The Principles of Creating Modern TSR Programs Running under the Microsoft Windows Operating System

ZBIGNIEW A. NOWACKI
Institute of Applied Computer Science, Lodz University of Technology

The article presents the principles of creating memory-resident applications, called TSR programs in a modern sense, running under the Microsoft Windows operating system. In this context, a new explanation of the TSR abbreviation has been suggested. Much attention has been devoted to showing why it is worthwhile to write such programs. All typical modules of a modern TSR have been discussed in detail. We have pointed out some differences between such residents and usual Windows applications. Author's functions and procedures facilitating the creation of the former have been provided. We have demonstrated that in some cases a specific program called the residents' supervisor can be found useful. It has been shown that the use of a semaphore gives here a number of advantages.

Key Words: **Windows application, memory-resident, TSR, operating system, multitasking, semaphore**

1. Introduction

Let us recall that in DOS [1], the first operating system for IBM PC-compatible computers, merely one program was running at any given time under normal circumstances. If, however, a program ended its work with the software interrupt INT 27H (called 'Terminate But Stay Resident') or INT 21H/31H [2, 3], the system kept the whole or a portion of program's code in the memory. This created a substitute for multitasking [4], because the control could be transferred back to the pieces of code in the case of occurrence of some events, such as pressing a certain key on the keyboard. Utilities accomplishing this approach were termed TSR ('Terminate and Stay Resident') or memory-resident (shortly resident) programs [5, 6]. They could be treated as a part of the operating system, and even DOS itself, especially in versions 5.0 and later, applied this technique to perform a number of useful functions.

Recently, with DOS no longer required to use Microsoft Windows [7], the majority of PC users have abandoned the former in favor of the latter. However, problems being solved by the DOS applications of the TSR type have remained and even intensified in the age of the Internet. Although writing such programs in the true multitasking environment of Windows is easier (no special programming tricks are needed), not every developer knows how to do it. Let us note that the problem cannot be reduced to hiding the main window (we assume that the word 'resident' means just 'running in background', which seems to be most consistent with the general judgment [8]). Our application still has to test externally or internally generated events and react to them appropriately, i.e., behave like a component of the operating system.

This article has been intended to conduct, among others, a comparative analysis between those traditional TSR programs, and utilities running under Microsoft Windows (albeit not in its DOS box) and normally not revealing their presence to users, but playing the role of a just-in-time tool whenever a specific event occurs in the system or the world around. Of course, our discussion focuses on applications of the latter type, so the reader may benefit from reading this article even if they have never programmed under DOS. We suggest to call the utilities Windows TSR or modern TSR (also embracing applications in other multitasking systems) or briefly TSR programs. Obviously, they have to reside in memory, so the last member of the word-by-word reconstruction of the acronym is justified. But they never terminate work (unless they cease to reside), so the first part must be somehow altered or eliminated. The paper clarifies, inter alia, how this can be done. In this manner TSR programs, so near and dear to the hearts of many people (including the author), will be able to start their life after life.

Obviously, it would have been better if the paper had been written in the second half of the nineties. It is hard to say why this was not done (the author simply did not have the time). But, of course, applications using mechanisms discussed here (hot keys, timers, synthesizing events, etc.) were still being created, at least by some developers, without hindrance. The main contribution of the author is the distinction of a class of programs that are specialized in this matter.

The set of such utilities contains, in particular, Unix daemons. In fact, they also set up a mechanism for being called up either periodically or by an application at a later time, and otherwise remain idle in the background until explicitly stopped. However, we would not want to call programs depicted in detail here 'Windows daemons' because a common practice is exactly the opposite. For instance, in [8] you can find the phrase 'our daemon (TSR in Linux)'. Comparisons of this type are very popular, but they are inaccurate, for the daemon function does not cause the termination of the program as the DOS interrupt did. And only after publishing this text they will be fully justified.

This work can also be regarded as a kind of an implementation report [9]. Our target audience contains, in particular, the group of people that do not like to perform tedious repetitive tasks. It is our hope that the paper will be understandable even to developers not building Windows applications yet. The most effective language to create historical memory-resident programs was Assembler, while it is most practical to write modern TSR utilities in the native language of Windows, i.e., C. Therefore, our procedures and examples have been expressed in it (or in C++). Some rules formulated in the article (involving, for instance, an introduction to the basics of the Windows API approach to programming) are known, but striving for self-sufficiency and clarity we have decided to keep them. (They are also needed to define the class of modern TSR programs.) On the other hand, we offer a number of functions that cannot be found in MSDN [10].

2. The usefulness of modern TSR programs in Microsoft Windows

Do you remember the TSR program running in DOS which pursued a screenshot after pressing *Alt* and *W* at the same time? (Anyway, you do not even have to, because currently another popular utility - being just TSR in our modern sense - does the same in response to *Ctrl-Shift-R*.) This is the example of a hot key denoted usually by *Alt-W* or *Alt+W*. Using methods described in the article you will be able to write your own program of this type, which runs under Windows and satisfies special requirements.

Everything, in principle, that is normally typed on the keyboard or clicked with the mouse can be automatically entered or performed by a TSR after pressing a hot key. This accelerates and simplifies work enabling us to eliminate mistakes. Of course, the larger the sequence of synthesized keys or mouse events, the more profitable the resident use. It should be also made clear that the mechanism can be applied solely to highly repeatable processes.

Another important functionality domain of TSRs embraces loading and starting, also in response to pressing a hot key, common software applications. This method displays evidence of the following advantages, compared to start-up icons on the desktop or in the taskbar: no place on the screen is needed and the resident can provide command-line arguments.

It should be pointed out that in choosing to apply a TSR program we replace the multiple use of tedious routine operations by a single performance of a more intellectual process, i.e., programming. Someone for whom such an exchange is not important should not take to write TSRs.

Computer game players sometimes buy special (rather expensive) keyboards with extra programmable keys being able to replace sequences of moves. Using a TSR is likely to achieve the same purpose at much lesser cost. Furthermore, the action of a hot key can depend on the current game state.

