	<b>7</b>	current_page	34
<b>3. SYSTEM REQUIREMENTS</b>	<b>7</b>	current_window	34
<b>4. HOW TO USE PC_UNOS</b>	<b>7</b>	draw_bar	34
4.1. Access to Functions	7	draw_box	35
4.2. Required Sequence of Events	7	draw_line	35
4.3. Tasks	8	get_screen	35
4.4. Task Monitoring	8	get_window	36
4.5. Serial Communications	8	grid_columns	36
4.6. The Screen System	8	grid_rows	37
4.6.1. Pages	9	go_to_xy	37
4.6.2. Windows	9	out_text	37
4.6.3. Positioning Objects	9	out_text_xy	37
4.6.4. Editable Items	10	put_intersection	38
<b>5. DEBUG FACILITIES</b>	<b>13</b>	release_screen	38
5.1. Stack Check	13	release_zone	39
5.2. Memory Checks	13	set_grid	39
<b>6. THE LIBRARY</b>	<b>15</b>	set_key	39
INITIALISATION	15	show_for_change	39
create_graphics_page	15	SEMAPHORES and TIMING	41
create_graphics_text	15	create_semaphore	41
create_graphics_window	16	create_timer	41
create_serial_channel	18	init_semaphore	41
create_task	19	reset_timer	41
create_text_page	19	return_semaphore_value	42
create_text_window	20	_signal	42
GoUNOS	21	start_timer	42
InitPC_UNOS	21	stop_timer	43
PC_UNOSRevision	21	timed_wait	43
set_as_first_page	22	wait	44
HARDWARE-RELATED FUNCTIONS	23	SERIAL COMMUNICATIONS	45
disable_int_mask	23	prog_16450_uart	45
enable_int_mask	23	reset_comms_hardware	45
return_interrupt_status	23	get_serial_errors	45
Set_vector	23	reset_serial_errors	46
MEMORY MANAGEMENT	25	TASK COMMUNICATIONS	47
ret_free_mem	25	connect_to_keyboard	47
ucalloc	25	flush_mbx	47
umalloc	25	free_mbx	47
ufree	25	rcv_mess	48
SCHEDULING & TASK MANAGEMENT	27	send_mess	48
at_exit	27	send_qik_mess	48
change_task_priority	27	size_mbx	49
chg_base_ticks_per_time_slice	27	size_mbx_mess	49
chg_task_tick_delta	27	used_mbx	49
disable_task_switch	28	APPENDIX A FUNCTIONS INDEX	51
enable_task_switch	28	APPENDIX B GLOBAL VARIABLES	53
exitUNOS	28	B1. UNOS CONTROL	53
UNOSTasksRunning	29	B2. SERIAL CHANNEL CONTROL	55
preemptive_schedule	29	B2.1 Hardware Settings	55
reschedule	30	B2.2 Software Settings	55
rtn_current_task_name_ptr	30		
rtn_current_task_num	30		
rtn_task_priority	30		
start_time_slice	31		
stop_time_slice	31		
task_switch_enabled	31		
SCREEN OPERATIONS	33		
clear_window	33		
c_printf	33		

**THIS PAGE IS INTENTIONALLY LEFT BLANK**



**THIS PAGE IS INTENTIONALLY LEFT BLANK**

## 2. INTRODUCTION

- 2.1. **PC\_UNOS** - is based on **UNOS - University of Newcastle Operating System**. The complete UNOS system is included in the PC\_UNOS library, and provides the whole basis of real-time operations for PC\_UNOS.
- 2.2. The PC\_UNOS package adds a complete screen, keyboard and serial channel interface to UNOS. However, unlike UNOS, these are specific to the x86 processor and IBM-compatible **Personal Computer (PC)**.
- 2.3. PC\_UNOS is written in the C language, and has been developed in the Borland C environment.
- 2.4. This manual assumes that the reader has a working knowledge of C and the standard PC arrangement. It describes, in outline, the use of PC\_UNOS, and provides details of all of its available functions.

## 3. SYSTEM REQUIREMENTS

- 3.1. PC\_UNOS and the included version of UNOS are designed for use on the following platform:-
  - a. 80n86 processor with coprocessor or with in-built coprocessor. Clock speed of 25 MHz or greater is recommended.
  - b. IBM-compatible chip set.
  - c. At least 640 Kilobytes of RAM.
  - d. At least 300 Kbytes of program disk space.
  - e. MS-DOS 4 or above, installed.
- 3.2. The installed PC\_UNOS requires 125 Kbytes of disk space.

## 4. HOW TO USE PC\_UNOS

### 4.1. Access to Functions

- 4.1.1. PC\_UNOS is provided as a library and a set of C-language header files. The complete list of PC\_UNOS files is given in Table 1.
- 4.1.2. To use any PC\_UNOS function, include the PC\_UNOS header files which list the function prototype, PC\_UNOS global variables and structure definitions required, in the appropriate source code file. During construction of the executable, link in the PC\_UNOS library.

**TABLE 1**  
**PC\_UNOS FILES**

FILENAME	FUNCTION
pc_unos.lib	Contains all PC_UNOS functions in a C library.
keyboard.h	Defines all character codes including non-printable and extended ASCII characters (NOT the keyboard scan codes) and contains keyboard-related function prototypes. All Ctrl- and Alt- key combinations are defined, with appropriate names for use as constants ( eg CTRL_B is defined as 0x02 ).
pc_unos.h	Contains the prototype of the PC_UNOS initialisation function.
screen.h	Defines constants and structures, and provides prototypes of functions, all relating to screen operations.
serial.h	Defines constants and structures, and provides prototypes of functions, all relating to serial communications operations.
unos.h	Defines constants and structures, and provides prototypes of functions, all relating to UNOS operations.
unos_hw.h	Defines constants used by UNOS, the values of which are specific to PC hardware, and provides function prototypes for functions related to UNOS but which are PC hardware-specific.

### 4.2. Required Sequence of Events

- 4.2.1. Because UNOS is a multi-tasking, real-time system PC\_UNOS operates almost independently of DOS. It has certain memory allocation procedures which must be completed before UNOS takes charge, and therefore there is a strict sequence of events to be followed when the user's program is started:-
  - a. Set any PC\_UNOS and compiler global variables required to be changed from their default values.
  - b. Call InitPC\_UNOS() to initialise the PC\_UNOS system, UNOS and serial channels.
  - c. Create the tasks, display pages and windows needed.
  - d. Call GoUNOS(). This starts the UNOS multi-tasking system.

4.2.2. Once UNOS is started ( by GoUNOS() ), the program may be terminated with the Ctrl-Alt-Del or Ctrl-Alt-Home keyboard combinations. This DOES NOT reboot the computer. The call to GoUNOS() should be the last operation undertaken by main(), as any statements after it will not be executed.

4.2.3. Program termination should be effected by calling exitUNOS(). A call direct to the standard C function exit() will also work, but may leave the computer in an unusable state.

### 4.3. Tasks

4.3.1. All tasks run as independent programs. They communicate via the UNOS mailbox system and by functions provided by the user for the purpose of making available the value of, or allowing the changing of the value of variables whose scope is restricted to a task. (The use of universally global variables - other than those provided by PC\_UNOS - is discouraged for program development administration reasons, but is technically acceptable.)

4.3.2. PC\_UNOS initialisation ( by executing the function InitPC\_UNOS() ) automatically creates a screen task and a keyboard task. Their functions are, respectively:-

a. to schedule, and control access to, the screen via page display functions, and

b. to interpret keyboard entries and send the resulting character bytes to the screen task.

The call to InitPC\_UNOS() also creates any serial channel tasks required.

### 4.4. Task Monitoring

4.4.1. PC\_UNOS includes a facility for determining if any task is running. This is done by a special 'Task-Monitor' task which, by default, cycles once every 5 seconds. During each cycle of its operation, it checks the status of each PC\_UNOS task (except the keyboard task and itself), and stores an indicator for each in an array. The user may obtain a copy of this array by calling the function PCUNOSTasksRunning(), using the address of a 9-byte array as parameter. The array indicated is filled with flags which indicate whether the task defined for each position in the array has cycled in the last 5 seconds.

4.4.2. The Task-Monitor task also calls a function, once per cycle, declared in PC\_UNOS.H as:-

```
void UserTasksRunningCheck( void );
```

The user may define this function to do whatever is required. It is intended for use in checking the user's tasks, and taking any action necessary should a task stop. This function is defined within the PC\_UNOS library, to do nothing. Provided the user includes a definition of it in a file in the link list before the PC\_UNOS library, the linker will use the user's definition, not that in the library.

### 4.5. Serial Communications

4.5.1. PC\_UNOS provides for up to four serial channels, operating on standard PC ports.

4.5.2. Each serial channel appears to the user's program as two tasks - a 'send' task and a 'receive' task.

The 'send' tasks are identified by their names:-

```
ch_0_tx; ch_1_tx; ch_2_tx, and ch_3_tx.
```

Similarly, the 'receive' tasks are named:-

```
ch_0_rx; ch_1_rx; ch_2_rx, and ch_3_rx.
```

Each receive task may be 'told' to send its output to a particular user's task by that task sending a message (which may be simply a character) to the nominated receive task, for instance:-

```
send_mess( "?", 1, ch_1_rx, );
```

will cause serial channel 1 to send any characters received, to the task which executes the above statement. Serial receive tasks wait indefinitely for such a message. Once it is received, the task operates, and is locked to the task which sent the above message.

Character strings are sent to a serial channel for transmission just as any message is sent to any other task:-

```
send_mess( pointer_to_char_string, sizeof( char_string ), ch_1_tx, );
```

### 4.6. The Screen System

4.6.1. The screen system provided by PC\_UNOS is intended to control all access to the PC's VDU screen, and supports text and graphics modes. The system is controlled by one PC\_UNOS task - the screen task - and provides for up to 24 user-created display 'pages', each of which occupies the entire screen (not including the top character line). Each page may contain any number of text 'windows', and each 'window' may contain any number of 'items' of editable data.

4.6.2. PC\_UNOS's screen system operates in standard 80x25 text mode or 640x480 VGA graphics mode.

4.6.3. The top line of the screen is reserved for use by the screen task, to display PC\_UNOS' **Action Prompt Bar**. This contains three directives:-

Page Menu: Ctrl-M

Go Back: Ctrl-B

Quit: Ctrl-Alt-Del

These directives indicate the key-stroke combinations required to perform the indicated functions:-

Alt-M	brings down a menu of available pages, on which selection of a page is indicated by a reverse video color cursor, moveable with the up-down cursor keys;
Alt-B	returns the display to the previously displayed page, and
Ctrl-Alt-Del	terminates the program (but does not re-boot the computer).

Whereas it is possible to overwrite the Action Prompt Bar, this is not recommended. The Action Prompt Bar is full page width, and in text mode is one row deep. In graphics mode it is 18 pixels deep - approximately twice the depth of the default graphics text.

#### 4.6.1. Pages

4.6.1.1. The creation of a display 'page' requires the definition of a function which will accept any keyboard-generated character via a single 'unsigned char' parameter. Normally it would contain all the operations necessary to display the whole page, but it is possible write to the screen from outside the page-defining function.

4.6.1.2. Pages are displayed in the menu in ascending alphabetical order, top to bottom.

4.6.1.3. A page-defining function nominated in a page creation must NOT be called by the user. The screen task will call the function as required.

4.6.1.4. When a new page is selected for display, the screen task hands its defining function a CTRL\_P character. This may then be used as an indication of a first-call - for instance to write a background which does not then have to be re-written during every subsequent call to the function.

4.6.1.5. The screen system thus reserves the three 'characters' ALT\_M, ALT\_B (for the Action Prompt Bar) and CTRL\_P. Their use for other purposes should be avoided.

#### 4.6.2. Windows

4.6.2.1. PC\_UNOS windows are simple windows. They do not contain scroll bars, title bars or any items associated with the 'MS Windows' GUI. The user may provide borders as required - they are not built-in. Once defined, a window cannot be moved.

