[illegible]

NASA-CR-201452

FAST INTERRUPT PLATFORM FOR EXTENDED DOS

T. W. Duryea
Rocketdyne Division, Rockwell International Corp.
Canoga Park CA

ABSTRACT

Extended DOS offers the unique combination of a simple operating system which allows direct access to the interrupt tables, 32 bit protected mode access to a 4096 MByte address space, and the use of industry standard C compilers. The drawback is that fast interrupt handling requires both 32 bit and 16 bit versions of each real-time process interrupt handler to avoid mode switches on the interrupts. A set of tools has been developed which automates the process of transforming the output of a standard 32 bit C compiler to 16 bit interrupt code which directly handles the real mode interrupts. The entire process compiles one set of source code via a make file, which boosts productivity by making the management of the compile-link cycle very simple. The software components are in the form of classes written mostly in C. A foreground process written as a conventional application which can use the standard C libraries can communicate with the background real-time classes via a message passing mechanism. The platform thus enables the integration of high performance real-time processing into a conventional application framework.

INTRODUCTION

Various operating systems are available to facilitate the programming of real-time systems on standard IBM PC compatible hardware and other hardware. Most of these options suffer the drawback of 16 bit operation and reliance on specialized development tools (i.e., compilers and linkers). The methodology presented here originated from a neophyte's need to write a fast control system for a specialized imaging system which images Hydrogen leaks near a rocket engine. It has become apparent that the components already written are usable in a variety of applications. Two examples of situations where the software could be quickly adapted are an infrared scene generator which requires fast control of an optical scanner to generate scene objects and an imaging system to quickly inspect metallic surfaces for subtle defects such as scratches. The software is also embeddable, as Phar Lap has available an embedded version the 386|DOS. The trend in aerospace hardware in fact seems to be towards supplying embedded software modules on various hardware assemblies.

Work prior to the present developments was on a 16 bit platform. After receiving advice from several sources concerning the necessity of using expensive, specialized, or unobtainable hardware and software platforms to host real-time development work, it was decided that the standard 286 PC computer offered the advantages of availability, simplicity, and standard software development tools which were cheap and which worked well. A control system was written using Microsoft QuickAssembler. The control system worked well, although the software was somewhat cumbersome. The architecture was a simple interrupt driven background process for both a millisecond timer interrupt and a serial port interrupt. The bulk of the program was a foreground process which was written as an ordinary application, and which communicated with the background processes via message passing functions called in a loop in the shell input routine which serviced the keyboard.

Several observations were made in the course of the 16 bit work. The use of standard compilers is desirable due to commercial pressures to produce a reliable product for mass consumption. The use of standard hardware is desirable in times of reduced budgets and of the necessity of rapid response to new challenges. The use of stub interrupt handlers is useful as these handlers can call handlers written in C as ordinary functions, which allows most of the difficult work to be written in C. The avoidance of special C keywords (such as interrupt) should be avoided, as bugs and lack of standardization across compilers will become a problem. The use of a MAKE file is important; lack of ability to recompile a fresh working set of code leads to serious bug problems and is a drain on productivity. Finally, the existence of some sort of coherent framework for the programming effort boosts productivity, eliminates many serious and time-consuming bugs, and most importantly offers modularity and reusability.

The above changes were implemented when the software was converted to 32 bit 386 operation. Previous familiarity with the Phar Lap 386|DOS extender product[1] suggested that it would be a good platform for 32 bit real-time development. The product is straightforward and reliable, and has a simple and fast interrupt structure which is well documented. The Phar Lap extender does not run the real mode portion of a program in virtual 86 mode, which avoids timing problems with things such as indirect interrupt vectoring. If an application hooks a real mode or protected mode interrupt, it gains control directly through an extender stub handler which changes the ring level if necessary. Thus one can write real-time 32 bit code almost as easily as one would for 16 bit usage, except for the problem of mode switches.