Let us note that our TSR utility does not need to confine itself to testing keyboard-related events. For example, it can examine the values of some pixels on the screen and, depending on the outcome, automatically take the appropriate action.

It is worth paying attention to some side (albeit very positive) effects connected with TSR programs. They concern security on the Internet. For instance, protection against the *man-in-the-browser* and *phishing* attacks [11, 12] may be easier, because applying a hot key you do not need to type the address of the bank website into the browser window (thus, no virus will be able to read it), and you do not need to click on suspicious links.

3. The main section of a TSR

A typical TSR program consists of the following components:

- Creating and hiding the main window.
- Registering hot keys.
- The loop testing events.
- Procedures responding to hot keys.
- Procedures invoked periodically by timers.

The first three elements are contained, as a rule, in the main function starting from the following header:

```
int WINAPI WinMain(HINSTANCE task, HINSTANCE, LPSTR, int )
```

The use of *WinMain* [13, 14] instead of *main* means that we deal here with a so-called Windows application [15-17] requiring the inclusion of the *windows.h* file defining, inter alia, types of the above parameters. In the TSR program we only need the task handle being used in, e.g., the following procedure creating the window [18, 14]:

```
WNDCLASS WindowClass;
memset(&WindowClass,0,sizeof(WindowClass));
WindowClass.hInstance = task;
WindowClass.lpszClassName = “class_name”            // arbitrary name
WindowClass.lpfnWndProc = WindowProc;        // user function declared earlier
if(!RegisterClass(&WindowClass)) return 1;
HWND window;
window = CreateWindowEx(WS_EX_TOOLWINDOW|WS_EX_TOPMOST,
“class_name”, “TSR_name”, WS_OVERLAPPED,
0, 0, 300, 21, NULL, NULL, task, NULL);
ShowWindow(window, SW_HIDE);
```

The arguments of *CreateWindowEx* [19] are, in principle, insignificant (except the task handle and class name), because the window is being hidden [20] at once and usually there is no need to display it later.

At the end of the main function one should put the loop testing events (manifested in coming messages), that is:

```
MSG event;
while(GetMessage(&event, NULL, 0, 0))
{
DispatchMessage(&event);
}
return 0;
```

In contrast to a normal loop, this sequence of instructions does not overburden the system even if it is being performed for a long time and by many programs simultaneously.

Let us note that the message loop in TSRs differs slightly from the analogous one in regular Windows applications. For in the latter [14, 21] the call of *DispatchMessage* [22] is preceded by that of *TranslateMessage* [23] responsible for providing ASCII codes of characters entered from the keyboard. But typical memory-resident programs do not allocate the entry of this device, since this is not needed for handling hot keys. Hence *TranslateMessage* can be omitted here.

In the context of problems discussed in this article, the last procedure is a counterpart of DOS interrupts INT 27H and INT 21H/31H. However, those mechanisms had to finish the program execution, while an application implementing the above loop is treated as a still existing task. A common feature of all the three cases is that programs reside in the memory and test real-world events or events generated by the system. Taking the facts into account we propose to call the counterparts of DOS memory-resident programs running in Windows (and in other multitasking environments, e.g., Linux with its system daemons) 'Test and Stay Resident'. Thus, the TSR abbreviation will be able to remain unchanged.

4. The registration of hot keys

Knowing the TSR window handle (returned by *CreateWindowEx*) we can proceed to submitting the keys whose pressing will wake the resident up. This is done using the call [24]

```
RegisterHotKey(window, id, modifiers, key);
```

where *id* is a unique identifier of the hot key (from 0 to 0xBFFF). The *key* parameter specifies a basic key designated to be pressed; it must be its so-called virtual code being used in Windows. For the letter or number keys it is simply an ASCII code; e.g. 'A' (but not 'a') or '9'. In other cases, you may use a constant (e.g. *VK_F10*) defined in the *winuser.h* header file included automatically by *windows.h* or apply the call [25]

```
MapVirtualKeyEx(scan,3,0)
```

returning the virtual code that corresponds to a scan code given in the *scan* parameter. Finally, the *modifiers* argument (if different from zero) specifies modifiers that must be pressed in combination with *key*. In order to list them, it is possible to use the constants *MOD_ALT, MOD_CONTROL, MOD_SHIFT* and *MOD_WIN* connected (if more than one) by the bitwise inclusive OR operator. If the registration fails (because the same hot key has already been registered by another program or it is being used by the operating system), the return value is zero. Let us add that the statement [26]

```
UnregisterHotKey(window,id);
```

cancels the hot key with the *id* identifier.

5. Handling of hot keys

When discussing the procedure for creating a window we mentioned an application-defined function called *WindowProc* (the name is, of course, arbitrary). It should be declared at using the header

```
LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
```

The function will be invoked whenever an event specified by an integer (termed Windows message) in the second argument, related to the created window, occurs in the system. In the case of TSR programs it is only essential to handle *WM_HOTKEY* (a constant defined in the header file) generated [27] by pressing a hot key whose identifier is passed in the third parameter. Hence, a typical window procedure will be:

```
LRESULT CALLBACK WindowProc(HWND window, UINT WindowsMessage,
    WPARAM wParam, LPARAM lParam)
{if(WindowsMessage==WM_HOTKEY)
    {
    // releasing modifiers
    keybd_event( VK_MENU,0,KEYEVENTF_KEYUP,0);
    keybd_event( VK_SHIFT,0,KEYEVENTF_KEYUP,0);
    keybd_event( VK_CONTROL,0,KEYEVENTF_KEYUP,0);
    keybd_event( VK_LWIN,0,KEYEVENTF_KEYUP,0);
    keybd_event( VK_RWIN,0,KEYEVENTF_KEYUP,0);
    switch(wParam)
         {
         case id:                    // response to the hot key with identifier id
         …
         }
    return 0;                        // return after break in a case label
    }
return DefWindowProc(window, WindowsMessage,
    wParam, lParam);                 // return for unprocessed messages
}
```

Let us add that using the *MSG* structure [28, 29] hot keys can be handled even without the window procedure. For this purpose, yet another version of the message loop should be applied:

```
MSG event = {0};
while(GetMessage(&event, NULL, 0, 0))
{
if(event.message == WM_HOTKEY)
    {
    // releasing modifiers (as above)
    …
    switch(event.wParam)
          {
          case id:                         // response to the hot key with identifier id
          …
          }
    }
DispatchMessage(&event);
}
return 0;
```

In this case, no window should be created, and even the TSR could be a console application. This approach has been used in the example of [24]. However, it is not recommended by the author, especially in more advanced applications.