4.6.2.2. It is good discipline to create a 'whole-page' window for each display page. This window would extend in text mode from character coordinates (1,2) to (80,25) - remember that the top line is reserved for PC\_UNOS' Action Prompt Bar - and would be used for writing the background detail of a page. For graphics mode, the coordinates of the corners of the whole-page window will depend on a grid which is defined in the page-creation function call. In graphics mode, the PC\_UNOS Action Prompt Bar extends from pixel (0,0) to pixel (639,18). Smaller windows might then be used to contain detailed information, or the 'whole screen' window might be used on its own.

4.6.2.3. Each window created operates as a standard C text window or graphics viewport. It has its own colours, and positions within it are relative to the window's top left corner.

#### 4.6.3. Positioning Objects

4.6.3.1. In text mode, objects are positioned on the screen by text rows and columns, in the normal manner. In graphics mode, standard C functions provide positioning by pixel number. PC\_UNOS provides a more flexible and easier system. Within the screen system, a graphics-mode grid is maintained. This has an adjustable resolution, determined by the number of rows and columns per full screen, set by the user. When a graphics page is created (see create\_graphics\_page() ) the grid settings for the page are nominated. Similarly when a graphics window is created (see create\_graphics\_window() ) the grid settings for that window are nominated. Another function - set\_grid() - allows the current grid settings to be changed at any time.

4.6.3.2. Note that the graphics-mode grid settings are the number of rows and columns per **whole** page. This means that, for instance, if 80 columns are set for a window's grid, and the window is half a page width, 40 columns will exist inside the window.

4.6.3.3. In graphics mode, the origin for the grid in a page (or whole-page sized window) is pixel (0,0) - top left-hand corner of the page. The origin in a graphics mode window is again pixel (0,0) - the top left-hand corner of the window. Thus, if say 80 columns were set for the grid in a page, they would be referred to as column 0 to 79 - NOT column 1 to 80, as in text mode.

4.6.3.4. This graphics-mode grid is only a means of positioning objects - it does not reduce the drawing resolution. Thus if a grid of say 80 columns and 50 rows was set, the origin of an object could be located at one of 80 horizontal positions and 50 vertical positions, but the object itself is drawn with a resolution of one pixel in both directions, and thus does not necessarily have width and depth which are integer multiples of column width and row height.

4.6.3.5. The default graphics text is based on an 8x8 pixel matrix for each character, thus a convenient grid setting for graphics pages and windows which carry a lot of text is 50 rows by 80 columns - approximating the text-mode resolution in the horizontal, and just over one character height for each row in the vertical.

#### 4.6.4. Editable Items

4.6.4.1. PC\_UNOS' **SHOW-FOR-CHANGE** system provides for the selection and editing of data on-screen with very simple programming. SHOW-FOR-CHANGE uses one PC\_UNOS function specifically designed for this purpose - *show\_for\_change()*. Details of each item - its location in a window, which window it is in, and information about the variable, are registered during the first two executions of the page-defining function, after the page is selected. Thus *show\_for\_change()* should only be used within a display page-defining function, and should be called between a *get\_window()* and *release\_screen()* pair of calls. It cannot be used outside a page-defining function.

4.6.4.2. Each call to *show\_for\_change()* displays an item on the screen, and makes the item a candidate for selection and editing. The currently selected item is shown in reverse-video colours ie the window's background colour is used for the characters and the window's foreground colour becomes the background colour for each character of the selected item. (Note that in text mode, if the window's foreground colour is 'bright', in reverse video the background will be the equivalent normal colour, and will flash.)

4.6.4.3. The item selection is changed within a window by use of the cursor motion arrow keys (page up and page down are not effective), and the window containing the selected item may be changed by use of the Tab key. Movement around the windows is cyclic, and the order is the same as the order in which the windows are 'opened' by *get\_window()* calls within the page-defining function. The first window 'opened' will contain the initially selected item whenever the page is selected. If there is only one window containing editable items on the current page, a tab key-stroke will be ignored.

4.6.4.4. Item selection changes within a window are in order by location. The selected item when the window is 'entered' will always be that which is identified by the first call to *show\_for\_change()* within the *get\_window()* 'if' statement. The cursor moves - one item for each arrow key-stroke - in the direction of the pressed arrow key. The cursor will move to the next item with the same y-location for right/left movement, or x-location for up/down movement. If another item is not located at the same x-location as the current item, an up or down request will result in selection of the left-most item in the next occupied y-location which is above or below the current one, respectively. If no new selection is possible, a movement request will be ignored.

4.6.4.5. Editing of the selected item is carried out in the normal manner, with appropriate key-strokes. The value of the item is displayed according to the format provided in the call to *show\_for\_change()*, in the display field. As soon as an entry is started, the displayed item value is blanked out and the entered keys are displayed instead, left-justified in the display field. These entries may be edited by use of the back-space key and retyping, until an Enter key-stroke is made. Key entries incompatible with the format specified are automatically discarded, and the bell sounded. The Enter key causes the new value typed in to be set, and displayed according to the format provided in the call to *show\_for\_change()*. Until the Enter key is pressed, the old value is retained and may be re-displayed by moving to another editable item, or to another page, then back.

4.6.4.6. Whilst *show\_for\_change()* is primarily designed for editing values of displayed variables, it also allows for 'toggling'. When an item set up for this operation (see details in the **show\_for\_change** entry in Library) is selected, the only effective key is the space-bar. A space-bar entry causes *show\_for\_change()* to return a 'changed' indication, but the variable supplied to it (in this case normally a text string) is not changed. The 'changed' return can then be used to carry out some action, and perhaps change the text string to be displayed. Thus a switch with 2 or more conditions can be easily constructed. A 'do-nothing' condition is also allowed for, in which *show\_for\_change()* displays the variable according to the format provided, but no change can be made. This allows the programmer to dynamically change the capability to edit an item.

4.6.4.7. Finally, *show\_for\_change()* is provided with a condition in which it does not even display the variable. This condition is provided so that the item can be registered within the *show\_for\_change()* system without a display resulting. Thus a later change to the edit, toggle or display-only condition will result in normal *show\_for\_change()* facilities being available.

4.6.4.8. The following example illustrates the use of the **SHOW-FOR-CHANGE** facilities. Note the use of argument-type modifiers in the format string. These **must** always be provided when necessary to completely define the variable type - just as they are required by the standard scanf() functions.

```
void page_defining_fn( char key_char )
{
int x, changed;
unsigned int change_control;

    if( some criterion )
        change_control = EDIT;
```

```

else if( some other criterion )
    change_control = SHOW;
else
    change_control = REGISTER_ONLY;

if ( get_window (pointer_to_window_1) )
{
    x = 5;
    changed = show_for_change( x, 3, "%5s", charstring1, EDIT );

    changed |= show_for_change( x, 3, "%4s", charstring2, TOGGLE );

    if( changed )
        Do anything necessary if a change has taken place during the
        last 2 calls to show_for_change().

    changed = show_for_change( x, 5, "%3d", &intvar, EDIT );

    changed |= show_for_change( x, 9, "%6.2lf", &doublevar1, change_control );

    changed |= show_for_change(x+8,5, "%8.2lf", &doublevar2, change_control );

    release_screen();

    if( changed )
        Do anything necessary if a change has taken place during the
        last 3 calls to show_for_change().
}
}

```

**THIS PAGE IS INTENTIONALLY LEFT BLANK**

## 5. DEBUG FACILITIES

### 5.1. Stack Check

5.1.1. By a special addition to UNOS, as the UNOS kernel switches tasks it changes the value of the Borland C global variable `_stklen`, to match the size of the stack reserved for the new task.

5.1.2. The PC\_UNOS library is compiled with stack checking option of the Borland C compiler turned on. If the user's program is compiled with this option turned on, full stack overflow checking will be carried out at run-time. No trap for the stack overflow error is provided, so a crash will result if such an overflow occurs.

### 5.2. Memory Checks

5.2.1. If the PC\_UNOS global `_task_debug` is turned on ( set to 1) during initialisation of the UNOS global variables, a special monitoring facility is activated. At normal program termination, this facility will list stack length, maximum mailbox message length and maximum mailbox queue size encountered by each task, during the running of the program. The maximum amount of memory used from the UNOS memory pool will also be shown.

5.2.2. Note that the stack length reported is not necessarily the maximum stack length used, as the monitoring of stack length takes place at task switching time, for the task being switched out. The actual maximum stack length used may be somewhat in excess of the figure reported.

5.2.3. The memory use recorded is a total of memory allocations made, and does not include any unused memory left between allocations because of block size requirements. Thus the total amount of memory actually needed in the UNOS memory pool will be somewhat in excess of the figure reported. A ratio of 2:1 has proved to be generally adequate.

**THIS PAGE IS INTENTIONALLY LEFT BLANK**

## 6. THE LIBRARY

6.1. This section of the Manual is arranged in functional groups. A cross-reference to functions in alphabetical order is provided in Appendix A.

6.2. A listing of PC\_UNOS global variables is given in Appendix B, which also provides a list of PC\_UNOS-defined strings.

### INITIALISATION

#### **create\_graphics\_page**

---

<b>Function</b>	Creates a graphics-mode display page.						
<b>Syntax</b>	<pre>#include "screen.h" struct display_page *create_graphics_page( void *page_display_fn, char *page_name,  int grid_rows, int grid_columns );</pre>						
<b>Remarks</b>	Generates a package of information that the screen task uses to include the page in a pull-down menu list of all available display pages, and bring the page onto the screen when required. The information is contained in a display_page structure:-  <pre>struct display_page {     void * display_fn_ptr;     char * page_name_ptr;     struct display_page * next_page_ptr;     struct display_page * last_page_ptr;     char display_mode; // set to GRAPHICS.     int rows, columns; // set to the values of grid_rows, grid_columns. }</pre>						
<b>Parameters</b>	<table><tr><td>page_display_fn:</td><td>Pointer to a function which writes the display to the screen</td></tr><tr><td>page_name:</td><td>Pointer to a null-terminated character string containing the name of the page to be shown in the pull-down menu of pages. A maximum of 20 characters is allowed.</td></tr><tr><td>grid_rows , grid_columns:</td><td>Defines the number of columns in an invisible grid, which may then be used to set the positions of objects on the page. A value of 0 will set the maximum possible number - 480 and 640 respectively.</td></tr></table>	page_display_fn:	Pointer to a function which writes the display to the screen	page_name:	Pointer to a null-terminated character string containing the name of the page to be shown in the pull-down menu of pages. A maximum of 20 characters is allowed.	grid_rows , grid_columns:	Defines the number of columns in an invisible grid, which may then be used to set the positions of objects on the page. A value of 0 will set the maximum possible number - 480 and 640 respectively.
page_display_fn:	Pointer to a function which writes the display to the screen						
page_name:	Pointer to a null-terminated character string containing the name of the page to be shown in the pull-down menu of pages. A maximum of 20 characters is allowed.						
grid_rows , grid_columns:	Defines the number of columns in an invisible grid, which may then be used to set the positions of objects on the page. A value of 0 will set the maximum possible number - 480 and 640 respectively.						
<b>Return value</b>	Pointer to a display_page structure containing information about the page.						