SOFTWARE ARCHITECTURE

High performance real-time software under extended DOS requires a minimal number of mode switches during process interrupts, especially when fast (millisecond or less) interrupt rates are involved. The usual method for avoiding mode switches is to write separate real mode and protected mode handlers for a software component. This approach generates efficient code but has the drawback that two versions of the same algorithm must be kept congruent. Also, it is difficult to write the real mode code in any language except assembly language. These constraints make it difficult develop real mode components in a productive manner and greatly increase the risk of bugs which are difficult to isolate.

The software architecture detailed here is driven by the necessities of writing high performance real-time code in C, of compiling through a make file, of being able to make changes simply without generating bugs, of having 32 bit access to memory, and of component reusability. The two critical ingredients of the architecture are the ability to compile a .C source file to run in real mode and the adoption of what is essentially an object oriented structure to the components.

Thus far the compilers used in this work are the MetaWare HighC compiler and the Phar Lap 386ASM assembler. The MetaWare compiler has an option which generates assembly source files. The files will not directly assemble without problems, as correct assembly of the files is currently not supported. Also, the instructions in the files are not necessarily the same object code instructions generated by the compiler. Nonetheless several non-trivial classes were written using these assembly source files.

It was noticed upon early examination of the MetaWare assembly files that the instruction sequences contained within would assemble real mode code if a number of subtle changes were made to the files. The principal change is to change the segment declaration lines (containing the SEGMENT keyword) to change the segment names and convert the segments to USE16 'CODE' attributes. All standard 32 bit 80386 instructions using any addressing mode will assemble for USE16 segment operation. The only constraint is that any dereferenced address offset which exceeds 64K will generate a processor exception in real mode.

Fig. 1 portrays the execution flow of a real-time interrupt under the new architecture. One real mode stub handler and one protected mode stub handler (written in assembler) are installed to intercept all real-time hardware interrupts at several different entry points. Because the Phar Lap handler provides a fresh stack, the stub handlers are reentrant. There are no mode switches in the interrupt generation process. The invoked stub handler in turn calls the class interrupt handler implied by the interrupt vector. Each real mode and protected mode class handler pair is derived from one .C source file. The handlers are ordinary C functions which use no special keywords at all (which avoids the problems associated with obscure keywords). The process of generating two handlers from one source file is termed "dualing", and the handlers are referred to as dual mode functions.

A hardware interrupt either occurs while the processor is in real mode or protected mode. The static data addressed by a class handler must be addressable in either mode; it is thus necessary to have all static data used by the handlers resident in conventional memory (i.e., below 1 MB). Both of the dual mode handlers should execute precisely the same algorithm on precisely the same data. Fig. 1 shows that the class static data is arranged as instance slots grouped into class slots which are bundled with the real-mode code. Fig. 2 provides more detail on the conventional memory arrangement. The simplest way to get this arrangement loaded into conventional memory is to use a simple assembly source file to force the segment ordering, (which gets the whole fig. 2 arrangement into the bottom of the load image) and to use the -REALBREAK extender switch to ensure that the necessary portion gets loaded below 1 MB.

Fig. 3 displays the parts of a real-time class. Typically a trivial assembly source file declares space for N instances to a segment named _REALDATA of class

'CODE'. A dual mode .C source file contains the handler source code and code for specialized dual mode functions. These functions may be called by the handler and by other real-time software classes. Dual mode functions which operate on static data must have some kind of semaphored access to the data, as the functions must not operate on static data when a handler already operating on the data is itself interrupted. The class handlers are written without regards to possible interruption (except where timing problems are a concern), but some dual mode functions can possibly return with a Data_Access_Denied return code. The calling process must be able to work around this.

The protected mode class functions (i.e., the foreground functions) in fig. 3 are conventional C functions which present the high level class interface to the conventional part of the application, called the foreground process. The real-time classes form the part of the application called the background processes. The division of the application into these two parts makes the majority of the work no more difficult than writing an ordinary C program. Thus the kind of reusability associated with object oriented programming is imparted to the real-time components. Development of the real-time classes is simplified due to the fact that one is writing ordinary C code.