6. Synthesizing events on the keyboard

In order to synthesize pressing the key with a virtual code *key* the following can be used

```
keybd_event(key,0,0,0);
```

Somewhere further the simulation of its releasing

```
keybd_event(key,0,KEYEVENTF_KEYUP,0);
```

should occur [30]. If the TSR program uses these options after pressing a hot key, releasing modifiers (cf. the previous section) may be necessary, since otherwise keys belonging to actually introduced combinations can affect the outcome of the simulation. However, the reader ought to be warned not to apply the *SetKeyboardState* function [31] for this purpose, because it influences only the current process, that is, the resident. It is also possible to use additional instructions such as

```
if(GetKeyState(VK_CAPITAL) & 1)                    // if CAPS is active
    {
    keybd_event( VK_CAPITAL,0,0,0);
    keybd_event( VK_CAPITAL,0,KEYEVENTF_KEYUP,0);
    }
```

Moreover, the *GetKeyState* function [32] can be used to verify if the key with a virtual code *key* is pressed, in which case the expression

```
GetKeyState(key) < 0
```

returns 1. This allows us to make the operation of hot keys dependent on the status of additional modifiers such as *Insert*.

It is worth mentioning that synthesized keyboard events participate in the activation of hot keys on a par with actual events. If it is negative (e.g., we would like the *F10* key with active

Scroll Lock to be modified, but otherwise unchanged), apply *UnregisterHotKey.* On the other hand, it is possible to synthesize the hot keys of other programs or of the operating system.

7. Synthesizing mouse events

To synthesize mouse motion and button clicking you may use the *mouse_event* function [33]. In most cases clicking the left button is sufficient. For this purpose, it is advisable to include in the resident source code the following function (it is not in MSDN):

```
void ClickLeft  (int x, int y)                    // click the left button at (x,y)
{
SetCursorPos(x, y);                               // set the cursor at (x,y)
mouse_event(MOUSEEVENTF_ABSOLUTE | MOUSEEVENTF_LEFTDOWN,
        x,y,0,NULL);
mouse_event(MOUSEEVENTF_ABSOLUTE | MOUSEEVENTF_LEFTUP,
        x,y,0,NULL);
}
```

Double-clicking can be obtained by calling *ClickLeft* twice (maybe separated by a certain time interval). If necessary, the C*lickRight* function might be implemented via replacing *LEFT* by *RIGHT.*

8. Optimized synthesizing of events

Windows also has the *SendInput* function [34] being able to synthesize input from both the keyboard and the mouse. However, writing its parameters is a bit more complicated. For instance, the implementation of *ClickLeft* using *SendInput* would look as follows:

```
void ClickLeft  (int x, int y)
{
INPUT a[2];
a[0].type=INPUT_MOUSE;
a[0].mi.dx=x;
a[0].mi.dy=y;
a[0].mi.dwFlags=
              MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_LEFTDOWN ;
a[1].type=INPUT_MOUSE;
a[1].mi.dx=x;
a[1].mi.dy=y;
a[1].mi.dwFlags=MOUSEEVENTF_ABSOLUTE|MOUSEEVENTF_LEFTUP ;
SetCursorPos(x,y);
SendInput(2,&a[0],sizeof(INPUT));
}
```

In the case of the keyboard, to apply *SendInput* sensibly we should create an array for each sequence of synthesized keystrokes. To this end you might use a very convenient function defined (it is not in MSDN) by

```
#include <stdarg.h>
#include <stdlib.h>
int SendKeys(int number, ... )
{
int sum = 0, i, key;
va_list marker;
if(!number)
  {
  va_start(marker,number);
  while (va_arg(marker, int)) number++;
  va_end(marker);
  }
va_start(marker,number);
#if defined(INPUT_KEYBOARD) && \
  (!defined(__WATCOMC__) || __WATCOMC__>1290)   // for some compilers
INPUT *array = new INPUT[number];
for(i=0; i<number; i++)
  {
  key = va_arg(marker, int);
  sum += key;
  array[i].type=INPUT_KEYBOARD;
  array[i].ki.wVk = abs(key);
  array[i].ki.dwFlags = key<0 ? KEYEVENTF_KEYUP : 0;
  }
SendInput(number, &array[0], sizeof(INPUT));
delete [ ] array;
#else
for(i=0; i<number; i++)
  {
  key = va_arg(marker, int);
  sum += key;
  keybd_event(abs(key), 0, key<0 ? KEYEVENTF_KEYUP : 0, 0);
  }
#endif
va_end(marker);
return sum;
}
```

The first parameter of *SendKeys* specifies the number of virtual codes passed in succeeding arguments. It may equal zero, which indicates that the same value will occur at the end of the variable parameter list. Note that the releasing of a key is indicated by the arithmetic negation of its code. For example, the very concise and readable call

```
SendKeys(4, VK_CONTROL, 'W',  -'W', -VK_CONTROL)
```

or, equivalently,

```
SendKeys(0, VK_CONTROL, 'W',  -'W', -VK_CONTROL, 0 )
```

synthesizes *Ctrl-W*. The function returns the sum of its variable arguments enabling us to check easily in more complicated cases if all pressed keys have been released (i.e., zero has been returned).