#### **create\_graphics\_text**

---

<b>Function</b>	Creates a graphics-mode text style.
<b>Syntax</b>	<pre>#include "screen.h" struct graphics_text_info *create_graphics_text( int size, int hor_just, int vert_just,   int foreground_colour, int background_colour, int font);</pre>
<b>Remarks</b>	Generates a structure containing data about a graphics-mode text style which may be used when writing to the screen in a graphics-mode page. The data is held in a graphics_text_info structure:-

```

struct graphics_text_info
{
    int size;           Character height in current (invisible) grid rows.
    int charsize;      Borland-C character size.
    int horjust;       horizontal justification.
    int verjust;       vertical justification.
    Int fgcolour;      Foreground colour.
    Int bkcolour;      Background colour.
    Int font;          Font identifier.
}

```

**Parameters**

size: Vertical size of largest character, in current grid rows.  
hor\_just: Justification:- LEFT\_TEXT, RIGHT\_TEXT or CENTER\_TEXT.  
vert\_just: Justification:-TOP\_TEXT, BOTTOM\_TEXT or CENTER\_TEXT.  
foreground\_colour, background\_colour: The required colours according to the standard colour list:-

BLACK	0	DARKGRAY	8
BLUE	1	LIGHTBLUE	9
GREEN	2	LIGHTGREEN	10
CYAN	3	LIGHTCYAN	11
RED	4	LIGHTRED	12
MAGENTA	5	LIGHTMAGENTA	13
BROWN	6	YELLOW	14
LIGHTGRAY	7	WHITE	15

font: Which Borland-C font to use. Fonts provided via the PC\_UNOS library are:-  
DEFAULT\_FONT  
TRIPLEX\_FONT

Other fonts may be used, but will require registration during initialisation, and their code linked to the application - usually via .OBJ files created by the BorlandC utility BGIOBJ.

**Return value** Pointer to a graphics\_text\_info structure containing information about the text style.

### create\_graphics\_window

**Function** Creates a graphics-mode window.

**Syntax**

```

#include "screen.h"
struct window_info *create_graphics_window( int winleft, int wintop, int winright,
                                           int winbottom, unsigned char foreground_colour,
                                           unsigned char background_colour,
                                           int grid_rows, int grid_columns);

```

**Remarks** Generates a structure containing data about a graphics-mode window which may be used to write to the screen. The data is held in a window\_info structure:-

```

struct window_info
{
    int winleft, wintop, winright, winbottom;
    int grid_rows, grid_columns;
    char winbkground, winfrground,
    char display_mode; // set to GRAPHICS.
    int curx, cury, cursortype;
    char do_not_cover;
    struct win_data window_data;
}

```

The elements curx, cury hold the current location of the cursor within the window. (This information is used by the PC\_UNOS on-screen editing function show\_for\_change().)  
The element do\_not\_cover is set to FALSE (0) by default, but can be set or reset, for any particular window at any time, by direct action on the structure owned by the window in

question. When set, a call to `get_screen()` or `get_window()` nominating the window owning a structure in which `do_not_cover` is set, prevents any other window which has been defined to overlap, from writing to the screen - even after screen control by the window has been released. It 'freezes' the screen zone occupied by the window. A call to `release_zone()` is needed to 'unfreeze' the zone, and allow overlapping windows to write to the screen. (This capability is used, for instance, when the page menu - actually a window which overlaps others - is activated.)

The `win_data` structure stores data about editable items within the window.

**Parameters**

`winleft, wintop, winright, winbottom:` Coordinates of the top-left and bottom-right corners (relative to a full screen display) of the window, in terms of the current (invisible) grid. This grid is defined according to the parameters provided in the previous page creation action, or by the last execution of the function `set_grid()`.

`background_colour, foreground_colour:` The required colours according to the standard colour list:-

BLACK	0	DARKGRAY	8
BLUE	1	LIGHTBLUE	9
GREEN	2	LIGHTGREEN	10
CYAN	3	LIGHTCYAN	11
RED	4	LIGHTRED	12
MAGENTA	5	LIGHTMAGENTA	13
BROWN	6	YELLOW	14
LIGHTGRAY	7	WHITE	15

`grid_rows, grid_columns:` Defines the number of rows and columns in an invisible grid, which may then be used to set the positions of objects in the window. These values reflect the numbers of rows and columns which would be generated if the window covered a whole page - they do NOT indicate the numbers of rows and columns which will be generated within the window. A value of 0 will set the maximum possible number - 480 and 640 respectively.

**Return value** Pointer to a structure containing data about the window.

## create\_serial\_channel

---

<b>Function</b>	Create a serial channel system.																																																
<b>Syntax</b>	<pre>#include "serial.h" int create_serial_channel( int port_number, unsigned int baud_rate,                           float stop_bits, char data_bits, char parity, char xonxoff,                           unsigned int command_reg_address,                           unsigned int data_reg_address, unsigned char intr_number,                           unsigned char tx_enable_disable_bit,                           unsigned char intr_ctrl_reg_offset,                           unsigned int rx_int_buffer_length,                           unsigned int tx_int_buffer_length,                           unsigned int buffer_lower_limit, unsigned int buffer_upper_limit,                           unsigned int mbx_q_size, unsigned int mbx_mess_size,                           char number_of_8259s, unsigned int address_of_8259_master,                           unsigned int address_of_8259_slave, char * rx_taskname,                           char * tx_taskname, unsigned char rx_task_priority,                           unsigned int tx_task_priority );</pre>																																																
<b>Remarks</b>	This function will not usually be required by the user. It is called by InitPCOS() to generate each required serial communications channel, and is provided for the programmer's use where special hardware arrangements are being serviced.																																																
<b>Parameters</b>	<table><tr><td>port_number:</td><td>Hardware port for this channel.</td></tr><tr><td>baud_rate:</td><td>As required.</td></tr><tr><td>stop_bits:</td><td>As required.</td></tr><tr><td>data_bits:</td><td>As required.</td></tr><tr><td>parity:</td><td>For odd parity:- 0. Even parity:- 1. No parity: 2.</td></tr><tr><td>xonxoff:</td><td>Channel is to operate software handshake - YES or NO.</td></tr><tr><td>command_reg_address:</td><td>UART's control register's I/O address.</td></tr><tr><td>data_reg_address:</td><td>UART's data register's I/O address.</td></tr><tr><td>intr_number:</td><td>UART's received data interrupt number.</td></tr><tr><td>tx_enable_disable_bit:</td><td>UART's tx interrupt enable bit in its control byte.</td></tr><tr><td>intr_ctrl_reg_offset:</td><td>UART's control register offset.</td></tr><tr><td>rx_int_buffer_length:</td><td>Receive task's data buffer size (bytes).</td></tr><tr><td>tx_int_buffer_length:</td><td>Transmit task's data buffer size (bytes).</td></tr><tr><td>buffer_lower_limit:</td><td>Used with software handshaking to send signal to the data source to restart transmission.</td></tr><tr><td>buffer_upper_limit:</td><td>Used with software handshaking to send signal to the data source to cease transmission.</td></tr><tr><td>mbx_q_size:</td><td>tx and rx tasks' UNOS mailbox queue lengths.</td></tr><tr><td>mbx_mess_size:</td><td>tx and rx tasks' UNOS mailbox envelope size.</td></tr><tr><td>number_of_8259s:</td><td>1 or 2, depending on the hardware.</td></tr><tr><td>address_of_8259_master:</td><td>I/O address of the master 8259.</td></tr><tr><td>address_of_8259_slave:</td><td>I/O address of the slave 8259 (if any). Ignored if there is only one.</td></tr><tr><td>rx_taskname:</td><td>Address of a char string identifying the port's channel 1 rx task.</td></tr><tr><td>tx_taskname:</td><td>Address of a char string identifying the port's channel 1 tx task.</td></tr><tr><td>rx_task_priority:</td><td>UNOS priority to be assigned to the receive tasks.</td></tr><tr><td>tx_task_priority:</td><td>UNOS priority to be assigned to the transmit tasks.</td></tr></table>	port_number:	Hardware port for this channel.	baud_rate:	As required.	stop_bits:	As required.	data_bits:	As required.	parity:	For odd parity:- 0. Even parity:- 1. No parity: 2.	xonxoff:	Channel is to operate software handshake - YES or NO.	command_reg_address:	UART's control register's I/O address.	data_reg_address:	UART's data register's I/O address.	intr_number:	UART's received data interrupt number.	tx_enable_disable_bit:	UART's tx interrupt enable bit in its control byte.	intr_ctrl_reg_offset:	UART's control register offset.	rx_int_buffer_length:	Receive task's data buffer size (bytes).	tx_int_buffer_length:	Transmit task's data buffer size (bytes).	buffer_lower_limit:	Used with software handshaking to send signal to the data source to restart transmission.	buffer_upper_limit:	Used with software handshaking to send signal to the data source to cease transmission.	mbx_q_size:	tx and rx tasks' UNOS mailbox queue lengths.	mbx_mess_size:	tx and rx tasks' UNOS mailbox envelope size.	number_of_8259s:	1 or 2, depending on the hardware.	address_of_8259_master:	I/O address of the master 8259.	address_of_8259_slave:	I/O address of the slave 8259 (if any). Ignored if there is only one.	rx_taskname:	Address of a char string identifying the port's channel 1 rx task.	tx_taskname:	Address of a char string identifying the port's channel 1 tx task.	rx_task_priority:	UNOS priority to be assigned to the receive tasks.	tx_task_priority:	UNOS priority to be assigned to the transmit tasks.
port_number:	Hardware port for this channel.																																																
baud_rate:	As required.																																																
stop_bits:	As required.																																																
data_bits:	As required.																																																
parity:	For odd parity:- 0. Even parity:- 1. No parity: 2.																																																
xonxoff:	Channel is to operate software handshake - YES or NO.																																																
command_reg_address:	UART's control register's I/O address.																																																
data_reg_address:	UART's data register's I/O address.																																																
intr_number:	UART's received data interrupt number.																																																
tx_enable_disable_bit:	UART's tx interrupt enable bit in its control byte.																																																
intr_ctrl_reg_offset:	UART's control register offset.																																																
rx_int_buffer_length:	Receive task's data buffer size (bytes).																																																
tx_int_buffer_length:	Transmit task's data buffer size (bytes).																																																
buffer_lower_limit:	Used with software handshaking to send signal to the data source to restart transmission.																																																
buffer_upper_limit:	Used with software handshaking to send signal to the data source to cease transmission.																																																
mbx_q_size:	tx and rx tasks' UNOS mailbox queue lengths.																																																
mbx_mess_size:	tx and rx tasks' UNOS mailbox envelope size.																																																
number_of_8259s:	1 or 2, depending on the hardware.																																																
address_of_8259_master:	I/O address of the master 8259.																																																
address_of_8259_slave:	I/O address of the slave 8259 (if any). Ignored if there is only one.																																																
rx_taskname:	Address of a char string identifying the port's channel 1 rx task.																																																
tx_taskname:	Address of a char string identifying the port's channel 1 tx task.																																																
rx_task_priority:	UNOS priority to be assigned to the receive tasks.																																																
tx_task_priority:	UNOS priority to be assigned to the transmit tasks.																																																
<b>Return value</b>	1 if successful, 0 otherwise.																																																

