Support Group Application Note
Number: 237
Issue: 1.00
Author: James Bye

Acorn

Writing Wimp Module Tasks using Desktop C

This application outlines how to write Wimp Module Tasks using Desktop C.

Applicable Hardware :	All RISC OS computers.	Related Application Notes:	None

Copyright © Acorn Computers Limited 1993

Neither whole nor any part of the information contained in this note may be adapted or reproduced in any form except with the prior written approval of Acorn Computers Limited.

Every effort has been made to ensure that the information in this leaflet is true and correct at the time of printing. However, the products described in this leaflet are subject to continuous development and improvements and Acorn Computers Limited reserves the right to change its specifications at any time. Acorn Computers Limited cannot accept liability for any loss or damage arising from the use of any information or particulars in this leaflet. ACORN, ECONET and ARCHIMEDES are trademarks of Acorn Computers Limited.

Support Group
Acorn Computers Limited
Acorn House
Vision Park
Histon, Cambridge
CB4 4AE

1. Introduction

This application note outlines how to write Wimp Module Tasks in C using the CMHG (C Module Header Generator) and presumes that the reader has experience of using Desktop and its associated libraries and utilities.

Although the creation of Wimp Module Tasks in C is relatively stimple, there are a few problem areas that need to be made clear to the programmer. This application note covers the following areas :-

- Overview of CMHG and constructing the header files
- Implementing simple the C routines defined in C header
- Handling Service_Memory
- Linking to RISC_OSLib
- Compiling C source files
- Details of other relevant documentation

A disc containing an example C Module task is included for reference with this application note.

2. Overview of CMHG and constructing the header file

Desktop C provides a tool called !CMHG (C Module Header Generator) for generating module headers for C modules. CMHG produces a header that veneers onto your C program and this tool should always be used when constructing modules in C. Producing your own header in assembly language is relatively complex and this is why CMHG has been provided.

The example on the enclosed disc is a simple example, and the CMHG file looks like the following :-

```
; /*
;  * C Module Header for Capture
;  */
module-is-runnable:
; /*
;  * service call handler
;  */
service-call-handler: MyService
; /*
;  * title string
;  */
title-string: Capture
; /*
;  * help string
;  */
help-string: Capture 1.00
```

It is worth noting at this point that, when constructing C Module Tasks, the module must be runnable and implement a service call handler. You will see from this example that not all the fundamental components

of a C Module header have been implemented. If you wish to claim RMA workspace using the OS_Module calls or claim vectors etc, then you will need to implement an initialisation routine. Full details on how the other fundamental components of a C Module are implemented can be found in the Desktop C User Guide (Chapter 14, page 337).

3. Implementing C routines defined in Module Header

As shown in section 2, our simple example implements a runnable entry point and a service call handler.

When a module is runnable, the C entry interface is as follows :-

```
int main ( int argc, char *argv[]);
```

Where the value **argc** is the number of arguments passed to the module and ***argv** is a pointer to the command line arguments. In our example, the run entry for the module looks like the following :-

```
int main ( void ) {

   wimpt_init("Capture");
   task_handle = wimpt_task();
   res_init("Capture");
   resspr_init();
   flex_init();
   template_init();
   dbox_init();
   visdelay_init();

   event_setmask(wimp_EMPTRLEAVE | wimp_EMPTRENTER);

   trace_on();

   iconbar_buffer = malloc(iconbar_buffer_len);
   strcpy(iconbar_buffer,"Ready");

   iconbar_icon =baricon_textandsprite("!capture",iconbar_buffer,
                iconbar_buffer_len,1,click_proc);

   iconbar_menu = menu_new(m_IconBar_Title,m_IconBar_Hits);
   event_attachmenu(win_ICONBAR,iconbar_menu,iconbar_menu_events,NULL);

   while(TRUE)
         event_process();
  }
```

The above is a simple application initialisation routine that is common to many applications written using RISC_OSLib.

It is worth noting that the runnable entry point for a module is entered in User Mode and not in SVC mode. This means that any calls to **malloc()** in the main routine will cause memory to be allocated in the application workspace and not in the RMA. If you wish to allocate RMA space in the main routine then you will need to call the relevant OS_Module SWI.

The C entry for Service Calls is as follows :-

```
void service_handler( int service_number, _kernel_swi_regs *r, void *pw );
```

The service call handler for our example is covered more deeply in the next section.

4. Handling Service_Memory

When writing a module task in C, the service call Service_Memory (0x11) needs to be claimed. The way to do this is as follows (this is also shown in our simple example) :-

```
#define Service_Memory       0x11

extern void MyService ( int service_number, _kernel_swi_regs *r, void *pw )
{
   pw = pw;

   /*-- keep application workspace (r2 holds CAO pointer) */

   switch(service_number)
   {
      case service_memory : if(r->r[2] == (int)Image__RO_Base)
                            {
                                /*--refuse to release app. workspace --*/
                                r->r[1] = 0;
                            }
                            break;
   }
}
```

The service call handler needs to compare the contents of register R2 with the address of the base of you module containing it. However, this is not a value directly available to C and the following assembler language fragment is used to gain access to the symbol **|Image$$RO$$Base|**. The fragment is as follows :-

```
        IMPORT  |Image$$RO$$Base|
        EXPORT  Image__RO_Base

            AREA    Code_Description, DATA, REL
Image__RO_Base      DCD     |Image$$RO$$Base|

            END
```

This object can be found pre-assembled on the example disc.

One of the constraints of writing module tasks in C is that the module task will appear in the list of applications and not in the list of module tasks.

5. Linking to RISC_OSLib

If you are using RISC_OSLib in the construction of your C Module task, then you will need to link to a special version of RISC_OSLib that has been compiled as module code. However, a module code version of RISC_OSLib is not supplied with Desktop C. If you have the RISC_OSLib sources, then you can build a module version by entering your source directory and typing the following :-

```
amu -f makemod
```

The relevant RISC_OSLib object can be found on the enclosed disc.

6. Compiling C source code

When compiling your C source code for inclusion in a module, then you must indicate to the C Compiler that you wish it to generate module code. This can be done by either specifying the '-zM' option if you are compiling from the command line, or by selecting the module code option from the CC window. The same applies when linking.

7. Other relevant documentation

This application note is intended only as an introduction to writing module tasks in C. There are many relevant areas which are beyond the scope of this document. It is therefore recommended that the reader consults the following manuals :-

1. The RISC OS 3 Programmer's Reference Manual
2. The Acorn ANSI C Release 4 Manual

The RISC OS 3 Programmers Reference Manual contains detailed information and the Modules and Window Manager chapters should be read. The Window Manager section contains information on the service call Service_StartWimp (&49) which is important when writing module tasks that are permanently resident (Volume 3, page 70).