However, the reader should be warned that some high-performance compilers do not contain *SendInput* (but *SendKeys* should work). Thus we do not recommend an unconditional use of *SendInput* instead of *keybd_event* and *mouse_event* although MSDN says that the latter have been superseded by the former. The author uses the *...event* functions with no problems.

9. Installing TSR programs

A TSR can be started with any method known under Windows. In particular, it may be located in the *Startup* folder. This option is good if the resident is to be treated as a component of the operating system. Such a TSR is able not to reveal its presence and can remain unnoticed by the user unless during the session its hot keys are pressed or other respective events happen.

The reader may be concerned that a user could start their TSR under another account and intercept their keystrokes. This will be impossible unless the utility is located in the subfolder *Startup* of the *All Users* folder.

A TSR can be sometimes installed as a Windows Service [35-37]. Such a solution is a preferred technique to build the equivalent of a UNIX daemon, but the class of modern TSR programs is essentially broader than the class of Windows Services. This is so because, according to [35], "These services … do not show any user interface." Precisely speaking, a utility of this type runs in a windows station distinct from the interactive station of the logged-on user. This implies that a Windows Service cannot register hot keys and display anything (hence it may be a console application). Even error messages should not be raised in the user interface, since dialog boxes will not be visible and can cause the program to stop responding.

In [38, 39] there is available a typical modern TSR that is able to be set up as a Windows Service. The description of the program explains how to accomplish this goal.

10. Starting software applications by TSR programs

In multitasking systems residents may start software applications relieving, in the case of Windows, the desktop and taskbar. A number of programs accept command-line arguments which can be easily passed on to them using the functions of the *spawn* family [40]. For example, the connection to a Web site is accomplished via [41, 42]

```
// at the beginning of the program:
char ie[MAX_PATH+1];
// in the main section:
GetSystemDirectory(ie,MAX_PATH+1);                    // get the system disk
strcpy(ie+2,"\\program files\\internet explorer\\iexplore.exe");
        GetShortPathNameA(ie,ie,MAX_PATH+1);   // replace long name by
        short one
// for a hot key:
spawnl (P_NOWAIT,ie,ie,"http://www.nova.pc.pl",NULL); // connect to network
```

This method (it is not presented in [10]) enables you to get a very fast access to large numbers of Web pages.

11. The removal of the resident

Erasing a TSR program running under DOS was feasible albeit realized via a relatively cumbersome procedure, and sometimes it required a prior removal of other residents. For Windows the matter is quite simple, since it is sufficient to invoke [43]

```
PostQuitMessage(0);
```

(Do not use the *exit* function [44] unless you want the system to start behaving in an undefined fashion.) Thus, you may book a hot key to perform the operation of deleting.

Furthermore, since the TSR is shown up as a process in the Windows Task Manager, it can be, if necessary, terminated in a brutal manner.

The question arises whether and when erasing our TSR utility before shutting down the whole system is useful. In the case of DOS the answer was determined by the all-powerful need to save memory (limited, in principle, to 640 kB even if the actual physical memory was much greater). In Windows this solution can be convenient if the resident executable file has been located in the *Startup* folder. It needs to be remembered that when the application is running, its exe file cannot be opened, whence also deleted or replaced by another. Otherwise, it may be an unpleasant surprise when after compiling a new version of the resident the user is not able to move it to the *Startup* folder.

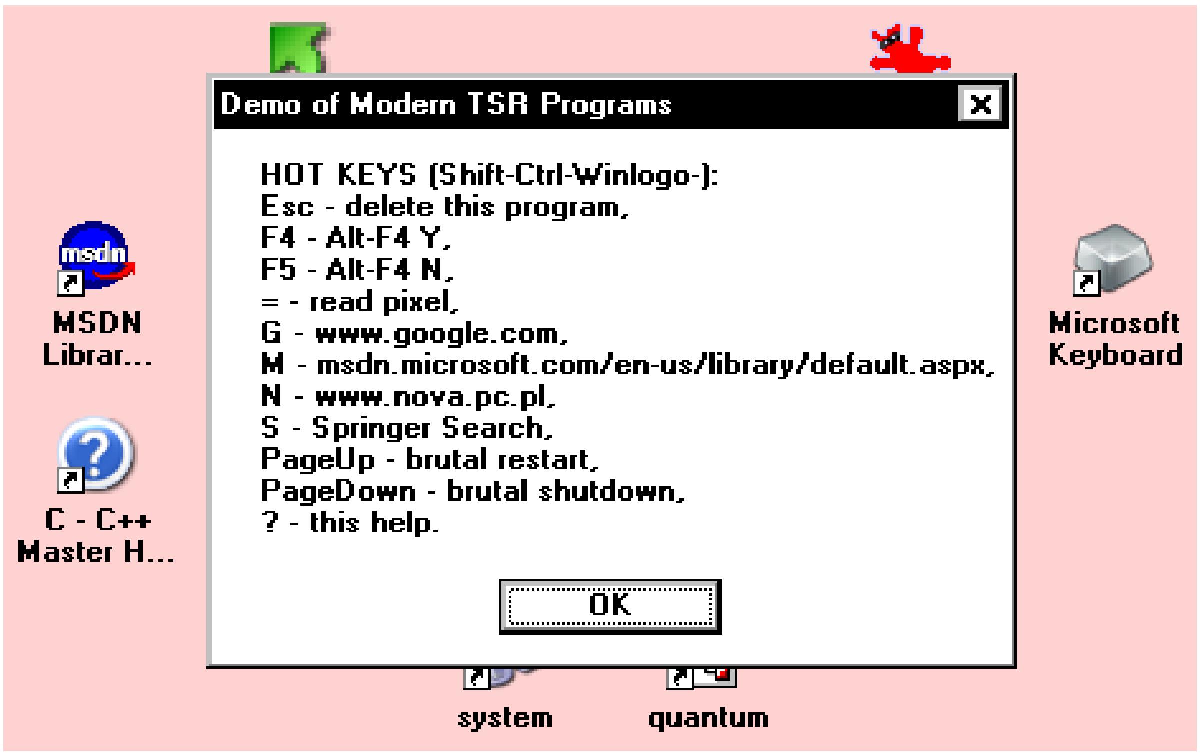
We emphasize once more that modern TSR programs differ from an average Windows application in only two respects: they hide their window (they *stay resident*), and their main task is to listen and respond to some events even when they are not in focus (they *test*). The fact that TSRs do not terminate until explicitly requested to do so is obviously irrelevant - most applications continue running until the user requests termination.