## create\_task

---

<b>Function</b>	To create a UNOS task.																								
<b>Syntax</b>	<pre>#include "unos.h" int create_task( char * task_name_ptr, unsigned char task_priority, int task_tick_delta,                 unsigned char task_status, unsigned char q_type,                 unsigned int semaphore_number, unsigned int task_stack_size,          unsigned int mess_q_size, unsigned int mess_size, void( *init_task)(void),                 void(*task)(void), void *local_var_ptr);</pre>																								
<b>Remarks</b>	This function carries out all the operations necessary for the generation of a new task to run under UNOS control. (Separate code is required to define the task function.)																								
<b>Parameters</b>	<table><tr><td>task_name_ptr:</td><td>Pointer to a text string, which will be used to identify the task.</td></tr><tr><td>task_priority:</td><td>Controls the proportion of CPU time which the UNOS kernel allocates to the task. 1 to 7. 1 is the highest priority.</td></tr><tr><td>task_tic_delta:</td><td>Variation allowed in time slice ticks for this task.</td></tr><tr><td>task_status:</td><td>May be TASK_BLOCKED or TASK_RUNNABLE.</td></tr><tr><td>q_type:</td><td>The type of queue on which the task will be initially placed. May be PRIORITY_Q_TYPE, SEMAPHORE_Q_TYPE, or DONOT_Q_TYPE.</td></tr><tr><td>semaphore_number:</td><td>Effective if q_type is a semaphore queue.</td></tr><tr><td>task_stack_size:</td><td>Size of stack to be allocated to this task, in bytes.</td></tr><tr><td>mess_q_size:</td><td>Mailbox queue length to be allocated to this task.</td></tr><tr><td>mess_size:</td><td>Mailbox envelope size to be allocated to this task.</td></tr><tr><td>init_task:</td><td>Pointer to the function that will initialise this task before create_task() completes this call. May be NULL.</td></tr><tr><td>task:</td><td>Pointer to the function which defines this task.</td></tr><tr><td>local_var_ptr:</td><td>Pointer to a data structure for use by the task function. It is passed to the task function as its void * parameter when it starts. May be NULL.</td></tr></table>	task_name_ptr:	Pointer to a text string, which will be used to identify the task.	task_priority:	Controls the proportion of CPU time which the UNOS kernel allocates to the task. 1 to 7. 1 is the highest priority.	task_tic_delta:	Variation allowed in time slice ticks for this task.	task_status:	May be TASK_BLOCKED or TASK_RUNNABLE.	q_type:	The type of queue on which the task will be initially placed. May be PRIORITY_Q_TYPE, SEMAPHORE_Q_TYPE, or DONOT_Q_TYPE.	semaphore_number:	Effective if q_type is a semaphore queue.	task_stack_size:	Size of stack to be allocated to this task, in bytes.	mess_q_size:	Mailbox queue length to be allocated to this task.	mess_size:	Mailbox envelope size to be allocated to this task.	init_task:	Pointer to the function that will initialise this task before create_task() completes this call. May be NULL.	task:	Pointer to the function which defines this task.	local_var_ptr:	Pointer to a data structure for use by the task function. It is passed to the task function as its void * parameter when it starts. May be NULL.
task_name_ptr:	Pointer to a text string, which will be used to identify the task.																								
task_priority:	Controls the proportion of CPU time which the UNOS kernel allocates to the task. 1 to 7. 1 is the highest priority.																								
task_tic_delta:	Variation allowed in time slice ticks for this task.																								
task_status:	May be TASK_BLOCKED or TASK_RUNNABLE.																								
q_type:	The type of queue on which the task will be initially placed. May be PRIORITY_Q_TYPE, SEMAPHORE_Q_TYPE, or DONOT_Q_TYPE.																								
semaphore_number:	Effective if q_type is a semaphore queue.																								
task_stack_size:	Size of stack to be allocated to this task, in bytes.																								
mess_q_size:	Mailbox queue length to be allocated to this task.																								
mess_size:	Mailbox envelope size to be allocated to this task.																								
init_task:	Pointer to the function that will initialise this task before create_task() completes this call. May be NULL.																								
task:	Pointer to the function which defines this task.																								
local_var_ptr:	Pointer to a data structure for use by the task function. It is passed to the task function as its void * parameter when it starts. May be NULL.																								
<b>Return value</b>	1 if successful, 0 if unsuccessful.																								

## create\_text\_page

---

<b>Function</b>	Creates a text-mode display page.
<b>Syntax</b>	<pre>#include "screen.h" struct display_page *create_text_page( void *page_display_fn, char *page_name );</pre>
<b>Remarks</b>	Generates a package of information that the screen task uses to include the page in a pull-down menu list of all available display pages, and bring the page onto the screen when required. The information is contained in a display_page structure:- <pre>struct display_page {     void * display_fn_ptr;     char * page_name_ptr;     struct display_page * next_page_ptr;     struct display_page * last_page_ptr;     char display_mode; // set to TEXT.     int rows, columns; // set to 80, 25. }</pre>



BLACK	0	DARKGRAY	8
BLUE	1	LIGHTBLUE	9
GREEN	2	LIGHTGREEN	10
CYAN	3	LIGHTCYAN	11
RED	4	LIGHTRED	12
MAGENTA	5	LIGHTMAGENTA	13
BROWN	6	YELLOW	14
LIGHTGRAY	7	WHITE	15

Note that the colours numbered 8 to 15 may be used ONLY for foreground.

**Return value** Pointer to a structure containing data about the window.

### GoUNOS

---

<b>Function</b>	Starts the UNOS time-slice operation.
<b>Syntax</b>	<code>#include "unos.h"</code> <code>void GoUNOS( void );</code>
<b>Remarks</b>	Until this function is called, the program is operating under DOS. After it is called, the program operates under UNOS.
<b>Parameters</b>	None.
<b>Return value</b>	None.

### InitPC\_UNOS

---

<b>Function</b>	To initialise the PC_UNOS system.
<b>Syntax</b>	<code>#include "pc_unos.h"</code> <code>void InitPC_UNOS( void );</code>
<b>Remarks</b>	Initialises UNOS, the screen, keyboard and serial channel systems using the values of the PC_UNOS global variables.
<b>Parameters</b>	None.
<b>Return value</b>	None.

### PC\_UNOSRevision

---

<b>Function</b>	Determine the Revision Number of the PC_UNOS library.
<b>Syntax</b>	<code>#include "pc_unos.h"</code> <code>char * PC_UNOSRevision( void );</code>
<b>Remarks</b>	PC_UNOS is a controlled library. Its Revision number is provided as a text string.
<b>Parameters</b>	None.

**Return value** Pointer to a character string representing the PC\_UNOS library Revision number.

### **set\_as\_first\_page**

---

**Function** To define which page shall initially have the screen.

**Syntax** #include "screen.h"  
int set\_as\_first\_page(struct display\_page \*pointer\_to\_page );

**Remarks** The pointer must have been initialised by a call to create\_display\_page(). The page owning the pointer will have the screen on UNOS startup. The function is effective only once.

**Parameters** pointer\_to\_page: Address of the display\_page structure of the page to initially have the screen.

**Return value** 0 if successful, -1 if unsuccessful (for instance on an inadvertent second call).

## HARDWARE-RELATED FUNCTIONS

### disable\_int\_mask

---

<b>Function</b>	To disable a particular interrupt.
<b>Syntax</b>	<pre>#include "unos_hw.h" void disable_int_mask( unsigned char interrupt_bit );</pre>
<b>Remarks</b>	Disables the interrupt controlled by interrupt_bit in the interrupt controller's control byte.
<b>Parameters</b>	interrupt_bit: The bit number (0-7) of the interrupt to be disabled.
<b>Return value</b>	None.

### enable\_int\_mask

---

<b>Function</b>	To enable a particular interrupt.
<b>Syntax</b>	<pre>#include "unos_hw.h" void enable_int_mask( unsigned char interrupt_bit );</pre>
<b>Remarks</b>	Enables the interrupt controlled by the nominated bit in the interrupt controller's control byte.
<b>Parameters</b>	interrupt_bit: The bit number (0-7) of the interrupt to be enabled.
<b>Return value</b>	None.

### return\_interrupt\_status

---

<b>Function</b>	To determine whether all interrupts are currently disabled.
<b>Syntax</b>	<pre>#include unos_hw.h" char return_interrupt_status( void );</pre>
<b>Remarks</b>	This function relates to the status of interrupt operations controlled by the standard C functions enable() and disable().
<b>Parameters</b>	None.
<b>Return value</b>	1 if interrupts are enabled, 0 otherwise.

### Set\_vector

---

<b>Function</b>	To determine whether all interrupts are currently disabled.
<b>Syntax</b>	<pre>#include unos_hw.h" Void set_vector( unsigned int interrupt, void interrupt (interrupt_routine)() );</pre>
<b>Remarks</b>	This function performs the same operationas the standard C function setvect(), but does not use DOS, and is therefore safe to use after UNOS' multi-tasking operation has started.
<b>Parameters</b>	interrupt:            The interrupt number to be affected.

interrupt\_routine: Address of the new interrupt handler function.

**Return value** Address of the old interrupt handler function.

## MEMORY MANAGEMENT

### ret\_free\_mem

---

<b>Function</b>	To determine the free memory in the UNOS heap.
<b>Syntax</b>	<pre>#include "unos.h" unsigned long ret_free_mem( void );</pre>
<b>Remarks</b>	A call to InitUNOS causes a block of memory (size set by the UNOS global <code>_memory_pool_size</code> ) to be reserved for use by UNOS. During various operations, some of this 'heap' may be temporarily or permanently allocated. This function tells the caller the amount of the UNOS heap which is presently unassigned.
<b>Parameters</b>	None.
<b>Return value</b>	Numbr of bytes of unassigned memory in the UNOS heap.

### ucalloc

---

<b>Function</b>	To allocate memory from the UNOS heap.				
<b>Syntax</b>	<pre>#include "unos.h" char huge * ucalloc( unsigned long number_of_objects, unsigned long size_of_object );</pre>				
<b>Remarks</b>	This function is similar to the standard C function <code>calloc()</code> . The difference is that the memory is allocated from the UNOS heap, which itself is obtained at initialisation - when operations are still under DOS control - from the DOS heap.				
<b>Parameters</b>	<table><tr><td><code>number_of_objects:</code></td><td>The number of objects that the allocated memory should be able to accommodate.</td></tr><tr><td><code>size_of_object:</code></td><td>Size of the objects to be accommodated.</td></tr></table>	<code>number_of_objects:</code>	The number of objects that the allocated memory should be able to accommodate.	<code>size_of_object:</code>	Size of the objects to be accommodated.
<code>number_of_objects:</code>	The number of objects that the allocated memory should be able to accommodate.				
<code>size_of_object:</code>	Size of the objects to be accommodated.				
<b>Return value</b>	On success, address of the lowest memory byte in the block allocated. On failure, the NULL pointer.				

### umalloc

---

<b>Function</b>	To allocate memory from the UNOS heap.
<b>Syntax</b>	<pre>#include "unos.h" char huge * umalloc( unsigned long number_of_bytes );</pre>
<b>Remarks</b>	This function is similar to the standard C function <code>malloc()</code> . The difference is that the memory is allocated from the UNOS heap, which itself is obtained at initialisation - when operations are still under DOS control - from the DOS heap.
<b>Parameters</b>	<code>number_of_bytes:</code> The number of bytes that the allocated memory should contain.
<b>Return value</b>	On success, address of the lowest memory byte in the block allocated. On failure, the NULL pointer.

### ufree

---

<b>Function</b>	To free memory previously allocated from the UNOS heap.
<b>Syntax</b>	<pre>#include "unos.h" void ufree( char huge* block_ptr );</pre>
<b>Remarks</b>	This function is similar to the standard C function free(), but operates within the UNOS heap.
<b>Parameters</b>	block_ptr:           Address of the lowest memory byte in the block to be freed.
<b>Return value</b>	None.

## SCHEDULING & TASK MANAGEMENT

### at\_exit

---

<b>Function</b>	To register a function for execution on exit.
<b>Syntax</b>	<pre>#include "unos.h" int at_exit( void (*ptr_to_function)(void) );</pre>
<b>Remarks</b>	The function registered will be executed if extUNOS() is called. Up to 32 functions may be registered, and they are executed in a last-in, first-out sequence immediately prior to the reset of all UNOS-effected hardware and interrupt vectors. Task switching stops before the first registered function is executed. The function is similar to the standard C function atexit(), but has no effect unless the function exitUNOS() is called. A call to the standard C function exit() will not cause functions registered by at_exit() to be executed.
<b>Parameters</b>	(*ptr_to_function)(void): Pointer to a void function with void parameter, which is to be executed if a call to exitUNOS() is made.
<b>Return value</b>	0 if successful, -1 if failed. (For instance, too many attempted registrations.)