Most non-trivial applications require some sort of communication between the foreground and background on a regular basis. Two examples are the need to drain and process serial port input and the need to transfer interrupt trace debugging data from a small conventional memory buffer to a much larger extended memory trace buffer. A simple way to ensure this communication is to execute a servicing function in a keyboard input loop, which allows a shell-driven user input mode. Any long foreground execution path should call the same function at various times. Another possibility is to hook the timer tick to always gain control in protected mode and call the servicing function.

IMPLEMENTATION DETAILS

The following details relate to the software architecture implementation. One important consideration is that the procedure used to generate real mode code from the .C source file is general in the sense that only one algorithm is needed to convert all files: i.e., only one single-pass editor need be written. The details regard only the use of the MetaWare compiler.

The real-time stub handlers are part of a special single-instance class named IS (which mimics a C++ PUBLIC base class). The role of IS is to provide a standard interface for installation and usage of the real-time classes. When a real-time class instance constructs, it provides its inherited IS base class the interrupt vector, the addresses of both dual handlers, and a pointer to its conventional memory static data. The IS class stores both real mode and protected mode far addresses of these quantities in tables for quick access by the stub handlers. The only class handler access to static data is through its structure pointer.

The current constraint on conventional memory length is that all of the code be less than 64K in length. If far data pointers in the C source are used, the IS stack pool may be up to 64K in length and any class instance may be up to 64K in length. If near data pointers are used, the stack pool and all class instance conventional static data must fit in less than 64K. Regarding far pointer usage, MetaWare and Watcom compilers support the far keyword, while Microsoft and Borland do not. Also, use of far pointers in C code may result in many segment loading instructions, which slow down execution considerably. (Class handlers rewritten in assembly language need only set the ES register once per pass and use segment overrides.) In general, a good strategy is to make the conventional memory usage small and use near pointers.

The role of the stub handlers is mainly to secure a stack from a stack pool, to call the appropriate class handler, and to send the EOI signal and return. The stack pool is not necessary (the Phar Lap handler uses a fresh stack), but facilitates debugging, as the stacks exist in the application data segment. The stub handler pushes a far pointer to the correct static data instance and calls the handler function. Several quantities such as the interrupted address are pushed before the static data pointer, which allows ready access of debugging parameters to the class handlers.

If the called class handler uses near pointers, the data addressing is successful, as the real mode DS register and SS register both point at or just below the beginning of the stack pool portion. The real mode static data pointer offset is fixed up to this value. If far pointers are used, the real mode DS points just below the correct instance, allowing 64K per instance. It is important when calling the real mode class handler that two padding bytes be pushed just before the call instruction, to maintain the proper stack frame across the call.

To generate the dualed real mode 32 bit object code the following procedure works: first the C compiler is run to generate the assembly output file, which should have a different root file name. The file is then processed by a single pass editor which regenerates an altered file which then is assembled by the assembler. The make file ensures that any changes to the .C source file generates new copies of both object code files.

The single pass editor was written to read in a text file to a heap. A directory contains the offsets of each text line, which makes the editing process straightforward.

The following set of commands has been found to convert MetaWare HighC .S assembly output files successfully. The following lines should be included into the dual mode files:

```
#define Glue(x, y) x ## y
#ifndef Realmode
#define Dual(x) x
#else
#define Dual(x) Glue(__, x)
#endif
```

Thus a function int func(int arg) declared as int Dual(func)(int arg) will obtain the name __func when the switch -DRealmode is used on the compilation. This prevents name recurrences. The function should be called in the form Dual(func)(value);.