12. Output messages of TSR programs

Although our resident application generally keeps silent, revealing its existence only by its influence on other programs or by starting them, there are two situations in which output messages are useful. Firstly, if you have a lot of hot keys, it is recommended to create an online help pertinent to their action, preferably with one more, easy-to-remember (e.g. '?' or *F1* with modifiers), key. It is convenient to display this information using the following scheme [45]:

```
MessageBox(NULL,"The help content (it may contain newline characters)",
           "The window title (e.g. the program name)",
      MB_OK|MB_TOPMOST|MB_SETFOREGROUND);
```

A demonstration [9] of such a help is shown in Figure 1.



HOT KEYS (Shift-Ctrl-Winlogo-):
Esc - delete this program,
F4 - Alt-F4 Y,
F5 - Alt-F4 N,
= - read pixel,
G - www.google.com,
M - msdn.microsoft.com/en-us/library/default.aspx,
N - www.nova.pc.pl,
S - Springer Search,
PageUp - brutal restart,
PageDown - brutal shutdown,
? - this help.

Fig. 1. An example of the hot keys' help

The second case is, e.g., the testing phase of the program, when it might be helpful to get information about the current values of variables. The easiest way would be to use the *printf* function [46] or the *cout* or *cerr* streams [47], but unfortunately these facilities do not work in Windows applications. Luckily, we can write ... an own function *printf*, whereby this is done in a surprisingly simple way. Namely, it suffices to include the following subroutine [48] in the application source code, somewhere at its beginning:

```
# define  MAXP            256  // maximal output message length, may be changed
#include <stdarg.h>
void printf(char *format,...)
{char string[MAXP];
va_list argptr;
va_start(argptr, format);
vsprintf(string, format, argptr);                 // always possible but a bit unsafe
_vsnprintf(string, MAXP, format, argptr);    // use instead if _vsnprintf is available
vsnprintf(string, MAXP, format, argptr);      // use instead if vsnprintf is available
va_end(argptr);
MessageBox(NULL, string, "Program",          // write down, e.g., the TSR name
MB_OK|MB_TOPMOST|MB_SETFOREGROUND);}
```

The only difference between the implementation of *printf* and standard one is that after displaying a message our version will be waiting for the user to press *Enter* (or *Esc*) or click *OK*.

Furthermore, if your compiler has *vsnprintf* accepting *NULL* for the first parameter [49], you may write a still more convenient function outputting messages:

```
#include <stdarg.h>
void printf(char *format,...)
{va_list argptr;
va_start(argptr, format);
int len = vsnprintf(NULL, 0, format, argptr)+1;
va_end(argptr);
char *string = new char[len];
va_start(argptr, format);
vsnprintf(string, len, format, argptr);
va_end(argptr);
MessageBox(NULL, string, "Program",          // insert, e.g., the TSR name
MB_OK|MB_TOPMOST|MB_SETFOREGROUND);
delete [ ] string;}
```

This version avoids not only potential buffer overflows but also the need to adjust the compile-time constant *MAXP* when the length of a text to be displayed increases.

13. Timers

Some tasks of TSRs cannot be resolved with the mechanism of hot keys, but they require rather a periodic testing of certain conditions and undertaking an action only when they are satisfied. To this end, when working under DOS, we were capturing the interrupt 08H hardware-generated every 55 ms. Under Windows, we can define a number of similar 'interrupts' known as timers by the call [50]

```
SetTimer(window, id, time, ResidentProc);
```

where *id* is a nonzero timer identifier, and *ResidentProc* is the name of an application-defined function (with the same header as that of the window procedure) that will be every *time* milliseconds invoked with the parameters *window*, *WM_TIMER*, *id* and the current system time. One of the *window* and *ResidentProc* arguments can equal *NULL*, which in the latter case implies that the window procedure will be called. The timer may be at any time destroyed with the use of the statement [51]

```
KillTimer(window, id);
```

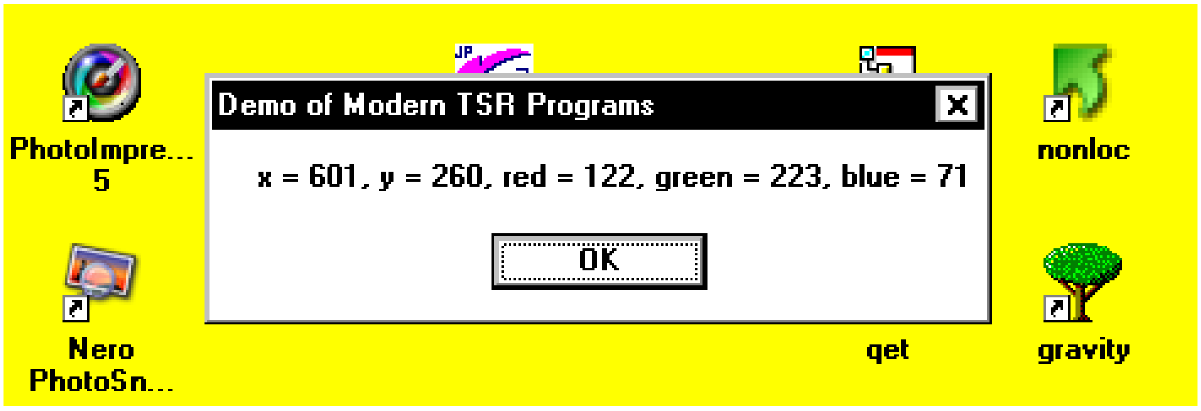
with parameters equal to the first two ones used during creating it (but this option requires *window* to be nonzero).