### change\_task\_priority

---

<b>Function</b>	To change the current UNOS priority of a task.
<b>Syntax</b>	<pre>#include "unos.h" int change_task_priority( char *taskname_ptr, unsigned int new_priority );</pre>
<b>Remarks</b>	Depending upon the state of the task's dynamic priority, the change may be slightly delayed. Do not use within an interrupt routine.
<b>Parameters</b>	taskname_ptr: Pointer to the char string containing the name of the task whose priority is to be changed. new_priority: The required new (static) priority ( 1-7 ).
<b>Return value</b>	1 if successful, 0 if failed.

### chg\_base\_ticks\_per\_time\_slice

---

<b>Function</b>	To change the number of ticks that a task executes before a time slice entry into the kernel occurs.
<b>Syntax</b>	<pre>#include "unos.h" void chg_base_ticks_per_time_slice( int new_value );</pre>
<b>Remarks</b>	The default value of ticks per time slice is 2.
<b>Parameters</b>	new_value: Required new setting of ticks per time slice.
<b>Return value</b>	None.

### chg\_task\_tick\_delta

---

<b>Function</b>	To change the allowed variation in ticks per time slice for a task.
-----------------	---

**Syntax** `#include "unos.h"`  
`int chg_task_tick_delta char *taskname_ptr, int new_value );`

**Remarks** By changing the allowed variation in ticks per time slice, the CPU time spent in this task between task switches, relative to that spent in tasks of the same priority, may be varied.

**Parameters** `taskname_ptr:` Pointer to the char string containing the name of the task whose tick delta is to be changed.  
`new_value:` New value to be assigned to task's tick delta.

**Return value** 1 if successful, 0 if failed.

### **disable\_task\_switch**

---

**Function** To stop task switching.

**Syntax** `#include "unos.h"`  
`void disable_task_switch();`

**Remarks** Prevents any task switching until `enable_task_switch()` is called.

**Parameters** None.

**Return value** None.

### **enable\_task\_switch**

---

**Function** To start task switching.

**Syntax** `#include "unos.h"`  
`void enable_task_switch();`

**Remarks** Causes task switching to start.

**Parameters** None.

**Return value** None.

### **exitUNOS**

---

**Function** To exit the program running under UNOS.

**Syntax** `#include "unos.h"`  
`void exitUNOS( char *message );`

**Remarks** This function causes an orderly shutdown of a program running under UNOS. It is called by the keyboard task when CTRL-ALT-DEL or CTRL-ALT-HOME is pressed, and is provided publicly in case some other instigation of termination is desired. It is functionally similar to the standard C function `exit()` - which it calls - but does more. First, it stops task switching. It then executes functions registered by `at_exit()` (not `atexit()`). This is followed by a reset of all hardware settings and interrupt vector table entries affected by UNOS, then the message indicated by the argument to `exitUNOS()` is sent to the screen (preceded and followed by a new-line and carriage-return pair). The standard C function `exit()` is then called (which will of course execute any functions registered by `atexit()` (not `at_exit()`)).

**Parameters** message: Pointer to a character string to be displayed.

**Return value** None.

### UNOSTasksRunning

---

**Function** To monitor the running status of tasks within PC\_UNOS.

**Syntax** #include "pc\_unos.h"  
void UNOSTasksRunning( char RunningArray[] );

**Remarks** Fills an array with boolean flags. The state of each flag indicates whether the PC\_UNOS task allocated to the byte in the array has run within the last 5 seconds.

The positions in the array are assigned as follows:-

Serial receiver task 0	0
Serial receiver task 1	1
Serial receiver task 2	2
Serial receiver task 3	3
Serial transmitter task 0	4
Serial transmitter task 1	5
Serial transmitter task 2	6
Serial transmitter task 3	7
Screen task	8

**Parameters** Address of an array of 9 bytes.

**Return value** None.

### preemptive\_schedule

---

**Function** To cause a task switch to the highest priority task.

**Syntax** #include "unos.h"  
void preemptive\_schedule( void );

**Remarks** A task switch to the runnable task with the highest allocated priority will occur, if that task's priority exceeds the priority of the calling task.

**Parameters** None.

**Return value** None.

## reschedule

---

<b>Function</b>	To force a switch to another task.
<b>Syntax</b>	<pre>#include "unos.h" void reschedule( void );</pre>
<b>Remarks</b>	The kernel will switch to the runnable task with the highest priority, other than the calling task.
<b>Parameters</b>	None.
<b>Return value</b>	None.

## rtn\_current\_task\_name\_ptr

---

<b>Function</b>	To identify the task currently being executed, by name.
<b>Syntax</b>	<pre>#include "unos.h" char* rtn_current_task_name( void );</pre>
<b>Remarks</b>	The task currently being executed, will be the calling task.
<b>Parameters</b>	None.
<b>Return value</b>	Pointer to the char string containing the name of the task the task currently being executed.

## rtn\_current\_task\_num

---

<b>Function</b>	To identify the task currently being executed, by number.
<b>Syntax</b>	<pre>#include "unos.h" char* rtn_current_task_num( void );</pre>
<b>Remarks</b>	The task currently being executed, will be the calling task.
<b>Parameters</b>	None.
<b>Return value</b>	The (UNOS-assigned) number of the task currently being executed.

## rtn\_task\_priority

---

<b>Function</b>	To determine the static priority of a task.
<b>Syntax</b>	<pre>#include "unos.h" unsigned int trn_task_priority( char *taskname_ptr );</pre>
<b>Remarks</b>	The value returned is the allocated (static) priority of the nominated task - this may be different to the current priority, due to dynamic priority variations effected by the kernel.
<b>Parameters</b>	taskname_ptr: Pointer to the char string containing the name of the task whose allocated priority is required.
<b>Return value</b>	Allocated (static) priority of the nominated task.

### **start\_time\_slice**

---

<b>Function</b>	To allow a restart of time-slice operations by the kernel.
<b>Syntax</b>	<pre>#include "unos.h" void start_time_slice( void );</pre>
<b>Remarks</b>	This function should be used in conjunction with stop_time_slice().
<b>Parameters</b>	None.
<b>Return value</b>	None.

### **stop\_time\_slice**

---

<b>Function</b>	To stop of time-slice operations by the kernel.
<b>Syntax</b>	<pre>#include "unos.h" void stop_time_slice( void );</pre>
<b>Remarks</b>	This function should be used in conjunction with start_time_slice(). It prevents time-slice operations by the kernel, thus effectively allocating 100% of CPU time to the calling task (excluding time used by interrupt routines). It has no effect on interrupts.
<b>Parameters</b>	None.
<b>Return value</b>	None.

### **task\_switch\_enabled**

---

<b>Function</b>	To identify the state of task switching.
<b>Syntax</b>	<pre>#include "unos.h" void task_switch_enabled();</pre>
<b>Remarks</b>	None.
<b>Parameters</b>	None.
<b>Return value</b>	1 if task switching is enabled, 0 otherwise.

**THIS PAGE IS INTENTIONALLY LEFT BLANK**

## SCREEN OPERATIONS

### clear\_window

---

<b>Function</b>	To clear the current window and set it to its background colour.
<b>Syntax</b>	<pre>#include "screen.h" void clear_window( void );</pre>
<b>Remarks</b>	This function should be used instead of the standard C function <code>clearviewport()</code> , in Graphics mode, and may be used instead of <code>clrscr()</code> in text mode.
<b>Parameters</b>	None.
<b>Return value</b>	None.

### c\_printf

---

<b>Function</b>	Types a formatted string on the screen in text or graphics mode.				
<b>Syntax</b>	<pre>#include "screen.h" int c_printf( const char *format [,argument,...]);</pre>				
<b>Remarks</b>	Has the same function as the standard C function <code>cprintf()</code> , plus graphics mode capability, in which it uses the default text style. See <code>cprintf()</code> specification for details of format and argument.				
<b>Parameters</b>	<table><tr><td>format:</td><td>pointer to a format string.</td></tr><tr><td>argument(s)</td><td>argument list.</td></tr></table>	format:	pointer to a format string.	argument(s)	argument list.
format:	pointer to a format string.				
argument(s)	argument list.				
<b>Return value</b>	Number of characters output.				

### current\_key

---

<b>Function</b>	Provides the character currently in use as the keyed input.
<b>Syntax</b>	<pre>#include "screen.h" char current_key( void );</pre>
<b>Remarks</b>	The character returned is that being used within the screen task. This will be the last keystroke or the value set by a call to <code>set_key()</code> ( whichever occurred last) if the screen task pass in which that event took place is current. Otherwise the returned value is the NULL character.
<b>Parameters</b>	None.
<b>Return value</b>	Character currently in use as the last keystroke.

## current\_page

---

<b>Function</b>	To determine the page which currently has the screen.
<b>Syntax</b>	<pre>#include "screen.h" struct display_page *current_page( void );</pre>
<b>Remarks</b>	Has no internal effect.
<b>Parameters</b>	None.
<b>Return value</b>	Pointer to the structure containing data about the display page currently controlling the screen.

## current\_window

---

<b>Function</b>	To determine the window which currently has screen control.
<b>Syntax</b>	<pre>#include "screen.h" struct window_info *current_window( void );</pre>
<b>Remarks</b>	Has no internal effect.
<b>Parameters</b>	None.
<b>Return value</b>	Pointer to the window_info structure owned by the window currently controlling the screen.

## draw\_bar

---

<b>Function</b>	Display a length-controlled bar on the screen in text mode.												
<b>Syntax</b>	<pre>#include "screen.h" void draw_bar(char * label_text, double value, double max_value,               unsigned int x_start, unsigned int y_start, int direction,               unsigned int max_length);</pre>												
<b>Remarks</b>	The bar is built from the extended ASCII 'dotted' character (code hex B1), and extends in the proportion of the parameters value to max_value.												
<b>Parameters</b>	<table><tr><td>label_text:</td><td>Pointer to a character string to be displayed as a label to the bar.</td></tr><tr><td>value:</td><td>The variable controlling the length of the bar.</td></tr><tr><td>max_value:</td><td>Value at which the bar will be at its maximum length.</td></tr><tr><td>x_start, y_start:</td><td>Location within the current window, of the start of the bar.</td></tr><tr><td>direction:</td><td>If 0, bar will be horizontal. If 1, bar will be vertical.</td></tr><tr><td>max_length:</td><td>Length of the bar (in characters) when value = max_value.</td></tr></table>	label_text:	Pointer to a character string to be displayed as a label to the bar.	value:	The variable controlling the length of the bar.	max_value:	Value at which the bar will be at its maximum length.	x_start, y_start:	Location within the current window, of the start of the bar.	direction:	If 0, bar will be horizontal. If 1, bar will be vertical.	max_length:	Length of the bar (in characters) when value = max_value.
label_text:	Pointer to a character string to be displayed as a label to the bar.												
value:	The variable controlling the length of the bar.												
max_value:	Value at which the bar will be at its maximum length.												
x_start, y_start:	Location within the current window, of the start of the bar.												
direction:	If 0, bar will be horizontal. If 1, bar will be vertical.												
max_length:	Length of the bar (in characters) when value = max_value.												
<b>Return value</b>	None.												

## draw\_box

---

<b>Function</b>	Used in text mode to display a rectangle on the screen, which may contain a title.						
<b>Syntax</b>	<pre>#include "screen.h" void draw_box(unsigned int x_start, unsigned int y_start, unsigned int x_end,               unsigned int y_end, char * text );</pre>						
<b>Remarks</b>	The rectangle is made using the appropriate extended ASCII character.						
<b>Parameters</b>	<table><tr><td>x_start, y_start:</td><td>Top left-hand position, within the window, of the rectangle.</td></tr><tr><td>x_end, y_end:</td><td>Bottom right-hand position, within the window, of the rectangle.</td></tr><tr><td>text:</td><td>Pointer to a text string which will be displayed in the top left-hand corner of the rectangle.</td></tr></table>	x_start, y_start:	Top left-hand position, within the window, of the rectangle.	x_end, y_end:	Bottom right-hand position, within the window, of the rectangle.	text:	Pointer to a text string which will be displayed in the top left-hand corner of the rectangle.
x_start, y_start:	Top left-hand position, within the window, of the rectangle.						
x_end, y_end:	Bottom right-hand position, within the window, of the rectangle.						
text:	Pointer to a text string which will be displayed in the top left-hand corner of the rectangle.						
<b>Return value</b>	None.						