The exact set of needed editor commands changes from release to release:

1: Change the "TITLE name.c" line to "TITLE rname.c" to not confuse the librarian.
2: Replace the "_TEXT segment" line with "_REALTEXT SEGMENT PUBLIC DWORD USE16 'CODE'" to change the segment declaration.
3: Change the "_TEXT ends" line to "_REALTEXT ENDS".
4: Delete the "CONST segment" and "CONST ends" lines (there should be no literal strings in the source file).
5: Insert the line ".386" near the file beginning.
6: Delete the lines "extrn _mwargstack", "extrn _mwargstack:NEAR", "extrn _mwgoc", and "extrn _mwgoc:NEAR" to prevent warnings.
7: Replace the line "CGROUP GROUP _TEXT" with "CGROUP GROUP _REALTEXT"
8: Replace the line "DGROUP GROUP " with "DGROUP GROUP _DATA".
9: Move lines containing the word "extrn" such that they follow the next line which contains the word "public".
10: For all lines containing the word "call", precede the line by "sub esp, 2", and follow it by "add esp, 2". This aligns the stack frame for calls to dual mode functions.
11: Replace the lines "leave" with the set of lines "mov esp, ebp" and "pop ebp". This prevents the 16 bit form of leave from being assembled.
12: Delete the lines "mov eax,ds".
13: Replace the lines "mov es,eax" with the pair of lines "mov ax, ds", "mov es, ax" to facilitate far pointer usage.
14: Replace the lines "les ecx,dword ptr 8[ebp]" with "les ecx, FWORD PTR [ebp][8]" to facilitate far pointer usage.

The above dualing procedure was used to write several real-time classes, most notably a 8250-16550 serial port interface which could be called from within other real-time classes. The required set of editor commands required little maintenance as the amount of high level source code increased, which was the hoped-for result when the work began. Some additions need to be made to facilitate things such as argument passing when dualed functions are called. The only assembler instructions in MetaWare which never worked involved bit fields. The changes needed above do not correct errors in the outputted assembler instructions, but in fact ensure that 32 bit instructions are not misinterpreted within a USE16 segment. Obviously an assembly output file which is guaranteed to assemble correct object code (identical to .OBJ output) and output instructions which specifically refer to 32 bit operations would be ideal.

A number of simple considerations must be kept in mind as dual mode .C files are being written. Stack checking must be off. A handler should have a static data structure pointer as its sole argument, unless debugging arguments are to be added. No standard library functions can be called from dual mode functions (all called functions must be of dual mode form). Things such as function pointers must be handled carefully. The volatile keyword should be used on the static data structure, but this alone will not prevent instruction overhang which causes real-time bugs. For example, suppose a one-bit flag is to be changed in a flags word. High level compilers typically copy the word to a register, alter the register, then replace the word. If a flag changing interrupt occurs during this sequence, the word changed by the interrupt will be overwritten by the interrupted process. The volatile keyword will reduce the instruction overhang, but will not eliminate it. These bugs are prevented by knowing the behavior of the compiler and using short (assembly) functions with names such as cl_i() and st_i() to control the interrupt flag when necessary. Good design technique can avoid many situations where overhang could be a problem.

The fastest way to develop a new class is to start with a trivial set of working code and test a minimal set of interacting classes as incremental changes are made to the new class. Ironically, a slow machine such as a 386 helps out, as the processor is more easily overwhelmed than a faster processor. Reliability of the class is enhanced by driving the test program at a rate which generates overhead problems. The class may be made fault tolerant to many of the errors which occur. Such behavior enhances the reliability of the class when it interacts with other classes later on. The ability to store large amounts of trace debugging information and to have access to interrupt stacks, interrupted addresses, and timing information can help pinpoint subtle class interaction bugs. All interrupt trace debugging features should be triggered on a single #define to allow the complete removal of the feature.

The implementation of this compilation technique using other compilers should be straightforward, although the task has yet to be performed. The assembly output files of the compiler must be examined, and a new set of editor commands which converge to one set of universal commands must be identified. Most standard 32 bit compilers don't support the far keyword or the int386 function calls. Most 32 bit compilers and debuggers which run on the Phar Lap extender are Windows NT products or WIN32S products, which cause problems with the obsolescent REALBREAK switch. A lack of reliance on this switch is therefore desirable.

CONCLUSIONS

The methodology described here has been implemented and is known to work. The tasks of converting to other major C compilers and of avoiding REALBREAK have not been implemented, but are expected to be straightforward. The actual speed or maximum interrupt rate of an application depend on the nature and complexity of the processes involved. This architecture has the property that the overhead is minimized in a system which provides a uniform framework for installing real-time components. Thus as the x86 processor family advances programs written in this platform gain the same speed increases.