14. Testing pixels

Suppose we wish to connect to a Web site, and we have prepared a set of keys and buttons that we want to synthesize using our TSR program. It is obvious that we cannot do that at any time, but we have to recognize in a way that the selected page is already loaded. In this and other cases, the TSR should be able to read the RGB components of pixels currently displayed on the screen. This can be done using the *GetPixel* and other functions [52-55], as shown below:

```
{
POINT pt;                                   // for cursor position
HDC screen = CreateDC("DISPLAY", NULL, NULL, NULL);
GetCursorPos(&pt);
COLORREF col = GetPixel(screen, pt.x, pt.y);
DeleteDC(screen);
printf("x = %d, y = %d, red = %d, green = %d, blue = %d",    // own printf
        pt.x , pt.y, col&255, (col>>8)&255, (col>>16)&255);
break;
}
```

This procedure displays the coordinates and the RGB components of the pixel located at the cursor position (see Fig. 2). It is worth to implement it as a response to pressing a hot key, because that way, e.g., you will be able to acquire easily any nice color visible on the screen to your program or page. A utility pursuing this goal can be found in [9].



```
x = 601, y = 260, red = 122, green = 223, blue = 71
```

Fig. 2. Displaying RGB components in response to pressing a hot key

15. The suspension of the resident action

If after starting the browser program your TSR tests in an ordinary loop whether RGB components of some pixels have attained values characteristic for the desired page, the system processor may be overburdened. One sees that it suffices to perform only one such verification within a given time interval. We should, therefore, have a mechanism to stop program execution for a given time period. In C/C++ under DOS we had the *delay* and *sleep* functions [49], but they cannot be used in Windows applications. Instead, there is a new function *Sleep* [56] whose single argument specifies for how many milliseconds the program is to be suspended. For instance, the following procedure

```
HDC screen = CreateDC("DISPLAY",NULL,NULL,NULL);
do Sleep(100);
while(RGB(192,192,192 )!= GetPixel(screen,8,13));
DeleteDC(screen);
```

tests every 100 milliseconds (this timeout value can be recommended for most applications) whether the pixel with the screen coordinates (8,13) has all the RGB components equal to 192.

16. The residents' supervisor

A similar sequence of instructions may be used, for example, in order to wait for the desired Web page. This method, however, has a major flaw: exceptional circumstances (e.g., a server crash) can cause our program to perform a practically infinite loop. (It will be easily recognized that the resident is in this state, because it will not react to other hot keys, e.g., it will not display the help.) Introducing a maximal waiting time is not a perfect solution; the case may be that the time period is too long or too short. One would like to be able to stop the execution of this loop upon request at any time and, surprisingly, there is such a simple possibility. However, instead of the *Sleep* function you should apply (at least in more extensive implementations) semaphores or another method of interprocess communication.

Let us recall that in Microsoft Windows parallel programming has been implemented in the form of so-called multithreading [57]. A detailed discussion of concurrent processes can be found in [58, 59], whereas here we confine ourselves to the description of the minimum set of required instructions. First of all, somewhere at the beginning of the TSR program you should put the semaphore handle declaration and definition of the *SLEEP* macro (it is not in MSDN):

```
HANDLE sema;
#define SLEEP(time) \
  {if(WaitForSingleObject(sema,time) != WAIT_TIMEOUT) return 0;}
```

Further, in the main section *sema* should be initialized by [60]

```
sema = CreateSemaphore(NULL,0,1, semaphore_name);
```

where *semaphore_name* is an arbitrarily chosen nonempty character string. You may now use the statement of the form of *SLEEP(time)* that normally will work exactly like *Sleep(time)*, although the suspension of the TSR action will be accomplished by the *WaitForSingleObject* function [61]. If all of the delays needed for handling *WM_HOTKEY* are realized via the *SLEEP* macro, in another program, called the residents' supervisor, you can perform the same (including *semaphore_name*) semaphore creation and register a hot ('emergency') key executing the following procedure [62]:

```
ReleaseSemaphore(sema,1,NULL);
// two next statements are  essential only in case of false alarm, i.e. pressing
// the emergency key without any pending state
Sleep(100);                                    // parameter  may be changed
WaitForSingleObject(sema,0);
```

Then pressing it causes to abandon the performing of the window function containing *SLEEP*. Let us note that the use of the residents' supervisor as a separate process is necessary here; the emergency key cannot be registered in the same single-threaded TSR program whose pending state we want to stop.

One might wonder whether in the definition of *SLEEP* we could replace *return 0* by *break*. To answer, it should be noted that after finishing the loop we usually synthesize some mouse and keyboard events. If the loop is interrupted by pressing the emergency key, these events occur in the wrong context, e.g., some random programs having an icon on the desktop are started. Thus, in general *return 0* is better.

The situation becomes complicated somewhat whenever such delays are needed in functions being invoked from the window procedure. In this case you may use the *LongSleep* function defined (it is not in MSDN), together with ancillary things, by

```
#include <setjmp.h>
jmp_buf jumper;                                    // arbitrary  name
void LongSleep(DWORD time)
  {if(WaitForSingleObject(sema,time) == WAIT_OBJECT_0) longjmp(jumper,1);}
```

It requires also the following statement

```
If(setjmp(jumper))
  return 0;                      // the emergency action, may be changed
```

being performed somewhere at the beginning (before the first use of *LongSleep*) of the window procedure. Then it is no longer needed to use *SLEEP*, because *LongSleep* will be able to replace it everywhere. (As a result, a macro is superseded here by a function, which is generally recommended.) We see that if the emergency key is pressed, *LongSleep* performs a nonlocal *goto* to finish the window procedure.

Note that *LongSleep* can be called with the *INFINITE* parameter. For instance, the following sequence of instructions

```
if(setjmp(jumper)) goto label;
LongSleep(INFINITE);
label:
```

suspends the program execution until the moment when the emergency key is pressed.