## draw\_line

---

<b>Function</b>	In text mode, draw a line on the screen.								
<b>Syntax</b>	<pre>#include "screen.h" void draw_line(unsigned int length, unsigned int x, unsigned int y, int direction,               int move );</pre>								
<b>Remarks</b>	The line is built from the appropriate extended ASCII characters.								
<b>Parameters</b>	<table><tr><td>length:</td><td>Number of character spaces to be covered by the line.</td></tr><tr><td>x, y:</td><td>Location within the current window, of the start of the line</td></tr><tr><td>direction:</td><td>If 0, the line will be horizontal. If 1, the line will be vertical.</td></tr><tr><td>move:</td><td>Number of character positions to move after 'drawing' each character. For example:- move = 1: Line is continuous and extends right or down. move = -1: Line is continuous and extends left or up. move = 3: Line is broken - only each 3rd character is drawn.</td></tr></table>	length:	Number of character spaces to be covered by the line.	x, y:	Location within the current window, of the start of the line	direction:	If 0, the line will be horizontal. If 1, the line will be vertical.	move:	Number of character positions to move after 'drawing' each character. For example:- move = 1: Line is continuous and extends right or down. move = -1: Line is continuous and extends left or up. move = 3: Line is broken - only each 3rd character is drawn.
length:	Number of character spaces to be covered by the line.								
x, y:	Location within the current window, of the start of the line								
direction:	If 0, the line will be horizontal. If 1, the line will be vertical.								
move:	Number of character positions to move after 'drawing' each character. For example:- move = 1: Line is continuous and extends right or down. move = -1: Line is continuous and extends left or up. move = 3: Line is broken - only each 3rd character is drawn.								
<b>Return value</b>	None.								

## get\_screen

---

<b>Function</b>	To obtain access to the screen within a nominated page and window.		
<b>Syntax</b>	<pre>#include "screen.h" int get_screen( struct display_page *page, struct window_info * window );</pre>		
<b>Remarks</b>	<p>Used to obtain screen access from anywhere. A call to <code>release_screen()</code> MUST follow the required screen operations, otherwise the screen will never be available for any other use. Use this function as follows:-</p> <pre>if( get_screen( page, window ) ) {     <i>required screen operations</i>     release_screen(); }</pre> <p>The function will fail if the requested page is not being displayed, or the display mode of the requested window does not match that of the requested page. If the required window is a graphics mode window, the (invisible) grid is set to the required window's values.</p>		
<b>Parameters</b>	<table><tr><td>page:</td><td>Pointer to the structure describing the display page in which the</td></tr></table>	page:	Pointer to the structure describing the display page in which the
page:	Pointer to the structure describing the display page in which the		

subsequent screen operations are required to occur.  
window: Pointer to the structure describing the window in which the subsequent screen operations are required to occur.

**Return value** 1 if successful, 0 if failed.

### get\_window

---

**Function** To access the screen within a nominated window.

**Syntax** #include "screen.h"  
int get\_window( struct window\_info \* window );

**Remarks** This function is similar to get\_screen(), but does not require a page as a parameter. It **MUST ONLY BE USED** within a page display function unless the window is to be displayed on ALL pages - it is like a call to get\_screen() with the currently active display page function used as the required page. A call to release\_screen() **MUST** follow the required screen operations, otherwise the screen will never be available for any other use. Use this function as follows:-

```
if( get_window( window ) )  
{  
    required screen operations  
    release_screen();  
}
```

The function will fail if the display mode of the requested window does not match that of the page being displayed.

**Parameters** window: Pointer to the structure describing the window in which the subsequent screen operations will occur.

**Return value** 1 if successful, 0 if failed.

### grid\_columns

---

**Function** Provides information about the currently set (invisible) grid on a graphics screen.

**Syntax** #include "screen.h"  
int grid\_columns( void );

**Remarks** This function relates to the invisible positioning grid for graphics mode operations. Whilst the grid has no effect in text mode, it exists in memory even when the screen is in text mode, so information about it can still be provided.

**Parameters** None.

**Return value** Number of coulms in the currently set grid.

## grid\_rows

---

<b>Function</b>	Provides information about the currently set (invisible) grid on a graphics screen.
<b>Syntax</b>	<pre>#include "screen.h" int grid_rows( void );</pre>
<b>Remarks</b>	This function relates to the invisible positioning grid for graphics mode operations. Whilst the grid has no effect in text mode, it exists in memory even when the screen is in text mode, so information about it can still be provided.
<b>Parameters</b>	None.
<b>Return value</b>	Number of rows in the currently set grid.

## go\_to\_xy

---

<b>Function</b>	Performs the 'go to' function in text and graphics mode.
<b>Syntax</b>	<pre>#include "screen.h" void go_to_xy( int x, int y );</pre>
<b>Remarks</b>	This function may be used instead of the standard C functions gotoxy() and movetoxy() in text and graphics modes, respectively. Note however that in graphics mode, the coordinates relate to the current grid, rather than pixels as in movetoxy(). The reference position in each rectangle defined by the grid is the top left corner. Thus whereas a value of 0 for y is legal, the use of bottom- or centre-justified text output from such a position would not produce a sensible result.
<b>Parameters</b>	x,y: Horizontal and vertical coordinates to which the cursor is moved. In text mode, this is in terms of text rows and columns. In graphics mode, it is in terms of rows and columns of the current (invisible) grid.
<b>Return value</b>	None.

## out\_text

---

<b>Function</b>	Places text on a text or graphics screen at the current cursor position.
<b>Syntax</b>	<pre>#include "screen.h" void out_text( struct graphics_text_info *ptr_to_text_type, char *message );</pre>
<b>Remarks</b>	This function combines the text style-setting operation with the standard C function outtext(), if the current mode is Graphics. In Text mode, the first parameter is ignored, and the function behaves as cputs().
<b>Parameters</b>	ptr_to_text_type: Pointer to a structure containing the required graphics text style settings. message: Pointer to a character string containing the required message.
<b>Return value</b>	None.

## out\_text\_xy

---

**Function** Places text on a text or graphics screen at a nominated position.

**Syntax** `#include "screen.h"`  
`void out_text_xy(int x, int y, struct graphics_text_info *ptr_to_text_type, char *message);`

**Remarks** This function combines the text style-setting operation with the standard C function `outtextxy()`, if the current mode is Graphics. Note however that the coordinates relate to the current grid, rather than pixels as in `outtextxy()`. The reference position in each rectangle defined by the grid is the top left corner. Thus whereas a value of 0 for y is legal, the use of bottom- or centre-justified text output from such a position would not produce a sensible result. In Text mode, the function behaves as a call to `gotoxy()` followed by `cputs()`, and the third parameter is ignored.

**Parameters** x, y: Horizontal and vertical coordinates to which the cursor is moved. In Graphics mode, it is in terms of rows and columns of the current (invisible) grid.  
ptr\_to\_text\_type: Pointer to a structure containing the required text style settings.  
message: Pointer to a character string containing the required message.

**Return value** None.

### put\_intersection

---

**Function** Place a line intersection character on the screen in text mode.

**Syntax** `#include "screen.h"`  
`void put_intersection( unsigned int x, unsigned int y, unsigned int intersection);`

**Remarks** The intersection is made using the appropriate extended ASCII character, and matches the line operations of `put_line()`.

**Parameters** x, y: Location within the current window, of the corner.  
intersection: Controls which type of intersection is placed:-  
0 lower left;  
1 upper right;  
2 upper left;  
3 lower right;  
4 left-pointing T;  
5 right-pointing T;  
6 down-pointing T;  
7 upward-pointing T, and  
8 cross intersection.

**Return value** None.

### release\_screen

---

**Function** To release a window's control of the screen.

**Syntax** `#include "screen.h"`  
`void release_screen( void );`

**Remarks** This function should only be called in association with a call to `get_screen()` or `get_window()`. It is mandatory to call `release_screen()` after each call to `get_screen()` or `get_window()`.

**Parameters** None.

**Return value** None.

### **release\_zone**

---

**Function** To release a screen zone frozen by a window.

**Syntax** `#include "screen.h"`  
`void release_zone( void );`

**Remarks** This function may be called after a call to `get_screen()` or `get_window()` when the nominated window's structure had its `do_not_cover` element set. It 'unfreezes' the screen zone previously 'frozen'.

**Parameters** None.

**Return value** None.

### **set\_grid**

---

**Function** Sets grid values for locating points in graphics mode displays.

**Syntax** `#include "screen.h"`  
`void set_grid( int rows, int columns );`

**Remarks** The grid setting is held in memory until changed by another call to this function, or entry into a page or window with different grid row and/or column settings. A value of 0 will set the maximum possible number - 480 and 640 respectively.

**Parameters** rows, columns: Number of rows and columns in a whole page.

**Return value** None.

### **set\_key**

---

**Function**

**Syntax** `#include "screen.h"`  
`void set_key( unsigned char key_char );`

**Remarks** The character provided replaces that being used within the screen task for the rest of the current screen task pass.  
This function is useful for clearing the key character before calls to `show_for_change()`, to prevent unwanted keystrokes reaching that function.

**Parameters** key\_char: keyboard character (see keyboard.h for definitions).

**Return value** None.

### **show\_for\_change**

---

**Function** Places an item within a window, for on-screen editing.

**Syntax** #include "screen.h"  
int show\_for\_change( int x, int y, char \*format, void \*data, unsigned int change\_control );

**Remarks** This function implements PC\_UNOS's on-screen editing operations. It places an editable item within the current window and determines the type of data from the format specification, WHICH MUST MATCH THE DATA TYPE EXACTLY. Note that an input-size modifier MUST BE INCLUDED if the matching argument is not of the default size short integer or floating-point. Legal values for change\_control are defined in screen.h as:-

TOGGLE	0
EDIT	1
SHOW	2
REGISTER_ONLY	3

**Parameters**

x, y:	location of the start of the item within the window. In graphics mode, this relates to the currently set (invisible) grid.
format:	format specification string, as in 'printf'. Size flags MUST be used if appropriate to the matching argument.
data:	pointer to the data to be displayed and /or edited.
change_control:	If 0: Function can return 1, without changing the data. If 1: Function can return 1 and will change the data. If 2: Function will always return 0 and will not change the data. If 3: Function will always return 0, will not change the data and will not display the data.

**Return value** 1 if the data is changed, or change\_control is 0 and the space-bar is hit. Otherwise 0.

## SEMAPHORES and TIMING

### create\_semaphore

---

<b>Function</b>	To create a new semaphore.
<b>Syntax</b>	<pre>#include "unos.h" unsigned int create_semaphore( void );</pre>
<b>Remarks</b>	May be called at any time to create a new semaphore. The semaphore created is initialised to a value of 1 with a multiplier value of 1.  <b>NOTE:</b> When declaring a semaphore variable, ALWAYS initialize it to 0xFFFF. In this manner, unintentioned operations on a semaphore which has not been created will be avoided, as the UNOS semaphore functions will be able to detect the illegal semaphore number.
<b>Parameters</b>	None.
<b>Return value</b>	The created semaphore's number. If unsuccessful, 0xFFFF.

### create\_timer

---

<b>Function</b>	To create a new timer.
<b>Syntax</b>	<pre>#include "unos.h" timer_struct * create_timer( void );</pre>
<b>Remarks</b>	After allocating the required timer structure from the UNOS heap, the timer is inserted into the inactive timer queue.
<b>Parameters</b>	None.
<b>Return value</b>	Pointer to the new timer data structure. This structure is defined in unos.h.