Several classes have been written besides the IS interface class. They include a real-time serial port class and a trace buffer debugging class which can store large amounts of interrupt trace data. A simple scheduler class to administrate several control system algorithms is under development. The classes are easily reused in other applications.

ACKNOWLEDGMENT

The work reported here was supported by NASA Marshall Space Flight Center (contract NAS8-40000).

REFERENCE

Baker, M. Steven, and Schulman, Andrew, "80386-based Protected Mode DOS Extenders", in Extending DOS (Ray Duncan, ed.), Addison-Wesley Publishing Company, Inc., Reading, Mass, p. 193, 1990.

FIG. 1: HARDWARE INTERRUPT ARCHITECTURE

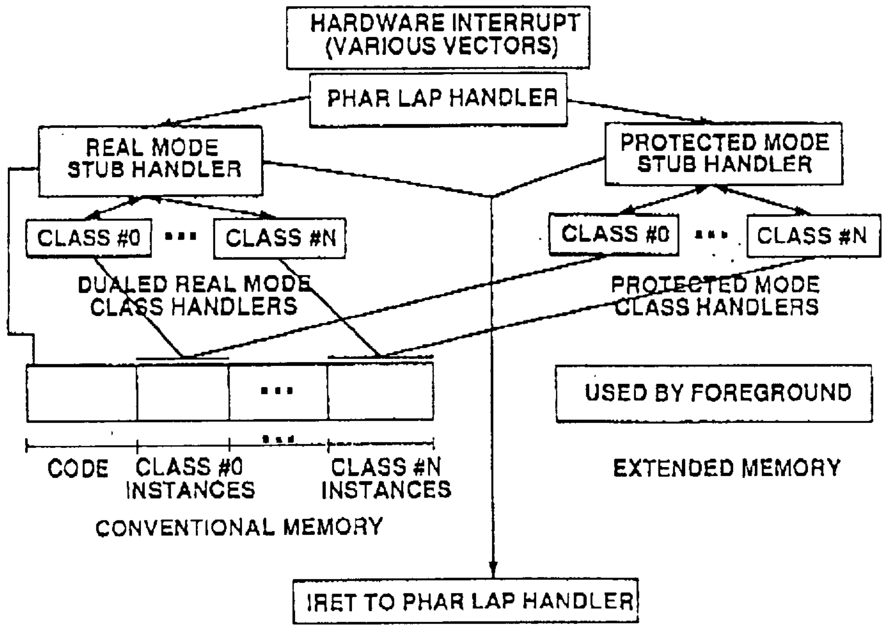


FIG. 2: CONVENTIONAL MEMORY MAP

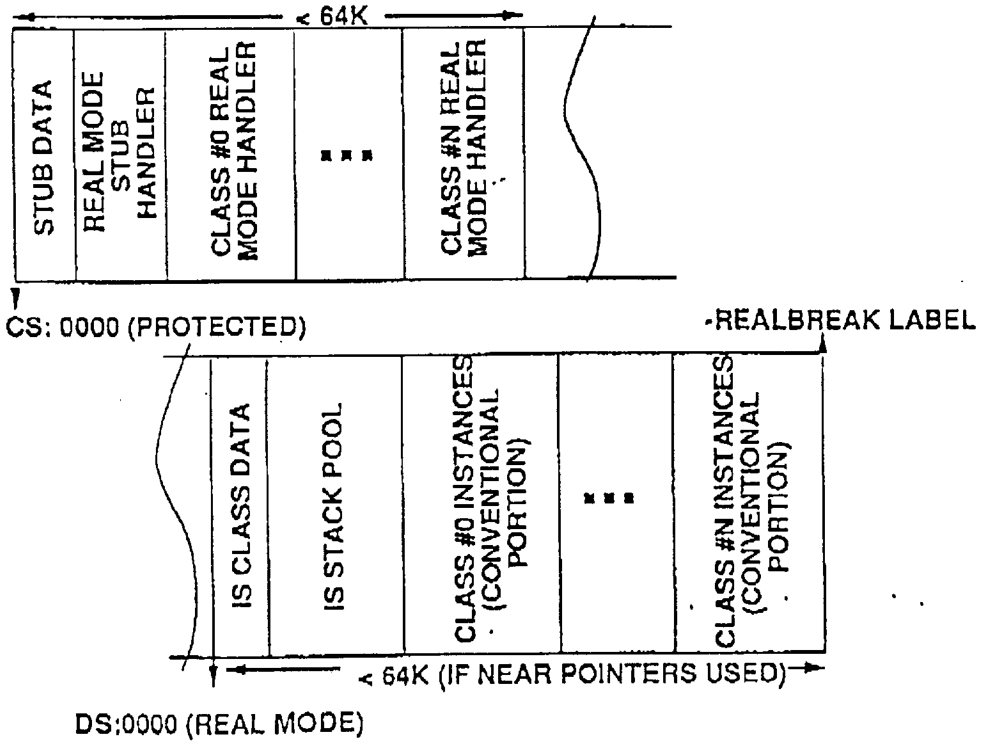
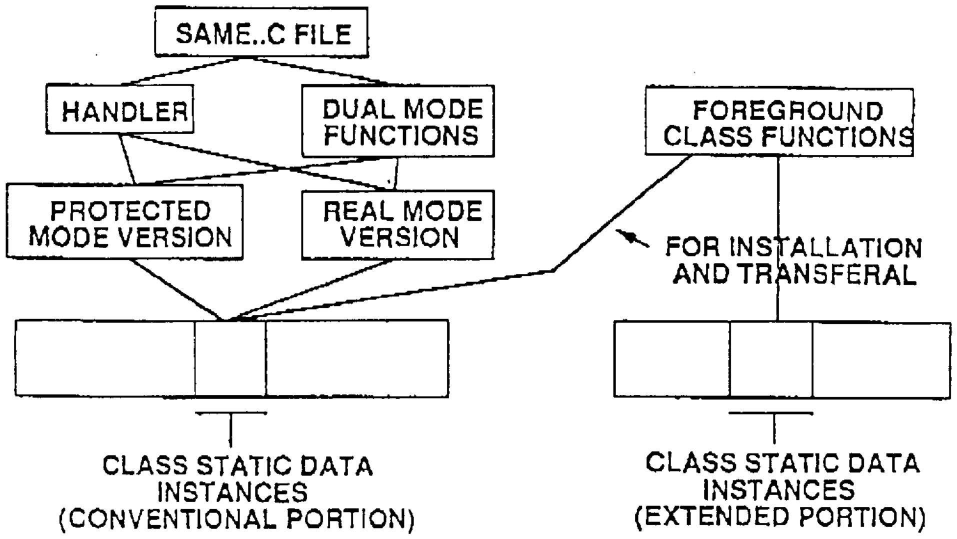

< 64K
STUB DATA
REAL MODE STUB HANDLER
CLASS #0 REAL MODE HANDLER
CLASS #N REAL MODE HANDLER
CS: 0000 (PROTECTED)
REALBREAK LABEL
IS CLASS DATA
IS STACK POOL
CLASS #0 INSTANCES (CONVENTIONAL PORTION)
CLASS #N INSTANCES (CONVENTIONAL PORTION)
< 64K (IF NEAR POINTERS USED)
DS:0000 (REAL MODE)

FIG. 3: REAL-TIME CLASS STRUCTURE


SAME..C FILE
HANDLER
DUAL MODE FUNCTIONS
FOREGROUND CLASS FUNCTIONS
PROTECTED MODE VERSION
REAL MODE VERSION
FOR INSTALLATION AND TRANSFERAL
CLASS STATIC DATA INSTANCES (CONVENTIONAL PORTION)
CLASS STATIC DATA INSTANCES (EXTENDED PORTION)