Let us add that this method can be equally well applied to stop, on request, the pending state of normal programs, even console applications including *windows.h.* It only requires specifying the alternative action (e.g., the call of *PostQuitMessage* or *exit*). Of course, the supervisor registering the emergency key still has to be TSR.

Developing his modern TSR application, the author recognized at one point that it would be better to have three different programs (for the screen, clipboard, and the Internet). Later he added the residents' supervisor with one universal emergency key, and in the other programs he simply replaced *Sleep* by *SLEEP*. Afterwards, he substituted *LongSleep* for *SLEEP*. It was easy and pleasant. On the other hand, a modification of the loop testing events or the use of the *PeekMessage* function [63] do not live up to our expectations in this area, although such solutions would be able to keep our TSR utility single-threaded. However, they would require significant changes in the source code creating an opportunity to make mistakes. For example, you could write

```
cancel = false;
do {
    Sleep(100);
    recursing = true;
    while (PeekMessage(&event, NULL, 0, 0, PM_REMOVE))
           {
           DispatchMessage(&event);
           }
    recursing = false;
    if(cancel) …          // an emergency action (e.g., return 0, break, etc.)
} while (RGB(…) != GetPixel(…));
```

where *recursing* is tested in *WindowProc* to ignore the hotkey if its handler has already been active, and *cancel* is set by the window procedure when the emergency key is pressed. This piece of code seems to be too extensive, and a possible macro would be rather too elaborate. Finally, last but not least, the methods of this type would necessitate the use of several emergency keys, one for each TSR program.

17. Conclusion

In the paper we define and study a class of memory-resident applications, called contemporary TSR programs, working under the Microsoft Windows operating system. A comparative analysis between them and historical residents running in DOS has been conducted. In the latter case, the TSR acronym was being deciphered as 'Terminate and Stay Resident'. Adapting the concept to the philosophy of Windows, we have proposed to explain TSR as 'Test and Stay Resident'. This expansion of the initialism [64] points out the fact that in a multitasking environment (including that of Linux) the utilities do not terminate work before 'staying resident'; they test events in the system and surrounding world and perform a suitable action. Thus, a TSR behaves like an element of the operating system. Let us add that recursive versions 'TSR Starts and Resides' or 'TSR Stays Resident' might be also accepted, especially in the hacker community [65].

This article has been conceived as a self-contained guide [cf. 9] to write such memory-resident programs by those who know the C/C++ language [66-68] even if they have no experience in building so-called Windows applications. Much attention has been devoted to showing why it is worthwhile to write modern TSRs. Their all modules, i.e., creating and hiding the main window, registering hot keys, the loop testing events, procedures being executed in response to hot keys, and timer routines have been discussed in detail. We have pointed out the differences between the TSRs and usual Windows applications. Author's functions and procedures facilitating the creation of the former have been given. We have obtained the most advanced result stating that in some circumstances the usage of a specific program called the residents' supervisor can be necessary. It has been demonstrated that its design is able to be greatly simplified by the use of semaphores.

References

1. Murdock EE. *DOS the Easy Way.* Downloadable E-books: 2007.

2. Ralf Brown's Interrupt List.
http://www.ctyme.com/rbrown.htm [27 July 2013].

3. Scanlon JL. *8086/8088/80286 Assembly Language.* Brady Books: 1988.

4. Stallings W. *Operating Systems: Internals and Design Principles.* Prentice Hall: New Jersey, 2009.

5. Hyde R. *Art of Assembly Language. Second Edition.* No Starch: San Francisco, 2010.

6. Tomczyk M. *The Methods of Creating Resident Programs in the DOS and Windows Systems*, Master's thesis supervised by Z. A. Nowacki. Lodz University of Technology: 2001.

7. Kominiak L. *The Genesis, Evolution and Structure of Microsoft Windows Operating Systems*. Master's thesis supervised by Z. A. Nowacki. Lodz University of Technology: 2007.

8. Writing resident programs under Linux.
http://rudy.mif.pg.gda.pl/~bogdro/linux/tsr_tut_linux_en.html [27 July 2013].

9. Modern TSR programs by Zbigniew Andrzej Nowacki.
http://www.nova.pc.pl/software.htm [27 July 2013].

10. MSDN Library.
http://msdn.microsoft.com/en-us/library/default.aspx [27 July 2013].

11. Lance J, Stewart J. *Phishing Exposed.* Syngress Publishing: 2005.

12. Zimoch M. *Privacy and Security on the Internet*. Master's thesis supervised by Z. A. Nowacki. Lodz University of Technology: 2009.

13. WinMain Entry Point.
http://msdn.microsoft.com/en-us/library/ms633559(v=vs.85).aspx [27 July 2013].

14. WinMain - application entry point.
http://www.toymaker.info/Games/html/winmain.html [27 July 2013].

15. Petzold C. *Programming Windows.* Microsoft Press: 1998.

16. Richter J. *Programming Applications for Microsoft Window.* Microsoft Press: 1999.

17. Prosise J. *Programming Windows with MFC.* Microsoft Press: 1999.

18. WNDCLASS Structure.
http://msdn.microsoft.com/en-us/library/ms633576(v=vs.85).aspx [27 July 2013].

19. CreateWindowEx Function.
http://msdn.microsoft.com/en-us/library/ms632680(v=vs.85).aspx [27 July 2013].

20. ShowWindow Function.
http://msdn.microsoft.com/en-us/library/ms633548(v=vs.85).aspx [27 July 2013].

21. Creating a Message Loop.
http://msdn.microsoft.com/en-us/library/ms644928(v=vs.85).aspx#creating_loop
[27 July 2013].