### init\_semaphore

---

<b>Function</b>	To initialise a semaphore's variables.
<b>Syntax</b>	<pre>#include "unos.h" void init_semaphore( unsigned int semaphore_number, unsigned int semaphore_value,                     unsigned int multiplier_value);</pre>
<b>Remarks</b>	Sets the semaphore value and multiplier value of the nominated semaphore. (Multiplier value is the number of signals on the semaphore generated by a single signal call on the semaphore.)
<b>Parameters</b>	semaphore_number: The number by which the semaphore to be affected is known. semaphore_value: The new count value of the semaphore. multiplier_value: The new value of the semaphore's multiplier.
<b>Return value</b>	None.

### reset\_timer

---

<b>Function</b>	Reset the tick count on a timer to its initial value.
<b>Syntax</b>	<pre>#include "unos.h" timer_struct * reset_timer( timer_struct * timer_ptr );</pre>
<b>Remarks</b>	This function may be used following a call to start_timer(), to restart the timer.
<b>Parameters</b>	timer_ptr: Pointer to a timer structure created with create_timer(), and started by a call to start_timer().
<b>Return value</b>	Pointer to the timer structure reset. If the reset was unsuccessful, the NULL pointer is returned.

### return\_semaphore\_value

---

<b>Function</b>	To determine the current count on a semaphore.
<b>Syntax</b>	<pre>#include "unos.h" unsigned int return_semaphore_value( unsigned int semaphore_number );</pre>
<b>Remarks</b>	This function has no effect on the semaphore.
<b>Parameters</b>	semaphore_number: The number by which the semaphore is known.
<b>Return value</b>	The value of the count on the semaphore. If the nominated semaphore does not exist, 0xFFFF is returned.

### \_signal

---

<b>Function</b>	Generates a signal on a semaphore.
<b>Syntax</b>	<pre>#include "unos.h" void _signal( unsigned int semaphore_number );</pre>
<b>Remarks</b>	<p>Unblocks tasks waiting on the nominated semaphore, up to the multiplier value of the semaphore. If this is less than the multiplier value, sets the semaphore's multiplier count to the remainder. If no task was waiting on the semaphore, adds the multiplier count to the semaphore's value.</p> <p>Note that this function's name begins with an underline, to distinguish it from the standard C function <b>signal()</b>.</p> <p>If the semaphore does not exist or is number 0xFFFF, the function returns without taking any action. <b>IT IS A GOOD IDEA TO INITIALISE SEMAPHORE VARIABLES TO 0xFFFF</b>, to prevent inadvertent signals on semaphore 0.</p>
<b>Parameters</b>	semaphore_number: The number by which the semaphore to be affected is known.
<b>Return value</b>	None.

### start\_timer

---

<b>Function</b>	To start a timer.
<b>Syntax</b>	<pre>#include "unos.h" timer_struct *start_timer( unsigned char timer_type, unsigned long initial_tick_count,</pre>

```
void ( *timeout_handler_fn )( void ), void *data_ptr );
```

**Remarks** This function takes a timer from the queue of inactive timers, initialises it to the nominated values, and places it on the queue of active timers. The type of timer may be REPETITIVE or SINGLE\_SHOT (defined in unos.h as 1 or 0 respectively). A REPETITIVE timer is automatically reset to its initial value on timeout. A SINGLE\_SHOT timer becomes inactive on timeout.

**Parameters**

timer_type:	Type of timer required - REPETITIVE or SINGLE_SHOT.
initial_tick_count:	Required number of ticks to time out.
timeout_handler_fn:	Pointer to a function which is to be executed on timeout.
data_ptr:	Pointer to a data structure of any type, which may be used by the timeout handler function to return data to the calling function.

**Return value** On success, pointer to the activated timer. If unsuccessful, the NULL pointer.

### stop\_timer

---

**Function** To de-activate an active timer.

**Syntax** #include "unos.h"  
timer\_struct\* stop\_timer( timer\_struct\* timer\_ptr );

**Remarks** Removes the nominated timer from the active timer queue and places it on the inactive timer queue.

**Parameters** Pointer to the timer structure to be de-activated.

**Return value** If successful, Pointer to the timer structure de-activated. Otherwise, the NULL pointer.

### timed\_wait

---

**Function** Block a task on a semaphore, for a predetermined maximum time.

**Syntax** #include "unos.h"  
int timed\_wait( unsigned int semaphore\_number, unsigned long ticks\_to\_wait );

**Remarks** This function causes the calling task to become blocked on the nominated semaphore, until the indicated number of ticks has occurred - this effectively generates a signal on the semaphore, unblocking the task.

**Parameters**

semaphore_number:	The number by which the semaphore to use is known.
ticks_to_wait:	The number of tick interrupts to occur before the calling task will become unblocked.

**Return value** If task is unblocked by a signal on the semaphore other than the one generated by the wait timer:- 0  
If unblocking occurs due to the nominated ticks having occurred:- 1  
If no timers are available to carry out the wait operation:- 2

## wait

---

**Function** Block a task on a semaphore.

**Syntax** #include "unos.h"  
void wait( unsigned int semaphore\_number );

**Remarks** This function causes the calling task to become blocked on the nominated semaphore. Primarily used for synchronisation of access to a resource shared with other tasks, the calling task becomes blocked, and any task which is claiming the resource will, if necessary, have its priority temporarily lifted (this is a *dynamic priority adjustment*) above that of the calling task, so that the resource will be released within a reasonable time.

**Parameters** semaphore\_number: The number by which the semaphore to use is known.

**Return value** None.

## SERIAL COMMUNICATIONS

### prog\_16450\_uart

---

<b>Function</b>	Program a 16540-compatible UART.
<b>Syntax</b>	<pre>#include "serial.h" unsigned char prog_16450_uart (unsigned int baud_rate, float stop_bits,                                char parity, char char_lgth, unsigned int command_reg_address,                                unsigned int data_reg_address);</pre>
<b>Remarks</b>	This function may be used if the UNOS serial channel facilities are not used. Note that a call to create_serial_channel programs the appropriate UART, thus there is no need to call prog_16540_uart() as well.
<b>Parameters</b>	baud_rate: As required. stop_bits: As required. data_bits: As required. parity: For odd parity:- 0; even parity:- 1; no parity: 2. command_reg_address: UART's control register's I/O address. data_reg_address: UART's data register's I/O address.
<b>Return value</b>	The value set into the Interrupt Enable register - currently 5 (bits 0,2 set ON, the rest OFF).

### reset\_comms\_hardware

---

<b>Function</b>	Re-establish serial communications hardware settings.
<b>Syntax</b>	<pre>#include "serial.h" unsigned char reset_comms_hardware( unsigned int port_number );</pre>
<b>Remarks</b>	Resets the UART and the interrupt vector for the nominated serial port to the values used in the call to create_serial_channel() for the same port. This function must not be called before a call to create_serial_channel() for the same port.
<b>Parameters</b>	port_number: As required - 0 to 3.
<b>Return value</b>	The value set into the Interrupt Enable register - currently 5 (bits 0,2 set ON, the rest OFF).

### get\_serial\_errors

---

<b>Function</b>	Obtain accumulated serial channel error data.
<b>Syntax</b>	<pre>#include "serial.h" void get_serial_errors( int port, serial_errors *ptr_to_error_struct );</pre>
<b>Remarks</b>	This function copies accumulated serial error data into the serial_error structure nominated. The serial_errors structure is defined in serial.h as:- typedef struct serial_errors { unsigned int    overrun; unsigned int    parity; unsigned int    framing; unsigned int    break_rec;

```
        unsigned int    rx_buf_overrun;  
        unsigned int    uart_reset;  
    }serial_errors;
```

Note that whilst break\_rec is not actually an error, it is included for convenience.

**Parameters** port: The serial port number for which error data is required.  
ptr\_to\_error\_struct: Pointer to a serial\_errors structure provided by the caller.

**Return value** None.

### reset\_serial\_errors

---

**Function** To set all elements of the serial error accumulator to zero.

**Syntax** #include "serial.h"  
void reset\_serial\_errors( int port );

**Remarks** Calling this function clears the contents of the structure within PC\_UNOS which accumulates serial error data. It has no effect on the port hardware.

**Parameters** port: The serial port number for which error data is required.

**Return value** None.

## TASK COMMUNICATIONS

### connect\_to\_keyboard

---

<b>Function</b>	Cause the keyboard task to send its output to the nominated task.
<b>Syntax</b>	<pre>#include "keyboard.h" void connect_to_keyboard( char *task_name );</pre>
<b>Remarks</b>	The use of this function IS NOT ADVISED, as it will disconnect the keyboard from the screen task. If it is used, the keyboard task should be re-connected to the screen task when it is no longer required to communicate directly with the nominated task. Keyboard output should normally be obtained through the screen task, via the single parameter required in all display page functions.
<b>Parameters</b>	task_name:           Pointer to the name of the task to which keyboard output should be sent. For instance:- connect_to_keyboard( screen_task_name );
<b>Return value</b>	None.

### flush\_mbx

---

<b>Function</b>	To clear all messages from the calling task's mailbox.
<b>Syntax</b>	<pre>#include "unos.h" void flush_mbx( void );</pre>
<b>Remarks</b>	If other tasks were blocked trying to send messages to the mailbox, they will be unblocked.
<b>Parameters</b>	None.
<b>Return value</b>	None.

### free\_mbx

---

<b>Function</b>	Returns the number of free (available) message slots in a mail box.
<b>Syntax</b>	<pre>#include "unos.h" unsigned int free_mbx( char *taskname_ptr );</pre>
<b>Remarks</b>	This function has no effect on the mail box.
<b>Parameters</b>	Pointer to the char string containing the name of the task whose mail box data is required.
<b>Return value</b>	Number of available message slots in the nominated task's mailbox. If the mail box does not exist, returns 0xFFFF.

## rcv\_mess

---

<b>Function</b>	Collect the contents of a mailbox.						
<b>Syntax</b>	<pre>#include "unos.h" char *rcv_mess( unsigned char * mess_ptr, unsigned int * mess_length_ptr,                 unsigned long time_limit );</pre>						
<b>Remarks</b>	If the mail box does not contain a message, the calling task becomes blocked until either a message arrives, or the nominated time limit expires.						
<b>Parameters</b>	<table><tr><td>mess_ptr:</td><td>Pointer to a byte array which has sufficient length to accommodate the longest possible message.</td></tr><tr><td>mess_length_ptr:</td><td>Pointer to a variable to hold the length of the message.</td></tr><tr><td>time_limit:</td><td>The maximum number of kernel tick routine entries for which the calling task may be blocked. If zero, the time limit is infinite.</td></tr></table>	mess_ptr:	Pointer to a byte array which has sufficient length to accommodate the longest possible message.	mess_length_ptr:	Pointer to a variable to hold the length of the message.	time_limit:	The maximum number of kernel tick routine entries for which the calling task may be blocked. If zero, the time limit is infinite.
mess_ptr:	Pointer to a byte array which has sufficient length to accommodate the longest possible message.						
mess_length_ptr:	Pointer to a variable to hold the length of the message.						
time_limit:	The maximum number of kernel tick routine entries for which the calling task may be blocked. If zero, the time limit is infinite.						
<b>Return value</b>	a. Pointer to the char string containing the name of the sending task, or b. if a timer was not available, 0xFFFF:000F, or c. if the time limit expires, the NULL pointer.						

## send\_mess

---

<b>Function</b>	sends a byte string to a task's mailbox.						
<b>Syntax</b>	<pre>#include "unos.h" int send_mess( unsigned char * mess_ptr, unsigned int mess_length,                char *taskname_ptr );</pre>						
<b>Remarks</b>	The calling task will become blocked if the mailbox fills, remaining blocked until space is again available. Note that the byte string making up the message to be sent does not have to be NULL-terminated.						
<b>Parameters</b>	<table><tr><td>mess_ptr:</td><td>Pointer to the byte string to be sent.</td></tr><tr><td>mess_length:</td><td>Length of the message, in bytes.</td></tr><tr><td>taskname_ptr:</td><td>Pointer to the char string containing the recipient task's name.</td></tr></table>	mess_ptr:	Pointer to the byte string to be sent.	mess_length:	Length of the message, in bytes.	taskname_ptr:	Pointer to the char string containing the recipient task's name.
mess_ptr:	Pointer to the byte string to be sent.						
mess_length:	Length of the message, in bytes.						
taskname_ptr:	Pointer to the char string containing the recipient task's name.						
<b>Return value</b>	1 if successful, 0 if the message is too big.						

## send\_qik\_mess

---

<b>Function</b>	sends a byte string to a task's mailbox, with highest priority.						
<b>Syntax</b>	<pre>#include "unos.h" int send_qik_mess( unsigned char * mess_ptr, unsigned int mess_length,                    char *taskname_ptr );</pre>						
<b>Remarks</b>	This function is similar to send_mess, except that it places the message at the top of the message queue owned by the receiving task.						
<b>Parameters</b>	<table><tr><td>mess_ptr:</td><td>Pointer to the byte string to be sent.</td></tr><tr><td>mess_length:</td><td>Length of the message, in bytes.</td></tr><tr><td>taskname_ptr:</td><td>Pointer to the char string containing the recipient task's name.</td></tr></table>	mess_ptr:	Pointer to the byte string to be sent.	mess_length:	Length of the message, in bytes.	taskname_ptr:	Pointer to the char string containing the recipient task's name.
mess_ptr:	Pointer to the byte string to be sent.						
mess_length:	Length of the message, in bytes.						
taskname_ptr:	Pointer to the char string containing the recipient task's name.						
<b>Return value</b>	1 if successful, 0 if the message is too big. 2 if the special quick mailbox envelope is full.						

## size\_mbx

---

<b>Function</b>	To determine the number of slots in a task's mailbox.
<b>Syntax</b>	<pre>#include "unos.h" unsigned int size_mbx( char *taskname_ptr );</pre>
<b>Remarks</b>	This function has no effect on the mail box.
<b>Parameters</b>	Pointer to the char string containing the name of the task about whose mailbox the information is sought.
<b>Return value</b>	Size of the mailbox queue belonging to the nominated task. If the mail box does not exist, returns 0xFFFF.

## size\_mbx\_mess

---

<b>Function</b>	To determine the size of the envelopes in a task's mailbox.
<b>Syntax</b>	<pre>#include "unos.h" unsigned int size_mbx_mess( char *taskname_ptr );</pre>
<b>Remarks</b>	This function has no effect on the mail box.
<b>Parameters</b>	Pointer to the char string containing the name of the task about whose mailbox the information is sought.
<b>Return value</b>	Number of bytes in each envelope in the mailbox belonging to the nominated task. If the mail box does not exist, returns 0xFFFF.

## used\_mbx

---

<b>Function</b>	To determine how many message slots in a mail box contain uncollected messages.
<b>Syntax</b>	<pre>#include "unos.h" unsigned int used_mbx( char *taskname_ptr );</pre>
<b>Remarks</b>	This function has no effect on the mail box.
<b>Parameters</b>	Pointer to the char string containing the name of the task about whose mailbox the information is sought.
<b>Return value</b>	Number of used message slots in the nominated task's mailbox. If the mail box does not exist, returns 0xFFFF.

**THIS PAGE IS INTENTIONALLY LEFT BLANK**

## APPENDIX A FUNCTIONS INDEX

Function	Page	Function	Pa
_signal	42	out_text	37
at_exit	27	out_text_xy	37
c_printf	33	PC_UNOSRevision	21
change_task_priority	27	preemptive_schedule	29
chg_base_ticks_per_time_slice	27	prog_16450_uart	45
chg_task_tick_delta	27	put_intersection	38
clear_window	33	rcv_mess	48
connect_to_keyboard	47	release_screen	38
create_graphics_page	15	release_zone	39
create_graphics_text	15	reschedule	30
create_graphics_window	16	reset_comms_hardware	45
create_semaphore	41	reset_serial_errors	46
create_serial_channel	18	reset_timer	41
create_task	19	ret_free_mem	25
create_text_page	19	return_interrupt_status	23
create_text_window	20	return_semaphore_value	42
create_timer	41	rtn_current_task_name_ptr	30
current_key	33	rtn_current_task_num	30
current_page	34	rtn_task_priority	30
current_window	34	send_mess	48
disable_int_mask	23	send_qik_mess	48
disable_task_switch	28	set_as_first_page	22
draw_bar	34	set_grid	39
draw_box	35	set_key	39
draw_line	35	Set_vector	23
enable_int_mask	23	show_for_change	39
enable_task_switch	28	size_mbx	49
exitUNOS	28	size_mbx_mess	49
flush_mbx	47	start_time_slice	31
free_mbx	47	start_timer	42
get_screen	35	stop_time_slice	31
get_serial_errors	45	stop_timer	43
get_window	36	task_switch_enabled	31
go_to_xy	37	timed_wait	43
GoUNOS	21	ucalloc	25
grid_columns	36	ufree	25
grid_rows	37	umalloc	25
init_semaphore	41	UNOSTasksRunning	29
InitPC_UNOS	21	used_mbx	49
		wait	44

**THIS PAGE IS INTENTIONALLY LEFT BLANK**

## APPENDIX B GLOBAL VARIABLES

### NOTE:

These variables control PC\_UNOS and UNOS initialisation operations. Their names are RESERVED when using PC\_UNOS. It is not advisable to change their values after PC\_UNOS has been initialised. In some cases such an action could have unpredictable results.

If a change to a PC\_UNOS global variable's value from its default value is required, it must be made at run-time, BEFORE a call to InitPC\_UNOS() or create\_serial\_channel().

### B1. UNOS CONTROL

#### \_max\_number\_of\_semaphores

<b>Function</b>	Specifies the maximum number of semaphores available to the user. Initialising PC_UNOS automatically increases its value by twice the maximum number of tasks allowed (including PC_UNOS tasks), to allow for the two semaphores used by each task (in its mailbox operations).
<b>Syntax</b>	extern int _max_number_of_semaophores
<b>Remarks</b>	Declared in unos.h
<b>Default Value</b>	10

#### \_max\_number\_of\_tasks

<b>Function</b>	Specifies the maximum number of tasks which the user may create. Initialising PC_UNOS automatically increases its value to allow for serial channel receive and transmit tasks for each channel required, the keyboard task, the screen task and UNOS' Null task.
<b>Syntax</b>	extern int _max_number_of_tasks
<b>Remarks</b>	Declared in unos.h
<b>Default Value</b>	10

#### \_max\_number\_of\_timers

<b>Function</b>	Specifies the maximum number of timers that the user may use.
<b>Syntax</b>	extern int _max_number_of_timers
<b>Remarks</b>	Declared in unos.h
<b>Default Value</b>	10

#### \_memory\_pool\_size

<b>Function</b>	Specifies the size, in bytes, of the memory pool available to UNOS for the creation of structures etc.
-----------------	--

**Syntax** extern int \_memory\_pool\_size

**Remarks** Declared in unos.h

**Default Value** 0x30000

### **\_task\_debug**

---

**Function** Determines if task stack and mailbox monitoring will be turned on. Note that the stack monitoring occurs at task switch time, so is only a conservative measurement of maximum stack length.

**Syntax** extern int \_task\_debug

**Remarks** Declared in unos.h

**Default value** 0 (ie OFF)

### **\_task\_monitor\_rate**

---

**Function** Determines the rate at which the Task Monitor task cycles. The value indicates seconds.

**Syntax** extern unsigned int \_task\_monitor\_rate

**Remarks** Declared in pc\_unos.h

**Default value** 5

### **\_tick\_interrupts\_per\_second**

---

**Function** Determines the rate of kernel entry interrupts.

**Syntax** extern int \_tick\_interrupts\_per\_second

**Remarks** Declared in unos.h

**Default value** 32

### **\_ticks\_before\_kernel\_entry**

---

**Function** Determines the default number of ticks a task is run before a time slice entry into the kernel occurs..

**Syntax** extern int \_ticks\_before\_kernel\_entry

**Remarks** Declared in unos.h

**Default value** 2

## B2. SERIAL CHANNEL CONTROL

Most of the functions fulfilled by the global variables associated with the serial channel system are common to all four channels. In these cases, their names differ only in one character, the channel number (which ranges from 0 to 3). Further, most of their names inherently make their function clear.

To ease the burden on the reader (and the forests supplying the paper for this manual), where possible the variables will be listed generically, with a lower-case n replacing the channel number in each name. Thus the listing here for baud rate settings is

\_CHANNEL\_n\_BAUD

The actual baud rate for channel 2 would be set by setting the value of the variable \_CHANNEL\_2\_BAUD to the desired value at run-time.

All serial channel global variables are declared in serial.h.

The default value of each variable is given in brackets after the variable's name.

The header file serial.h defines NO = 0, YES = 1.

### B2.1 Hardware Settings

```
extern int _channel_n_required;          ( YES )
        Setting to NO prevents the creation of tasks associated with Channel n, and no hardware
        associated with Channel n is then initialised.
extern int _UART_0_COM_REG_ADDR;        ( 0x3f8 )
extern int _UART_0_DATA_REG_ADDR;      ( 0x3f8 )

extern int _UART_1_COM_REG_ADDR;        ( 0x2f8 )
extern int _UART_1_DATA_REG_ADDR;      ( 0x2f8 )

extern int _UART_2_COM_REG_ADDR;        ( 0x3e8 )
extern int _UART_2_DATA_REG_ADDR;      ( 0x3e8 )

extern int _UART_3_COM_REG_ADDR;        ( 0x2e8 )
extern int _UART_3_DATA_REG_ADDR;      ( 0x2e8 )

extern int _CHANNEL_0_IRQ_NUM;          ( 4 )
extern int _CHANNEL_1_IRQ_NUM;          ( 3 )
extern int _CHANNEL_2_IRQ_NUM;          ( 5 )
extern int _CHANNEL_3_IRQ_NUM;          ( 7 )

extern int _CHANNEL_n_BAUD;             ( 9600 )

extern int _CHANNEL_n_STOP_BITS;        ( 1 )

extern int _CHANNEL_n_DATA_BITS;        ( 8 )

extern int _CHANNEL_n_PARITY;           ( NO_PARITY )
        May also be set to ODD_PARITY or EVEN_PARITY. These constants are defined in serial.h
        as:-
                ODD_PARITY      0
                EVEN_PARITY     1
                NO_PARTY        2

extern int _CHANNEL_n_XONXOFF;          ( NO )
```

### B2.2 Software Settings

```
extern int _FEEDTHROUGH_ERRORS;        ( NO )
```

If set to YES, a serial error will cause an 'error byte' to be inserted in the receive buffer. These bytes are defined in serial.h as:-

OVERRUN_ERROR_BYTE	0x80
PARITY_ERROR_BYTE	0x81
FRAMING_ERROR_BYTE	0x82
BREAK_INTERRUPT_ERROR_BYTE	0x83

```
extern int _CHANNEL_n_RX_BUF_SIZE;    ( 100 )
extern int _CHANNEL_n_TX_BUF_SIZE;    ( 100 )
```

```
extern int _CHANNEL_n_BUF_LOW_LIM;    ( 10 )
extern int _CHANNEL_n_BUF_UPPER_LIM;  ( 80 )
```

These limits are used if XONXOFF is set to YES. If a receive buffer use reaches its set upper limit, Ctrl-Q is transmitted to the sender. Subsequent reduction of the buffer's use below the lower limit causes Ctrl-S to be transmitted to the sender.

```
extern int _CHANNEL_n_MBX_MESS_SIZE;  ( 100 )
extern int _CHANNEL_n_MBX_Q_SIZE;     ( 2 )
```

These variables set the sizes of the UNOS mailboxes for both the transmit and receive tasks.