22. DispatchMessage Function.
http://msdn.microsoft.com/en-us/library/ms644934(v=vs.85).aspx [27 July 2013].

23. TranslateMessage Function.
http://msdn.microsoft.com/en-us/library/ms644955(v=vs.85).aspx [27 July 2013].

24. RegisterHotKey Function.
http://msdn.microsoft.com/en-us/library/ms646309(v=vs.85).aspx [27 July 2013].

25. MapVirtualKeyEx Function.
http://msdn.microsoft.com/en-us/library/ms646307(v=vs.85).aspx [27 July 2013].

26. UnregisterHotKey Function.
http://msdn.microsoft.com/en-us/library/ms646327(v=vs.85).aspx [27 July 2013].

27. WM_HOTKEY Message.
http://msdn.microsoft.com/en-us/library/ms646279(v=vs.85).aspx [27 July 2013].

28. MSG Structure.
http://msdn.microsoft.com/en-us/library/ms644958(v=vs.85).aspx [27 July 2013].

29. GetMessage Function.
http://msdn.microsoft.com/en-us/library/ms644936(v=vs.85).aspx [27 July 2013].

30. keybd_event Function.
http://msdn.microsoft.com/en-us/library/ms646304(v=vs.85).aspx [27 July 2013].

31. SetKeyboardState Function.
http://msdn.microsoft.com/en-us/library/ms646314(v=vs.85).aspx [27 July 2013].

32. GetKeyState Function.
http://msdn.microsoft.com/en-us/library/ms646301(v=vs.85).aspx [27 July 2013].

33. mouse_event Function.
http://msdn.microsoft.com/en-us/library/ms646260(v=vs.85).aspx [27 July 2013].

34. SendInput Function.
http://msdn.microsoft.com/en-us/library/ms646310(v=vs.85).aspx [27 July 2013].

35. Introduction to Windows Service Applications.
http://msdn.microsoft.com/en-us/library/d56de412(VS.80).aspx [27 July 2013]

36. Services.
http://msdn.microsoft.com/en-us/library/ms685141.aspx [27 July 2013].

37. How To Create a User-Defined Service.
http://support.microsoft.com/default.aspx?scid=kb;en-us;137890 [27 July 2013].

38. Nowacki ZA. File Duplicator for the Microsoft Windows Operating System. *Elektryka* 2011; **1108** (123): 39-50.

39. File Duplicator: An Example of Windows Services by Zbigniew A. Nowacki.

http://www.nova.pc.pl/filedupl.pdf [27 July 2013].

40. spawn functions.
http://www.users.pjwstk.edu.pl/~jms/qnx/help/watcom/clibref/src/spawn.html
[27 July 2013].

41. GetSystemDirectory Function.
http://msdn.microsoft.com/en-us/library/ms724373(v=vs.85).aspx [27 July 2013].

42. GetShortPathName Function.
http://msdn.microsoft.com/en-us/library/aa364989(v=vs.85).aspx [27 July 2013].

43. PostQuitMessage Function.
http://msdn.microsoft.com/en-us/library/ms644945(v=vs.85).aspx [27 July 2013].

44. exit Function.
http://msdn.microsoft.com/en-us/library/k9dcesdd(v=vs.80).aspx [27 July 2013].

45. MessageBox Function.
http://msdn.microsoft.com/en-us/library/ms645505(v=vs.85).aspx [27 July 2013].

46. printf.
http://www.cplusplus.com/reference/clibrary/cstdio/printf/ [27 July 2013].

47. Open Watcom C++. Class Library Reference.
http://www.openwatcom.org/ftp/manuals/current/cpplib.pdf [27 July 2013].

48. vsprintf.
http://www.cplusplus.com/reference/clibrary/cstdio/vsprintf/ [27 July 2013].

49. Watcom C Library Reference. Volume 1.
ftp://ftp.heanet.ie/disk1/openwatcom/manuals/1.5/clib.pdf [27 July 2013].

50. SetTimer Function.
http://msdn.microsoft.com/en-us/library/ms644906(v=vs.85).aspx [27 July 2013].

51. KillTimer Function.
http://msdn.microsoft.com/en-us/library/ms644903(v=vs.85).aspx [27 July 2013].

52. GetPixel Function.
http://msdn.microsoft.com/en-us/library/dd144909(v=vs.85).aspx [27 July 2013].

53. GetCursorPos Function.
http://msdn.microsoft.com/en-us/library/ms648390(v=vs.85).aspx [27 July 2013].

54. CreateDC Function.
http://msdn.microsoft.com/en-us/library/dd183490(v=vs.85).aspx [27 July 2013].

55. DeleteDC Function.
http://msdn.microsoft.com/en-us/library/dd183533(v=vs.85).aspx [27 July 2013].

56. Sleep Function.
http://msdn.microsoft.com/en-us/library/ms686298(v=vs.85).aspx [27 July 2013].

57. Multiple Threads.
http://msdn.microsoft.com/en-us/library/ms684254(v=vs.85).aspx [27 July 2013].

58. Concurrent Processes: Basic Issues.
http://cnx.org/content/m12312/latest/ [27 July 2013].

59. Szurgot R. *Parallel Programming in Windows.* Master’s thesis supervised by Z. A. Nowacki, Lodz University of Technology: 2009.

60. CreateSemaphore Function.
http://msdn.microsoft.com/en-us/library/ms682438(v=vs.85).aspx [27 July 2013].

61. WaitForSingleObject Function.
http://msdn.microsoft.com/en-us/library/ms687032(v=vs.85).aspx [27 July 2013].

62. ReleaseSemaphore Function.
http://msdn.microsoft.com/en-us/library/ms685071(v=vs.85).aspx [27 July 2013].

63. PeekMessage Function.
http://msdn.microsoft.com/en-us/library/ms644943(v=vs.85).aspx [27 July 2013].

64. *Merriam-Webster's Dictionary of English Usage.* Merriam-Webster, Inc: 1994.

65. Recursive Acronym.
http://catb.org/jargon/html/R/recursive-acronym.html [27 July 2013].

66. Kernighan BW, Ritchie DM. *The C Programming Language. Second Edition.* Prentice Hall: New Jersey, 1988.

67. Stroustrup B. *The C++ Programming Language. Third Edition.* Addison Wesley: New Jersey, 1997.

68. C Language Tutorial.
http://einstein.drexel.edu/courses/Comp_Phys/General/C_basics/ [27 July 2013].